

HOWTO zum MySQL-Einsatz

(C) 2006-2011 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>
 OSTC Open Source Training and Consulting GmbH, <http://www.ostc.de>
 \$Id: mysql-HOWTO.txt,v 1.167 2011-10-21 14:26:13 tsbirn Exp \$

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

HINWEIS: MySQL-Spezialitäten sind durch "MY!" oder "MY!N.M" gekennzeichnet
 (falls sie erst 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) Client "mysql" starten (Verbindung/Session)
 - 1d) Client "mysql" effizient bedienen (interaktiv)
 - 1e) Client "mysql" verlassen
 - 1f) Client "mysql" von der Kommandozeile aus nutzen (Batch-Modus)
 - 1g) MySQL-Clients: Allgemeine Optionen
 - 1h) Client "mysql": Spezielle Optionen
 - 1i) Client "mysql": Sonstige Optionen
 - 1j) Client "mysql": Umgebungsvariablen
 - 1k) Client "mysql": Kommandos und Escape-Sequenzen
 - 1l) Client "mysql": Hilfe zu Kommandos anzeigen
 - 1m) Grafische MySQL-Programme (GUI)
 - 1n) MySQL-Kommandozeilen-Programme
- 2) Syntax
 - 2a) Leerraum und Formatierung
 - 2b) Zeichenketten (Strings) und Quotierung
 - 2c) Escape-Sequenzen
 - 2d) Zahlen
 - 2e) Datenbank-Objekte
 - 2f) Bezeichner (Identifizier)
 - 2g) GROSS/kleinschreibung
 - 2h) Kommentare
- 3) Typische MySQL-Datenbankbefehle (Tutorial)
- 4) MySQL-Datentypen
 - 4a) Datentyp-Optimierung (Performance/Speicherplatz)
 - 4b) Fixes/Variables Rowformat
 - 4c) AUTO_INCREMENT
- 5) MySQL-Operatoren
- 6) Boolesche Logik
- 7) Der Wert NULL
 - 7a) Vergleiche mit NULL
 - 7b) Boolesche Logik mit NULL (dreiwertig)
- 8) Reguläre Ausdrücke in MySQL
- 9) MySQL-Funktionen
 - 9a) Aggregatfunktionen und Gruppierung (Aggregation)
- 10) Schlüsselfelder (Keys) und Indices
- 11) Index-Optimierung
- 12) Fremdschlüssel (Foreign Keys) und Referenzielle Integrität
- 13) Joins
- 14) Mengenoperationen
- 15) Unterabfragen (Subqueries/Subselect)
- 16) Transaktionen
- 17) Locking
- 18) Views (Sichten)
- 19) Variablen
- 20) Prepared Statements (Vorbereitete Anweisungen)
- 21) Routinen (Stored Procedures)
 - 21a) Lokale Variablen
 - 21b) Wertzuweisung an Variablen
 - 21c) Ausgabe von Variablen oder Daten
 - 21d) Kontrollstrukturen (Block/Compound Statement)
 - 21e) Kontrollstrukturen (Verzweigungen)
 - 21f) Kontrollstrukturen (Schleifen)
 - 21g) Prozeduren
 - 21h) Funktionen

zum Absetzen von SQL-Befehlen an einen MySQL-Server und zur Ausgabe ihrer Ergebnisse (MySQL-"Terminal/Monitor"). Trotzdem ist er sehr leistungsfähig, weil MySQL nahezu vollständig über SQL-Befehle steuerbar ist. Um diese Schnittstelle nutzen zu können, sind allerdings gute Kenntnisse der SQL-Befehle 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-Befehle 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 verhinderbar durch:

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

Typische Aufrufe des "mysql"-Clients:

Kommando	Bedeutung
mysql	Als angemeldeter Benutzer aufrufen
mysql -utom	Benutzer "tom" ohne Passwort (unsinnig!)
mysql -utom -p	Benutzer "tom" (Passwort abfragen)
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 verb. (/var/lib/mysql/mysql.sock bzw. /var/run/mysqld/mysqld.sock als Std.)
mysql -utom -pgeheim -Dprod	Datenbank "prod" auswählen (Variante A)
mysql -utom -pgeheim prod	Datenbank "prod" auswählen (Variante B)

Nach der Anmeldung erscheint ein PROMPT folgender Form, der die interaktive Eingabe beliebiger SQL-Anweisungen (Statements) und ihre Ausführung durch Abschluss der Eingabe mit <RETURN> erlaubt (alle SQL-Kommandos --- außer "USE <Db>" und "QUIT" --- sind mit ";" abzuschließen!)

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

Um z.B. alle Datenbanken anzuzeigen, ist folgender Dialog zu führen (Kommando wird hier GROSS geschrieben, GROSS/kleinschreibung ist aber eigentlich egal):

```
mysql> SHOW DATABASES;<RETURN> # Prompt + Benutzer-EINGABE (";" notwendig)
+-----+ # Rahmen (AUSGABE)
| Database | # Spaltenüberschriften (AUSGABE)
+-----+ # Rahmen (AUSGABE)
| information_schema | # 1. Ergebniszeile (AUSGABE)
| mysql | # 2. Ergebniszeile (AUSGABE)
| phpmyadmin | # 3. Ergebniszeile (AUSGABE)
| ... | # ... (AUSGABE)
+-----+ # Rahmen (AUSGABE)
15 rows in set (0.01 sec) # Statistik (15 Ergeb.zeilen + Laufzeit)
mysql> # (Leerzeile)
mysql> # Prompt nächste Benutzer-EINGABE
```

oder (leeres Ergebnis):

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

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 Spaltenüberschriften. Am Ende wird die Anzahl der erhaltenen Zeilen und die Dauer der Abfrage 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
| SIN(PI()/2) | (4+1)*5 |
+-----+-----+
|          1 |        25 |
+-----+-----+
1 row in set (0.00 sec) # Statistik (eine Ergeb.zl. + Laufzeit)
```

Kommandos müssen nicht auf einer Zeile stehen, sondern dürfen über mehrere Zeilen verteilt werden. MySQL erkennt das Kommandoende am abschließenden ";" :

```
mysql> SELECT user,
->         password,
->         host
-> FROM mysql.user
-> ;
```

Unvollständige SQL-Kommandos (z.B. wird häufig einfach der ";" vergessen) führen zur Anzeige eines der folgenden Fortsetzungs-Prompts (dann einfach die restlichen Kommandoteile oder den fehlenden ";" tippen und <RETURN> drücken):

Prompt	Bedeutung
->	Kommando unvollständig bzw. ";" vergessen
">	"..." begonnen aber noch nicht beendet (String)
'>	'...' begonnen aber noch nicht beendet (String)
`>	`...` begonnen aber noch nicht beendet (Bezeichner)
/*>	/*...*/ begonnen aber noch nicht beendet (Kommentar)

Teilweise eingetipptes Kommando abbrechen (c=cancel, Escape-Sequenz):

```
mysql> CMD...\c
```

Weitere Einstellungen zur Verbindung:

```
--connect_timeout=<N> # Anzahl Sek. bis autom. Verbindungstrennung (Std: 0)
--max_allowed_packet=<N> # Max. Länge transf. Befehle/Ergebnisse (Std: 16MB)
--net_buffer_length=<N> # Puffergröße der TCP/IP+Socket-Komm. (Std: 16KB)
```

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

1d) Client "mysql" effizient bedienen (interaktiv)

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) erreichbar. Der ";" am Ende darf hier weggelassen werden (besser erst gar nicht daran gewöhnen!):

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

Die alten Befehle können mittels der Cursortasten durchgeblättert, editiert und erneut mit <RETURN> abgeschickt werden. Mehrzeilig eingegebene Befehle werden dabei zu einer (langen) Zeile zusammengezogen, die evtl. schwer zu editieren ist. Sie stehen auch nach dem Verlassen und erneuten Aufrufen des Clients "mysql" zur Verfügung, da sie PRO BENUTZER in einer Datei gespeichert werden (d.h. Unterschied root <-> normaler Benutzer):

```
~/.mysql_history
```

Befehle oder Befehlssteile können auch mit der Maus (linke Taste + mittlere Taste) in die Kommandozeile des MySQL-Clients kopiert werden. Wird bis zum rechten Terminalrand kopiert, ist der Befehlsabschluss <RETURN> gleich mit dabei (unter Linux).

Datenbank-, Tabellen- und Spalten-Namen (aber keine anderen SQL-Syntaxelemente) können per <TAB>-Taste automatisch vervollständig werden (Tab-Completion). Dazu per "-D <Db>" oder "use <Db>;" eine Default/Standard-Datenbank auswählen und die Option "--auto-rehash" setzen oder der Befehl "rehash" eingeben (verlangsamt den Start etwas) um den Datenbankinhalt einzulesen.

Der Bildschirm wird durch <Strg-L> gelöscht (deu: L=Löschen ;-).

Hilfe zum Aufruf des Client "mysql" (Optionen, Konfigurationsdateien, Variablen) anzeigen durch:

```
mysql --help
```

```
mysql -?
```

Hilfe im MySQL-Client durch folgende Befehle anzeigen (die Hilfetexte stehen direkt in der Datenbank in den Tabellen "mysql.help..."):

Befehl	Bedeutung
HELP	Liste der MySQL-Client-Befehle ausgeben
HELP help	Hilfe zur Hilfe ausgeben
HELP <SqlCmd>...	Hilfe zu SQL-Befehl ausgeben (eindeutiger Präfix genügt)
HELP contents	Themenliste ausgeben
HELP <Topic>	Hilfe zu einem Thema aus Themenliste ausgeben

TIPP: Mit Option `--i-am-a-dummy` / `--safe-updates` / `-U` werden UPDATE- und DELETE-Anweisungen gegen das Weglassen einer WHERE-oder LIMIT-Klausel geschützt (d.h. das versehentliche Ändern/Löschen ALLER Datensätze verhindert).

TIPP: Ebenso werden max. 1.000 Ergebniszeilen pro SELECT ausgegeben und ein SELECT auf mehreren Tabellen abgebrochen, das mehr als 1.000.000 Zeilenkombinationen zur Auswertung erfordert. Die beiden letzten Beschränkungen beim Aufruf von "mysql" ändern per:

```
--select_limit=<N>      # Std: 1.000
--max_join_size=<N>     # Std: 1.000.000
```

TIPP: SQL-Befehle aus externer Datei IM "mysql"-Client einlesen per (Include):

```
SOURCE <SqlFile>;      # kein "..." um <SqlFile>!
```

TIPP: SQL-Befehle aus externer Datei per "mysql"-Client einlesen (Umleitung):

```
mysql OPTIONEN < <SqlFile>      # USE <Db>; nicht vergessen
mysql OPTIONEN -D<Db> < <SqlFile> # USE nicht notwendig
```

le) Client "mysql" verlassen

Befehl	Bedeutung
quit	
exit	
\q	Quit (Escape-Sequenz)
Strg-C	Cancel (deaktivieren mit "--sigint-ignore")
Strg-D	D=Dateiende (nur Linux)
Strg-Z	Z=Dateiende (nur Windows)

TIPP: Das Drücken von Strg-C zum Abbrechen des aktuellen SQL-Befehls 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.

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

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

TIPP: Aliase für MySQL-Anmeldung + Umschalten auf aktuell relevante Datenbank definieren und in "~/.alias" oder "~/.bash_aliases" ablegen (-D=Standard-DB, -t=Linien 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-Befehlen einlesen
mye 'SELECT * FROM pers' # SQL-Befehl direkt ausführen
```

lf) Client "mysql" von der Kommandozeile aus nutzen (Batch-Modus)

Neben der interaktiven Nutzung kann der Client "mysql" auch im Batch-Modus benutzt werden. Dazu ist der SQL-Befehl direkt auf der Kommandozeile in "...", oder '...' gesetzt anzugeben. Mehrere SQL-Befehle sind durch ";" trennen, beim letzten SQL-Befehl darf der ";" weggelassen werden.

Alternativ können die SQL-Befehle per Eingabe-Umleitung aus einer Datei

<SqlFile> eingelesen werden (analog SQL-Befehl "SOURCE"). Ebenso können die Ergebnisse der SQL-Kommandos statt auf dem Bildschirm durch Ausgabe-Umleitung auch in eine Datei <OutFile> ausgegeben werden.

Kommando	Bedeutung
mysql -u<User> -p -e "<SqlCmd>" ... -e "USE first; SELECT * FROM pers" ... -e "USE first; INSERT INTO pers VALUES (100, 'Norbert', 'Böhm'); INSERT INTO pers VALUES (101, 'Hänschen', 'Klein)'"	execute execute execute (mehrere Kommandos durch ";" trennen!) (Statement in "... " setzen, Strings darin in '...')
mysql -u<User> -p < <SqlFile> ... -u<User> -p -D<Db> < <SqlFile> ... > <OutFile>	Eingabe von Datei <SqlFile> USE <Db> nicht vergessen! Ausgabe in Datei <Outfile>

Die Eingabedatei <SqlFile> wird meist per Hand erstellt und kann neben der Definition von Tabellen auch Daten zum Füllen der Tabellen enthalten. Häufig ist die Eingabedatei auch ein vollständiger Dump einer MySQL-Datenbank per "mysqldump".

```
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):

```
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
mysql -utom -pgeheim -Dtest < test-dump.sql # DB-Dump "test" einspielen
```

Alternativ folgende zwei Anweisungen am Anfang der Dumpdatei "test-dump.sql" hinzufügen (notwendig):

```
CREATE DATABASE test; # Datenbank "test" anlegen (leer)
USE test; # Als Default-Datenbank auswählen
```

Und dann die Sicherung der Datenbank einspielen:

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

1g) MySQL-Clients: Allgemeine Optionen

Optionen sind in Kurzform "-X" sowie in Langform "--XXX" angebar.

Bei ALLEN MySQL-Kommandozeilen-Clients sind folgende Optionen angebar (legen die Art und Weise der Verbindung fest):

Kurzform	Langform	Bedeutung
-u<User>	--user=<User>	Benutzer festlegen
-p[<Pass>]	--password[=<Pass>]	Passwort angeben/abfragen (besser!)
-h<Host>	--host=<Host>	Ziel-Host (Std: localhost)
-D<Db>	--database=<Db>	Datenbank auswählen
-P<Port>	--port=<Port>	Ziel-Port (Std: 3306)
-S<File>	--protocol=<Proto> --socket=<File>	Verb.protokoll (TCP/SOCKET/PIPE/MEMORY) Socket (Std: /var/lib/mysql/mysql.sock bzw. /var/run/mysqld/mysqld.sock)
-V	--version	Versioninformation ausgeben

1h) Client "mysql": Spezielle Optionen

Optionen sind in Kurzform "-X" sowie in Langform "--XXX" angebar.

Beim Client "mysql" sind folgende weiteren Optionen angebar (legen das Verhalten bei fehlerhaften SQL-Anweisungen und das Ausgabeformat fest):

Kurzform	Langform	Bedeutung
----------	----------	-----------

-e "<Cmd>"	--execute="<Cmd>"	SQL-Kommandos ausführen
-f	--force	Bei SQL-Fehler nicht abbrechen
	--show-warnings	Warnungen nach jeder Anw. anzeigen
-t	--table	ASCII-Rahmen (Std bei interaktiv)
-H	--html	HTML-Format (1 Zeile ohne NL!)
-X	--xml	XML-Format (mehrzeilig + eingerückt)
-E	--vertical	vVertikale-Ausgabe (nützlich!)
-B	--batch	Spalten TAB-getrennt (Std bei batch)
-r	--raw	Ohne Escape-Umwandlung ausgeben (*)
-s	--silent	Weniger Info ausgeben
-v	--verbose	Mehr Info ausgeben (mehrfach)
-V	--version	Versioninfo 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 -> Escape-Sequenzen NICHT bei Angabe der Optionen -t, -X, -E, -r, ansonsten schon.

Im interaktiven Betrieb (Eingabe-Gerät ist EIN Terminal) ist automatisch die Option -t (ASCII-Rahmen) gesetzt, d.h. die Ausgabedaten werden auf einheitliche Breite formatiert und mit "Rahmen" versehen.

Im Batch-Betrieb (Eingabe-Gerät ist KEIN Terminal) ist automatisch die Option -B (batch) gesetzt und die Ausgabedaten werden nicht auf einheitliche Breite formatiert, sondern durch je einen TABULATOR getrennt.

Für das menschliche Auge ist die erste Form angenehmer zu lesen, für den Computer ist die zweite Form besser zu verarbeiten.

Bei besonders breiten Tabellen ist das per Option "-E" aktivierte vVertikale Format sinnvoll. Die Spalten eines Datensatzes werden dann zeilenweise ausgegeben und jeder Datensatz durch eine Zeile "**** N. row ****" eingeleitet. Alternativ im MySQL-Client statt ";" als Anw.-Abschluss ein "\G" angeben.

