SelfSQL: SQL − Tutorial für Anfänger

Klaus Becker, becker−k@web.de, http://www.becker−k.de/

SelfSQL: Table of Content

SelfSQL: Table of Content

SQL ?

Diese kurze Einführung in SQL (Structured Query Language) soll einige "Basics" dieser Sprache erklären damit der praktische Einstieg in SQL für Anfänger einfacher wird. Nicht mehr. Fortgeschrittene Anwender werden hier deshalb vieles vermissen.

Die Beispiele beziehen sich auf die für einige Betriebssysteme frei erhältliche Datenbank "MySQL" der Firma "MySQL AB". Aber auch ohne diese Datenbank ist man nicht aufgeschmissen. Die allermeisten Beispiele funktionieren auch mit anderen Datenbanken (z.B. Microsoft Access, MS−SQL Server).

Detailliertere Informationen zu MySQL sowie die jeweils aktuellen Installationsdateien für Windows, Linux und diverse UNIX−Derivate findet man unter http://www.mysql.com/downloads/. Bitte unbedingt die Lizenzbestimmungen beachten!

Wer weitergehende Fragen zu SQL bzw. zum Funktionsumfang von MySQL hat findet unter der Adresse http://www.mysql.com/documentation/ umfangreiche Dokumentationen.

Was ist SQL?
Der Begriff SQL steht für **S**tructured **Q**uery **L**anguage. Übersetzt bedeutet das soviel wie "Strukturierte Abfrage Sprache". SQL sollte − zumindest von der Idee her − die Abfrage von Datenbanken standardisieren. Allerdings gibt es bei vielen Datenbanken mehr oder weniger große Abweichungen. U.a. auch bei MySQL.

Man kann SQL in zwei Teilbereiche unterteilen:

`. **SQL** (Structured Query Language)
a. **DML** (Data Manipulation Language)

Die Unterscheidung spielt in der Praxis allerdings kaum eine Rolle.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr
 becker−k@web.de, www.becker−k.de/mysql/

Datenbanken?

Wie sieht eine Datenbank aus?
Fast alle gängigen Datenbank sind relationale Datenbanken. Objektorientierte Datenbanken spielen momentan noch keine große Rolle. Hier soll es nur um die relationalen Kollegen gehen.

Eine Datenbank besteht aus mindestens einer − meistens jedoch mehrerer mit einander in Beziehung stehenden Tabellen. In den Tabellen sind die Daten "organisiert". In den Spaltenköpfen der Tabellen stehen die Feldnamen und in den Zeilen, genauer gesagt in den einzelnen Zellen, die Werte. Jede Zeile enthält einen Datensatz.

Beispiel:

1	Herr	Ludwig	Laberbacke	Laberstrasse 11	27861	Lordhausen
2	Frau	Paula	Tratschig	Sonnenstraße 56	89894	Augustburg
3	Frau	Petra	Timmers	Feldweg 1	65323	Langenort
...	...	...	...	...	...	...

Jeder Wert (jedes Feld!) ist − ähnlich wie beim Schachbrett − eindeutig bestimmbar. So ist z.B. der Wert "Timmers" über die dritte Spalte im Feld "nachname" definiert. Aus der Definition "zweite Spalte − plz" ergibt sich der Wert "89894". Diese Eindeutigkeit ist sehr zentral für das Verständnis von SQL.

Formen des Datenhandlings
Beim Umgang mit Daten in einer Datenbank kennt SQL vier Grundspielarten:

- SELECT−Anweisung: Ausqahl von Daten aus einer Tabelle bzw. aus mehreren Tabellen
- INSERT−Anweisung: Schreiben von Daten in eine Tabelle
- DELETE−Anweisung: Löschen von Daten aus einer Tabelle
- UPDATE−Anweisung: Verändern von Daten in einer Tabelle

Für jede dieser vier Anweisungsarten gilt, daß der Name der von der Anweisung betroffenen Tabelle mit angegeben werden muß.

Hinweise zur Syntax
Um die folgenden Beispiele leichter lesbar und nachvollziehbar zu machen, habe ich mich an folgende Syntax gehalten: Alle "SQL−Begriffe" sind groß geschrieben; alle Begriffe die Tabellen entstammen (z.B. der Name der Tabelle, die Feldnamen) sind grundsätzlich klein geschrieben.

Beispiel:

```
SELECT vorname, nachname FROM adressen WHERE
nachname='Becker'
```

Hinweis: Es empfiehlt sich, beim Anlegen einer Tabelle in einer Datenbank alle Feldnamen klein zu schreiben. Umlaute in den Feldnamen sollten vermieden werden (lieber oe statt ö). Gleiches gilt für die Benennung der Tabellen selbst. So kann man vermeiden, daß man später ständig in der Tabelle nachsehen muß, wie sich der Feldname, bzw. der Tabellenname denn nochmal schreibt...

Letzte Änderung: 17. Mai 2002, 21:26 Uhr
 becker−k@web.de, www.becker−k.de/mysql/

Select − Statement

1. Alle Daten aus einer Tabelle holen

2. Eine bestimmte Spalte aus einer Tabelle holen

3. Mehrere bestimmte Spalten aus einer Tabelle holen

4. Spalten aus mehreren Tabellen holen

5. Mehrere Tabellen aus verschiedenen Datenbanken abfragen

6. Tabellen exportieren: SELECT * INTO OUTFILE

7. Einschränken von Ergebnismengen

Bedeutung:
Mit der SELECT−Anweisung lassen sich alle oder bestimmte Daten aus einer oder mehreren Tabellen holen.

Alle Daten aus einer Tabelle holen
Allgemeine Form:

```
SELECT * FROM tabelle
```

Im Beispiel werden alle Daten aus der Tabelle *adressen* geholt (alle Werte aus allen Spalten). Das Sternchen in derAnweisung fungiert als Platzhalter. Da alle Daten aus der Tabelle geholt werden ist das Ergebnis dieser Abfrage identisch mit der Tabelle *adressen*.

```
SELECT * FROM adressen
```

Eine bestimmte Spalte aus einer Tabelle holen
Häufig benötigt man aber garnicht alle Daten aus einer Tabelle sondern nur einige Spalten. In diesem Fall muß der Name des gewünschten Feldes (Spalte) explizit angegeben werden.

```
SELECT nachname FROM adressen
```

Als Ergebnis der obigen Abfrage werden alle Werte des Feldes *nachname* der Tabelle *adressen* ausgegeben. Kurz: Alle Nachnamen werden ausgegeben. Das Ergebnis könnte dann z.B. wie in

der folgenden Abbildung aussehen.

Beispiel:

nachname
Laberbacke
Tratschig
Timmers

Mehrere bestimmte Spalten aus einer Tabelle holen
Will man mehrere Felder gleichzeitig abfragen müssen die Feldnamen − durch Komma und Leerzeichen getrennt − hintereinander geschrieben werden.

```
SELECT anrede, vorname, nachname, strasse, ort FROM
adressen
```

Als Ergebnis der obigen Abfrage werden alle Werte der angegebenen Felder der Tabelle *adressen* ausgegeben. Folgendes Ergebnis wäre denkbar:

Beispiel:

anrede	vorname	nachname	strasse	ort
Herr	Ludwig	Laberbacke	Laberstrasse 11	Lordhausen
Frau	Paula	Tratschig	Sonnenstraße 56	Augustburg
Frau	Petra	Timmers	Feldweg 1	Langenort

Außer den Feldern *id* und *plz* sind alle Felder − wie in der Anweisung angegeben − in der Abfrage enthalten.

Spalten aus mehreren Tabellen holen
Will man mehrere Felder aus mehreren Tabellen in einem Statement abfragen müssen die Namen der Tabellen durch Komma und Leerzeichen getrennt hintereinander (hinter FROM) geschrieben werden.

Beispiel:

```
SELECT vorname, nachname, verein, pkw FROM adressen,
interessen
```

Kommt ein Feld in beiden Tabellen vor, so muss jedem Feld der Tabellenname vorangestellt werden da der Feldname ansonsten nicht eindeutig ist.

Beispiel:

```
SELECT adressen.vorname, nachname,
interessen.vorname, verein, pkw FROM adressen,
interessen
```

Es können auch mehr als zwei Tabellen in einer Abfrage angesprochen werden. Hier ist jedoch Vorsicht angesagt: Je größer die abgefragten Tabellen sind und je mehr Tabellen und Felder abgefragt werden desto größer die Ergebnismengen. Hier können schnell riesengroße Ergebnismengen entstehen.

Beispiel:

```
SELECT adressen.vorname, adressen.nachname,
interessen.vorname, interessen.nachname,
interessen.verein, beruf.gehalt
FROM adressen, interessen, beruf
```

Mehrere Tabellen aus verschiedenen Datenbanken abfragen
Um Tabellen aus verschiedenen Datenbanken auf dem selben MySQL−Server abzufragen muß den einzelnen Tabellen ein Präfix mit dem Namen der Datenbanken vorangestellt werden.