```

***** 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)

```

li) Client "mysql": Sonstige Optionen

	Langform	Bedeutung
-?	--help	Hilfe anzeigen (Usage-Meldung)
	--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
	--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=OPTIONS	Debugoptionen angeben
-T	--debug-info	
	--debug-check	Einige Debuginfo am Programmende ausg.
	--default-character-set=NAME	Standard-Zeichensatz
	--character-sets-dir=PATH	Verz. der Zeichensätze
	--delimiter=STRING	SQL-Kmdo.begrenzer def. (Std: ";")
-L	--line-numbers	Zeilennummer bei Fehler ausgeben (Std)
-L	--skip-line-numbers	Zeilennummer bei Fehler NICHT ausgeb.
	--ssl*	Client + Server per SSL verbinden
	--tee=FILE	Ausgabe auf Datei FILE zusätzlich
	--no-tee	Ausgabe auf Datei wieder abschalten
-n	--unbuffered	Puffer nach jeder Abfrage leeren
	--buffered	??
-w	--wait	Warten+neu versuch. Verb. aufzubauen

	--reconnect	Bei Verb.verlust Verb. wieder aufbau.
	--skip-reconnect	Bei Verb.verlust Client verlassen
-c	--comments	Kommentare zum Server senden
	--skip-comments	Kommentare NICHT zum Server senden
-t	--tmpdir	Temporäres Verzeichnis
	--defaults-group-suffix	??
	--named-commands	??
-g	--no-named-commands	?? (alt)
-g	--disable-named-commands	?? (neu)
	--skip-named-commands	??
-i	--ignore-spaces	Leerzeichen nach Fkt.name ignorieren
	--pager=PATH	Seitenweises blättern
	--no-pager	Kein seitenweises blättern
-o	--one-database	Nur -D<Db> zählt, USE <DB> ignoriert
-q	--quick	Abfrageergebnisse nicht Cachen

1j) Client "mysql": Umgebungsvariablen

Folgende Umgebungsvariablen werden vom Client "mysql" ausgewertet:

MYSQL_DEBUG	Debug-Trace-Optionen
MYSQL_GROUP_SUFFIX	Analog --defaults-group-suffix
MYSQL_HISTFILE	MySQL-History-Dateipfad (Std: "\$HOME/.mysql_history")
MYSQL_HOME	Pfad zu Server-spezifischer "my.cnf"-Datei
MYSQL_HOST	Standard-Hostname
MYSQL_PS1	Client-Prompt-String (Std: "mysql>")
MYSQL_PWD	Standard-Passwort (unsicher!)
MYSQL_TCP_PORT	Standard-TCP/IP-Port (Std: 3306)
MYSQL_UNIX_PORT	Standard-Socket (bei "localhost" benutzt)
PAGER	Programm zum seitenweisen blättern (--prompt)
EDITOR	Editor für interaktives Bearbeiten (1.Wahl)
VISUAL	Editor für interaktives Bearbeiten (2.Wahl)
USER	Standard-Benutzer
HOME	Heimatverzeichnis des Benutzers
PATH	Shell-Suchpfad für MySQL-Programme
LD_RUN_PATH	Suchpfad für Bibliothek "libmysqlclient.so"
TMPDIR	Pfad wo Temporäre Tab./Dateien erzeugt (--tmpdir/-t)
UMASK	Maske für Dateien (and UMASK!)
UMASK_DIR	Maske für Verzeichnisse (and UMASK_DIR!)
TZ	Lokale Zeitzone
DBI_USER	Benutzername für Perl DBI-Modul
DBI_TRACE	Trace-Optionen für Perl DBI-Modul
CC	Name des C-Compilers (für "configure")
CXX	Name des C++-Compilers (für "configure")
CFLAGS	Flags für C-Compiler (für "configure")
CXXFLAGS	Flags für C++Compiler (für "configure")

1k) Client "mysql": Kommandos und Escape-Sequenzen

Neben den SQL-Kommandos versteht der Client "mysql" noch folgende internen Kommandos bzw. ihre Abkürzung in Form eines zweibuchstabigen ESC-Befehls:

Kommando	ESC	Bedeutung
? [<ARG>]	\?	Synonym für "help" (Argument <ARG>)
charset <C>	\C	Zeichensatz <C> einschalten (für binlog-Verarbeitung)
clear	\c	SQL-Kommando abbrechen
connect [<D><H>]	\r	Serverbindung neu aufbauen (Opt: <DB> und <HOST>)
delimiter <S>	\d	SQL-Kmdobegrenzer <S> def. (Zeilenrest, Std: ";")
edit	\e	Letztes SQL-Kommando mit \$EDITOR bearbeiten
ego	\G	SQL-Kommando ausführen (vErtikale Anzeige)
exit	\q	Client "mysql" verlassen (auch Strg-C, Strg-D)
go	\g	SQL-Kommando ausführen (Std: ASCII-Tabellen Anz.)
help	\h	Diese Hilfe anzeigen
nopager	\n	Seitenweise Anzeige aus (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	Aktuelles SQL-Kommando ausgeben
prompt [<STR>]	\R	"mysql"-Client-Prompt auf <STR> ändern (Std: "mysql>")

quit	\q	Client "mysql" verlassen (auch Strg-C, Strg-D)
rehash	\#	TAB-Vervollständigungs-Hash aufbauen (TAB-Completion)
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
warnings	\W	Warnungen nach jeder Anweisung anzeigen

1l) Client "mysql": Hilfe zu Kommandos anzeigen

Im Client "mysql" ist zu den MySQL-Befehlen jederzeit Hilfe anzeigbar (die Hilfetexte werden der internen Verwaltungs-Datenbank "mysql" aus den Tabellen "mysql.help_*" entnommen).

Hilfe-Kommando	Bedeutung
?	Obige Befehlsliste
\?	Obige Befehlsliste
\h	Obige Befehlsliste
help	Obige Befehlsliste
help help	Hilfe zur Hilfe
help <SqlCmd>	Hilfe zu SQL-Kommando (z.B. JOIN)
help contents	Hilfethemenliste
help <Topic>	Hilfe zu Thema aus Themenliste
... Account Management	Hilfethema (Kommandos)
... Administration	"
... Compound Statements	"
... Data Definition	"
... Data Manipulation	"
... Data Types	"
... Functions	"
... Functions and Modifiers for Use with GROUP BY	"
... Geographic Features	"
... Language Structure	"
... Storage Engines	"
... Stored Routines	"
... Table Maintenance	"
... Transactions	"
... Triggers	"
... User-Defined Functions	"
... Utility	"

1m) Grafische MySQL-Programme (GUI)

Neben dem bedienungstechnisch relativ einfachen aber trotzdem leistungsfähigen allgemeinen Client "mysql" sind noch eine Reihe anderer spezieller MySQL-Clients verfügbar, die alle eine GUI anbieten und mit der Maus bedienbar sind.

GUI-Programm	Beschreibung
MySQL GUI Tools MySQL Administrator MySQL Query Browser MySQL Migration Toolkit	Zusammenfassung von: Server-Konfiguration, -Verwaltung, -Wartung Grafischer SQL-Client Schemata + Daten aus anderen DB importieren
MySQL Workbench	Datenbank-Designer (Windows, kommerziell)
phpMyAdmin mysql-admin	Webbasiertes DB-Management (Webserver nötig) Webbasiertes DB-Management (Webserver nötig)
winMySQLadmin SQL-Front SQLyog MySQLManager Navicat for MySQL EMS MySQL Manager MyAdmin HeidiSQL (MySQL-Front)	GUI-Client (Windows, Std., MUSS lokal laufen!) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell) GUI-Client (Windows, kommerziell)
Knoda Rekall Ksql KMySQLadmin	GUI-Client analog Access (Linux) GUI-Client (Linux) GUI-Client (Linux) 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)

1n) MySQL-Kommandozeilen-Programme

Es gibt eine Reihe von mitgelieferten Verwaltungs- und Dienstprogrammen:

- * Zur Datenbank-Erstellung und -Bearbeitung
- * Zur Datenbank- bzw. Tabellen-Reparatur
- * Zur Administration + Überwachung

Damit ist der Fernzugriff auf MySQL betriebssystem-unabhängig möglich:

- * Netzwerk funktionsfähig
- * Dienstprogramm auf lokalem Rechner
- * Anmeldung -h<Host> -u<User> -p -> <User>@<Host>

Dienstprogramme (meist MySQL-Clients, manche arbeiten auch direkt auf den Datenbankdateien):

- S = Serverprogramm
- CS = Clientprogramm, das auf Server zugreift
- C = Clientprogramm unabhängig vom Server (Dateizugriff)
- I = Installationsprogramm
- H = Hilfsprogramm
- T = Testprogramm
- P = Programmierung

	Dienstprogramm	Zweck
CS	mysql	Interaktiver Befehlszeilen-Client (auch Batch, alle Kmdos)
CS	mysqlaccess	Zugriffsrechte für USER+HOST+DB-Kombinationen überprüfen
CS	mysqladmin	Datenbank-Administration (nur Admin-Befehle)
C	mysqldumpslow	Logdatei mit "Langsamen SQL-Anweisungen" anzeigen+zusfass.
CS	mysqlreport	Bericht über MySQL Status erstellen (SHOW STATUS)
CS	mysqlshow	Struktur von DB-Objekten ansehen (DB, Tabelle, Spalte)
CS	mysqlcheck	Tabellen analysieren/prüfen/optimieren/reparieren (online)
CS	mysqlanalyze	Symb. Link auf "mysqlcheck" (-a = analysieren)
CS	mysqloptimize	Symb. Link auf "mysqlcheck" (-r = optimieren)
CS	mysqlrepair	Symb. Link auf "mysqlcheck" (-o = reparieren)
C	innochecksum	InnoDB-Tabellen prüfen (offline)
C	myisamchk	MyISAM-Tabellen prüfen/optimieren/reparieren (offline)
C	myisampack	MyISAM-Tabellen komprimieren für ReadOnly-Zugr. (offline)
C	myisam_ftdump	Info über MyISAM Fulltext-Indices ausgeben
C	myisamlog	MyISAM-Logdateien verarbeiten
C	isamlog	MyISAM-Logdateien verarbeiten
C	mysqlbinlog	Anweisungen aus Binärlog lesen (Rebuild nach Absturz)
C	mysql_explain_log	SQL-Querylog analysieren (per EXPLAIN)
C	mysql_find_rows	SQL-Anw. aus Dateien extrahieren (Updatelog, Regex)
CS	mysqldump	Backup von Datenbanken/Tabellen in SQL-Befehlsform
CS	mysqlimport	Textdatei in div. Formaten importieren (Name = <Tab>name)
CS	mysqlhotcopy	Backup MyISAM/ARCHIVE-Tab. binär (online, Filezugriff = nur Serverlokal)
IS	mysql_install_db	Datenverz. + System-Tab. (GRANT) des Servers erstellen
S	mysqlmanager	MySQL Instance Manager ("mysqld_multi"-Ersatz, -MY!5.5)
S	mysqld	Server (zum Start "mysqld_safe" verwenden)
S	mysqld_safe	Server-Startskript (früher "safe_mysqld")
S	mysqld.server	Server-Startskript (SysV-Systeme)
S	mysqld_multi	Server-Startskript (Manager für mehrere Server)
I	comp_err	MySQL Fehlernachrichten-Datei übersetzen (Installation)
C	perror	Bedeutung der Fehlercodes erklären
H	replace	Textersetzung in Dateien durchführen
H	resolveip	Abbildung Hostname <-> IP-Adresse auflösen
I	mysqlbug	Interaktiv Fehlerbeschreib. erstellen + an MySQL mailen
I	mysql2mysql	mSQL-C-Funktionsaufrufe -> mysql-C-Funkt. konvertieren
I	my_print_defaults	Konfig.opt. einer Konf.Gruppe ([client], ...) ausgeben
S	mysql_waitpid	Server-Prozess beenden (und darauf warten)
S	mysql_zap	Server-Prozesse beenden (per Regex-Muster)
	mysql_convert_table_format	Tabellen in andere Storage Engine konvertieren

	mysql_fix_extensions	MyISAM-Dateiext. konvert. frm/MYI/MYD/ISD/ISM
	mysql_secure_installation	Sicherheit einer Installation verbessern
	mysql_setpermission	Zugriffsrechte in GRANT-Tab. interaktiv setzen
	mysql_tableinfo	Tab. aus Metadaten erz. (ab MY!5.0 INFORMATION_SCHEMA)
	mysql_tzinfo_to_sql	Zeitzone-Tabelle in Server laden
	mysql_fix_privilege_tables	Rechte-Systemtab. upgraden (veraltet)
	mysql_upgrade	(Rechte-)Systemtab. upgraden (neue Version)
	mysql_upgrade_shell	" " " " " "
T	mysqlslap	Simulation mehrerer Clients für Lasttests
T	mysql-test-run.pl	Test des Servers steuern
T	mysqltest	Test des Servers und Vergleich mit and. Erg.
T	mysqltest_embedded	Test des Servers und Vergleich mit and. Erg.
T	mysql_client_test	Test der MySQL-Client-API durchf. (Skript)
T	mysql_client_test_embedded	Test der MySQL-Client-API durchf. (Skript)
P	mysql_config	Compiler-Optionen zum Übersetzen extrahieren
P	make_binary_distribution	Binärpaket aus MySQL-Install. erzeugen (Unix)
P	make_win_bin_dist	Binärpaket aus MySQL-Install. erzeugen (Windows)
P	resolve_stack_dump	Numerischen Stack Trace Dump in Symbole auflösen
	mysql-binlog-dump	?
	mysql-mysam-dump	?

Zur Überwachung der MySQL-Prozesse und -Engines sind folgende Kommandozeilentools verfügbar:

Tool	Bedeutung
innotop	MySQL- und InnoDB Transaktions/Threadliste (analog "top")
mtop	MySQL-Threadliste (analog "top")
mytop	MySQL-Threadliste (analog "top")

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 = vertikale Ausgabe) schließen eine SQL-Anweisung ab (einzige Ausnahme: 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 '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: "..."
```

Zwischen Zeichenketten in "..." und '...' gibt es KEINEN Unterschied (wie in anderen Programmiersprachen bei den Escape-Sequenzen \C), allerdings ist nur die

Form '...' ANSI-SQL-kompatibel ("..." wird dort verwendet, um Bezeichner mit Sonderzeichen zu quotieren).

2c) Escape-Sequenzen

Escape-Sequenzen \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 "...")
\b	Backspace
\n	Newline (Linefeed)
\r	Carriage Return
\t	Tabulator
\Z	ASCII 26 (Control-Z, unter Windows evtl. notwendig)
\\	Backslash (\)
\%	%-Zeichen (Zeichen "%" selbst in LIKE)
_	_Zeichen (Zeichen "_" selbst in LIKE)

2d) Zahlen

Wird eine Zeichenkette in einem Zahlensammenhang verwendet, dann wird das max. Anfangsstück (PRÄFIX), das noch wie eine Zahl aussieht, AUTOMATISCH in diese Zahl KONVERTIERT (und eine Warnung ausgegeben). 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, "ghi" + 100;             # -> 0, 100
```

Kommazahlen sind (unabhängig von Sprach- und Collation-Einstellungen) immer mit DEZIMALPUNKT zu schreiben (Dezimalkomma und Tausendertrennzeichen sind nicht erlaubt). Die Ausgabe von Dezimalkommata ist per FORMAT(<Zahl>, <Nkst>) erreichbar. Rundung auf feste Anzahl Nachkommastellen oder ganze Zahlen 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"), 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>	Vertikale Zerlegung einer Tabelle
Prepared Statement	<PrepStm>	Vorkompilierte Anweisung
Stored Function	<Func>	Funktion (Rückgabewert)
Stored Procedure	<Proc>	Prozedur (kein Rückgabewert)
Table	<Tbl>	Tabelle
Tablespace	<TblSpace>	Datei für Tabellen (analog Partition)
Trigger	<Trig>	Trigger einer Tabelle
User	<User>	Benutzeraccount
View	<View>	Sicht (vordef. gespeicherte Abfrage)

2f) Identifizier (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 Hallo                FROM test;    # OK (Std-Form)
SELECT `Hallo`             FROM test;    # OK (Std-Form)
SELECT _Hallo_             FROM test;    # OK (Std-Form)
SELECT $Hallo$             FROM test;    # OK (Std-Form)
SELECT Und_Ein_Name$       FROM test;    # OK (Std-Form)
SELECT `mit Leer zeichen`  FROM test;    # OK (Sonderzeichen)
SELECT `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! Generiert MySQL SQL-Anweisungen (z.B. mit "mysqldump"), so werden alle Bezeichner grundsätzlich prophylaktisch in `...` gesetzt (auch wenn das gar nicht notwendig wäre).

```

CREATE TABLE alter (...);           # FALSCH!
CREATE TABLE `alter` (...);         # OK
SELECT user, host FROM mysql.user;   # OK
SELECT `user`, `host` FROM `mysql`.`user`; # OK
SELECT `user`, `host` FROM `mysql.user`; # FALSCH!
SELECT "user", "host" FROM "mysql"."user"; # FALSCH!
SELECT `user`, `host` FROM `mysql`.`user`; # FALSCH!

```

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

Bezeichner	Max
Datenbank	64
Tabelle	64
Spalte	64
Routine	64
Alias	255
Host	60
Benutzer	16
Passwort	32

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

Bezeichner	Bedeutung
<Db>.<Tbl> <Db>.<Tbl>.<Col> <Db>.<Tbl>.*	A) Vollständiger Pfad (immer OK!) Analog Analog (alle Spalten der Tabelle)
<Tbl> <Tbl>.<Col> <Tbl>.*	B) Relativ zu Default/Standard-Datenbank (USE <Db>) Analog 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 Aliasen OK. Analog (alle Spalten einer Tabelle)
<Db>.<Proc> <Proc>	D) Vollständiger Pfad E) Relativ zu Default/Standard-Datenbank (USE <Db>)

```

+-----+
SELECT test.pers.name FROM test.pers; # A) OK (Datenbank "test")
USE test; # Default/Standard-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 Spaltennamenkollision
SELECT pers.name, age.name FROM pers, age; # C) OK (auch bei zusätzl. Sp.)

```

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-Kommandos zu Syntaxfehlern, wenn darin auf diese Spalte zugegriffen wird. Daher am besten mit Variante B) arbeiten und zur Abkürzung ALIASe einführen.

Die Bestandteile eines relativen oder voll qualifizierten Bezeichners müssen getrennt mit `...` quotiert werden, falls Quotierung notwendig ist:

```

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

```

Analog müssen die beiden Bestandteile "User" und "Host" eines Benutzernamens "User@Host" getrennt quotiert werden (allerdings mit `...` oder `...`, da es sich um keinen Bezeichner handelt):

```

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

```

2g) GROSS/kleinschreibung

Beim Zugriff auf und beim Sortieren von Daten in Tabellen IGNORIERT MySQL standardmäßig die GROSS/kleinschreibung. BINARY vor einem String in den Vergleichen = != <> <=> < <= > >= BETWEEN IN LIKE REGEX und in ORDER BY ERZWINGT aber die Beachtung der GROSS/kleinschreibung im Vergleich. Ebenso erzwingen die Datentypen BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB und LONGBLOB die Beachtung der GROSS/kleinschreibung.

Die GROSS/kleinschreibung in SQL-Anweisungen ist meist egal AUSSER bei den Bezeichnern von Datenbanken, Tabellen, Views und Logfilegroups unter bestimmten Betriebssystemen mit case-sensitiven Dateisystemen (da als Verzeichnis/Datei abgelegt). Die Namen von Triggern, Labeln (Marken), Benutzernamen und Passwörtern sind immer "case-sensitive". Es gibt daher gewissen KONVENTIONEN bzgl. 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
Dateiname/pfad	JA (Linux), NEIN (Windows)	klein
Host-Name	NEIN	klein
Benutzer-Name	JA	klein
Passwort	JA	---
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
View-Bezeichner	JA (Linux), NEIN (Windows)	klein
Prepared-Stm-Bezeichner	NEIN	klein
Prozedur-Bezeichner	NEIN	klein
Funktions-Bezeichner	NEIN	klein
Trigger-Bezeichner	JA	klein
Event-Bezeichner	NEIN	klein
Logfilegroup-Name	JA (Linux), NEIN (Windows)	klein
Partitions-Bezeichner	NEIN	klein
String-Vergleich	NEIN (abhängig von Collation)	---
Regex-Vergleich	NEIN (abhängig von Collation)	---
Tabellen-Alias	NEIN	klein
Spalten-Alias	NEIN	klein
Label (Marke)	JA	GROSS

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 mehrere 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 von MySQL ausgeführt (MY!)
/*!NXXYY...*/	C	Inhalt ab MySQL-Version N.XX.YY ausgeführt (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 Schalter "--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;      # Dies ist ein Kommentar bis zum Zeilenende
SHOW TABLES;      #Dies ist ein Kommentar bis zum Zeilenende
SHOW TABLES;      -- Dies ist ein Kommentar bis zum Zeilenende
SHOW TABLES;      --Dies ist KEIN Kommentar (da Leerzeichen fehlt!)
SHOW /* mehrzeiliger
      Kommentar
      */ TABLES;
SELECT * FROM user /*! STRAIGHT_JOIN */ host      # Von MySQL ausgeführt
ON user.Host = host.Host;
CREATE /*!32302 TEMPORARY */ TABLE t1 (a INT);     # Ab MY!3.23.02 ausgeführt
CREATE /*199999 Kommentar */ TABLE t2 (a INT);    # NIE ausgeführt, aber erh.
SHOW TABLES; // Dies ist KEIN Kommentar!         # Fehler
```

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 vertiefte Beschreibung dieser und anderer SQL-Kommandos ist in den folgenden Kapiteln und in der Datei --> mysql-admin-HOWTO.txt zu finden.

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

TIPP: Zu jeder Datenbank einen EIGENEN MySQL-Benutzer (hier: "tom") anlegen (meist als MySQL-Administrator "root"), ihm ein Passwort (hier: "geheim") sowie geeignete Zugriffsrechte (hier: GRANT ALL = sämtliche Rechte) geben, nur mit dieser Datenbank (hier "first") zu arbeiten (muss noch nicht existieren!):

```
GRANT ALL          # Alle Zugriffsrechte...
ON first.*         # ...auf Objekte der Datenbank "first"
TO "tom"@"localhost" # ...für Benutzer "tom@localhost"
IDENTIFIED BY "geheim" # ...mit Passwort "geheim"
WITH GRANT OPTION; # ...mit Rechteweitergabe (optional)
```

TIPP: Username != Datenbankname != Tabellenname != Spaltenname wählen, sonst besteht Verwechslungsgefahr.

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):

```
RENAME DATABASE first TO new;
RENAME SCHEMA new TO first;      # Analog
```

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

```
mysql -utom -pgeheim           # Keine Default/Standard-Datenbank -> "USE"
mysql -utom -pgeheim -Dfirst   # Default/Standard-Datenbank -> "first"
mysql -utom -pgeheim first     # Default/Standard-Datenbank -> "first"
```

Abfrage der Verbindungsdaten:

```
SELECT DATABASE(), SCHEMA();           # Standard-Datenbank (Schema)
SELECT USER(), CURRENT_USER(),        # 4x Benutzername + Hostname
      SESSION_USER(), SYSTEM_USER();   # (user@host)
SELECT CONNECTION_ID();                # Sitzungs-ID (Thread, Prozess)
SELECT VERSION();                      # MySQL-Server Version
```

Abfrage des Server-Status (ID, DB, User, Version, Conn-ID, Character set, ...):

```
status
\s
```

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 (NUR hier darf der ";" weggelassen werden, besser erst gar nicht daran gewöhnen!). Alle Bezeichner (Objektnamen) ohne weitere Qualifizierung beziehen sich dann auf diese Datenbank (dieses Schema):

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

Tabellen aus einer Datenbank auflisten:

```
SHOW TABLES;                       # Alle aus Standard-DB "first"
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 "MyISAM"; Spalten ohne/mit Defaultwert und mit NULL erlaubt/nicht erlaubt):

```
CREATE TABLE pers (                 # Std-Engine = MyISAM
  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,         #
  vorname VARCHAR(30) DEFAULT "",    # Defaultwert leere Zeichenkette
  name    VARCHAR(30) NULL          # NULL explizit erlaubt (sowieso Std)
) ENGINE = "InnoDB";                # Engine = InnoDB (TYPE veraltet!)

CREATE TEMPORARY TABLE pers (       # Temporäre Tabelle (stirbt bei Sitzungsende)
  nr      INT      DEFAULT 0         # (ÜBERDECKT evtl. gleichnamige echte Tab.!)
  vorname VARCHAR(30),
  name    VARCHAR(30)
) CHARACTER SET latin1;              # Standard-Zeichensatz für Textspalten festl.
```

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

HINWEIS: Die Reihenfolge der Spalten in einer Tabelle ist irrelevant - bis auf Abfragen mit "*" und bei Verwendung mehrerer TIMESTAMP-Spalten (hier ist die erste Spalte ausgezeichnet, sie wird bei jeder Datensatz-Änderung aktualisiert).

HINWEIS: Temporäre Tabellen dürfen genauso heißen wie echte Tabellen. Sie überdecken die gleichnamige echte Tabelle, bis sie wieder gelöscht werden. Temporäre Tabellen sind sitzungsbezogen (d.h. pro Sitzung kann der gleiche Name verwendet werden, ohne dass es zu Kollisionen kommt). Sie werden am Sitzungsende automatisch gelöscht.

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 Tabellendefinition 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 in einer Tabelle anzeigen (aktuell noch leer):

```
SELECT * FROM pers;           # Alle Spalten in Definitionsreihenfolge
SELECT nr, vorname, name FROM pers; # (analog)
SELECT vorname, nr, name FROM pers; # 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 belegt; Wertangabe NULL oder DEFAULT ... in der Tabellendefinition verwenden, um Wert NULL oder einen bestimmten Defaultwert bei Angabe von NULL als Wert in eine Spalte einzufügen):

```
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 Tabellendefinition übereinstimmen:

```
INSERT INTO pers
VALUES (9, "Richard", "Seiler");
```

Andere Einfüge-Syntax (MY!):

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

Einfügen von mehr als einem Datensatz gleichzeitig (sehr effizient, MY!):

```
INSERT INTO pers (nr, vorname, name)
VALUES (1, "Thomas", "Birnthaler"), # Nur 1x VALUES!
      (2, "Markus", "Mueller"),     # ", " zw. Datensätzen!
      (8, "Andrea", "Bayer"),
      (9, "Richard", "Seiler"),
      (7, "Heinz", "Bayer"),
      (NULL, "Hans", "Dampf"),
      (NULL, NULL, "Unbekannt");    # Anweisungsende
```

Einfügen eines Datensatzes mit allen Spalten (in Definitionsreihenfolge!):

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

Laden der Daten aus einer externen CSV-Datei (Comma Separated Values, enthält pro Datensatz eine Zeile abgeschlossen durch <NL> oder <CR> <NL>. Die Elemente jedes Datensatzes stehen in der Spaltenreihenfolge im Datensatz und sind durch je ein Tabulatorzeichen <TAB> getrennt, die bei CSV-Dateien notwendige 1. Zeile mit den Spaltennamen weglassen.

```
LOAD DATA LOCAL INFILE "/path/to/pers-data.txt" INTO TABLE pers; # Linux A
LOAD DATA LOCAL INFILE "/path/to/pers-data.txt" INTO TABLE pers # Linux B
  LINES TERMINATED BY "\n"
  IGNORE 1 LINES; # 1.Zl ign.
LOAD DATA LOCAL INFILE "/path/to/pers-data.txt" INTO TABLE pers # Windows
  LINES TERMINATED BY "\r\n"
  IGNORE 1 LINES; # 1.Zl ign.
LOAD DATA LOCAL INFILE "/path/to/pers-data.txt" INTO TABLE pers # MacOS
```

```

LINES TERMINATED BY "\r"
IGNORE 1 LINES; # 1.Zl ign.

```

Inhalt der Datei "/path/to/pers-data.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)
+-----+

```

Kopieren einer vollständigen Tabelle (ALLE Spalten + Datensätze - MY!):
 ACHTUNG: Primary Key und Indices werden NICHT mitkopiert (ausser bei LIKE!).

```

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

```

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):

```

SELECT * FROM copy;
SELECT ALL * FROM copy; # Analog (ALL ist Std)

```

Tabelleninhalt abfragen (alle Datensätze mit bestimmten Spalten in der
 angegebenen Reihenfolge - PROJEKTION!):

```

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

```

Benutzerspezifische Spaltennamen (Aliase) für Ausgabe als Überschrift
 festlegen, Standard ist der Spaltenname laut Tabellendefinition (bei
 Spalten-Aliassen ist GROSS/kleinschreibung nicht relevant, "AS" ist auch
 weglassbar):

```

SELECT vorname AS "Vorname", # mit AS
       name AS "Familiename" # (besser lesbar)
FROM pers;

```

```

SELECT vorname "Vorname", name "Familiename" FROM pers; # ohne AS

```

```

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', #
       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;

```

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

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

Tabelleninhalt nach Spalte "nr" 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 Tabellendefinition):

```
SELECT * FROM pers ORDER BY name;           # GROSS/klein.. ignorieren (unklar!)
SELECT * FROM pers ORDER BY name ASC;      # (analog, besser!)
SELECT * FROM pers ORDER BY BINARY name ASC; # GROSS/klein.. beachten
SELECT * FROM pers ORDER BY 3, 2;         # Nach 2+3. selekt. Sp. sort.
SELECT * FROM pers ORDER BY name, vorname; # (analog per Sp.name statt Nr)
```

Tabelleninhalt nach Spalte "nr" ABsteigend sortiert abfragen
(DESC = descending, ohne GROSS/kleinschreibung zu berücksichtigen):

```
SELECT * FROM pers ORDER BY nr DESC;
```

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 (SELEKTION!):

- * Sogenannte WHERE-Klausel/clause
- * Textvergleiche ignorieren GROSS/kleinschreibung (außer BINARY vor Operator)
- * LIKE = Unschärfe Suche mit Wildcards % (beliebig viele belieb. Z.)
 _ (genau ein beliebiges Zeichen)
- * REGEX/RLIKE = Unschärfe Suche mit Regulären Ausdrücken (s.u.)
- * i/- = Index nutzbar/NICHT nutzbar (kann Suche beschleunigen)

```
SELECT nr
FROM pers
  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 = NULL; # - FALSCH! immer NULL -> FALSE!
... WHERE name IS NOT NULL; # i Spalte NICHT LEER
... 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 Werteliste
... WHERE name LIKE "a%"; # i "a" am Anfang
... WHERE name NOT LIKE "%a"; # - KEIN "a" am Ende
... WHERE name LIKE "%a%"; # - "a" irgendwo
... WHERE name NOT LIKE "__a%"; # - KEIN "a" an 3. Z.pos.
... WHERE name REGEXP "abc"; # - "abc" drin
... WHERE name NOT REGEXP "abc"; # - KEIN "abc" drin
... WHERE name RLIKE "abc"; # - "abc" drin
... WHERE name NOT RLIKE "abc"; # - KEIN "abc" drin
```

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

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

In echter JOIN-Syntax lautet die gleiche Abfrage (Variante 2+3 ist nur möglich, falls die Verknüpfungsspalte in beiden Tabellen gleich heißt, wie hier "nr"):

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

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

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

Nur Teil der Datensätze ab bestimmter Position holen ("**<Rows>**" Datensätze ab Datensatz "**<Offset>**", Std: alle ab 0), die Datensätze sind beginnende mit "0" durchnummeriert - MY!). WICHTIG: Abfrage sortieren, sonst ist die Reihenfolge der Datensätze zufällig gemäß ihrer "physikalischen" Engine-Reihenfolge.

```
SELECT * FROM pers LIMIT <Rows>;        # Ab Datensatz "0"
SELECT * FROM pers LIMIT <Offset>, <Rows>; # Ab Datensatz "<Offset>"
SELECT * FROM pers LIMIT <Rows> OFFSET <Offset>; # Ab Datensatz "<Offset>"
```

Index oder Primary Key auf diversen Spalten anlegen (Variante A; damit läuft obige SELECT-Abfrage mit Verknüpfung beider Tabellen SEHR VIEL SCHNELLER!):

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

Index gleich bei Erstellung einer Tabelle mit anlegen Variante A):

```
CREATE TABLE pers (
  nr INT NOT NULL,
  ...
  PRIMARY KEY      ON (nr),
  INDEX           idx1 ON pers (name),
  INDEX           idx2 ON pers (vorname),
  INDEX           idx3 ON pers (name, vorname)
);
```

Index nach Erstellung 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 (nur MyISAM!)
CREATE SPATIAL INDEX   idx5 ON age (nr);  # Geometriedaten (MyISAM!)
```

Index / Primary Key zu einer Tabelle hinzufügen (Variante C, jederzeit mögl.):

```
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); # Mehrd. Spalte(n)
ALTER TABLE age  ADD FULLTEXT INDEX  idx4 (name); # Volltextsuche (nur MyISAM!))
ALTER TABLE age  ADD SPATIAL INDEX   idx5 (nr);  # Geometriedaten (MyISAM!)
```

Okt 22, 11 3:00

mysql-HOWTO.txt

Page 21/74

Indices zu einer Tabelle anzeigen:

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

Index oder Primary Key einer Tabelle löschen (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
```

Index auf Spalten-Präfix bzw. AUF/ABsteigend erzeugen (Std: ASC - MY!), nur für Tuningzwecke wirklich interessant.

```
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!):

```
UPDATE pers
SET nr = nr + 100;
```

Bestimmte Datensätze ändern:

```
UPDATE pers                                # 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, 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 ist optional
VALUES (1, "Thomas", "Birnthaler"),        # Mehrere Datensätze erlaubt
      (2, "Hans", "Dampf");
```

```
REPLACE INTO copy (nr, vorname, name)      # INTO ist optional
SELECT nr, vorname, name
FROM pers,
WHERE nr < 100;
```

```
REPLACE pers                                # INTO ist 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";
```

ACHTUNG: REPLACE ändert AUTO_INCREMENT-Wert bereits vorhandener Datensätze, INSERT INTO ON DUPLICATE KEYS UPDATE macht das nicht (aber 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:

```
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;
```

TIPP: 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 verwendet).

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

```
SELECT COUNT(*) FROM pers;          # Anzahl Datensätze in Tabelle (inkl. NULL!)
SELECT COUNT(nr) FROM pers;         # Anz. Werte von Spalte "nr" (ohne NULL!)
SELECT COUNT(DISTINCT nr) FROM pers; # Anz. versch. Werte von Spalte "nr" (ohne NULL!)
SELECT name, COUNT(name) FROM pers  # Häufigkeit der Werte in Spalte "name"
GROUP BY name;                      #
SELECT name, COUNT(name) FROM pers  # Analog, nur Werte mit Häufigkeit > 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          # 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); # B) Variable @max füllen
SELECT * FROM pers WHERE nr = @max;   # B) (: = / = 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 * FROM pers WHERE nr IN (@min, @max);         # D1) ... Menge
SELECT * FROM pers WHERE nr = @min                  # D2) ... 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), 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 noch das Anhängen von "FROM dual" notwendig!). Die bei MySQL (und Oracle) immer 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 angebar
- * 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 die Tabellendaten so weit möglich erhalten (evtl. abgeschnitten oder mit Leerzeichen aufgefüllt)
- * Tabelle/Engine ändern NICHT auf Systemtabellen in DB "mysql" anwenden!

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

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

Beispiele:

```
ALTER TABLE pers ...
... ADD COLUMN preis DECIMAL(10,2); # Std: Spalte hinten einfügen
... 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 löschen (inkl. Daten)
... MODIFY COLUMN vorname VARCHAR(50); # Typ ändern (Daten mögl. erhalten)
... ... FIRST; # Als 1. Spalte verschieben
... ... AFTER nr; # Nach Spalte "nr" verschieben
... CHANGE COLUMN vorname name CHAR(30); # Name+Typ änd. (Daten mögl. erh.)
... 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 = InnoDB; # Engine ändern (neu!)
... TYPE = InnoDB; # Engine ändern (veraltet!)

... ADD PRIMARY KEY (nr); # Primärschlüssel hinzu
... DROP PRIMARY KEY; # Primärschlüssel entf.
... INDEX idx1 (name, vorname); # Index hinzu
... DROP INDEX idx1; # Index entfernen
... UNIQUE idx2 (name, vorname); # Sekundärschlüssel hinzu
... DROP UNIQUE idx2; # Sekundärschlüssel entf.
... DISABLE KEYS; # Alle Indices abschalten
```

```
... ENABLE KEYS; # Alle Indices aktivieren

... ADD FOREIGN KEY age(nr) REFERENCES pers(nr); # Fremdschlüssel hinzufügen
... DROP FOREIGN KEY age; # Fremdschlüssel entfernen
... ADD CONSTRAINT pruef CHECK (nr >= 1); # Spalten-Beschränkung hinzufügen
... DROP CONSTRAINT pruef; # Nicht verfügbar!
```

Tabelle(n) umbenennen (auch in andere Datenbank verschieben, wenn die beiden Tabellen auf dem gleichen Dateisystem liegen; stellt eine "atomare" Operation dar, auch bei gleichzeitiger Umbenennung mehrerer Tabellen!):

```
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 Tabellennamen vertauschen (FALSCH!)
copy TO pers; # (-> ERROR: Table 'copy' already exists)
RENAME TABLE pers TO tmp, # Zwei Tabellennamen 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 einiger Daten der vorhergehenden SQL-Anweisung (pro Sitzung):

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

Fehlermeldungen, Warnungen und Bemerkungen (Notes) zu vorhergehenden SQL-Anweisungen ausgeben (pro Sitzung):

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

Abhängig von den darin zu speichernden Werten ist für jede Tabellenspalte einer der folgenden Datentypen auszuwählen. Die Auswahl des Datentyps ist nicht immer eindeutig bestimmt (z.B. könnte ein Jahr als INT oder YEAR gespeichert werden). Bei sinnvoller Auswahl des Datentyps sind aber spezifische MySQL-Operatoren und -Funktionen einsetzbar, der Platzbedarf der Daten wird minimiert sowie die Zugriffsgeschwindigkeit maximiert. Auch an evtl. zukünftige Erweiterungen denken und z.B. nicht wieder für das Speichern einer Jahreszahl nur 2 Stellen vorsehen ("Y2K-Problem").

Datentyp	Typname	Byte	Wertebereich
Ganzzahl (mit Vorzeichen)	TINYINT	1	-127..128
	SMALLINT	2	-32768..32767
	MEDIUMINT	3	-8388608..8388607
	INT	4	-2Mrd..2Mrd
	BIGINT	8	-9*10 ¹⁸ ..+9*10 ¹⁸
Ganzzahl (ohne Vz.)	TINYINT UNSIGNED	1	0..255
	SMALLINT UNSIGNED	2	0..65535
	MEDIUMINT UNSIGNED	3	0..16777215
	INT UNSIGNED	4	0..4 Mrd
	BIGINT UNSIGNED	8	0..18*10 ¹⁸
Fließkommazahl	FLOAT	4	10 ³⁸ (7 Dez.stellen)
	DOUBLE	8	10 ³⁰⁸ (15 Dez.stellen)
Fließkommazahl	FLOAT UNSIGNED	4	10 ³⁸ (7 Dez.stellen)

Okt 22, 11 3:00

mysql-HOWTO.txt

Page 25/74

(ohne Vorz.)	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!)
Bitfeld Boolean	BIT(N) BOOL	N+7/8 1	1..64 Bits 0/1 FALSE=0/TRUE=1
Zeichenkette (Länge fest) (Länge fest) (Länge variabel) (Länge variabel)	CHAR(L) BINARY(L) VARCHAR(L) VARBINARY(L)	L L L+1/2 L+1/2	L=0..255 L=0..255 L=0..255/65535 ab 5.01 L=0..255/65535 ab 5.01
Datum + Zeit (Sek. seit 1.1.1970 00:00)	YEAR DATE TIME DATETIME TIMESTAMP	1 3 3 8 4	JJJJ/JJ (1900-2155) JJJJ-MM-TT (1000-9999) hh:mm:ss JJJJ-MM-TT hh:mm:ss JJJJMMThhmmss
Text (Sonderzeichen interpretiert)	TINYTEXT TEXT MEDIUMTEXT LONGTEXT	L+1 L+2 L+3 L+4	0-255 0-65535 0-16777216 (16 Mio) 0-4294967296 (4 Mrd)
Binary large object (binäre Daten, keine Interpretation von Sonderzeichen)	TINYBLOB BLOB MEDIUMBLOB LONGBLOB	L+1 L+2 L+3 L+4	0-255 0-65535 0-16777216 (16 Mio) 0-4294967296 (4 Mrd)
Aufzählung Menge	ENUM(V1, ...) SET(V1, ...)	1/2 1-4/8	1..255, 2=256-65535 1..8/16/24/32/64 Elem.
Unicodestring	NCHAR(L) NATIONAL VARCHAR(L)	L L+1	Synonym für CHAR Synonym für VARCHAR

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)

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

Attribut	Bedeutung
<N>	Ausgabebreite (INT)
<N>, <M>	Ausgabebreite N und Nkst. M (FLOAT, DOUBLE)
AUTO_INCREMENT	Pro neu eingefügtem Datensatz automatisch hochzählen (künstl. Key, nur 1x pro Tabelle, nicht Festkomma!)
UNSIGNED	Wert immer positiv
ZEROFILL	Ausgabe mit führenden Nullen gemäß Breite (nicht DEC!)

Beispiel:

```
CREATE TABLE data (
  nr          INT NOT NULL AUTO_INCREMENT,      # Spalte "nr" automatisch füllen
  anz        TINYINT(3) UNSIGNED,
  temper     FLOAT(10,2),
  betrag     DECIMAL(10,2),
  vorname    VARCHAR(30) NOT NULL,
  name       CHAR(30),
  stamp      TIMESTAMP,
  datum      DATE DEFAULT '2000-01-01',
  uhrzeit    TIME DEFAULT '12:00:00',
  zeitpunkt DATETIME,
  jahr       YEAR(4),
  flags      BIT(8),
  opt        BOOL,
  dokument   TEXT,
  antwort    ENUM('Ja', 'Nein', 'Vielleicht'),
  menge      SET('gross', 'reich', 'maechtig'),
  file       BLOB,
  PRIMARY KEY (nr) # Kein Komma, notwendig wg. AUTO_INCREMENT
);
```

```

INSERT INTO data VALUES (
  NULL,          # Nächste freie Nummer verwendet wg. AUTO_INCREMENT
  123,
  12.3456,
  9876543.21,
  "thomas",
  "birnthaler",
  NULL,          # Aktueller Zeitpunkt eingetragen (TIMESTAMP)
  "2010-08-03",
  "23:59:59",
  "2010-08-03 23:59:59",
  2010,
  b'01000001',  # ASCII-Code Buchstabe 'A' = 65
  TRUE,
  "Dokumenttext",
  "Vielleicht", # Aufzählungswert als String schreiben (EINER)
  "gross,maechtig", # String mit Element-Liste durch "," getrennt (" " = leere Menge)
  "Dateiinhalt"   # OK!
  LOAD_FILE("/home/tsbirn/my/data/pers-data.txt") # NULL (wieso)
  LOAD_FILE("/etc/passwd") # OK!
);

```

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
BIT	j	b'10101100' 0x10EF x'10EF'
BOOL	-	TRUE FALSE 0 1 "aaa" ""
CHAR	j	"Hallo" 'Hallo' "" ''
VARCHAR	j	"Hallo" 'Hallo' "" ''
BINARY	j	0x10EF x'10EF' _utf8'uc' "" ''
VARBINARY	j	0x10EF x'10EF' _utf8'uc' "" ''
YEAR	-	69 99 13 1972 2010 2100
DATE	j	"2009-10-18" "0000-00-00" '2009-10-00'
TIME	j	"23:59:59" '09:45' "00:00:00"
DATETIME	j	"2009-10-18 22:30:00"
TIMESTAMP	j/-	"2009-10-18 22:30:00" 20091018223030
TEXT	j	"Hallo" 'Hallo' "" '' _utf8'uc'
BLOB	j	0x10EF x'10EF' b'11010' "Daten..." 'Daten...'
ENUM	j	"1.Wert" '5.Wert'
SET	j	"1.Wert,5.Wert,8.Wert"

Defaultwert für die Datentypen:

Datentyp	Defaultwert
INT	0
FLOAT	0.0
DECIMAL	0.0
BIT	b'' (kein Bit gesetzt)
BOOL	0 (FALSE)
CHAR	"" (leerer String)
VARCHAR	"" (leerer String)
BINARY	"" (leere Bytefolge)
VARBINARY	"" (leere Bytefolge)

YEAR	00/0000
DATE	"0000-00-00"
TIME	"00:00:00"
DATETIME	"0000-00-00 00:00:00"
TIMESTAMP	Aktuelles Datum + Uhrzeit (1.Spalte) "0000-00-00 00:00:00" (restliche Spalten)

TEXT	"" (leerer String)
BLOB	"" (leerer String)

ENUM	0 (falls NULL), "1. Wert" (falls NOT NULL)
SET	"" (leere Menge)

HINWEISE:

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

```

BINARY <String>           # GROSS/kleinschreibung berücksichtigen
CAST(<Val> AS <Type>)     # Datentyp von <Val> umwandeln
CONVERT(<Val> AS <Type>)  # Datentyp von <Val> umwandeln
CONVERT(<Val> USING <Transcode>) # Zeichensatz von <Val> konvertieren

```

Ziel_datentyp <Type> 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 Umwandlungen sind möglich:

```

DECIMAL    -> CHAR  VARCHAR  DATE
CHAR  VARCHAR -> DECIMAL  DATE
DATE      -> CHAR  VARCHAR
BLOB  TEXT  -> (keine!)

```

- * INT/FLOAT/DOUBLE: Bei Ganzzahlen ist in Klammern eine "Darstellungsgröße" (1-255) angebar, 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 Nachkommastellenanzahl die AUSGABE erfolgen soll.

Ganzzahlen können keine führenden 0-en darstellen -> evtl. CHAR verwenden
+ Damit kann man trotzdem rechnen (Preis: Konvertierung zur Ausführungszeit)
+ Alternativ ZEROFILL (feste Länge gemäß Darstellungsbreite)

- * 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 für DECIMAL "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. Ähnliches Verhalten wie BCD-Format (Binary Coded Digits), aber anders realisiert.
- * Kaufmännische oder Finanzmathematische Daten wie Preise, Umsatz, ... mit DECIMAL darstellen (vermeidet Rundungsfehler von FLOAT/DOUBLE) oder statt Euro mit 2 Nkst besser Cent ohne Nkst speichern.
- * Dezimalkomma in Zahl (z.B. 1,50) ist (meist) ein Syntaxfehler, Dezimalkomma in String (z.B. "1,50") schneidet Rest nach Komma ab (ignoriert).
- * BOOL: Kennt die Konstanten TRUE (Wahr, Wert 1) und FALSE (Falsch, Wert 0). Jeder Wert außer 0, FALSE (und NULL) ist TRUE (Wahr).
- * YEAR: Werte decken 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 -> 1900-1999 + 100-255 -> 2000-2155)
- * DATE/DATETIME: Werte decken die Jahre 1000-9999 ab. Jeder Teil (Tag, Monat, Jahr) ist auf "0" setzbar, um auszudrücken, dass ein Datumteil nicht bekannt ist. Derartige Teile werden bei Sortierung als 1. Wert einsortiert. Ein ungültiger Datumswert wird als "0000-00-00" gespeichert. Mit "SET @@sql_mode

= "... " ist steuerbar, welche "falschen" Datumswerte erlaubt sind:

@@sql_mode = ...	Bedeutung
ALLOW_INVALID_DATES	Beliebige Komb. von Mon=1..12 + Tag=1..31 erlaubt
NO_ZERO_IN_DATE	Mon=0 und Tag=0 nicht erlaubt
NO_ZERO_DATE	0000-00-00 verboten (aber 1999-01-00 + 1999-00-01)

- * **TIMESTAMP**: Werte decken Zeitraum 1.1.1970 00:00 bis 31.12.2037 23:59 ab (UNIX-Zeitrechnung mit Anzahl Sekunden seit 1.1.1970 00:00 mit 32-Bit Zahl).
- * **TIMESTAMP**: Wert der 1. **TIMESTAMP**-Spalte in einer Tabelle wird bei JEDER Änderung einer Zeile auf aktuelle(s) Datum + Uhrzeit gesetzt (alle weiteren **TIMESTAMP**-Spalten bleiben unverändert).
- * **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.
VARBINARY(n) speichert 0-n Byte PLUS eine Länge.
- * **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. Werden nicht im Datensatz gespeichert, sondern dort nur ein "Zeiger" der Größe L Byte. Die Daten selber stehen pro Objekt an einer anderen Stelle. **TEXT** sind Textdaten, Vergleich und Sortierung erfolgt anhand Zeichensatz und Collation (ohne Berücksichtigung der GROSS/kleinschreibung). **BLOB** sind Binärdaten, Vergleich und Sortierung erfolgt anhand der Byte-Codes.
- * **TEXT/BLOB** entsprechen **VARCHAR/VARBINARY** mit folgenden Unterschieden:
 - + Indices auf **TEXT/BLOB**-Spalten MÜSSEN eine Präfix-Länge haben
 - + **TEXT/BLOB**-Spalten können KEINE Defaultwerte haben
 - + Speicherung erfolgt nicht in Datensatz sondern außerhalb, im Datensatz steht nur ein Zeiger.
- * **ENUM**: Als Werte sind nur die Elemente der definierten Werteliste speicherbar, anstelle des Elements wird platzsparend seine Nummer gespeichert. Speicherung eines anderen Wertes entspricht **NULL** (insbes. 0). Anstelle der Werte sind auch ihre Indices gemäß der Werteliste verwendbar (1=1.Wert, 2=2.Wert, ...).
- * **SET**: Als Wert ist eine beliebige Kombination ("Menge") der Werte aus der definierten Werteliste (max. 64 Werte) erlaubt (auch die "leere Menge"). Anstelle der Elementtexte wird platzsparend ein Bitmuster gespeichert, jede Bitposition entspricht einem Element der Menge. Bitwert = 1 heißt, das Element ist in der Menge, Bitwert = 0 heißt, es ist nicht in der Menge.

4a) Datentyp-Optimierung (Performance/Speicherplatz)

- * Kleinstmöglichen Datentyp nutzen (z.B. **MEDIUMINT** statt **INT**)
-> Weniger Festplattenplatz belegt -> Weniger Platten-I/O -> schneller
- * **WICHTIG**: Spalten möglichst "NOT NULL" deklarieren!
-> Weniger Platzverbrauch (**NULL** benötigt zusätzl. Byte) + Index schneller!
- * Mind. EINE **VARCHAR/VARBINARY/TEXT/BLOB**-Textspalte -> Variable Record-Länge
NUR **CHAR/BINARY**-Textspalten -> Fixe Record-Länge
- * In **SELECT** keine Spalten-Konvertierungen durchführen
-> Index nutzbar + schneller
-> **ACHTUNG**: Wird implizit gemacht, wenn mit Textspalten gerechnet wird
- * In **JOIN** verwendete Spalten sollten exakt gleichen Datentyp + Länge haben, damit Index nutzbar (bei (VAR)CHAR/(VAR)BINARY/TEXT/BLOB wird unterschiedliche Länge toleriert)
- * **MySQL**-Optimierer macht selbständig folgende Änderungen bei **CREATE/ALTER TABLE** (Speicherplatz minimal, Tabellenstruktur optimiert, autom. Konvertierung)
 - + **VARCHAR(<=4)** -> **CHAR**
 - + **VARBINARY(<=4)** -> **BINARY**
 - + **CHAR(>4)** -> **VARCHAR** (falls mind. eine **VARCHAR**-Spalte vorhanden)
 - + **TIMESTAMP(N)** -> **TIMESTAMP (2 <= + N geradzahlig + <= 14)**

4b) Fixes/Variablen Rowformat

-
- * Datensätze (Rows) können max. 65535 Byte lang werden.
 - * Fixes Rowformat (feste Länge): Alle Spalten NICHT von folgenden 4 Datentypen
Variables Rowformat (unterschiedl. Länge): Mind. eine Spalte dieser Datentypen
 - VARCHAR > 4 Zeichen
 - VARBINARY > 4 Zeichen
 - ...BLOB
 - ...TEXT
 - * Fixes Rowformat hat Vorteile bei der Adressierung der Datensätze, nutzt aber evtl. den Plattenplatz schlechter (wg. Füllbytes)
 - * Variables Rowformat nutzt den Plattenplatz besser aus, benötigt aber zusätzliche Bytes (<=255 -> 1, >255 -> 2) zur Längeninformation
 - * Bei vielen Löschen/Einfügeoperationen in Tabelle führt variables Rowformat leichter zur "Zersplitterung" der Tabellendaten
-> OPTIMIZE TABLE durchführen
 - * Für Zugriff per Index ist fixes/variables Rowformat egal
 - * Besondere Einstellungen bei Tabellendefinition
 - + MyISAM: ROW_FORMAT=FIXED/DYNAMIC (myisampack -> COMPRESSED)
 - + InnoDB: ROW_FORMAT=REDUNDANT/COMPACT (20% weniger Platz, CPU erhöht)
 - * TIPP: Evtl. eine Tabelle in mehrere aufsplitten (fixer + variabler Anteil)

4c) AUTO_INCREMENT

-
- * Gibt jedem Datensatz automatisch eine eindeutigen ("künstlichen") Key
 - * Eindeutige positive Zahl pro Datensatz (negativ -> große positive Zahl)
 - + Startwert beim Anlegen der Tabelle wählbar (Std: 1):
 CREATE TABLE ... (...) ... AUTO_INCREMENT = <N> ...
 ALTER TABLE <Tbl> AUTO_INCREMENT = <N>;
 (bzw. eine Zeile mit Wert "Start-1" einfügen und wieder löschen)
 - * Nur in EINER Spalte pro Tabelle erlaubt
 - + Spalten-Datentyp muss INT, FLOAT oder DOUBLE sein
 (DEDECIMAL 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
 - + NO_AUTO_VALUE_ON_ZERO: 0 als gültiger Wert erlaubt (nicht empfohlen!)
 - + Spaltenwert angegeben
 - Schon vorhanden -> Einfügen wird verweigert
 - Nächster automatisch ermittelter um 1 höher (Lücken entstehen!)
 - * Letzter verwendeter AUTO_INCREMENT-Wert direkt nach INSERT (in derselben Sitzung/Transaktion) mit LAST_INSERT_ID() abfragbar.
 - * Damit Master-Master-Replikation möglich ist (2 gegenseitige Master oder Ring aus mehreren Masters), müssen AUTO_INCREMENT-Spalten auf allen Masters garantiert unterschiedliche Werte erzeugen. Dazu gibt es pro 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 Masters der Increment auf (mindestens) 5 gesetzt werden.
 - * In PostgreSQL dafür Datentyp "SERIAL" verwenden.
 - * In Oracle durch "Sequence"-Objekt nachbilden.

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

```

5) MySQL-Operatoren

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

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

Beispiele:

```

SELECT 1 + 2 * 3 / 4;
SELECT 17 DIV 3, 17 % 3, 17 MOD 3;
SELECT 10 < 4, 2 <= 5, 1 <> 3, 1 != 3;
SELECT TRUE AND FALSE OR FALSE AND NOT TRUE;
SELECT TRUE && FALSE || FALSE && ! TRUE;
SELECT ~(b'11110001' | b'00001111') ^ b'00110011';

```

Hinweise:

- * Division "/" ist immer Fließkommadivision
- * Es gibt keinen Stringverkettungsoperator (z.B. & + . ||)
-> CONCAT verwenden, damit sind beliebig viele Strings verkettbar
(oder "SET @@sql_mode = 'ANSI'" -> Operator "||" verkettet Strings)
CONCAT("abc", <Col>, "xyz")
- * Falls in Stringverkettung NULL vorkommen kann:
CONCAT("abc", IF(ISNULL(<Col>), "", <Col>), "xyz")
CONCAT("abc", IFNULL(<Col>, ""), "xyz")
- * BINARY vor einem String in den Vergleichen = != <> <=> < <= > >= BETWEEN IN
LIKE REGEX und in ORDER BY erzwingt Beachtung der GROSS/kleinschreibung.
SELECT * FROM pers WHERE BINARY name = "abc";
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: % steht für beliebig viele beliebige Zeichen (auch 0!)
_ steht für GENAU EIN beliebiges Zeichen
- * ESCAPE "<C>" am Ende von LIKE setzt das Entwertungs-Zeichen von "%" und "_"
auf das Zeichen "C". D.h. um mit LIKE nach den Zeichen "%" und "_" selbst
zu suchen, muss man "C" davorsetzen (Std: "\\")
SELECT * FROM test WHERE name LIKE "_%" ESCAPE "\\\";

Hinweise zu NULL:

- * Kommt NULL in einem Ausdruck vor, ist sein Ergebnis NULL ("schwarzes Loch")
- * Kommt NULL in einem Vergleich vor, ist sein Ergebnis NULL (außer bei <=>)
- * Logische Ausdrücke ergeben 1 für TRUE und 0 für FALSE
- * Logische Ausdrücke werden verkürzt ausgewertet (Short-Cut/Circuit Evaluation)
(FALSE AND ... ist sofort FALSE, TRUE OR ... ist sofort TRUE, d.h. hat ...
den Wert NULL, ist das Gesamtergebnis trotzdem FALSE bzw. TRUE!)

```
* Ist NULL das Ergebnis einer WHERE-Bedingung, entspricht dies dem Wert FALSE
* IS UNKNOWN entspricht 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
* NULL-sicherer Vergleich <=>
  NULL <=> NULL      -> TRUE   (bei "=" -> NULL)
  NULL <=> TRUE/FALSE -> FALSE (bei "=" -> NULL)
  Rest wie bei "="
```

Rangfolge/Priorität der Operatoren (durch Klammerung mit "(...)" umgehbar):

Nr	Typ	Bemerkung
1	BINARY COLLATE	GROSS/kleinschreibung beachten, Zeichensatz anpassen
2	! (NOT)	Negation (falls HIGH_NOT_PRECEDENCE)
3	- ~	Unäres Minus, Bitweise Flip
4		Falls @@sql_mode="PIPES_AS_CONCAT"
5	^	Bitweise XOR
6	* / DIV % MOD	% = Modulo
7	+ -	Addition/Subtraktion
9	<< >>	Bitweise Shift
9	&	Bitweise AND
10		Bitweise OR
11	= != <> <=> < <= > >=	Vergleich
	IN IS LIKE REGEXP	Vergleich
12	BETWEEN CASE WHEN THEN ELSE	Bereich, Fallunterscheidung
13	NOT	(ab MY!5.0.2 vom Vorrang her hier!)
14	AND &&	Logisch AND
15	XOR	Logisch XOR
16	OR	Logisch OR
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 Priorität erhält man mit "SET @@sql_mode = 'HIGH_NOT_PRECEDENCE'".

HINWEIS: Mit "SET @@sql_mode = 'ANSI'" entspricht der Operator "||" nicht mehr OR, sondern verkettet Strings (ANSI-SQL Standard).

6) Boolesche Logik

In Boolescher Logik gibt es nur die beiden Konstanten TRUE und FALSE mit der numerischen Bedeutung 1 und 0. Umgekehrt werden alle numerischen Werte ungleich 0 auf TRUE abgebildet und der numerische Wert 0 auf 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
```

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). D.h.

```
<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

Eigenschaften von NULL:

- * NULL steht für den Wert "unbekannt" oder "undefiniert"
- * NULL ist weder die Zahl 0 noch der leere String "", noch TRUE oder FALSE, sondern etwas völlig anderes
- * Alle Operationen mit NULL ergeben NULL (außer: <=> AND OR IS NULL) ("Schwarzes Loch")
- * Benötigt extra Speicherplatz (pro NULL-Spalte ein Byte oder ein Bit)
 - > Spalten wann immer möglich mit NOT NULL kennzeichnen (Std: NULL)
- * NULL-sicherer Vergleich mit <=> statt = (MY!)