Allgemeine Form:

```
SELECT db1.tabelle3.feldname, db2.tabelle2.feldname
FROM db1.tabelle3, db2.tabelle2.feldname
```

Tabellen exportieren: SELECT * INTO OUTFILE
Mit MySQL ist es möglich, ganze Tabelleninhalte (oder auch Teile) in Textdateien zu exportieren. Und das in einer hohen Geschwindigkeit. Diese Dateien können dann als Sicherungsdatei dienen oder aber auch in Access, StarCalc oder ähnliches importiert werden.

Die grundsätzliche Syntax ist recht einfach:

```
SELECT * INTO OUTFILE 'C:/temp/data.csv' FROM tabelle
```

Es ist auch möglich, die Trennzeichen zwischen den Feldwerten und am Ende jedes Datensatzes zu bestimmen:

```
SELECT * INTO OUTFILE 'C:/temp/data.csv'
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n'
FROM tabelle
```

Das obige Beispiel würde eine CSV−Datei (Felder getrennt durch Semikolon, Zeilenende durch Zeilenumbruch) generieren, die später z.B. mit Excel weiter bearbeitet werden kann.

Hinweis: Der Name der Datei, in der das Ergebnis der Abfrage gespeichert werden soll, muß vor FROM stehen. Wenn die Datei nicht existiert versucht MySQL sie anzulegen.

Beim Exportieren von Daten können wie beim 'normalen' SELECT alle Optionen benutzt werden. Im folgenden Beispiel werden nur bestimmte Spalten aus zwei Tabellen exportiert.

```
SELECT tabelle1.feld1, tabelle1.feld2, tabelle2.feld1
INTO OUTFILE 'C:/temp/data.csv'
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n'
FROM tabelle
```

Einschränken von Ergebnismengen
Wenn man Tabellen abfragt will man im seltensten Fall den gesamten Tabelleninhalt anzeigen. In MySQL gibt es dazu (wie bei anderen Datenbanken auch) weitere Möglichkeiten, die u.a. in Kombination mit dem SELECT−Statement benutzt werden können:

- WHERE
- DISTINCT
- GROUP BY
- Vergleichsoperatoren wie '=', '!=', '>=', '<=', ''

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Insert − Statement

1. Einen Datensatz in eine Tabelle einspielen
2. Mehrere Datensätze auf einmal in eine Tabelle einspielen
3. Fehler bei doppelten Werten vermeiden (INSERT IGNORE INTO...)
4. Daten aus einer bereits bestehenden Tabelle in eine andere Tabelle übernehmen
5. Daten aus einer CSV−Datei in eine Tabelle übernehmen

Bedeutung:
Mit der INSERT−Anweisung lassen sich neue Datesätze in eine Tabelle einspielen.

Allgemeine Form:

a) Kurzform

```
INSERT INTO tabelle ('wert1', 'wert2', 'wert3',
'wert4')
```

b) Vollständige Form

```
INSERT INTO tabelle
(feld1, feld2, feld3, feld4)
VALUES
('wert1', 'wert2', 'wert3', 'wert4')
```

Bei der Kurzform muss die Anzahl der Werte genau der Anzahl der Felder in der Tabelle entsprechen. Die Werte werden in der Reihenfolge der in der Tabelle vorkommenden Felder in die Tabelle eingespielt. Um Fehler zu vermeiden sollte man hier immer auf die vollständige Form der INSERT−Anweisung zurückgreifen.

Einen Datensatz in eine Tabelle einspielen
Beispiel:
In folgende Tabellenstruktur soll ein Datensatzeingefügt werden. Das Feld `id` ist ein INTEGER−Feld, das eine eindeutige Nummer − die automatisch von MySQL vergeben wird (Autoincrement) − für jeden Datensatz enthält.

id	anrede	vorname	nachname	strasse	ort
1	Herr	Ludwig	Laberbacke	Laberstrasse 11	Lordhausen
2	Frau	Paula	Tratschig	Sonnenstraße 56	Augustburg
3	Frau	Petra	Timmers	Feldweg 1	Langenort

Die Verwendung der folgenden Kurzform wird hier fehlschlagen:

```
```

```
INSERT INTO adressen
('Herr', 'Klaus', 'Becker', 'Feldweg 1', 'Berlin')
```

Grund: Der Wert für das Feld `id` ist in der Werteliste nicht vorhanden. Die Anzahl der Werte stimmt nicht mit der Anzahl der Felder überein.

Aber auch wenn der Wert für das Feld `id` mit angegeben wird kann die INSERT−Anweisung fehlschlagen. Da man nicht immer sicher sein kann, welche Werte in der Spalte `id` bereits vergeben sind kann es passieren, daß man versucht einen in der Spalte bereits vergebenen Wert einzufügen. Dies führt zu einem Fehler und damit auch dazu, daß die Anweisung nicht ausgeführt wird.

Die vollständige Anweisung würde hier in jedem Fall eher zum Erfolg führen:

```
INSERT INTO adressen
('anrede', 'vorname', 'nachname', 'strasse', 'ort')
VALUES
('Herr', 'Klaus', 'Becker', 'Feldweg 1', 'Berlin')
```

Der Wert für das Feld `id` braucht hier nicht mit angegeben werden da klar ist welcher Wert in welches Feld eingespielt werden soll. MySQL würde automatisch einen DEFAULT−Wert für alle nicht im Statement genannten Felder mit einfügen, in diesem Fall also auch für das Feld `id`.

Mehrere Datensätze auf einmal in eine Tabelle einspielen
Hat man sehr viele Datensätze in eine Tabelle einzuspielen kann man sich einiges an Schreibarbeit sparen indem man einfach alle Wertelisten jeweils durch Komma getrennt hintereinander schreibt. Auch hier kann man entweder die Kurzfrm oder die vollständige Form verwenden.

a) Kurzform

```
INSERT INTO tabelle
('wert1', 'wert2', 'wert3', 'wert4'),
('wert1', 'wert2', 'wert3', 'wert4'),
('wert1', 'wert2', 'wert3', 'wert4')
```

b) Vollständige Form

```
INSERT INTO tabelle
(feld1, feld2, feld3, feld4)
VALUES
('wert1', 'wert2', 'wert3', 'wert4'),
('wert1', 'wert2', 'wert3', 'wert4'),
('wert1', 'wert2', 'wert3', 'wert4')
```

Hinweis:
"Multiple Inserts" werden wesentlich schneller abgearbeitet als einzelne Inserts (vorrausgesetzt die Anzahl der einzufügenden Datensätze ist gleich).

Fehler bei doppelten Werten vermeiden (INSERT IGNORE INTO...)
Wenn Daten in eine Tabelle mit eindeutigen Schlüsseln eingespielt werden kommt es zu Fehlern, wenn versucht wird einen bereits enthaltenen Wert erneut in die Tabelle bzw. Spalte einzufügen. Diese Fehler und damit den Abbruch der SQL−Anweisung kann man vemeiden wenn man im Statement IGNORE verwendet.

```
INSERT IGNORE INTO tabelle (...) VALUES (...)
```

Die einzufügende Zeile, die den Wert in einer Spalte verdoppeln würde, wird bei IGNORE verworfen. Dabei wird keine Fehlermeldung ausgegeben.

Daten aus einer bereits bestehenden Tabelle in eine andere Tabelle übernehmen
Mit dem folgenden Statement kann man MySQL anweisen, die Daten aus einer bereits bestehenden Tabelle in eine andere einzufügen. Voraussetzung dafür ist allerdings, daß die betroffenen Tabellen eine idetische Struktur besitzen.

```
INSERT INTO tabelleEins
SELECT * FROM tabelleZwei
```

Selbstverständlich kann man alle Möglichkeiten des SELECT−Statements wie z.B. die WHERE−Bedingung, die GROUP BY−Klausel oder auch ORDER BY hier auch nutzen:

```
INSERT INTO tabelleEins
SELECT * FROM tabelleZwei
WHERE feld1 = 'wert'
```

Daten aus einer CSV−Datei in eine Tabelle übernehmen
In MySQL gibt es die Möglichkeit, Daten aus einer CSV−Datei in eine bestehende Tabelle zu übernehmen. Jede Zeile in der CSV−Datei entspricht später einem Datensatz in der Tabelle.

```
LOAD DATA INFILE 'C:/temp/data.csv'
INTO TABLE tabelle
FIELDS TERMINATED BY ';'
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
```

Wichtig:
Die Datenstruktur in der Datei muss der der Tabelle entsprechen (Anzahl der Werte pro Zeile muss der Anzahl der Felder in der Tabelle entsprechen).

Als Feldtrenner und als Zeilenbegrenzer sollte man solche Zeichen verwenden, die mit Sicherheit nicht in den Daten der Datei vorkommen. Ist 'ENCLOSED BY' angegeben, so nimmt MySQL an, dass die Daten in der Datei jeweils in das angegebene Begrenzungszeichen eingeschlossen sind. Die Angabe ist optional.