Kommt der Wert NULL in einem Ausdruck vor, so hat in der Regel der gesamte Ausdruck als Ergebnis den Wert NULL (Ausnahme: <=>, AND, OR):

```
SELECT 3 * 4 - 1 + NULL AS "Wert";           # -> NULL
SELECT 1 = NULL, 1 <> NULL, 1 < NULL, 1 > NULL;   # -> 4x NULL
SELECT 1 <=> NULL, NULL <=> NULL, 0 <=> NULL, "" <=> NULL; # -> 0 1 0 0
```

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

```
<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> falls ungleich NULL, sonst <Expr2>
NULLIF(<Expr1>, <Expr2>) # NULL falls <Expr1>=<Expr2>, sonst <Expr1>
COALESCE(<Expr1>, <Expr2>, ...) # Erster Ausdruck <Expr_i> ungleich NULL
```

Beispiel:

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

7a) Vergleiche mit NULL

Vergleiche ergeben einen Booleschen Wert außer mit NULL wird verglichen, dann resultiert der Wert NULL als Ergebnis. 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	

Alle 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

7b) Boolesche Logik mit NULL (dreiwertig)

Die Booleschen Operatoren AND, OR, XOR und NOT werden auf die Verknüpfung mit NULL-Werten erweitert. Aufgrund der "verkürzten Auswertung" von AND und OR gilt:

- * FALSE links von AND ergibt sofort FALSE
- * TRUE links von OR ergibt sofort TRUE

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 ! | TRUE | FALSE | NULL | # NULL invertiert ergibt NULL!

	FALSE	TRUE	NULL*
--	-------	------	-------

8) Reguläre Ausdrücke in MySQL

Eigenschaften:

- * Reguläre Ausdrücke nennt man auch "RegExp" oder "Pattern" (Muster) (von englisch "Regular Expressions")
- * Vergleichen wie LIKE/NOT LIKE Texte mit einem Muster -> wahr/falsch (stark erweiterte Fähigkeiten gegenüber LIKE/NOT LIKE)
- * Operatoren: <Expr> REGEX "<RegExp>"
<Expr> RLIKE "<RegExp>"
<Expr> NOT REGEX "<RegExp>"
<Expr> NOT RLIKE "<RegExp>"
- * Bei Regulären Ausdrücken ist KEIN Index für Datenzugriff benutzbar!
- * Muss IRGENDWO in Zeichenkette passen, damit wahr (außer bei Verwendung von ^ bzw. \$, dann am Anfang bzw. Ende)

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

Zeichenklassen fassen Zeichen zusammen (hängen von locale-Einstellung 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$";           # Genau "abc"
... WHERE name = "abc";              # (identisch)
... WHERE name REGEXP "abc$";       # "abc" am Ende
... WHERE name LIKE "%abc";         # (identisch)
... WHERE name REGEXP "abc";        # "abc" drin
... WHERE name LIKE "%abc%";        # (identisch)
... WHERE name NOT REGEXP "abc";     # KEIN "abc" drin
... WHERE name NOT LIKE "%abc%";    # (identisch)
... WHERE name RLIKE "^a*$";        # "", "a", "aa", "aaa", ...
... WHERE name = "" OR name = "a" OR ... # (identisch)
... WHERE name RLIKE "^a+$";        # "a", "aa", "aaa", ...
... WHERE name = "a" OR name = "aa" OR ... # (identisch)
... WHERE name RLIKE "thomas|hans"; # Enth. "thomas" oder "hans"
... WHERE name = "thomas" OR name = "hans"; # (identisch)
... WHERE name RLIKE "ab{1,8}a";    # "aba", "abba", ...max. 8 "b"
... WHERE name = "aba" OR name = "abba" OR ...; # (identisch)

```

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 bietet eine große Menge an eingebauten Funktionen, die auch in einer SELECT-Anweisung zum Konvertieren/Verknüpfen von Spalten und in Bedingungen verwendet werden dürfen. 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 in MySQL programmierte benutzerdefinierte Funktionen (Stored Procedures) und durch Laden von externen User Defined Functions (UDF):

Klasse	Eingebaute Funktionen + ihre Parameter
Datenbank	USER() CURRENT_USER() SESSION_USER() SYSTEM_USER() DATABASE() SCHEMA() CONNECTION_ID() VERSION()
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) EXP(zahl) GREATEST(zahl1, zahl2, ...) LEAST(zahl1, zahl2, ...) LN(zahl) LOG(zahl [, basis]) LOG10(zahl) MOD(x, y) PI() POW(num, exp) POWER(num, exp) RAND([seed]) # Zufallszahl 0 <= z < 1 SIGN(zahl) SQRT(zahl)
Trigonometrie	ACOS(zahl) ASIN(zahl) ATAN(zahl) ATAN2(x, y) COS(zahl) COT(zahl) DEGREES(rad) RADIANS(dec) SIN(zahl) TAN(zahl)
Rundung	CEILING(zahl) CEIL(zahl) FLOOR(zahl) FORMAT(zahl, dezstellen)

	<p>ROUND(zahl [, dezstellen]) TRUNCATE(zahl [, dezstellen])</p>
Bit	<p>BIT_COUNT(zahl) BIT_LENGTH(string) EXPORT_SET(zahl, ein, aus, [trennzeichen, [numbits]]) MAKE_SET(bits, string1, string2, ...)</p>
String	<p>COERCIBILITY(string) COLLATION(string) CONCAT(string1, string2, ...) CONCAT_WS(trennzeichen, string1, string2, ...) INSERT(string, pos, laenge, neu) INSTR(string, teil) LCASE(string) LOWER(string) LEFT(string, laenge) LENGTH(string) CHAR/CHARACTER/OCTET_LENGTH(string) LPAD(string, laenge, fuellstring) LTRIM(string) MID(string, pos, laenge) REPEAT(string, anzahl) REPLACE(string, alt, neu) REVERSE(string) RIGHT(string, laenge) RPAD(string, laenge, fuellstring) RTRIM(string) SOUNDEX(string) SPACE(anzahl) SUBSTRING(string, pos) SUBSTRING(string, pos, laenge) SUBSTRING(string FROM pos FOR laenge) SUBSTRING(string FROM laenge) SUBSTRING_INDEX(string, zeichen, anzahl) TRIM([BOTH LEADING TRAILING] [zeichen] [FROM] string) UCASE(string) UPPER(string)</p>
Stringvergl.	<p>FIELD(string, string1, string2, ...) FIND_IN_SET(string, menge) LOCATE(teil, string [, zahl]) MATCH (spaltel, ...) AGAINST (string [modifier]) POSITION(teil, string) STRCMP(string1, string2)</p>
Datum	<p>CURDATE() CURRENT_DATE() -> Datum YYYY-MM-DD DATE(datum/zeit) DATE_FORMAT(datum, format) DAYNAME(datum) DAYOFMONTH(datum) DAY(datum) DAYOFWEEK(datum) DAYOFYEAR(datum) EXTRACT(zeitabschnitt FROM datetime) FROM_DAYS(tage) LAST_DAY(datum) LOCALTIME() MAKEDATE(jahr, tag) MONTH(datum) MONTHNAME(datum) QUARTER(datum) STR_TO_DATE(string, formatmuster) TO_DAYS(datum) UTC_DATE() WEEK(datum) WEEKDAY(datum) WEEKOFYEAR(datum) YEAR(datum) YEARWEEK(datum [, modus])</p>
Datumrechnung	<p>ADDDATE(datum, INTERVAL anzahl typ) analog DATE_ADD DATE_ADD(datum, INTERVAL zeitspanne typ) DATE_SUB(datum, INTERVAL zeitspanne typ) DATEDIFF(neu_datum, alt_datum) -> TAGE PERIOD_ADD(datum, monate) PERIOD_DIFF(datum1, datum2) SUBDATE(datum, INTERVAL anzahl typ) analog DATE_SUB</p>
Zeit	<p>CONVERT_TZ(datum/zeit, zeitzone, zeitzone) CURTIME() CURRENT_TIME() HOUR(zeit) MAKETIME(stunde, minute, sekunde) MICROSECOND(zeit) MINUTE(zeit) SEC_TO_TIME(sekunden)</p>

	<pre>SECOND(zeit) TIME_FORMAT(zeit, format) TIME_TO_SEC(zeit) UTC_TIME()</pre>	
Zeitrechnung	<pre>ADDTIME(zeit, zeit) SUBTIME(datum/zeit, datum/zeit) TIMEDIFF(zeit, zeit)</pre>	
Timestamp	<pre>FROM_UNIXTIME(sekunden [,format]) -> Datum LOCALTIMESTAMP() NOW() CURRENT_TIMESTAMP() TIMESTAMP(datum, zeit) TIMESTAMPADD(intervall, wert, datum/zeit) TIMESTAMPDIFF(intervall, wert, datum/zeit) UNIX_TIMESTAMP([datum]) -> Sekunden UTC_TIMESTAMP()</pre>	
Bedingung	<pre>CASE wert WHEN auswahl THEN wert ... ELSE wert END ELT(zahl, string1, string2, ...) IF(test, wert1, wert2) INTERVAL(x, grenze1, grenze2, ...)</pre>	
NULL-Vergleich	<pre>COALESCE(wert1, wert2, ...) IFNULL(wert1, wert2) ISNULL(ausdruck) NULLIF(wert1, wert2)</pre>	
Konvertierung	<pre>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(string) QUOTE(string) UNHEX(string)</pre>	
Zeichen- codierung + sortierung	<pre>CHARSET() COERCIBILITY() COLLATION()</pre>	
Advisory Locking (User Level)	<pre>GET_LOCK(name, timeout) IS_FREE_LOCK(name) IS_USED_LOCK(name) RELEASE_LOCK(name)</pre>	
Komprimierung	<pre>COMPRESS(string) UNCOMPRESS(string) UNCOMPRESSED_LENGTH(string)</pre>	
Verschlüsseln Entschlüsseln	<pre>AES_DECRYPT(string, password) AES_ENCRYPT(string, password) DES_DECRYPT(string [, password]) DES_ENCRYPT(string [, password]) DECODE(blob, password) ENCODE(blob, password) ENCRYPT(string, salt) OLD_PASSWORD(string) PASSWORD(string)</pre>	
Hashing	<pre>CRC32(string) MD5(string) SHA(string) SHA1(string)</pre>	
Eindeutige ID	<pre>UUID() -> 128-Bit Zahl UUID_SHORT() -> 64-Bit Zahl</pre>	
Netzwerk	<pre>INET_ATON(adresse) INET_NTOA(zahl)</pre>	
Sonstige	<pre>BENCHMARK(anzahl, funktion) LOAD_FILE(dateiname) # Datei als String einlesen MASTER_POS_WAIT(dateiname, position [, timeout]) SLEEP(sekunden)</pre>	

9a) Aggregatfunktionen und Gruppierung (Aggregation)

Aggregatfunktionen fassen in einer SELECT-Anweisung die Werte einer Spalte über ALLE Datensätze zusammen. Kommt eine GROUP-BY-Klausel bezüglich einer (anderen) Spalte vor, erfolgt die Zusammenfassung PRO unterschiedlichem Wert dieser Spalte. Mehrere GROUP-BY-Spalten sind erlaubt. MySQL kennt folgende Aggregatfunktionen:

Funktion	Beschreibung
COUNT(*)	Anzahl ALLER Datensätze (auch NULL!)
COUNT(nr)	Anzahl aller Werte ungleich NULL
COUNT(DISTINCT nr)	Anzahl aller verschiedenen Werte ungleich NULL
SUM(nr)	Summe aller Werte
SUM(DISTINCT nr)	Summe aller verschiedenen Werte (MY!5.1)
AVG(nr)	Durchschnitt aller Werte
AVG(DISTINCT nr)	Durchschnitt aller verschiedenen Werte (MY!5.1)
MIN(nr)	Minimum aller Werte
MAX(nr)	Maximum aller Werte
STD(nr)	Standardabweichung Werte ungleich NULL (MY!)
STDDEV(nr)	Standardabweichung Werte ungleich NULL (MY!)
STDDEV_POP(nr)	Standardabweichung Werte ungleich NULL
STDDEV_SAMP(nr)	Standardabweichung aller Werte (auch NULL!)
VARIANCE(nr)	Varianz aller Werte ungleich NULL (MY!)
VAR_POP(nr)	Varianz aller Werte ungleich NULL
VAR_SAMP(nr)	Varianz aller Werte (auch NULL!)
BIT_AND(nr)	Bitweise UND-Verknüpfung aller Werte (MY!)
BIT_OR(nr)	Bitweise ODER-Verknüpfung aller Werte (MY!)
BIT_XOR(nr)	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 ANDEREN Spalten der jeweils zusammengefassten Datensätze Gesamtwerte.

Eigenschaften:

- * Pro UNTERSCHIEDLICHEM 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"
 - + Nicht zusammen mit Sortierung verwendbar
 - + NULL nicht in HAVING als Bedingung verwendbar

Syntax:

```

SELECT ...
FROM ...
GROUP BY ... [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)
FROM verkaeufe
GROUP BY jahr;

```

```

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

```

```

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

```

```

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

```

```

SELECT IFNULL(jahr, "ALLE_JAHRE") AS jahr, # Gesamtwert umbenennen
IFNULL(land, "ALLE_LÄNDER") AS land, # Gesamtwert umbenennen
IFNULL(produkt, "ALLE_PRODUKTE") AS produkt, # Gesamtwert umbenennen
SUM(gewinn)
FROM verkaeufe
GROUP BY jahr, land, produkt WITH ROLLUP; # Gruppen + Gesamt

```

10) Schlüsselfelder (Keys) und Indices

Schlüssel und Indices haben zunächst nichts miteinander zu tun:

- * Ein "Schlüssel" adressiert jeden Datensatz einer Datenbanktabelle EINDEUTIG
 - + Aus einer Spalte oder Kombination mehrerer Spalten gebildet
 - + Benutzt um Beziehungen zwischen Tabellen herzustellen
 - + NUR EIN (primärer) Schlüssel pro Tabelle möglich
- * Ein "Index" wird für eine Spalte oder Kombination von Spalten angelegt, nach denen häufig gesucht oder sortiert wird
 - + Ähnlich einem Inhaltsverz. zu einem Buch

- + Beschleunigt Suche nach Datensätzen zum Teil enorm
- + MEHRERE Indices pro Tabelle möglich
- * Für die Schlüssel einer Datenbank werden meist Indices erstellt

Unterscheidung:

- * Primärschlüssel
 - + Kennzeichnet JEDEN Datensatz EINDEUTIG (z.B. Abteilungsnummer)
 - + Meist eine (abstrakte) fortlaufende Nummer
 - + Häufig automatisch beim Anlegen eines Datensatzes erzeugt
 - + NUR EIN Primärschlüssel pro Tabelle möglich und notwendig
 - + Das Primärschlüsselfeld darf NIEMALS LEER sein (NOT NULL)
 - + Komplexe Primärschlüssel verlangsamen Datenbank-Operationen (am besten nur eine Spalte vom Typ kleine ganze Zahl verwenden)
- * Sekundärschlüssel
 - + Kennzeichnet ebenfalls Datensatz eindeutig (z.B. Abteilungsname)
 - + MEHRERE Sekundärschlüssel pro Tabelle möglich
 - + Ein Sekundärschlüsselfeld darf NIEMALS LEER sein (NOT NULL)
- * Fremdschlüssel
 - + 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
- * Indices
 - + Erleichtern Suche nach Datensätzen vor allem in großen Tabellen
 - + Einfügen, Ändern, Löschen von Datensätzen erfordert Index-Aktualisierung -> Kann viel Zeit kosten
 - + TIPP: Indices großer Tabellen erst NACH dem Füllen anlegen

Primärschlüssel erstellen/entfernen (automatisch/muss NOT NULL + UNIQUE sein):

- * Beim Erzeugen einer Tabelle (1.Form mehrspaltig, 1+2.Form = einspaltig)


```
CREATE TABLE <Tbl> (
  nr INT NOT NULL AUTO_INCREMENT,
  ...
  PRIMARY KEY (nr)
);
CREATE TABLE <Tbl> (
  nr INT AUTO_INCREMENT PRIMARY KEY,
  ...
);
```
- * Nachträglich erzeugen
 - Nur möglich, wenn Spalte "nr" NOT NULL + UNIQUE (KEINE doppelten Werte)!


```
ALTER TABLE pers ADD PRIMARY KEY (nr);
CREATE PRIMARY KEY ON pers (nr); # FALSCH! -> ALTER TABLE!
```
- * Löschen


```
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 erlaubt


```
ALTER TABLE pers ADD UNIQUE INDEX idx1 (name, vorname); # INDEX weglassbar
CREATE UNIQUE INDEX idx1 ON pers (name, vorname);
```
- * Löschen


```
ALTER TABLE pers DROP INDEX idx1; # immer nur INDEX (ohne UNIQUE, ...)
```

Index erstellen:

- * Allgemeine Syntax:


```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX <Idx>
  [USING {BTREE | HASH | RTREE}]
  ON <Tbl> (<Col> [(<Len>)] [ASC | DESC], ...);
```
- + Indexname <Idx> nur nötig, um Index gezielt löschen 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-Typ 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>, ...);

```

* Löschen

```

DROP INDEX <Idx> ON <Tbl>;
ALTER TABLE <Tbl> DROP INDEX <Idx>;

```

* Index 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;
```

FULLTEXT-Index:

- * Nur bei Engine "MyISAM" möglich!
- * Nur auf Spaltentypen CHAR, VARCHAR, TEXT möglich
- * Suche nach einzelnen Worten und beliebig langen Texten möglich
- * Nach bestimmten "Stopwords" wir NICHT gesucht (z.B. and, or, ...)
- * Suche NICHT mit LIKE/REGEX möglich
- * Spezielle Suchsyntax:

```

MATCH (<Coll>, ...) AGAINST ("<ConstString>" [<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 #

```

SPATIAL-Index:

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

Hinweise:

- * Nur 1 PRIMARY Index pro Tabelle erlaubt (automatisch UNIQUE + NOT NULL, mehr als 1 Spalte erlaubt)
 - + Primärschlüssel einer Tabelle ist automatisch auch ein Index
 - + Darf mehr als 1 Spalte enthalten!
 - + Kann keinen NULL-Wert enthalten!
- * Mehrere UNIQUE Indices anlegbar
 - + NULL-Werte darin erlaubt
- * Index auch auf Spaltenpräfix beschränkbar (z.B. name(5), vorname(5)) (nicht auf Suffix -> TRICK: Daten mit REVERSE() umgedreht speichern)
- * Tabelleninhalt wird durch Indices nicht verändert
- * Indices belegen zusätzlichen Speicherplatz
- * Indices enthalten KEINE zusätzlichen Daten, sondern erlauben nur schnelleren Lese-Zugriff auf bestimmte Tabellen
 - + Indices sind daher jederzeit löschar und aus den Tabellendaten wieder herstellbar
 - + Nur Abfragen über indizierte Spalten gehen schneller, andere nicht
- * Beim Erzeugen + Ändern von Datensätzen kosten Indices zusätzlichen Aufwand, da jeder Schreibvorgang auf Datensätzen die Indices mitpflegen muss
 - Während nachträglicher Index-Erzeugung ist Tabelle für Schreiben gesperrt!
- * Indices zur Abarbeitung von SELECT und JOIN-Anweisungen verwendet
- * Datenbank sucht zu jeder Abfrage SELBSTÄNDIG sinnvoll einzusetzende Indices (mit Hilfe des "Query Optimizer")
- * Tabellen/Indices enthalten Informationen über Spezifität und statistische Verteilung der indizierten Daten (Histogramm)
 - + Bsp: Ein Index auf Spalte "Geschlecht" ist wenig nützlich, da immer etwa auf die Hälfte der Daten zugegriffen wird
 - + Bsp: Ein Index auf Spalte "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)
 - + Dieser mehrspaltige Index kann immer dann genutzt werden, 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 Daten-Spalten gelesen, die in EINEM Index enthalten sind, dann KEIN Tabellenzugriff notwendig (Index enthält bereits alle Daten)
  - \* Index enthält "Zeiger" auf passende Datensätze in sortierter Reihenfolge
  - \* 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>

```
  - \* KEIN Index bei folgenden Vergleichen verwendbar:
 

```

!= # Ungleich
<> # Ungleich
LIKE "%abc" # Platzhalter "%" vorne
LIKE "__abc" # Platzhalter "_" vorne
RLIKE "abc" bzw. REGEX "abc" # Regulärer Ausdruck

```
  - \* Werden mehrere Bedingungen per AND/OR verknüpft, dann wird meist der Index zu den Spalten einer Bedingung verwendet, der am SPEZIFISCHSTEN 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

## 11) 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 sind dann nicht eindeutig identifizierbar)
- \* Spalten mit wenigen unterschiedlichen Werten (z.B. Anrede, Geschlecht) bringen mit Index KEINEN Geschwindigkeitsgewinn (Kardinalität gering)
- \* Indices auf Tab. mit wenig Datensätzen (<=12) bringen KEINEN Geschw.gewinn
- \* Mehr als 10% Datensätze einer Tab. ständig gelesen -> Index nicht sinnvoll
- \* Indices auf Tabellen einsetzen, die oft gefiltert/gruppirt/sortiert werden
- \* Zwei Indices notwendig für auf- + absteigendes Sortieren (ASC, DESC)
- \* Sortierkriterium an letzter Stelle im Index -> (teuren) Filesort sparen
- \* Aggregatfunktionen nutzen keinen Index (da alle Datensätze durchlaufen werden) (COUNT, SUM, MIN, MAX, AVG, STDDEV, ...)
- \* 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. Index-Tabelle ausreichend -> Daten lesen unnötig
  - + Weniger Platz für Datensätze mit gleichem Index nötig
  - + 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
  - + REVERSE verwenden, falls Anfang statisch

### Eigenschaften von Indices:

- \* Automatisch Präfix- und Längen-komprimiert (MY!)
- \* MySQL verwendet pro Tabelle in einer Query maximal EINEN Index, und zwar den mit der kleineren Treffermenge -> evtl. mehrspaltiger Index sinnvoll!
- \* Normalerweise verwendet: Index mit kleinster Anzahl passender Datensätze

### Indices werden verwendet für:

- \* Datensätze gemäß WHERE-Klausel
- \* Datensätze weglassen
- \* Doppelte Einträge rauswerfen (DISTINCT)
- \* Joins: 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

- \* Bei = > >= < <= 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!)
- \* Bei Verwendung von LIMIT

Indices werden NICHT verwendet für:

- \* Leseoperationen, die großen Prozentsatz einer Tabelle lesen
- \* OR auf Spalten in verschiedenen Indices -> IN, UNION [ALL]!

Verschiedene Indexverfahren möglich (MY!):

- \* B-Tree: Fast immer möglich
- \* R-Tree: Für geometrische Daten
- \* Hash: Bei MEMORY-Tabellen
  - + Einschränkung des Hash-Index
    - NUR für = und <=>
    - Nicht für != <> < <= > >= verwendbar
    - Nicht für ORDER BY und GROUP BY
    - Nur ganze Indices, keine Präfixe

Hinweis für Indexnutzung (Index-Hint) in SELECT-Anweisungen pro Tabelle (MY!):

- \* USE INDEX = NUR diese Indices benutzen ("Full Table Scan" möglich)
- FORCE INDEX = Analog USE INDEX ("Full Table Scan" NUR wenn nicht anders möglich)
- IGNORE INDEX = Diese Indices NICHT benutzen
- \* Name des Primären Index ist "PRIMARY"
- \* USE INDEX Liste darf auch leer sein -> keine Indices benutzen
- \* Fehlt FOR ..., dann für alle 3 Fälle (JOIN, ORDER BY, GROUP BY) gültig

```

+-----+
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;
```

Müssen Indices reorganisiert werden?

- \* JA: Nach dem Laden von Daten
  - Cardinality = Durchschnittliche Datensatzanzahl mit gleichem Wert berechnen
  - OPTIMIZE TABLE <Tbl>;
  - ANALYZE 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

## 12) Fremdschlüssel (Foreign Keys) und Referenzielle Integrität

Grund für Foreign Keys:

- \* Dokumentieren Beziehungen zwischen Tabellen (Master/Detail, Vater/Kind)
- \* Verhindern Einfügen von Inkonsistenzen in DB durch Programmierer
  - + "Referenzielle Integrität" erhalten
  - + Reihenfolge von Einfüge/Änderungs/Löschooperationen nicht zu beachten (verhindert "verwaiste Kinder" = orphaned childs)
  - + Fehlerbehandlung bei Unterbrechung
- \* Zentrale Constraint-Prüfung (in Anwendung verzichtbar + einheitlich)
- \* Kaskadierende UPDATES/DELETES vereinfachen Anwendungscode

In einer Datenbank werden die Zusammenhänge zwischen den Tabellen in Form einer Beziehung zwischen Primär- und Fremdschlüsseln abgelegt:

- \* Beim Löschen/Ändern eines Datensatzes aus einer (Eltern)Tabelle werden alle damit über den gleichen Fremdschlüssel verknüpften Datensätze in den zugeh. (Kind)Tabellen ebenfalls AUTOMATISCH von der Datenbank gelöscht/geändert
- \* Beim Einfügen eines Datensatzes in eine (Kind)Tabelle mit Fremdschlüssel

prüft das Datenbanksystem, ob eine Entsprechung in der referenzierten (Eltern)Tabelle vorhanden ist. Wenn nein, wird Fehlermeldung ausgegeben und Datensatz nicht eingefügt

## Eigenschaften:

- \* Grundsätzlich bei Tabellendefinition angebbbar
  - + Bei allen Engines außer "InnoDB" syntaktischer Zucker (reine Doku)
  - + Nur bei "InnoDB" gespeichert
  - + Speicherung und Implementierung für "MyISAM" geplant
  - + Erst in MY!6.0 einheitlich über alle Engines möglich
- \* Workaround falls nur ON DELETE nötig:
  - + Multi-Table DELETE
- \* Erzeugen zusätzlichen Overhead
  - + Performance: 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!

## Bedingungen:

- \* 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

## Erzeugen einer Tabelle mit Fremdschlüsseln:

```
CREATE TABLE <Tbl> (
 ma_id INT NOT NULL,
 pr_id INT NOT NULL,
 ...,
 FOREIGN KEY (ma_id) REFERENCES mitarbeiter (id)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
 FOREIGN KEY (pr_id) REFERENCES projekt (id)
 ON UPDATE CASCADE
 ON DELETE CASCADE
) ENGINE = "InnoDB"; # Engine = InnoDB (TYPE veraltet!)
```

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

- \* MATCH-Klauseln <MatchOpt> werden derzeit noch ignoriert (immer SIMPLE), sie definieren die Behandlung von NULL-Werten in mehrspaltigen Fremdschlüsseln beim Vergleich mit dem Primärschlüssel
  - MATCH SIMPLE # Mehrsp. Fremdschl. darf teilw. oder ganz NULL sein (Std)
  - MATCH PARTIAL # Mehrsp. Fremdschl. darf nicht vollständig NULL sein
  - MATCH FULL # Mehrsp. Fremdschl. muss vollständig ungleich NULL sein
- \* Verhalten beim Einfügen/Löschen steuern:
  - ON DELETE ... # Beim Löschen von Zeilen in Elterntabelle
  - ON UPDATE ... # 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/Aktualisieren transitiv auf Kindtabelle ausdehnen
  - SET NULL # Ref. Spalten in Kindtab. auf NULL setzen
  - SET DEFAULT # Ref. Spalten in Kindtab. auf Default setzen (nicht MY!)

```
ALTER TABLE <Tbl> ADD [CONSTRAINT <Name>] # Kindtabelle
 FOREIGN KEY <Idx> # 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
```

## Foreign-Keys löschen:

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

## Foreign Keys anzeigen:

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

## Foreign-Keys überprüfen vor Datenladen aus und danach wieder einschalten:

```
SET foreign_key_checks = 0;
SOURCE <SqlFile>; # kein "..." um <SqlFile>!
SET foreign_key_checks = 1;
```

-----  
 Ein Join ist eine Verknüpfung von zwei Tabellen zu einer Gesamttabelle über eine Abhängigkeit zwischen den Tabellen (Primärschlüssel + Fremdschlüssel). MySQL unterstützt folgende JOIN-Arten:

|                                       |                                                                                                                                                                                                                       |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [CROSS] JOIN                          | Kombination aller Datensätze der 1. Tabelle mit allen der 2. Tabelle ("Kreuzprodukt", "Kartesisches Produkt") (andere Schreibweise: <Tbl1>, <Tbl2>).                                                                  |
| [INNER] JOIN                          | Komb. der Datensätze der 1. Tab. mit dazu passenden der 2. Tab. bzgl. einer/mehrerer Spalten. Datensätze beider Tab. weglassen, deren Wert in Komb.spalte nicht gemeinsam vork.                                       |
| {LEFT   RIGHT} [OUTER] JOIN           | Analog INNER JOIN, aber JEDER Datensatz aus LEFT=linker bzw. RIGHT=rechter Tabelle ist im Ergebnis vorhanden (Spalten aus anderer Tabelle haben evtl. alle Wert NULL)                                                 |
| NATURAL [{LEFT   RIGHT} [OUTER]] JOIN | Automatisch verknüpft über ALLE gemeinsamen Spaltennamen (USING ... unnötig)                                                                                                                                          |
| FULL [OUTER] JOIN                     | Kombination aus LEFT + RIGHT JOIN, d.h. JEDER Datensatz der 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 wird immer zuerst VOR rechter Tab. gelesen), entspricht JOIN (d.h "Kreuzprodukt") |

## Hinweise:

- \* Die Schlüsselworte CROSS, INNER und OUTER dürfen weggelassen werden (aus Gründen der klareren Programmierung aber 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 Begriffe

| Begriff    | Bedeutung                                                   |
|------------|-------------------------------------------------------------|
| Self-Join  | Verknüpfung einer Tabelle mit sich selbst (rekursiv)        |
| Equi-Join  | Verknüpfung über "="-Relation                               |
| Theta-Join | Verknüpfung über and. Relation 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> [<JoinCond>]
 | <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] (<Idx>, ...)
 | FORCE {INDEX | KEY} [FOR JOIN] (<Idx>, ...)
 | IGNORE {INDEX | KEY} [FOR JOIN] (<Idx>, ...)

```

## Beispiele:

```

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

```

```

SELECT a.name, h.name
FROM artikel a LEFT OUTER JOIN hersteller h

```

```
USING (name); # Spalten gleichnamig
```

#### 14) Mengenoperationen

Mengenoperationen 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 vorh., MY!) |
| EXCEPT/MINUS     | Differenz = 1. minus 2. Selektion (NICHT vorh., 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 (DISTINCT ist Standard)
- \* Sortierung der Daten wird zerstört (Gesamtergebnis sortieren mögl.)

#### Beispiel:

```
SELECT name, vorname FROM pers # Selektion 1
UNION
SELECT name, vorname FROM copy; # Selektion 2
```

#### 15) Unterabfragen (Subqueries/Subselect)

#### Eigenschaften:

- \* SELECT-Statement eingebettet ("nested") in anderem Statement
- \* 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 JOINS und UNIONS einsetzbar
- \* Vorsicht bei NULL-Werten und Empty Tables!

#### Syntax:

```
---- outer query ---- --- inner query ---
SELECT * FROM t1 WHERE coll = (SELECT coll FROM t2);
SELECT * FROM t1 a WHERE a.coll = (SELECT b.coll FROM t2 b);
SELECT * FROM t1 WHERE t1.coll = (SELECT t2.coll FROM t2);
-- outer statement -- ----- subquery -----
```

#### Ergebnis einer Subquery kann sein:

- \* EIN Skalar (einzelner Wert):
  - = (<SubQuery>)
  - <> (<SubQuery>)
  - > (<SubQuery>)
  - < (<SubQuery>)
  - >= (<SubQuery>)
  - <= (<SubQuery>)
- \* EINE Datenzeile (ROW-Subquery mit "Row Constructor"):
  - ("wert1", "wert2", ...) = (<SubQuery>)
  - ROW("wert1", "wert2", ...) = (<SubQuery>)
  - EXISTS (<SubQuery>) # Wahr, wenn mind. 1 Datenzeile zurück
  - NOT EXISTS (<SubQuery>) # Wahr, wenn keine einzige Datenzeile 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

```

## 16) Transaktionen

Ein DBMS soll die Datenkonsistenz sichern bei:

- \* Gleichzeitigem (Schreib)Zugriff mehrerer Benutzer
- \* MySQL-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  
Zur Auflösung von Deadlocks (Über-Kreuz-Exklusiv-Zugriffen)

ACID-Eigenschaften einer Transaktion:

|              |                                                           |
|--------------|-----------------------------------------------------------|
| A)tomicity   | Komplett oder gar nicht                                   |
| C)onsistency | DB danach konsistent (nicht unbedingt während!)           |
| I)solation   | Gleichzeitig ablaufende Transakt. beeinflussen sich nicht |
| D)urability  | Ergebnis erfolgreicher Transakti. steht dauerhaft in DB   |

Transaktionen über mehrere SQL-Statements:

- \* NUR mit Engines "InnoDB" + "BDB" (MY!)
- \* Bei allen anderen Engines ist jedes einzelne SQL-Statement eine TA (sofern der MySQL-Server nicht abstürzt)

- \* Mit "MyISAM" nur Locken kompletter Tabellen möglich (LOCK/UNLOCK)
- \* Tabellenformat 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 abgeschaltet (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; # Std: Jede Aktion ist TA, BEGIN startet TA bis COMMIT
  - SET AUTOCOMMIT = 0; # Ständig TA offen, COMMIT/ROLLBACK beendet+startet neue

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

```
BEGIN [WORK]; BEGIN [WORK]; # TA beginnen
...
SAVEPOINT <Label>; ...
...
ROLLBACK [WORK] TO SAVEPOINT <Label>; ...
...
COMMIT [WORK]; ROLLBACK [WORK]; # Abschließen/zurücknehmen
```

Folgende Anweisungen beenden eine Transaktion ebenfalls (neben COMMIT/ROLLBACK), d.h. führen automatisch "COMMIT" durch:

```
DROP DATABASE ...
DROP TABLE ...
TRUNCATE TABLE ... # DELETE FROM <Tbl>; ohne WHERE beendet TA nicht
ALTER TABLE ...
RENAME TABLE ...
CREATE INDEX ...
LOCK TABLES ...
BEGIN ...
```

Hinweise:

- \* 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
- \* Bei Anwendung einer Transaktion auf nicht transaktionssichere Tabelle wird jede Änderung sofort durchgeführt (AUTOCOMMIT=1)
- \* "ROLLBACK" auf nicht transaktionssichere Tabelle führt zu Fehlermeldung
- \* Protokolldateien
  - + ASCII: Zeichnet "ROLLBACK" auf
  - + Binär: Zeichnet "ROLLBACK" NICHT auf

Transaktions-Level (Isolations-Ebene) einstellen (Std: REPEATABLE READ):

- \* SESSION = Für AKTUELLE Sitzung
- GLOBAL = Für ALLE danach neu gestartete Sitzungen (nicht für bestehende!)
- Sonst = Für NÄCHSTE Transaktions-Anweisung festlegen (1x!)
- \* Beim Start des MySQL-Servers festlegen durch:
  - transaction-isolation = <Level> # Std: GLOBAL

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

| <Level>          | N<br>r | Datensatz-<br>sperre | Lost<br>Update | Dirty<br>Read | Nonrepeatable<br>Read | Phantom<br>Read |
|------------------|--------|----------------------|----------------|---------------|-----------------------|-----------------|
| keiner           | -      | nein                 | ja             | ja            | ja                    | ja              |
| READ UNCOMMITTED | 0      | nein                 | nein           | ja            | ja                    | ja              |
| READ COMMITTED   | 1      | write                | nein           | nein          | ja                    | ja              |
| REPEATABLE READ  | 2      | read+write           | nein           | nein          | nein                  | ja              |
| SERIALIZABLE     | 3      | read+write           | nein           | nein          | nein                  | nein            |

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;
```

## 17) Locking

Tabellen/View-Lock:

- \* Tabellen/Views für exklusiven Zugriff sperren + wieder freigeben
- \* Differenzierung Schreib- oder Lese- und Schreiblock
- \* Alle angegebenen Tabellen werden vollständig gesperrt
- \* Funktioniert bei jeder Engine!