Update − Statement

Bedeutung:
Mit der UPDATE−Anweisung lassen sich alle oder bestimmte Werte bestimmter Felder in einer Tabelle abändern.

Allgemeine Form:

```
UPDATE tabelle SET feld1 = 'wert1', feld2 = 'wert2'
```

Ganz wichtig (!) bei der Ausführung von UPDATE−Statements ist eine angehangene 'WHERE−Bedingung'. Gibt man diese nicht an so werden immer **alle** Werte in den angegebenen Feldern abgeändert. Durch ein unbedacht abgeschicktes UPDATE−Statement kann es passieren, daß alle Daten in der Tabelle unbrauchbar werden!

Allgemeine Form mit WHERE−Bedingung:

```
UPDATE tabelle SET feld1 = 'wert1', feld2 = 'wert2'
WHERE feld3 = 'XYZ'
```

Beispiel:

```
UPDATE adressen SET vorname = 'Paul', nachname =
'Timmers'
```

Im Beispiel werden **alle** Werte in den Feldern *vorname* und *nachname* in *Paul* bzw. *Timmers* abgeändert. Dies wird in den wenigsten Fällen gewünscht sein. Was hier fehlt ist eine entsprechende WHERE−Bedingung'. Die Tabelle *adressen* könnte in diesem Fall so aussehen:

anrede	vorname	nachname	strasse	ort
Herr	Paul	Timmers	Laberstrasse 11	Lordhausen
Frau	Paul	Timmers	Sonnenstraße 56	Augustburg
Frau	Paul	Timmers	Feldweg 1	Langenort

Um das Ändern aller Werte in den Feldern *vorname* und *nachname* zu verhindern könnte man zum Beispiel die Bedingung 'wo strasse gleich Sonnenstraße 56' hinzufügen:

```
UPDATE adressen SET vorname = 'Paul', nachname =
'Timmers'
WHERE strasse = 'Sonnenstraße 56'
```

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Delete − Statement

Bedeutung:
Mit der DELETE−Anweisung lassen sich alle oder bestimmte Werte in einer Tabelle löschen.

Allgemeine Form:

```
DELETE FROM tabelle
```

Ganz wichtig (!) bei der Ausführung von DELETE−Statements ist eine angehangene 'WHERE−Bedingung'. Gibt man diese nicht an so werden immer **alle** Werte in der Tabelle **gelöscht**. Die Tabelle ist dann komplett leer!

Allgemeine Form mit WHERE−Bedingung:

```
DELETE FROM tabelle WHERE feld1 = 'XYZ'
```

Beispiel:

```
DELETE FROM adressen
```

Im Beispiel werden **alle** Werte in der Tabelle *adressen* gelöscht. Dies wird in vielen Fällen nicht gewünscht sein. Was hier fehlt ist wieder eine entsprechende 'WHERE−Bedingung'.

Um z.B. in der untenstehenden Tabelle nur die weiblichen Personen zu löschen müßte das Statement um eine entsprechende 'WHERE−Bedingung' erweitert werden:

```
DELETE FROM adressen WHERE anrede = 'Frau'
```

anrede	vorname	nachname	strasse	ort
Herr	Ludwig	Laberbacke	Laberstrasse 11	Lordhausen
Frau	Paula	Tratschig	Sonnenstraße 56	Augustburg
Frau	Petra	Timmers	Feldweg 1	Langenort

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Where−Klausel

Bedeutung:
Die WHERE−Klausel wird in Verbindung mit dem UPDATE− und dem SELECT−Statement verwendet. Mit ihr lassen sich Bedingungen formulieren, mit denen die Anzahl der betroffenen Datensätze (Zeilen) eingeschränkt werden. Innerhalb der WHERE−Klausel lassen sich Boolsche−Operatoren wie 'AND' oder 'OR' verwenden. Mit dem Klammern von Bedingungen kann man die Ergebnismenge genauer spezifizieren.

Allgemeine Form:
SELECT:

```
SELECT * FROM tabelle WHERE feld1 = 'wert1'
```

UPDATE:

```
UPDATE tabelle SET feld1 = 'wert1' WHERE feld1 =
'wert1'
```

Um zwei oder mehrere Bedingungen zu formulieren kann man 'AND' und/oder 'OR' benutzen:

```
SELECT * FROM tabelle
WHERE feld1 = 'wert1'
AND
 (
 feld2 = 'wert2'
 OR
 feld3 = 'wert3'
 )
```

Neben dem oben mehrmals verwendeten Operator '=' gibt es noch weitere wichtige Operatoren:

- '!=' (ungleich) bzw. 'NOT'
- '>=' (größer gleich)
- '<=' (kleiner gleich)
- 'LIKE' ("ähnlich wie")
- 'IN' ("Liste von Werten")
- 'BETWEEN' ("zwischen zwei Werten")

Beispiele:
Alle Beispiele gehen von der folgenden Tabelle aus:

anrede	vorname	nachname	strasse	plz	ort
Herr	Ludwig	Laberbacke	Laberstrasse 11	11223	Lordhausen
Frau	Paula	Tratschig	Sonnenstraße 56	34567	Augustburg
Frau	Petra	Timmers	Sonnenstraße 56	23226	Langenort

Die Verwendung der WHERE−Klausel ist bei SELECT und UPDATE identisch. Die Beispiele beziehen sich daher nur auf das SELECT−Statement.

Tipp:
Möchte man per UPDATE−Statement eine Anzahl von Spalten in einer Tabelle ändern so kann man sich − wenn man das Statement zu einer SELECT−Abfrage umformuliert − erst einmal ansehen, welche Datensätze von dieser Ändrung betroffen sein würden. Vorraussetzung dafür ist allerdings, das die Bedingungen in der WHERE−Klausel identisch sind.

a) Selektierung von Datensätzen über '=':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE anrede = 'Frau'
```

Die Ergebnismenge enthält genau zwei Datensätze: Paula und Petra jeweils mit den entsprechenden Nachnamen und den Orten. Das gleiche Resultat lässt sich über folgenden zwei Abfragen realisieren:

b) Selektierung von Datensätzen über '!=':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE anrede != 'Herr'
```

c) Selektierung von Datensätzen über 'NOT':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE NOT anrede = 'Herr'
```

Die Abfrage b) ist im Grunde identisch mit der Abfrage c). Nur die Schreibweise ist verkürzt. Wichtig bei der Verwendung von 'NOT' ist, daß 'NOT' vor dem Feldnamen steht, für den die Bedingung gelten soll.

d) Selektierung von Datensätzen über den Operator 'LIKE':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE ort LIKE '%g%'
```

Der Operator 'LIKE' unterscheidet sich insofern von den bisher dargestellten als dass er 'Wildcards' zuläßt. So werden mit dem obigen Statement z.B. alle Datensätze ausgegeben, die irgendwo(!) im Ortsnamen ein 'g' enthalten. Also Augustburg und Langenort inklusive der zum Datensatz gehörenden Vor− und Nachnamen.

Wichtig:
Bei der Verwendung von 'LIKE' ist die Angabe der 'Wildcards'. In diesem Fall das Prozentzeichen vor und nach dem gesuchten Wert (g). Läßt man diese Prozentzeichen weg, so verhält sich die Suche mit 'LIKE' genauso wie die mit '='. Die Ergebnismenge wäre dieselbe. Aus der obigen Tabelle würde kein Datensatz ausgewählt da kein Ort g in der Tabelle vorhanden ist.

Bedeutung der Wildcards:

- **%** (Prozent) steht für beliebig viele Zeichen
- _ (Unterstrich) steht für genau ein Zeichen

Beispiel für die Wildcard '?':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE vorname LIKE 'P____'
```

Die Ergebnismenge wäre wiederum identisch mit der aus den Abfragen a), b) und c).

f) Selektierung von Datensätzen über 'IN':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE vorname IN ('Paula', 'Petra')
```

Bei der Verwendung von 'IN' gibt man die zugelassenen Werte als kommaseparierte Liste innerhalb einer Klammer an. Die Ergebnismenge dieser Abfrage wäre wiederum identisch mit der aus den Abfragen a), b), c) und dem letzten Beispiel in d).

g) Selektierung von Datensätzen über 'BETWEEN':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE plz BETWEEN '20000' AND 40000
```

Die Ergebnismenge dieser Abfrage wäre wiederum identisch mit der aus den Abfragen a), b), c), f) und dem letzten Beispiel in d).

h) Selektierung von Datensätzen über '>=' bzw. '<=':

```
SELECT vorname, nachname, ort
FROM adressen
WHERE plz >= '20000'
AND plz <= '40000'
```

Die Ergebnismenge dieser Abfrage wäre wiederum identisch mit der aus den Abfragen a), b), c), f), g) und dem letzten Beispiel in d).

Eine kleine Änderung am Statement hätte eine andere Ergebnismenge zur Folge:

```
SELECT vorname, nachname, ort
FROM adressen
WHERE plz >= '20000'
OR plz <= '40000'
```

Wenn man − wie hier geschehen − 'AND' durch 'OR' ersetzt so würden alle Datensätze aus der Tabelle zurückgegeben da jeder der drei Datensätze mindestens eine der beiden Bedingungen erfüllt.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr
 becker−k@web.de, www.becker−k.de/mysql/

Order By

Bedeutung:
Mit der ORDER BY−Anweisung lassen sich die Werte in der Ergebnismenge auf− oder absteigend sortieren.

Allgemeine Form:

```
SELECT * FROM tabelle ORDER BY feld1
```

Wichtig:
ORDER BY kann nur dann ausgeführt werden, wenn die gesamte Ergebnismenge bereits feststeht. Alle WHERE−Bedingungen, GROUP BY− oder HAVING−Anweisungen müssen vorher ausgeführt worden sein!

Falsch:

```
SELECT * FROM adressen
ORDER BY nachname
WHERE vorname = 'Paula'
```

Richtig:

```
SELECT * FROM adressen
WHERE vorname = 'Paula'
ORDER BY nachname
```

Die Sortierung kann auf− oder absteigend erfolgen (ASC, DESC). Wenn weder *ASC* noch *DESC* angegeben ist verwendet MySQL *ASC* (aufsteigend). Die ORDER BY−Anweisung funktioniert sowohl bei numerischen Werten, Datumsangaben als auch bei Zeichenketten. Es können auch mehrere Sortierungen in einem Statement gleichzeitig vorgenommen werden.

Beispiel:

```
SELECT * FROM adressen
WHERE geburtsdatum > '1980−01−01'
ORDER BY geburtsdatum ASC, nachname DESC, vorname ASC
```

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Group By

Bedeutung:
Mit der GROUP BY−Anweisung lassen sich Datensätze in der Ergebnismenge gruppieren.

Allgemeine Form:

```
SELECT * FROM tabelle GROUP BY feld1
```

Beispiel:
Aus der untenstehenden Tabelle sollen alle **unterschiedlichen** Straßennamen selektiert werden

anrede	vorname	nachname	strasse	plz	ort
Herr	Ludwig	Laberbacke	Laberstrasse 11	11223	Lordhausen
Frau	Paula	Tratschig	Sonnenstraße 56	34567	Augustburg
Frau	Petra	Timmers	Sonnenstraße 56	23226	Langenort

Das Statement

```
SELECT strasse FROM adressen
```

würde sich nicht eignen da es alle drei Strassennamen als Ergebnis zurückliefern würde. U.a. wäre zweimal der Strassenname 'Sonnenstraße 56' im Ergebnis enthalten:

strasse
Laberstrasse 11
Sonnenstraße 56
Sonnenstraße 56

Erst die Verwendung von 'GROUP BY' würde hier zum gewünschten Ergebnis führen:

```
SELECT strasse FROM adressen GROUP BY strasse
```

Das Ergebnis dieser Abfrage würde wie folgt aussehen:

strasse
Laberstrasse 11
Sonnenstraße 56

Die Verwendung von 'GROUP BY' reduziert das Vorkommen von mehrfach vorhandenen Strassennamen auf ein einziges.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Create Table − Statement

1. Eine neue Tabelle per SQL−Statement erzeugen
2. Eine neue Tabelle aus bereits bestehenden Tabellen erzeugen
3. Beispiele zum Erstellen von Tabellen

Bedeutung:
Mit der CREATE TABLE−Anweisung lassen sich Tabellen **in einer bereits existierenden(!)** Datenbank anlegen.

Hinweis:
Wegen der Fülle von Optionen bei diesem Statement werde ich hier nur die grundlegensten (!) Möglichkeiten aufzeigen. Wichtig zum Verständnis von CREATE TABLE sind vor allem die Datentypen/Feldtypen von MySQL.

Eine neue Tabelle per SQL−Statement erzeugen

Allgemeine Form:

```
CREATE TABLE IF NOT EXISTS tabellenname
(
 feld1 int(10) NOT NULL auto_increment,
 feld2 char(2) NOT NULL DEFAULT 'AB' ,
 feld3 varchar(255) NOT NULL DEFAULT '' ,
 feld4 text NOT NULL DEFAULT '' ,
 PRIMARY KEY (feld1)
)
```

Der optionale Zusatz *IF NOT EXISTS* verhindert die Ausgabe einer Fehlermeldung falls die Tabelle *tabellenname* bereits besteht. Am Schluß des obigen Satements wird noch ein Primärschlüssel/Index angelegt. Alle Feldnamen sowie der jeweilige Feldtyp stehen zwischen den beiden Klammern. Hinter jedem Feldnamen steht die genauere Spezifizierung des Feldtyps.

Hinter dem jeweiligen Feldtyp ist bei drei Feldern noch ein DEFAULT−Wert angegeben. Werden später Werte in die Tabelle eingefügt so brauchen diese Felder beim INSERT−Statement nicht mit angegeben werden − sofern der einzutragende Wert dem DEFAULT−Wert des Feldes entspricht.

Das Feld *feld1* ist als *auto_increment* spezifiziert. Dieses Feld wird bei späteren INSERT−Statements automatisch mit einem (eindeutigen) Zahlenwert von 1 bis X gefüllt. Dieses Feld braucht bei INSERT−Statements deshalb nicht mit angegeben werden.

Hinweis:
Ein AUTO_INCREMENT Feld sollte in jeder Tabelle vorhanden sein. In vielen Fällen kann man bei einem späteren UPDATE−Statement nur so jeden Datensatz wirklich eindeutig adressieren.

Eine neue Tabelle aus einer bereits bestehenden Tabelle erzeugen
Es ist auch möglich, aus einer bereits bestehenden Tabelle eine neue Tabelle zu erzeugen. Die

Daten der bereits bestehenden Tabelle können wahlweise mit übernommen werden.

Allgemeine Form:
Ohne Datenübernahme

```
CREATE TABLE IF NOT EXISTS tabelleNeu
SELECT * FROM tabelleAlt
WHERE 0 = 1
```

Da die WHERE−Bedingung niemals zutrifft wird auch kein Datensatz aus der bestehenden Tabelle ausgewählt. Deshalb wird nur die Struktur (alle Felder!) der alten Tabelle in die neue übernommen.

Mit Datenübernahme

```
CREATE TABLE IF NOT EXISTS tabelleNeu
SELECT * FROM tabelleAlt
```

Hier werden sowohl die Struktur (alle Felder!) als auch die Daten der alten Tabelle übernommen. INDIZES werden nicht übernommen!

Möchte man nur bestimmte Felder aus der bereits bestehenden Tabelle bzw. aus den bereits bestehenden Tabellen übernehmen so müssen diese wie in einem normalen SELECT−Statement mit angegeben werden.

Beispiel:

```
CREATE TABLE IF NOT EXISTS tabelleNeu
SELECT
tabelleAltEins.feld1,
tabelleAltEins.feld2,
tabelleAltZwei.feld1,
tabelleAltZwei.feld4,
tabelleAltZwei.feld5
FROM tabelleAltEins, tabelleAltZwei
WHERE tabelleAltZwei.feld4 = 'wert'
```

Im obigen Beispiel werden nur die fünf angegebenen Felder in der neuen Tabelle angelegt. Es werden nur die Daten aus den beiden bestehenden Tabellen übernommen, die die WHERE−Bedingung erfüllen.

Beispiele zum Erstellen von Tabellen
Um den Einstieg in das Erstellen von Tabellen zu erleichtern sind unten beispielhaft drei CREATE−Statements wiedergegeben. Ich habe die Statements bewußt nach Numerischen−, String− und Datumstypen getrennt. Natürlich kann man die Feldtypen auch zusammen verwenden.

Teilweise sind optionale Angaben zu den jeweiligen Feldtypen gemacht worden. Jedes Statement enthält einen PRIMARY KEY vom Typ *tinyint(3)*. Dieses Feld ist jeweils zusätzlich als AUTOINCREMENT deklariert, wird also beim Füllen der Tabelle automatisch gefüllt. Neben dem PRIMARY KEY enthalten die beiden ersten Statements noch weitere INDIZES. Der eine INDEX davon ist als UNIQUE deklariert.

Hinter dem jeweiligen Feldnamen folgt der Feldtyp. Danach folgt entweder die maximale Länge,

das Format oder die Liste der aufnehmbaren Werte. In vielen Fällen ist danach der DEFAULT−Wert angegeben. Am Schluß fast jeder Feld−Definition ist nochmal angegeben, ob das Feld den Wert 'NULL' (nicht zu verwechseln mit mit der Zahl '0', NULL bedeutet hier soviel wie 'Nichts') haben kann. Soweit möglich sollte man aus Performancegründen alle Felder als NOT NULL deklarieren.

a) Erstellen einer Tabelle mit numerischen Feldtypen:

```
CREATE TABLE IF NOT EXISTS numerischeTabelle
 (
  feld1 tinyint(4) NOT NULL auto_increment,
  feld2 mediumint(9) DEFAULT '0' NOT NULL,
  feld3 int(11) unsigned DEFAULT '0' NOT NULL,
  feld4 bigint(20) unsigned zerofill,
  feld5 float DEFAULT '0' NOT NULL,
  feld6 double(4,2) DEFAULT '0.00' NOT NULL,
  PRIMARY KEY (feld1),
  UNIQUE feld5 (feld5),
  KEY feld2 (feld2)
 )
```

b) Erstellen einer Tabelle mit String−Feldtypen:

```
CREATE TABLE IF NOT EXISTS stringTabelle
 (
  feld1 tinyint(3) unsigned NOT NULL auto_increment,
  feld2 varchar(32) DEFAULT 'ABC',
  feld3 tinytext,
  feld4 mediumtext,
  feld5 text,
  feld6 longtext,
  feld7 tinyblob,
  feld8 mediumblob,
  feld9 blob,
  feld10 enum('A','B','C','D','E') DEFAULT 'A' NOT
NULL,
  feld11 set('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p'),
  PRIMARY KEY (feld1),
  KEY feld10 (feld10)
 )
```

c) Erstellen einer Tabelle mit Datums−Feldtypen:

```
CREATE TABLE IF NOT EXISTS datumTabelle
 (
  feld1 tinyint(3) NOT NULL auto_increment,
  feld2 datetime DEFAULT '0000−00−00 00:00:00' NOT
NULL,
  feld3 timestamp(14),
  feld4 time DEFAULT '00:00:00' NOT NULL,
```

```
 feld5 year(4) DEFAULT '0000' NOT NULL,
 PRIMARY KEY (feld1)
)
```

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Alter Table − Statement

1. Tabellentyp umwandeln
2. Tabelle umbenennen
3. Felder in einer Tabelle umbenennen
4. Feldtypen ändern
5. Felder hinzufügen
6. Felder löschen
7. Indizes hinzufügen
8. Indizes löschen

Bedeutung:
Mit der Anweisung ALTER TABLE lassen sich Änderungen an einer Tabelle vornehmen: Tabelle umbenennen, Tabellentyp ändern, Felder umbenennen, Feldtypen ändern, Felder hinzufügen und löschen, Indizes hinzufügen und löschen sowie Primary Keys hinzufügen und löschen.

Tabellentyp umwandeln
In MySQL gibt es verschiedene Tabellentypen. Momentan ist der Standardtyp MYISAM. Frühere MySQL−Versionen verwendeten ISAM als Standard. Daneben gibt es noch die Typen 'TEMPORARY' und 'HEAP' sowie einige andere. Legt man per CREATE TABLE eine neue Tabelle an so ist diese − sofern nichts anderes angegeben wurde − vom Typ MYISAM. Um den Tabellentyp in einen anderen umzuwandeln verwendet man ALTER TABLE.

Allgemeine Form:

```
ALTER TABLE tabelle TYPE = tabellentyp
```

Beispiel:

```
ALTER TABLE adressen TYPE = HEAP
```

Tabelle umbenennen
Mit ALTER TABLE kann man − wie mit der Anweisung RENAME TABLE auch − Tabellen umbenennen.

Allgemeine Form:

```
ALTER TABLE tabelleAlt RENAME AS tabelleNeu
```

Beispiel:

```
ALTER TABLE adressen RENAME AS adressbuch
```

Felder in einer Tabelle umbenennen
Allgemeine Form:

```
ALTER TABLE tabelle CHANGE feldnameAlt feldnameNeu
TYPANGABE
```

Beispiel:

```
ALTER TABLE adressen CHANGE ort stadt VARCHAR(32)
DEFAULT ""
```

Die Angabe des DEFAULT−Wertes ist optional. Die Angabe des Feldtyps dagegen nicht.

Feldtypen ändern

Allgemeine Form:

```
ALTER TABLE tabelle MODIFY feldname TYPANGABE
```

Beispiel:

```
ALTER TABLE adressen MODIFY ort VARCHAR(64) DEFAULT
"Berlin"
```

Die Angabe des DEFAULT−Wertes ist auch hier wieder optional. Der Feldname braucht hier − im Gegensatz zum Umbenennen von Feldern − nur einmal angegeben werden. Das Schlüsselwort MODIFY weist schon darauf hin, dass nur der Typ geändert werden soll.

Man kann auch mehrere Felder gleichzeitig ändern. Dazu listet man einfach alle zu ändernden Felder mit Komma getrennt an das Statement an:

```
ALTER TABLE adressen
MODIFY ort VARCHAR(64) DEFAULT "Berlin",
strasse VARCHAR(100),
plz VARCHAR(5),
vorname VARCHAR(32) DEFAULT "Klaus"
```

Eine Änderung des Feldtyps läßt sich auch mit dem folgenden Statement erreichen:

```
ALTER TABLE adressen CHANGE ort ort VARCHAR(32)
DEFAULT ""
```

Angenommen der Typ des Feldes "ort" war vor der Ausführung des Statements VARCHAR(16) so ist er nach der Ausführung vom Typ VARCHAR(32). Man beachte das doppelte Vorkommen des Feldnamens! (falls der nicht geändert werden soll).

Felder hinzufügen

Allgemeine Form:

```
ALTER TABLE tabelle ADD feld4 TYPANGABE
```

Beispiel:

```
ALTER TABLE adressen ADD telefon INT(8) UNSIGNED
DEFAULT "1"
```

Es können auch mehrere Felder gleichzeitig hinfugefügt werden. Die Felder sowie deren Typen müssen auch hier wieder durch Kommas getrennt hintereinander angegeben werden. Vor jedem Feldnamen muss das Schlüsselwort ADD stehen:

```
ALTER TABLE adressen
ADD telefon INT(8) UNSIGNED DEFAULT "1",
ADD fax INT(8) UNSIGNED DEFAULT "1"
```

Felder löschen
Allgemeine Form:

```
ALTER TABLE tabelle DROP feldname
```

Beispiel:

```
ALTER TABLE adressen DROP strasse
```

Will man mehrere Felder gleichzeitig löschen kommt wieder die altbekannte kommaseparierte Liste zum Zuge. Diesmal mit dem Schlüsselwort DROP vor jedem Feldnamen:

```
ALTER TABLE lexikonklaus DROP telefon, DROP fax
```

Anlegen von Indizes
In MySQL gibt es verschiedene Indizies: normaler INDEX, UNIQUE Index, PRIMARY KEY und − neuerdings − FULLTEXT. Alle drei Formen lassen sich mit ALTER TABLE anlegen und löschen.

Allgemeine Form

```
ALTER TABLE tabelle
ADD INDEXTYP indexName (feld1, feld2, feld3)
```

Hinter der Angabe des Indextyps folgt der Name des Index sowie die kommaseparierte Liste der Feldnamen, die indiziert werden sollen. Die Liste muss in Klammern gesetzt werden. Die Angabe des Namens ist optional. Wird kein Name angegeben so verwendet MySQL den ersten in der Liste angegebenen Feldnamen.

Beispiele:

```
ALTER TABLE adressen
ADD INDEX adressenindex (vorname, nachname, strasse)
```

```
ALTER TABLE adressen
ADD FULLTEXT volltext (ueberschrift, abstract,
textmenge)
```

```
ALTER TABLE adressen
ADD UNIQUE ort (ort)
```

```
ALTER TABLE adressen
ADD PRIMARY KEY primaerSchluessel (vorname, nachname)
```

Indizes löschen
Das Löschen von Indizies ist noch sehr viel einfacher als das Anlegen:

Allgemeine Form

```
ALTER TABLE tabelle DROP indexName
```

Beispiel:

```
ALTER TABLE adressen DROP adressenindex
```

Man kann auch gleichzeitig einen Index löschen und wieder anlegen:

```
ALTER TABLE adressen
DROP adressenindex,
ADD INDEX adressenindex (vorname, nachname, strasse)
```

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Rename Table − Statement

Bedeutung:
Mit der RENAME TABLE−Anweisung lassen sich Tabellen umbenennen.

Allgemeine Form:

```
RENAME TABLE tabellennameJetzt TO tabellennameNeu
```

Das obige Statement benennt die Tabelle *tabellennameJetzt* in *tabellennameNeu* um.

Möchte man mehrere Tabellen gleichzeitig umbenennen, so kann man die Tabellennamen mit Komma getrennt hintereinander angeben:

```
RENAME TABLE
tabellennameEins TO tabellennameOne,
tabellennameZwei TO tabellennameTwo,
tabellennameDrei TO tabellennameThree
```

Hinter dem letzten angegebenen Tabellennamen steht kein Komma!

Mit dem RENAME−Statement ist es auch möglich, eine Tabelle von einer Datenbank in eine andere zu **verschieben**:

```
RENAME TABLE
datenbankEins.tabellennameEins
TO
datenbanknameZwei.tabellennameZwei
```

Die Tabelle *tabellennameEins* ist nach der Ausführung dieses Statements in der Datenbank *datenbankEins* nicht mehr vorhanden!

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Drop Table − Statement

Bedeutung:
Mit der DROP TABLE−Anweisung lassen sich Tabellen in einer Datenbank löschen.

Allgemeine Form:

```
DROP TABLE tabellenname
```

Das obige Statement löscht die Tabelle *tabellenname* in der Datenbank, in der man sich gerade befindet. Ist die Tabelle nicht vorhanden wird eine entsprechende Fehlermeldung ausgegeben. Vermeiden kann man diese evtl. Fehlermeldung indem man das Statement ein wenig erweitert:

```
DROP TABLE IF EXISTS tabellenname
```

Das Statement löscht alle Dateien, die die angegebene Tabelle repräsentieren. Die Daten sind danach weg!

Sollen mehrere Tabellen in der Datenbank gelöscht werden so kann man die Tabellennamen durch Komma getrennt angeben. So kann man sich einige Statements ersparen:

```
DROP TABLE IF EXISTS tabelle1, tabelle1, tabelle3
```

Hinter dem letzten angegebenen Tabellennamen steht kein Komma!

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Show Tables − Statement

Bedeutung:
Mit der Show Tables−Anweisung lassen sich alle Tabellen aus einer Datenbank anzeigen für die man die entsprechende Berechtigung hat. Siehe dazu auch SHOW DATABASES und DESCRIBE TABLE.

Allgemeine Form:

```
SHOW TABLES
```

Die untenstehende Tabelle zeigt eine beispielhafte Ausgabe der obigen Anweisung.

tables_in_homepage
bestellt
statistik
suche
produkte
warenkorb
zugang

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Describe Table − Statement

Bedeutung:
Mit der Describe−Anweisung lassen sich Informationen über eine Tabelle anzeigen. Siehe dazu auch SHOW TABLES.

Allgemeine Form:

```
DESCRIBE tabellenname
```

Beispiel:

```
DESCRIBE user
```

Das Beispiel zeigt die Informationen über die Tabelle *'user'* aus der Datenbank *'mysql'* in der untenstehenden Form an. Neben dem Feldnamen werden der Feldtyp, ob das Feld NULL sein darf, etwaige Schlüssel inklusive Schlüsseltyp, der Default−Wert sowie die Extras (z.B. 'autoincrement') angezeigt.

Field	Type	Null	Key	Default	Extra
Host	char(60) binary		PRI		
User	char(16) binary		PRI		
Password	char(16) binary			N	
Select_priv	enum('N','Y')			N	
Insert_priv	enum('N','Y')			N	
Update_priv	enum('N','Y')			N	
Delete_priv	enum('N','Y')			N	
Create_priv	enum('N','Y')			N	
Drop_priv	enum('N','Y')			N	
Reload_priv	enum('N','Y')			N	
Shutdown_priv	enum('N','Y')			N	
Process_priv	enum('N','Y')			N	
File_priv	enum('N','Y')			N	
Grant_priv	enum('N','Y')			N	
Grant_priv	enum('N','Y')			N	
References_priv	enum('N','Y')			N	
Index_priv	enum('N','Y')			N	
Alter_priv	enum('N','Y')			N	

Optimize Table − Statement

Bedeutung:
Mit der OPTIMIZE TABLE−Anweisung lassen sich Tabellen in einer Datenbank optimieren. OPTIMIZE TABLE sollte vor allem dann angewendet werden, wenn viele Datensätze aus einer Tabelle gelöscht oder geändert wurden und wenn die Tabelle Felder mit variabler Länge enthält (Varchar, Text, Blob). MySQL kann den Platz in der Datendatei dann besser nutzen.

Allgemeine Form:

```
OPTIMIZE TABLE tabellenname
```

Man kann auch mehrere Tabellen in einem Statement angeben:

```
OPTIMIZE TABLE tabelleEins, tabelleZwei, tabelleDrei
```

Hinweis:
Bei sehr großen Tabellen von mehr als einigen hundert Megabyte (vor allem wenn diese zudem noch Indizes enthalten) kann die Ausführung OPTIMIZE TABLE schon mal ein paar Stunden dauern.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Repair Table − Statement

Bedeutung:
Mit der REPAIR TABLE−Anweisung lassen sich zerstörte Tabellen häufig wieder reparieren. Dies gilt jedoch nur für den Tabellentyp 'MyISAM' (Standard).

Allgemeine Form:

```
REPAIR TABLE tabellenname
```

Hinweis:
MySQL gibt nach der Auführung eine Tabelle mit vielen Informationen über die gefundenen Fehler zurück. Die letzte Spalte sollte die Zeichenkette 'Msg_type status' enthalten und ein 'ok' dahinter stehen haben.

Auch wenn die Tabelle repariert wurde heißt das nicht, das noch alle Datensätze in der Tabelle sind die auch vorher drin waren!

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Backup Table − Statement

Bedeutung:
Mit der Backup Table−Anweisung lassen sich Kopien von Tabellen−Dateien anlegen. Sie auch RESTORE TABLE.

Allgemeine Form:

```
BACKUP TABLE tabellenname
TO 'c:/pfad/zum/backup/verzeichnis'
```

Der Befehl BACKUP TABLE legt eine Kopie von zwei Dateien pro Tabelle im angegeben Verzeichnis an. Existieren die Dateien bereits gibt MySQL eine Fehlermeldung aus. Die Dateien werden nicht überschrieben. Man kann auch mehrere Tabellen in einem Statement angeben. Diese werden − jeweils durch Komma getrennt − nacheinander angegeben:

```
BACKUP TABLE tabelleEins, tabelleZwei, tabelleDrei
TO 'c:/pfad/zum/backup/verzeichnis'
```

Hinweis:
Der Befehl funktioniert nur mit Tabellen vom Typ MyISAM.

MySQL kopiert beim Ausführen von BACKUP TABLE nur die Tabellendefinition (Datei−Endung: .frm) und die Datendatei (Datei−Endung: .MYD). Evtl. bestehende INDIZES kann es aus den beiden Dateien wieder erzeugen.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Restore Table − Statement

Bedeutung:
Mit der Restore Table−Anweisung lassen sich Tabellen aus zuvor gemachten Kopien wieder anlegen. Sie dazu auch BACKUP TABLE.

Allgemeine Form:

```
RESTORE TABLE tabellenname
FROM 'c:/pfad/zum/backup/verzeichnis'
```

Der Befehl RESTORE TABLE legt die im Statement angegebene(n) Tabelle(n), die in den Dateien im angegebenen Backup−Verzeichnis liegen in der jeweils aktuellen Datenbank an − sofern diese Tabellen noch nicht bestehen. Falls die Tabellen bereits bestehen wird eine Fehlermeldung ausgegeben.

Man kann auch mehrere Tabellen in einem Statement angeben. Diese werden − jeweils durch Komma getrennt − nacheinander angegeben:

```
RESTORE TABLE tabelleEins, tabelleZwei
FROM 'c:/pfad/zum/backup/verzeichnis'
```

Hinweis:
Der Befehl funktioniert nur mit Tabellen vom Typ MyISAM.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Use − Statement

Bedeutung:
Mit der USE−Anweisung kann man die für die nächsten Statements von MySQL zu verwendene Datenbank auf dem MySQL−Server wechseln.

Allgemeine Form:

```
USE datenbankname
```

Das Statement *USE* sollte man immer dann verwenden, wenn man eine ganze Reihe von Abfragen an die jeweilige Datenbank stellen möchte. Bei einem oder wenigen Queries kann man sich auch behelfen in dem man den Namen der zur verwendenen Datenbank im Statement mit angibt:

```
SELECT feld1, feld2 FROM datenbankname.tabellename
```

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Create Database − Statement

Bedeutung:
Mit der CREATE DATABASE−Anweisung lassen sich neue Datenbanken auf dem MySQL−Server anlegen.

Allgemeine Form:

```
CREATE DATABASE datenbankname
```

Das obige Statement legt die Datenbank *datenbankname* an. Ist diese Datenbank auf dem MySQL−Server bereits vorhanden wird eine entsprechende Fehlermeldung ausgegeben. Vermeiden kann man diese evtl. Fehlermeldung indem man das Statement ein wenig erweitert:

```
CREATE DATABASE IF NOT EXISTS datenbankname
```

Der Zusatz *IF NOT EXISTS* kann vor allem für dynamisch generierte Datenbanken interessant sein.

Hinweis zum Namen der anzulegenden Datenbank:
Der Slash (*/*) und der Punkt (.) sind im Datenbanknamen nicht erlaubt. Desweiteren sollte man alle Sonderzeichen und Umlaute ebenfalls vermeiden.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Drop Database − Statement

Bedeutung:
Mit der DROP DATABASE−Anweisung lassen sich Datenbanken auf dem MySQL−Server löschen.

Allgemeine Form:

```
DROP DATABASE datenbankname
```

Das obige Statement entfernt die Datenbank *datenbankname* auf dem MySQL−Server. Ist diese Datenbank auf dem MySQL−Server nicht vorhanden wird eine entsprechende Fehlermeldung ausgegeben. Vermeiden kann man diese evtl. Fehlermeldung indem man das Statement ein wenig erweitert:

```
DROP DATABASE IF NOT EXISTS datenbankname
```

Das Statement löscht alle Dateien im entsprechenden Daten−Verzeichnis des MySQL−Servers mit den folgenden Endungen:

- .BAK
- .DAT
- .HSH
- .ISD
- .ISM
- .MRG
- .MYD
- .MYI
- .db
- .frm

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Show Databases − Statement

Bedeutung:
Mit der SHOW DATABASES−Anweisung lassen sich die vorhandenen Datenbanken auf dem MySQL−Server anzeigen. Siehe dazu auch SHOW TABLES und DESCRIBE TABLE.

Allgemeine Form:

```
SHOW DATABASES
```

Das obige Statement zeigt die vorhandenen Datenbanken für die man die entsprechende Berechtigung hat in der untenstehenden Form. Damit man dieses Statement absetzen kann ist keine Auswahl einer Datenbank notwendig.

Database
homepage
statistik
xml
mysql

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Funktionen in MySQL

In MySQL gibt es sehr viele nützliche Funktionen. Diese alle hier zu Listen würde die Seite sprengen. Deshalb habe ich hier zunächst die − aus meiner Sicht − wichtigsten. Nach und nach werde ich immer welche hinzufügen.

Hinweis zur Syntax:
Beim Benutzen von Funktionen ist es wichtig, daß zwischen dem Funktionsnamen und der öffnenden Klammer mit den Parametern **kein** Leerzeichen steht. Also nicht 'LOCATE ()' sondern 'LOCATE()' !!!

LOCATE()
Mit der Funktion LOCATE() läßt sich das die Position des ersten Auftretens eines Strings in einem anderen String (oder Feld) bestimmen.

```
SELECT LOCATE(nachname, 'Becker') AS position FROM
adressen
```

oder:

```
SELECT LOCATE('au', 'Klaus')
```

Im letzten Beispiel würde '3' als Position zurückgegeben.

Man kann LOCATE() auch mit drei Parametern aufrufen. Der letzte Parameter wird dann von MySQL als Offset angesehen.

```
SELECT LOCATE('au', 'Klaus ist zu Hause', 7)
```

Hier würde nicht '3' sondern '15' als Position zurückgegeben da erst ab dem 7ten Zeichen mit der Suche begonnen wurde.

REPLACE()
Mit der Funktion REPLACE() lassen sich Zeichen bzw. Zeichenketten in einer anderen Zeichenkette ersetzen. (oder Feld) bestimmen.

```
SELECT REPLACE('Becker', 'e', 'ae')
```

Hier würde 'Baecker' zurückgegeben.

UCASE()
Mit der Funktion UCASE() lassen sich Zeichen bzw. Zeichenketten in Großbuchstaben umwandeln. Siehe auch LCASE().

```
SELECT UCASE('Becker')
```

Hier würde 'BECKER' zurückgegeben.

```
SELECT UCASE(nachname) AS grossernachname FROM
```

```
adressen
```

LCASE()
Mit der Funktion LCASE() lassen sich Zeichen bzw. Zeichenketten in Kleinbuchstaben umwandeln. Siehe auch UCASE().

```
SELECT LCASE('Becker')
```
Hier würde 'becker' zurückgegeben.

FLOOR()
Die Funktion FLOOR() gibt den Integer−Wert für den angegebenen Parameter zurück, der nicht kleiner als der parameter selbst ist. Siehe dazu auch ROUND().

```
SELECT FLOOR(23.1734)
```
Hier würde '23' zurückgegeben.

```
SELECT FLOOR(−23.1734)
```
Hier würde '−24' zurückgegeben.

ROUND()
Die Funktion ROUND() rundet den angegebenen Parameter. Siehe dazu auch FLOOR().

```
SELECT ROUND(23.1734)
```
Hier würde '23' zurückgegeben.

```
SELECT ROUND(−23.1734)
```
Hier würde '−23' zurückgegeben.

```
SELECT ROUND(23.6734)
```
Hier würde '24' zurückgegeben.

```
SELECT ROUND(betrag) AS gerundeterbetrag FROM preise
```

CEILING()
Die Funktion CEILING() gibt den kleinsten Integer−Wert für den angegebenen Parameter zurück der nicht kleiner als der Parameter selbst ist. Siehe dazu auch FLOOR().

```
SELECT CEILING(23.1734)
```
Hier würde '24' zurückgegeben.

```
SELECT CEILING(−23.1734)
```
Hier würde '−23' zurückgegeben.

```
SELECT CEILING(betrag) AS neuerbetrag FROM preise
```

ABS()
Die Funktion ABS() gibt den absoluten Wert für den angegebenen Parameter zurück ("entfernt die Vorzeichen").

```
SELECT ABS(−23)
```

Hier würde '23' zurückgegeben.

```
SELECT ABS(23)
```

Hier würde auch '23' zurückgegeben.

SUM()
Mit der Funktion SUM() lassen sich Werte in einer Spalte aufsummieren.

```
SELECT SUM(alter) AS gesamtalter FROM adressen
```

AVG()
Mit der Funktion AVG() läßt sich der Durchschnittswert einer Spalte ermitteln.

```
SELECT AVG(alter) AS durchschnittsalter FROM adressen
```

MAX()
MAX() ermittelt den höchsten Wert in einer Spalte. MAX() ist ein Synonym für GREATEST().

```
SELECT MAX(alter), vorname, nachname FROM adressen
```

MIN()
MIN() ermittelt den niedrigsten Wert in einer Spalte.

```
SELECT nachname, MIN(alter) FROM adressen
```

VERSION()
VERSION() gibt die Version des MySQL−Datenbank−Servers zurück

```
SELECT VERSION()
```

Hier könnte z.B. '3.23.43−nt' zurückgegeben werden.

MATCH()
Die Funktion MATCH() kann zur Abfrage von Volltext−Indizes verwendet werden.

```
SELECT feld1,
MATCH(feld6) AGAINST ('suchbegriff') AS relevanz
FROM tabelle
```

Gibt man in der WHERE−Bedingung einer SQL−Anweisung dieselbe MATCH−Funktion erneut an, so werden die Ergebniszeilen nach Relevanz sortiert. Die Abfrage wird durch die doppelte Ausführung aber nicht langsamer. Näheres siehe unter ALTER TABLE und FULLTEXT.

```
SELECT feld1,
feld2,
MATCH(feld4, feld6) AGAINST ('suchbegriff1,
suchbegriff2') AS relevanz
FROM tabelle
MATCH(feld4, feld6) AGAINST ('suchbegriff1,
suchbegriff2')
```

SUBSTRING()
Mit der Funktion lassen sich Teile eines Feldes ermitteln. Die Funktion kann − ebenso wie alle anderen auch − in einer WHERE−Bedingung verwendet werden.

Allgemeine Form:

```
SELECT SUBSTRING(Feldname, Startposition, Länge) FROM
tabelle
```

Ist der optionale Parameter 'Länge' nicht angegeben, so wird die gesamte Zeichenkette ab der Position 'Startposition' zurückgegeben.

Beispiele:

```
SELECT SUBSTRING(vorname,1,1) as ersterBuchstabe FROM
adressen
```

Die obige Abfrage würde jeweils den ersten Zeichen des Vornamens zurückgeben.

```
SELECT SUBSTRING(vorname,1,3) as beginn FROM adressen
```

Die obige Abfrage würde jeweils die ersten drei Zeichen des Vornamens zurückgeben.

```
SELECT SUBSTRING(vorname,3,1) FROM adressen
```

Die obige Abfrage würde jeweils das dritte Zeichen des Vornamens zurückgeben.

```
SELECT SUBSTRING(vorname,4) FROM adressen
```

Die obige Abfrage würde jeweils alles ab dem vierten Zeichen des Vornamens zurückgeben.

CONCAT()
Mit CONCAT() lassen sich Inhalte von Feldern zusammenfassen.

```
SELECT CONCAT(vorname, " ", nachname) AS ganzerName
FROM adressen
```

Das obige Beispiel könnte beispielsweise "Paula Tratschig" zurückgeben. Durch die

Anführungszeichen lassen sich auch zuätzliche, nicht in der Tabelle vorhandene Zeichen mit ausgeben.

TRIM()
Mit TRIM() lassen sich Leerzeichen innerhalb von Feldern beseitigen.

```
SELECT TRIM(vorname) FROM adressen
```

Das obige Beispiel würde beispielsweise "Paula" zurückgeben wenn in der Tabelle der Wert 'Paula ' steht. Die Leerzeichen werden links und rechts entfernt. Nicht aber innerhalb des Wertes. So würde ' Anne Christin ' zu 'Anne Christin' werden.

RTRIM()
Wie TRIM(), nur werden dabei lediglich die Leerzeichen auf der rechten Seite des Parameters entfernt. Siehe auch LTRIM().

```
SELECT RTRIM(vorname) FROM adressen
```

LTRIM()
Wie TRIM(), nur werden dabei lediglich die Leerzeichen auf der linken Seite des Parameters entfernt. Siehe auch RTRIM().