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

Beispiel:

```
LOCK TABLES `pers` WRITE;
ALTER TABLE `pers` DISABLE KEYS;
INSERT INTO pers (nr, vorname, name)
 VALUE (1, "Thomas", "Birnthaler");
INSERT INTO pers (nr, vorname, name)
 VALUE (2, "Markus", "Mueller");
INSERT INTO pers (nr, vorname, name)
 VALUE (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 Transaktionsbeginn) 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 sitzungsspezifisch)
- \* LOCK auf View lockt alle ihre Basistabellen
- \* Nach LOCK sind nur gelockte Tabellen verfügbar (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, die sich nicht daran beteiligen, werden nicht mit einbezogen (können machen was sie wollen)
- \* Deadlock möglich

| Funktion                  | Bedeutung                                                                                      |
|---------------------------|------------------------------------------------------------------------------------------------|
| GET_LOCK(<Name>, timeout) | Lock <Name> beantragen (max. timeout Sekunden)<br>(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                                                                          |

## 18) Views (Sichten)

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

Zweck:

- \* Neue Spaltennamen einführen
- \* Zugriffsbeschränkung durchsetzen (über Sichtbarkeit)
- \* Zugriff vereinfachen (Verknüpfung mehrerer Tab. sieht wie eine Tab. aus)

Eigenschaften:

- \* Ableitung aus einer oder mehreren Basistabellen 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 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 "eingefroren"
  - + "Updatability Flag" wird erstellt (UPDATE/INSERT prinzipiell möglich)
  - + Änderungen an Unter-Tabellen oder -Views wirken sich nicht aus
  - + Löschen von Unter-Tab/Views führt bei View-Verwendung zu Fehlermeldung
- \* Beschränkung der Einfüge-Daten auf Erfüllung der View-WHERE-Klausel möglich (WITH CHECK OPTION -> prüft, ob Daten mit WHERE-Bed. der View ausgefiltert)

## Beschränkungen:

- \* Aktualisierbar (updateable, d.h. UPDATE/DELETE) wenn 1:1-Beziehung zwischen Datensätzen in View und in Basistabellen 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
  - + Nur EINE Basistabelle ist betroffen
  - + Nicht möglich bei Realisierung der View per temp. Tabelle
- \* Insertable (d.h. INSERT) wenn zusätzlich zu obigen Bedingungen
  - + Kein Basis-Spaltenname doppelt verwendet
  - + Alle Spalten der Basistabelle ohne Defaultwert in View enthalten
  - + Keine per Ausdruck abgeleiteten View-Spalten
  - + Nur EINE Basistabelle 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

## "Materialized" Views (nicht in MY!):

- \* Kosten Speicherplatz
- \* Werden nicht bei jeder Veränderung der Basistabellen aktualisiert
- \* Verhalten sich wie statische Tabellen -> schneller als normale Views
- \* Bei häufigen View-Abfragen, deren Basistabellen sich selten ändern
- \* Simulierbar per EVENTS, die "M.V." periodisch als Tabelle erstellen (MY!)

## 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  
Effizienter, updatebar, Locks länger gehalten,  
Nicht verwendbar bei Aggregat-Funktionen SUM, MIN, MAX, COUNT  
DISTINCT, GROUP BY, HAVING, LIMIT, UNION, UNION ALL, Subqueries
- \* 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):

```
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
```

Beispiele für Views:

```
CREATE TABLE t (
 anz INT,
 preis INT
);
INSERT INTO t VALUES
(3, 50),
(7, 80),
(9, 99),
(1, 12);
CREATE VIEW v1 AS
 SELECT anz, preis, anz*preis AS "wert"
 FROM t;
SELECT * FROM v1;
DROP VIEW v1;
DROP TABLE t;
```

```
CREATE TABLE t (
 nr INT,
 name CHAR(30)
);
INSERT INTO t VALUES
(1, "a"),
(2, "abcde"),
(5, "test"),
(4, "text");
CREATE VIEW v2 AS
 SELECT *, "konstant"
 FROM t WHERE nr = LENGTH(name);
SELECT * FROM v2;
DROP VIEW v2;
DROP TABLE t;
```

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

```
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";
```

```
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
```

```
DROP VIEW v1;
DROP VIEW v2;
DROP VIEW v3;
DROP VIEW v4;
DROP VIEW v5;
```

19) Variablen

MySQL kennt folgende Variablentypen:

| Typ   | Bedeutung                                                                                                            |
|-------|----------------------------------------------------------------------------------------------------------------------|
| <Var> | Lokale Variablen und Parameter in Stored Procedures:<br>+ Im Inneren mit Datentyp zu DEKLARIEREN und nur dort gültig |

|         |                                                                                                                                                                                                                         |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| @<Var>  | Session/Sitzungs-Variablen (benutzerspezifisch/userdefined):<br>+ Verlieren ihren Wert am Ende der Sitzung                                                                                                              |
| @@<Var> | Globale/System/Server-Variablen:<br>+ Definieren Zustände oder Attribute des MySQL-Servers<br>+ Manche dieser Variablen gibt es 2-fach:<br>- SESSION: Für aktuelle Verbindung<br>- GLOBAL: Als systemweiter Defaultwert |

## Eigenschaften:

- \* Name muss ungleich Objektnamen und SQL-Schlüsselworten sein
- \* Bis MY!4.1 wird GROSS/kleinschreibung unterschieden, ab MY!5.0 nicht mehr
- \* Für Änderung von Globalen Variablen ist SUPER-Recht notwendig

## Variablen-Zuweisung:

```
SET @var = 3;
SET @var := 3;
SELECT @var := 3;
SELECT @var := COUNT(*) FROM pers;
SELECT COUNT(*) FROM pers INTO @var;
SELECT name, vorname FROM pers WHERE nr = 1 INTO @var1, @var2; # MY!5.0
```

## Variablen anzeigen (einzelne, alle oder Teil davon):

```
SELECT @var;
SELECT @@binlog_cache_size;
SHOW SESSION VARIABLES;
SHOW GLOBAL VARIABLES;
SHOW VARIABLES LIKE "a%";
SHOW VARIABLES WHERE Variable_name LIKE "%myisam%" AND Value > 0;
```

## Unterscheidung Globale und Session-Variablen:

```
SELECT @wait_timeout; # Session
SELECT @session.wait_timeout; # Session
SELECT @@wait_timeout; # Global
SELECT @@global.wait_timeout; # Global

SET @wait_timeout = 10001; # Session
SET @session.wait_timeout = 10002; # Session
SET SESSION wait_timeout = 10003; # Session
SET @@global.wait_timeout = 10004; # Global
SET GLOBAL wait_timeout = 10005; # Global

SELECT @wait_timeout; # Session
SELECT @session.wait_timeout; # Session
SELECT @@wait_timeout; # Global
SELECT @@global.wait_timeout; # Global
```

## 20) Prepared Statements (Vorbereitete Anweisungen)

Prepared Statements sind vorbereitete SQL-Anweisungen mit PLATZHALTERN der Form "?", die erst beim Aufruf mit übergebenen WERTEN (Variablen) gefüllt werden. Ein Prepared Statement ist entweder ein Stringliteral oder eine Variable gefüllt mit dem Text eines SQL-Statements.

## Eigenschaften:

- \* Geschwindigkeitsgewinn, da SQL-Anweisung bereits vorkompiliert
- \* Sicherheitsgewinn
  - + SQL-Injection unmöglich (Benutzer-Eingabe manipuliert SQL-Statement)
  - + REGEL: "Never trust incoming Data!"
  - + Reguläre Ausdrücke zum Prüfen der Benutzer-Daten bieten sich an!
- \* Platzhalter nur für Datenwerte, NICHT aber für SQL-Schlüsselworte und Namen von Datenbanken, Tabellen, Spalten einsetzbar
  - Aber für LIMIT/OFFSET-Werte!
- \* Nur Variablen in Platzhalter einsetzbar, keine Ausdrücke oder Literale
  - + Anzahl der Variablen muss gleich Anzahl der Platzhalter sein
- \* Nur eine SQL-Anweisung pro Prepared Statement erlaubt (kein ";")
- \* Nicht verschachtelbar
- \* In Prozeduren verwendbar, nicht aber in Funktionen und Triggern
- \* Nur pro Sitzung existent (liegen nicht auf Platte!)
- \* Prepared Statement mit gleichem Namen wird überschrieben

## Links zu SQL-Injection:

- \* Tool zum Simulieren von SQL-Injections auf Webseiten  
<http://sqlmap.sourceforge.net/>
- \* SQL Injection Cheat Sheet  
<http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>

- \* SQL Injection Walkthrough  
<http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- \* Understanding and Preventing SQL Injection Attacks  
<http://www.silksoft.co.za/data/sqlinjectionattack.htm>
- \* SQL-Injection-Beispiele (Steve Friedl)  
<http://www.unixwiz.net/techtips/sql-injection.html>

Vorbereitete Anweisung mit Platzhaltern:

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

Vorbereitete Anweisung ausfüllen und ausführen:

```
EXECUTE <PrepStm>; # Ohne Platzhalter
EXECUTE <PrepStm>; # Falls Platzhalter "?" verwendet
USING <Var1>, ...; # MÜSSEN Variablen sein, keiner Ausdr./Literale!
```

Prepared-Statement anzeigen gibt es nicht!

Vorbereitete Anweisung löschen:

```
DROP PREPARE <PrepStm>;
DEALLOCATE PREPARE <PrepStm>;
```

Beispiel (? nicht in "..."/'...' setzen, auch wenn Datentyp dies erfordert!):

```
USE first;

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";
SET @autor = "Thomas Birnthaler";
SET @isbn = "3-987654-321-0";

EXECUTE neues_buch USING @titel, @autor, @isbn;
EXECUTE neues_buch USING "Carrie", "King", "9-876543-210-X"; # Geht NICHT!

SELECT * FROM buecher;

DROP PREPARE neues_buch;
```

## 21) Routinen (Stored Procedures)

---

Eigenschaften:

- \* Dienen der Kapselung von Geschäftslogik in der Datenbank (nur 1x), Anwendungen rufen sie nur auf (statt sie Nx selbst zu programmieren)
- \* Routinen = Prozeduren + Funktionen
  - + Funktionen haben EINEN Rückgabewert  
daher in Ausdrücken oder mit SELECT aufrufen (wird gerne vergessen!)
  - + Prozeduren haben KEINEN Rückgabewert (außer über OUT-Parameter),  
daher mit "CALL" aufzurufen (wird gerne vergessen!)
- \* An Datenbank <Db> GEBUNDEN (werden mit dieser Datenbank gelöscht!),  
eigentliche Definition liegt in der zentralen Datenbank "mysql"
- \* Anweisungen in Funk./Proz. beziehen sich auf einzige Datenbank  
(auf beim Aufruf aktuell mit USE ausgewählte)
- \* Körper (Body) besteht aus einer oder mehreren SQL-Anweisungen  
(mehrere in BEGIN...END einschließen)
- \* Erlaubt sind die meisten DDL-, DML- und DCL-Anweisungen  
(aber keine Transaktionssteuerung!)
- \* Rechte zum Erstellen/Ändern (CREATE ROUTINE, ALTER ROUTINE)  
und zum Ausführen (EXECUTE) notwendig
- \* Lokale Variablen definierbar (per SET...)  
+ Parameter sind ebenfalls automatisch lokale Variablen
- \* Aufruf:
  - CALL <Proc>; # In Default/Standard-Datenbank
  - CALL <Db>.<Proc> # In anderer Datenbank
  - SELECT <Func>; # Oder im Rahmen eines Ausdrucks
  - SELECT <Db>.<Func>; # Oder im Rahmen eines Ausdrucks
- \* Funktionen sind 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)
- \* Parameter NICHT als Platzhalter für Datenbank/Tabellen/Spaltennamen nutzbar (aber als LIMIT/OFFSET-Werte!)
- \* Macht "Prelocking" aller darin benötigten Tabellen!

## Hinweise:

- \* SET vor Variablenzuweisung gerne vergessen
- \* CALL vor Prozeduraufruf gerne vergessen
- \* SELECT auch vor Funktionsaufruf notwendig
- \* SQL-Delimiter ";" VOR der Definition einer Prozedur/Funktion temporär auf "/" oder ähnliches setzen und DANACH wieder auf ";" zurücksetzen, damit ";" in Definition verwendbar ist (MY!):

HINWEIS: Beispiele für alle Kontrollstrukturen folgen am Ende des Abschnitts

## 21a) Lokale Variablen

## Eigenschaften:

- \* Nur innerhalb Prozeduren/Funktionen erlaubt (mit <Type>!) (außerhalb per SET <Var> = <Val> ohne <Type>!)
- \* Nur am Körper-Anfang, direkt nach BEGIN!
- \* Jeder SQL-Datentyp verwendbar
- \* Kein "@" vor Variablenname (= lokal)
- \* Verschwinden beim Verlassen der Routine
- \* Datentypen INT, FLOAT, CHAR,

```
DECLARE anz INT DEFAULT 0;
DECLARE max FLOAT DEFAULT 0.0;
DECLARE sum FLOAT DEFAULT 0.0;
DECLARE str CHAR(30) DEFAULT "";
DECLARE str2 CHAR(30) DEFAULT "Maximale Bestellsumme";
```

## 21b) Wertzuweisung an Variablen

```
SET n = n - 1000;
SET s = CONCAT(s, "x");

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 func(1, 3.14, "text") INTO str; # Funktionsergebnis
```

## 21c) Ausgabe von Variablen oder Daten

```
SELECT "Fester Text: ", 17.00, " Euro"; # Fixe Werte
SELECT CONCAT("Fester Text: ", 17.00, " Euro") AS Text; # Verketteter Text
SELECT @str AS "Titel"; # Variable
SELECT func(1, 3.14, "text") AS "Funktionsergebnis" # Funktionsergebnis
```

## 21d) Kontrollstrukturen (Block/Compound Statement)

## Eigenschaften:

- \* Fasst mehrere SQL-Anweisungen zu einem Block zusammen
- \* Zu Beginn DECLARE-Anweisungen erlaubt
- \* Definiert Geltungsbereich für DECLARE-Variablen (danach vergessen)
- \* Marke <Label> kann max. 16 Zeichen lang sein
- \* Marke <Label> kann fehlen (falls verwendet, am Kontrollstruktur-Ende zu wiederholen!)

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

## 21e) Kontrollstrukturen (Verzweigungen)

Verzweigung: Die Anweisungen nach der von oben nach unten ersten wahren Bedingung <CondI> werden ausgeführt; oder falls keine <CondI> wahr wird, die Anweisungen nach ELSE; oder falls ELSE nicht vorhanden: gar keine Anweisung):

```

IF <Cond1> THEN
 <Statement>;
 ...
ELSEIF <Cond2> THEN # Beliebig oft (auch weglassbar)
 <Statement>;
 ...
ELSE # Wegglassbar
 <Statement>;
 ...
END IF; # ";" nicht vergessen!

```

Verzweigung: Die Anweisungen nach der von oben nach unten ersten wahren Bedingung <CondI> werden ausgeführt; oder falls keine <CondI> wahr wird, die Anweisungen nach ELSE; oder falls ELSE nicht vorhanden: gar keine Anweisung):

```

CASE # Variante A: Bedingungen prüfen
 WHEN <Cond1> THEN <Statement>; ... # Bedingung 1 prüfen
 WHEN <Cond2> THEN <Statement>; ... # Bedingung 2 prüfen
 ...
 ELSE <Statement>; ... # Besser immer angeben!
END CASE; # ";" nicht vergessen!

```

Fallunterscheidung: <Expr> wird ausgewertet und das Ergebnis von oben nach unten mit den Werten <ValI> verglichen und die Anweisungen nach dem gleichen Wert <ValI> 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 Werten vgl.
 WHEN <Val1> THEN <Statement>; ... # Ausdruckergebnis
 WHEN <Val2> THEN <Statement>; ... # mit Werten vergleichen
 ...
 ELSE <Statement>; ... # 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!

## 21f) Kontrollstrukturen (Schleifen)

-----

Hinweise (Beispiele folgen weiter unten):

- \* <Label> kann fehlen (falls verwendet, ist sie am Ende der Kontrollstruktur zu wiederholen!)
- \* LEAVE bricht eine Schleife vorzeitig ab
- \* ITERATE springt zum Schleifenanfang (nächster Durchlauf)
- \* Es gibt kein GOTO <Label> (wurde kurzzeitig getestet, aber wieder entfernt)

Endlosschleife:

```

[<Label>:] LOOP # Evtl. Positions-Marke setzen
 <Statement>;
 ...
 ITERATE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 LEAVE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 ...
END LOOP [<Label>]; # Marke wiederholen falls verwendet

```

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

```

[<Label>:] REPEAT # Evtl. Positions-Marke setzen
 <Statement>;
 ...
 ITERATE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 LEAVE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 ...
UNTIL <Cond> END REPEAT [<Label>]; # Wiederholen falls verwendet

```

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

```

[<Label>:] WHILE <Cond> DO # Evtl. Positions-Marke setzen
 <Statement>
 ...
 ITERATE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 LEAVE [<Label>]; # Innerste Schleife oder Schleife mit Marke
 ...
END WHILE [<Label>]; # Marke wiederholen falls verwendet

```

## 21g) Prozeduren

-----

## Hinweise:

- \* Name, Parameter (leere Liste erlaubt) und Prozedurkörper sind anzugeben
  - + Parameter sind in der Form [`<InOut>`] `<Name>` `<Type>` anzugeben, (z.B. INOUT wert TINYINT, IN ist Standard)
  - + `<Type>` ist ein beliebiger SQL-Datentyp (OHNE Längenangabe!)
- \* Eingabeparameter verhält sich wie initialisierte lokale Variable:
 

```
DECLARE <Var> <Type> DEFAULT <Val von außen>;
```
- \* Kann per OUT/INOUT-Parameter oder SELECT-Anweisung etwas zurückgeben
- \* Prozedur ohne Parameter kann ohne "()" aufgerufen werden

Parameter-Eigenschaft `<InOut>`:

| Syntax | Typ          | Bemerkung                                           |
|--------|--------------|-----------------------------------------------------|
| IN     | Eingabe(Std) | Ausdruck oder Literal (z.B. "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> <Type>, ...)
 [[NOT] DETERMINISTIC]
 {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
 [SQL SECURITY {DEFINER | INVOKER}]
 [COMMENT "<String>"]
BEGIN
 <Statement>;
 ...
END //
DELIMITER ;
```

## Aufruf einer Prozedur:

- \* Anzahl übergebener Argumente muss zu Definition passen
- \* Für OUT- und INOUT-Parameter muss eine Variable übergeben werden
- \* Für IN-Variable kann auch eine Konstante übergeben werden

```
DELIMITER //
CREATE PROCEDURE erste(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 ;

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

## 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>;
SHOW PROCEDURE CODE <Proc>;
```

## 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 in Funktion!
```

```

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

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

DROP PROCEDURE conv_xml_special //

DELIMITER ; # Nicht vergessen!

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

CREATE TABLE pers (# Variante A (Engine = MyISAM = 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") //

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 # Geht nicht
SELECT * FROM pers LIMIT grenze; # als Parameter!
 END IF;
END //

CALL select_limited(100) //
CALL select_limited(6) //
CALL select_limited(5) //
CALL select_limited(4) //

DROP PROCEDURE select_limited //

#-----
Prozedur 4
#-----
CREATE TABLE artikel (
 nr INT,
 bez CHAR(30),
 anz INT,
 preis FLOAT(6,2)
) //

INSERT INTO artikel (nr, bez, anz, preis)

```

Okt 22, 11 3:00

mysql-HOWTO.txt

Page 57/74

```

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) //

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) //

DROP PROCEDURE max_preis //
DROP TABLE artikel //

#-----
Prozedur 5
#-----
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; # Für Tests ganz nützlich

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) //

DROP PROCEDURE summe_artikel //
DROP TABLE artikel //

#-----
Prozedur 6
#-----
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") //
```

```
DROP PROCEDURE avg_sum_max_min //
```

```
DELIMITER ;
```

## 21h) Funktionen

```

```

### Hinweise:

- \* Name, Parameter (leere Liste erlaubt), Rückgabotyp, Funktionskörper nötig
- + Parameter sind in der Form <Name> <Type> anzugeben (z.B. Wert FLOAT)
- + <Type> 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

### Definition einer Funktion:

```
DELIMITER //
```

```
CREATE FUNCTION <Func> (<Param> <Type>, ...)
```

```
RETURNS <Type>
```

```
[[NOT] DETERMINISTIC]
```

```
{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
```

```
[SQL SECURITY {DEFINER | INVOKER}]
```

```
[COMMENT "<String>"]
```

```
BEGIN
```

```
<Statement>;
```

```
...
```

```
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>;
```

```
SHOW FUNCTION CODE <Func>;
```

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
#-----
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) //

DROP FUNCTION brutto_netto //

#-----
Funktion 2
#-----
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) //

DROP FUNCTION sumup_to_n //

#-----
Funktion 3
#-----
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) //

DROP FUNCTION area_or_volume //

#-----
Funktion 4
#-----
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) //

```

Okt 22, 11 3:00

mysql-HOWTO.txt

Page 61/74

```

SELECT roman(1001) //
SELECT roman(1962) //
SELECT roman(1999) //
SELECT roman(2000) //
SELECT roman(2007) //
SELECT roman(4567) //
SELECT roman(9999) //

DROP FUNCTION roman //

#-----
Funktion 5 (iterative Variante)
#-----
CREATE FUNCTION fakultaet(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 fakultaet(-10) AS "Fakultät von -10" // //
SELECT fakultaet(0) AS "Fakultät von 0" //
SELECT fakultaet(1) AS "Fakultät von 1" //
SELECT fakultaet(5) AS "Fakultät von 5" //
SELECT fakultaet(10) AS "Fakultät von 10" //
SELECT fakultaet(50) AS "Fakultät von 50" //
SELECT fakultaet(100) AS "Fakultät von 100" //

DROP FUNCTION fakultaet //

#-----
Funktion 6 (rekursive Variante)
ERROR 1424 (HY000): Recursive stored functions and triggers are not allowed
#-----
DELIMITER //

CREATE FUNCTION fakultaet(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 * fakultaet(n - 1));
 END IF;
END //

DELIMITER ;

#-----
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 Fktnamen verwendbar!)
#-----
DROP FUNCTION 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 ;

```

## 22) Trigger

Lösen automatische Reaktion bei der Aktion Einfügen, Ändern oder Löschen von Datensätzen in Tabelle aus. Kann VOR und/oder NACH der Aktion erfolgen.

### Zweck:

- \* Prüfung/Korrektur/Berechnung von Spaltenwerten mit BEFORE-Trigger
- \* Andere Tabellen updaten (History, Denormalisierung) mit AFTER-Trigger

### Eigenschaften:

- \* EINER einzelnen konkreten Tabelle zugeordnet (NICHT View, temp. Tabelle!) (Tabelle mit passenden Spalten muss existieren)
- \* VOR/NACH bestimmten Aktionen/Ereignissen autom. auf einer Tabelle ausgeführt (INSERT, UPDATE, DELETE, REPLACE = INSERT oder UPDATE)
  - + Pro Datensatz: FOR EACH ROW (Standard in MySQL)
  - + Pro SQL-Anweisung: FOR EACH STATEMENT (nicht in MySQL!)
- \* Gespeichert pro Tabelle in Datei "<Tbl>.TRG"
- \* Name eines Triggers muss pro Datenbank (Schema) eindeutig sein
- \* TIMESTAMP- und AUTO\_INCREMENT-Spalten haben eine Art "Default-Trigger"

### Beschränkungen:

- \* Trigger LOCKT während Ausführung ALLE beteiligten Tabellen vollständig! -> so schnell wie möglich durchlaufen!
- \* NUR EIN Trigger pro Kombination Zeitpunkt <Time> + Aktion <Event> möglich (aber mehrere Trigger insgesamt pro Tabelle möglich)
- \* Trigger haben gleiche Einschränkungen wie Stored Procedures (als <Statement> zw. BEGIN...END alle in Stored Procedures erlaubten mögl.)
- \* Im Trigger Zugriffe auf andere Tabellen erlaubt (keine Schreiboperationen auf Tabellen, die den Trigger auslösende Anweisung benutzt)
- \* Im Trigger Stored Prozeduren mit CALL und Funktionen aufrufbar (dürfen keine Daten produzieren, Datenaustausch nur per OUT/INOUT-Parameter)
- \* Aufruf von Stored Procedure per CALL als Trigger-Anweisung ist erlaubt
- \* Im Trigger können keine Transaktionen oder Locks beginnen/enden
- \* Im Trigger sind keine Prepared Statements erlaubt
- \* Derzeit nicht durch kaskadierte Foreign Key Aktionen ausgelöst (kommt noch!)
- \* Vorzeitiges Aktions-Ende in Trigger nur mit LEAVE, nicht mit RETURN möglich
- \* 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

### Fehlerbehandlung:

- \* Scheitert BEFORE-Trigger, wird Aktion NICHT durchgeführt
- \* BEFORE-Trigger wird durchgeführt, auch wenn eigentliche Aktion scheitert

- \* AFTER-Trigger wird nur durchgeführt, wenn BEFORE-Trigger (falls vorhanden) UND Aktion nicht gescheitert
- \* Fehler im BEFORE/AFTER-Trigger führt zu Fehler in auslösender Anweisung
  - + Transaktion: Anweisungs-Fehler nehmen ausgelöste Trigger wieder zurück
  - + Transaktion: Trigger-Fehler lösen Rücknahme der ganzen Anweisung aus
  - + Nicht-transakt. Tabellen: Ausgelöste Trigger werden bei Anweisungs-Fehler nicht zurückgenommen (und umgekehrt)

Hinweise zum zeitlichen Ablauf:

- \* INSERT: Immer Trigger BEFORE INSERT;  
nur falls Key NICHT vorhanden, Trigger AFTER INSERT.
- \* UPDATE: Immer Trigger BEFORE UPDATE + AFTER UPDATE falls Key vorhanden;  
falls Key NICHT vorhanden, findet gar kein Trigger statt.
- \* DELETE: Immer Trigger BEFORE DELETE + AFTER DELETE falls Key vorhanden;  
falls Key NICHT vorhanden, findet gar kein Trigger statt.
- \* REPLACE: Immer Trigger BEFORE INSERT und anschließend entweder Trigger  
AFTER INSERT (falls Key NICHT vorhanden) oder Trigger BEFORE  
DELETE + AFTER DELETE + AFTER INSERT (falls Key vorhanden).
- \* INSERT ... ON DUPLICATE KEY UPDATE ... (Variante von REPLACE):  
Immer Trigger BEFORE INSERT und anschließend entweder  
Trigger AFTER INSERT (falls Key NICHT vorhanden) oder  
Trigger BEFORE UPDATE + AFTER UPDATE (falls Key vorhanden).

| Aktion                                       | Ausgelöste Trigger                                             |                                   |
|----------------------------------------------|----------------------------------------------------------------|-----------------------------------|
|                                              | Key da                                                         | Key NICHT da                      |
| INSERT ...                                   | BEFORE INSERT                                                  | BEFORE INSERT<br>AFTER INSERT     |
| UPDATE ...                                   | BEFORE UPDATE<br>AFTER UPDATE                                  |                                   |
| DELETE ...                                   | BEFORE DELETE<br>AFTER DELETE                                  |                                   |
| REPLACE ...                                  | BEFORE INSERT<br>BEFORE DELETE<br>AFTER DELETE<br>AFTER INSERT | BEFORE INSERT<br><br>AFTER INSERT |
| INSERT ...<br>ON DUPLICATE<br>KEY UPDATE ... | BEFORE INSERT<br>BEFORE UPDATE<br>AFTER UPDATE                 | BEFORE INSERT<br><br>AFTER INSERT |

Definition eines Triggers:

```

DELIMITER //
CREATE
 [DEFINER {<User> | CURRENT USER}]
 TRIGGER <Trig> <Time> <Event>
ON <Tbl> FOR EACH ROW
BEGIN
 <Statement>;
 # oder nur <Statement>;
 ...
END //
DELIMITER ;

```

Mögliche Zeitpunkte <Time>:

| Time   | Bedeutung             |
|--------|-----------------------|
| BEFORE | Vor Tabellenänderung  |
| AFTER  | Nach Tabellenänderung |

Mögliche 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:

```
SELECT TRIGGER_NAME, EVENT_MANIPULATION, EVENT_OBJECT_TABLE, ACTION_STATEMENT
FROM INFORMATION_SCHEMA.TRIGGERS
WHERE TRIGGER_SCHEMA = <Db>;
SHOW TRIGGERS; # Default <Db>
SHOW TRIGGERS LIKE "a%"; # Default <Db>, Tabellenname
SHOW TRIGGERS FROM <Db> LIKE "a%"; # Spezelle <Db>, Tabellenname
```

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)
#-----
CREATE TABLE account (
 acct_num INT,
 amount DECIMAL(10,2)
);
SET @sum = 0;
CREATE TRIGGER t0 BEFORE INSERT ON account
FOR EACH ROW
 SET @sum = @sum + NEW.amount;
SHOW TRIGGERS LIKE "account"; # Tabellenname
INSERT INTO account VALUES
 (137, 14.98),
 (141, 1937.50),
 (97, -100.00);
SELECT @sum AS "Gesamt";
DROP TRIGGER t0;
DROP TABLE account; # Löscht auch Trigger

DELIMITER //

#-----
Trigger 1 (2 Spalten immer in GROSSschreibung umwandeln)
#-----
CREATE TRIGGER t1 BEFORE INSERT ON pers # für UPDATE gl. Trigger definieren
FOR EACH ROW
BEGIN
 SET NEW.vorname = UCASE(NEW.vorname);
 SET NEW.name = UCASE(NEW.name);
END //

#-----
Trigger 2 (Preis hat minimal Wert 0 und maximal Wert 100)
#-----
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)
#-----
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)
#-----
```

```

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)
#-----
CREATE TRIGGER t5 BEFORE DELETE ON pers
FOR EACH ROW
BEGIN
 SET @vorname = OLD.vorname; # Bei jeder Löschoperation überschrieben
 SET @name = OLD.name; # Nach Tabellenlöschen noch verfügbar
 SET @plz = OLD.plz;
 SET @ort = OLD.ort;
END //

#-----
Trigger 6 (Summe aller in eine Tabelle eingefügten Werte bilden)
#-----
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 //

Verwendung:
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)
#-----
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 //

DELIMITER ;

DROP TRIGGER t1;
DROP TRIGGER t2;
DROP TRIGGER t3;
DROP TRIGGER t4;
DROP TRIGGER t5;
DROP TRIGGER t6;
DROP TRIGGER t7;

```

### 23) 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 bestimmte SQL-Stati 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" dargestellt (wird mit dem externen Programm "perror" in eine entsprechende Textmeldung umgewandelt). Ihm entspricht ein MySQL-Fehlercode, aus dem er generiert wird. Die ersten beiden Zeichen des SQL-Status "XXXXX" legen die FEHLERKLASSE fest:

| Klasse | Bedeutung                 |
|--------|---------------------------|
| 00...  | Erfolg                    |
| 01...  | Warnung                   |
| 02...  | Nicht gefunden            |
| XX...  | Fehler (sonstige Präfixe) |

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 eine oder mehrere SQL-Stati. Trifft eine der Bedingungen zu (tritt einer der SQL-Stati 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>}
 <Statement>; BEGIN
 <Statement>;
 ...
 END;
```

```
<HandlerType> = CONTINUE # Umgebender Block (BEGIN ... END) ...
 EXIT # ... läuft nach Handler-Anweisung weiter
 UNDO # ... wird nach Handler-Anweisung verlassen (Std)
 # ... (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 und SQL-Stati (keine vollständige Liste, SQL-Stati sind durch ANSI SQL und ODBC standardisiert, MySQL-Fehlercodes sind nur für MySQL relevant):

| 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  |
| 1040 | "08004" | Too many connections                                 |
| 1047 | "08S01" | Unknown command                                      |
| 1022 | "23000" | Can't write; duplicate key in table ...              |
| 1046 | "3D000" | No database selected                                 |
| 1044 | "42000" | Access denied for user '...' to database '...'       |
| 1037 | "HY001" | Out of memory; restart server and try again          |
| 1038 | "HY001" | Out of sort memory; increase server sort buffer size |

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;
```

```

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;

```

#### 24) 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 Routinen, Triggern und Events erlaubt
- \* Asensitive (Kopie der Ergebnistabelle möglich aber nicht unbedingt nötig)
- \* Readonly (keine Updates möglich, MY!)
- \* Nonscrollable (nur eine Leserichtung, keine Datensätze überspringbar, MY!)
- \* Nonholdable (nach Commit geschlossen)
- \* Namenslos (Handler = Cursor-ID)
- \* Serverside (materialisierte temporäre Tabelle)

##### Syntax:

```

DECLARE <Cursor> CURSOR FOR <SelectStmt>; # Cursor mit SQL-Anw. verknüpfen
OPEN <Cursor>; # Abfrage durchführen (Puffer)
FETCH <Cursor> {INTO | USING} <Var>, ...; # Einen Ergebnissatz holen
CLOSE <Cursor>; # Pufferinhalt verwerfen

```

##### Hinweise:

- + Zielvariablen müssen zu Spalten passenden Datentyp haben
- + Prüfung per Handler über Statusvariable ob letzter Datensatz gelesen
- + Verknüpfung des Cursors bleibt bestehen (erneut öffnbar)
- + Deklarations-Reihenfolge: Erst Variablen, dann Cursors, dann Handler

HANDLER zur Ende-Erkennung notwendig (SQLSTATE 02000 bzw. NOT FOUND):

```

DECLARE done INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1; # A) Var. "done" setzen
DECLARE CONTINUE HANDLER FOR SQLSTATE "02000" BEGIN END; # B) Ignorieren

```

##### Beispiel:

```

CREATE TABLE res (
 nr INT NOT NULL,
 vorname VARCHAR(30),
 name VARCHAR(30)
);

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;

```

## 25) 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:

- \* Zugriff auf ALLE Spalten EINER Tabelle notwendig ("Full Table Scan")
- \* Schneller und flexibler als SELECT (solange nur Zugriff auf eine Tabelle)
  - + Weniger SQL-Kommando-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 Tabellename 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;
```

## 26) 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:

- \* Regelmäßige Protokollierung des DB-Zustandes
- \* Automatisch Backup oder Synchronisation auslösen
- \* Periodische Generierung von Tabellen mit Zwischenergebnissen (Top10)
  - > "Materialized" Views

Eigenschaften:

- \* Erst ab MySQL MY!5.1 verfügbar
- \* Name muss pro Datenbank eindeutig sein
- \* Einmalige (AT) oder wiederholt (EVERY) möglich (Schedule)
- \* Wiederholt ab (STARTS) und bis zu (ENDS) einem gewissen Zeitpunkt
- \* Evtl. autom. löschen, wenn komplett erledigt (ON COMPLETION NOT PRESERVE)
- \* Events haben die gleichen Einschränkungen wie Stored Procedures
  - + Kann keinen Trigger oder Stored Procedure oder Event erzeugen
- \* Kein SQL-Standard

Syntax:

```
CREATE
[DEFINER = {<User> | CURRENT_USER}]
EVENT [IF NOT EXISTS] <Event>
ON SCHEDULE <Schedule>
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT "<String>"]
DO <Statement>; # oder DO BEGIN <Statement>;... END;

<Schedule> = AT <Timestamp> [+ INTERVAL <N> <Interval>]
 | EVERY <Interval> [STARTS <Timestamp>] [ENDS <Timestamp>]

<Interval> = YEAR | QUARTER | MONTH | WEEK | DAY
 | HOUR | MINUTE | SECOND
 | YEAR_MONTH
 | DAY_HOUR | DAY_MINUTE | DAY_SECOND
```

```

| HOUR_MINUTE | HOUR_SECOND
| MINUTE_SECOND

```

```

<Timestamp> = CURRENT_TIMESTAMP
| "YYYY-MM-DD HH:MM:SS"

```

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                       |
| # | 5000 MICROSECOND              | m | Alle 5 Millisekunden                   |
|   | "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 02:10:30" DAY_MICROSECOND | m | 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          |
| # | "10:05:30" HOUR_MICROSECOND   | m | Alle 10 Std, 5 Min und 30 Sek          |
|   | "10:30" MINUTE_SECOND         | S | Alle 10 Min und 30 Sek                 |
| # | "10:30" MINUTE_MICROSECOND    | m | Alle 10 Min und 30 Sek                 |
| # | "10:30" SECOND_MICROSECOND    | m | Alle 10 Min und 30 Sek                 |

Beispiel (in einer Stunde UPDATEn):

```

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):

```

CREATE EVENT IF NOT EXISTS 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):

```

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):

```

DELIMITER //
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 Summe Anzahl Sitzungen sichern und Sitzungstabelle löschen):

```

DELIMITER //
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):

```

DELIMITER //
CREATE EVENT e_opt_tables
ON SCHEDULE
EVERY 1 WEEK
STARTS "2010-01-01 00:00:00"
ENDS "2012-12-31 00:00:00"
ON COMPLETION NOT PRESERVE
DO BEGIN
OPTIMIZE TABLE test.t1;
OPTIMIZE TABLE test.t2;
END //
DELIMITER ;

```

Zustand des Event-Schedulers anzeigen/ändern:

```

SET GLOBAL event_scheduler = 1; # 1=ON, 0=OFF, DISABLED
SELECT @@event_scheduler;
SHOW SCHEDULER STATUS;
SHOW VARIABLES LIKE "event_scheduler%";

```

Liste aller/einiger Ereignisse mit Eigenschaften anzeigen:

```

SHOW EVENTS;
SHOW EVENTS LIKE "optimize_%";

```

Definition eines Ereignisses anzeigen:

```

SHOW CREATE EVENT <Event>;

```

Ereignis 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 <Statement>]; # oder DO BEGIN <Statement>;... END;

```

Ereignis deaktivieren/aktivieren:

```

ALTER EVENT <Event> DISABLE;
ALTER EVENT <Event> ENABLE;

```

Ereignis umbenennen:

```

ALTER EVENT <Event> RENAME TO <NewEvent>;

```

Ereignis löschen:

```

DROP EVENT <Event>;
DROP EVENT IF EXISTS <Event>; # Kein Fehler falls nicht existent

```

Recht vergeben, Ereignisse zu erstellen, ändern und löschen:

```

GRANT EVENT ON <Db>.* TO <User>@<Host>;

```

Recht wegnehmen, Ereignisse zu erstellen, ändern und löschen:

```

REVOKE EVENT ON <Db>.* TO <User>@<Host>;

```

## 27) 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-Stati verwenden, keine MySQL-Fehlercodes
* Signal-Klassen:
"00..." = Erfolg # Beendet Programmfluss nicht
"01..." = Warnung # Beendet Programmfluss nicht
"02..." = Nicht gefunden # Beendet Programmfluss
"XX..." = Fehler (sonstige Präfixe) # Beendet Programmfluss
```

## Syntax:

```
SIGNAL <CondValue> # Zurückzugebener Fehlerwert
 [SET <SignalInfo> {, <SignalInfo>}]

<CondValue> = SQLSTATE [VALUE] "<XXXXX>" # SQL-Status "XXXXX"
 | <CondName> # Name einer vorher def. Condition
```

```
<SignalInfo> = <CondInfoItem> = <SimpleValueSpec>
```

```
<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;
```

## 28) Begrenzungen

Begrenzungen von MySQL (minimaler Wert, abhängig von der Engine auch mehr):

| Max.  | Bedeutung                                 |
|-------|-------------------------------------------|
| 65535 | Byte in einem Datensatz                   |
| 4096  | Spalten in einer Tabelle                  |
| 64    | Indices pro Tabelle                       |
| 16    | Spalten zu "Composite Index" kombinierbar |
| 1000  | Byte Index-Länge (InnoDB: 767)            |
| 61    | Tabellen in einer View                    |

|    |                        |
|----|------------------------|
| 61 | Tabellen in einem Join |
|----|------------------------|

Längenbegrenzung von Bezeichnern:

| Max | Bezeichner |
|-----|------------|
| 64  | Datenbank  |
| 64  | Tabelle    |
| 64  | Spalte     |
| 64  | Routine    |
| 255 | Alias      |
| 60  | Host       |
| 16  | Benutzer   |
| 32  | Passwort   |

Begrenzungen der InnoDB-Engine:

| Max. | Bedeutung                                          |
|------|----------------------------------------------------|
| 1000 | Spalten                                            |
| 8000 | Byte in einem Datensatz (etwa halbe Speicherseite) |
| 767  | Byte Index-Länge                                   |