```
SELECT LTRIM(vorname) FROM adressen
```

COUNT()
COUNT() ermittelt die Anzahl der Datensätze in der Tabelle

```
SELECT COUNT(*) AS gesamt FROM adressen
```

LENGTH()
LENGTH() ermittelt die Länge des Wertes in einem Feld.

```
SELECT nachname, LENGTH(nachname) AS laenge FROM
adressen
```

Das obige Beispiel liefert die Werte für 'nachname' und für die Länge des jeweiligen Namens zurück.

NOW()
Die Funktion NOW() gibt das aktuelle Datum und/oder die aktuelle Zeit zurück. Mögliche Formate sind 'YYYY−MM−TT HH:MM:SS' oder 'YYYYMMTTHHMMSS'. Ist ein Feld kürzer als der Wert den NOW() zurückgibt so wird der Teil beim INSERT eingefügt, den das Feld auch aufnehmen kann.

```
INSERT INTO statistiktabelle
 (ipadresse, hostname, datumzeit)
```

```
VALUES
 (now(), '221.118.112.22', 'www.hostname.de')
```

Die Funktion wird nicht in einfache Anführungszeichen gesetzt. Auch wenn es eine Zeichenkette zurückgibt.

PASSWORD()
Mittels PASSWORD() lassen sich Zeichenketten irreversibel verschlüsseln.

```
INSERT INTO benutzer
 (id, benutzername, passwort)
VALUES
 (3, 'tollerbenutzername',
PASSWORD('geheimesPasswort'))
```

PASSWORD() wird auch von MySQL selbst für die Speicherung von Passwörtern in der Benutzerdatenbank benutzt.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

INDEX / INDIZIES in MySQL

Bedeutung:
Indizes legt man an um schneller auf die Daten in einer Tabelle zugreifen zu können. Dies gilt allerdings nicht für Abfragen, die den kompletten Inhalt einer Tabelle abfragen. Mit Indizes lassen sich Abfragen an die Datenbank **erheblich** beschleunigen.
Faktor 1000 bis 1000000 (!) sind keine Seltenheit.

Anlegen eines INDEX:
Wie man einen INDEX bzw. mehrere INDIZIES anlegt und löscht ist unter ALTER TABLE beschrieben. Hier nochmal die allgemeine Form um einen INDEX anzulegen:

```
ALTER TABLE tabelle
ADD INDEXTYP indexName (feld1, feld2, feld3)
```

Typen von INDIZES:
MySQL unterscheidet vier Typen von INDIZES:

- **INDEX:**
  Ein 'normaler' INDEX

- **UNIQUE:**
  wie INDEX, mit der Bedingung, dass jeder Wert in dem indexierten Feld nur einmal vorkommt, also einzigartig (unique) ist

- **PRIMARY KEY:**
  wie UNIQUE, einziger Unterschied ist der, dass pro Tabelle nur ein einziger PRIMARY KEY zugelassen ist. Typischerweise verwendet man hierfür AUTOINCREMENT−Felder.

- **FULLTEXT:**
  Eine noch relativ neue Index−Variante von MySQL. Mit ihr lassen sich sehr große Textmengen indizieren und später wieder sehr schnell abfragen. Näheres dazu unter unter anderem unter FUNKTIONEN, ALTER TABLE und FULLTEXT.

  Der Typ FULLTEXT soll in der Version 4.01 und folgende noch weiter ausgebaut werden. U.a. werden dann auch Suchen mit boolschen Operatoren unterstützt (−>FULLTEXT).

Welche Felder sollen indexiert werden?
Bei der Auswahl der Felder für das Anlegen eines Indizes sind als erstes die Felder zu berücksichtigen, die häufig innerhalb einer WHERE−Bedingung abgefragt werden. Als weitere Kandidaten für die Indexierung kommen die Felder in Frage, auf die per ORDER BY oder GROUP BY zugegriffen wird.

Ganz besonders wichtig ist es, die Felder mit einem INDEX zu belegen, die in einer Kreuzabfrage

(Abfragen mehrerer Tabellen in einem Statement) auf ihren Wert abgeprüft werden. Hier sind die größten Geschwindigkeitsverbesserungen zu erwarten.

Beispiel:

```
SELECT eins.feld1, eins.feld2, zwei.feld1
FROM tabelle1 eins, tabelle2 zwei
WHERE eins.feld2 = zwei.feld1
```

Hier würde es Sinn machen, das Feld *feld2* aus Tabelle *tabelle1* und das Feld *feld1* aus Tabelle *tabelle2* jeweils mit einem INDEX zu belegen.

Bei der Indexierung von Text− oder Blob−Feldern sollte man vorsichtig sein. Selbst wenn man das Textfeld indexiert ist der Geschwindigkeitsvorteil unter Umständen nur gering. Häufig ist es besser, nur einen Teil des Feldes zu indexieren (z.B. nur die ersten 100 Zeichen).

Ist man sich sicher, das jeder Wert in einem Feld nur einmal vorkommt so sollte man als Indextyp *UNIQUE* wählen. MySQL kann so schneller auf die Werte in dem INDEX / Feld zugreifen.

Grundsätzlich gilt:

- Je weniger gleiche Werte es in einem Feld einer Tabelle gibt desto mehr eignet sich das Feld für einen INDEX.
- Bei Kreuzabfragen werden die Abfragen am schnellesten abgearbeitet, bei denen die zu vergleichenden Felder vom selben Feldtyp sind. Im letzten Beispiel wäre es zum Beispiel optimal, wenn *feld2* und *feld1* beide vom Typ INTEGER(8) oder vom Typ CHAR(4) wären und jeweils mit einem INDEX belegt wären.
- Bei Feldern mit sehr großem Inhalt kann es sinnvoller sein, nur einen Teil des Feldes mit einem Index zu belegen (z.B. nur die ersten 100 Zeichen). Der von MySQL angelegte INDEX wird dann nicht so groß und kann daher schneller durchsucht werden.
- Sollen sehr große Textmengen (bis zu mehreren hundert MB) indiziert werden so empfiehlt sich die Indizierung als FULLTEXT. Bei relativ kleinen Textmengen sollte man hiervon jedoch keinen Gebrauch machen da die Abfragen an diesen Indextyp dann zu sehr eigenartigen Ergebnissen führen kann.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

FULLTEXT − Index

Bedeutung:
Mit einem FULLTEXT − Index lassen sich in MySQL sehr große Textmengen sehr effizient indizieren und mit einigen netten Features wieder abfragen. Den FULLTEXT − Index sollte man allerdings nicht für kleinere Textmengen verwenden.

Allgemeine Form zum Anlegen eines FULLTEXT − Index:

```
ALTER TABLE tabelle
ADD FULLTEXT KEY indexName (feld1, feld2, etc.)
```

Allgemeine Form zum Abfragen eines FULLTEXT − Index (mit Sortierung nach Relevanz):

```
SELECT
feld1,
feld2,
MATCH(feld3, feld4, feld6, feld8) AGAINST
('suchbegriff1 suchbegriff2') AS relevanz
FROM tabellenName
MATCH(feld3, feld4, feld6, feld8) AGAINST
('suchbegriff1 suchbegriff2')
```

Allgemeine Form zum Abfragen eines FULLTEXT − Index (ohne Sortierung nach Relevanz):

```
SELECT
feld1,
feld2,
MATCH(feld3, feld4, feld6, feld8) AGAINST
('suchbegriff1 suchbegriff2') AS relevanz
FROM tabellenName
```

Bemerkungen zum Anlegens eines FULLTEXT − Index
Beim Anlegen eines FULLTEXT − Index nimmt MySQL eine Gewichtung der in den Feldern enthaltenen Wörter vor. Diese Gewichtung spielt u.a. bei einer späteren Abfrage eine große Rolle: Man kann sich daß Suchergebnis nach Relevanz absteigend sortieren lassen (siehe 'Allgemeine Form zum Abfragen eines FULLTEXT − Index'). Gibt man in der WHERE−Bedingung die MATCH−Funktion nicht erneut an, so werden die Ergebniszeilen unsortiert zurückgegeben.

Bei der Indizierung fallen standardmäßig alle Begriffe unter den Tisch, die kürzer als vier Zeichen sind (dies kann allerdings durch das Setzen der entsprechenden Variablen geändert werden). Wenn ein Begriff sehr häufig vorkommt bedeutet dies nicht unbedingt, daß er eine hohe Relevanz zugewiesen bekommt. Er kann unter Umständen sogar eine sehr niedrige Relevanz zugewiesen bekommen.

Die Begriffe werden entsprechend der Häufigkeit des Vorkommens in dem jeweiligen Datensatz und entsprechend der Häufigkeit des Vorkommens in der gesamten Tabelle gewichtet. Ein Volltext−Index ist somit immer in Beziehung zur jeweiligen Tabelle zu sehen. In einer anderen Tabelle mit anderen Inhalten kann ein Begriff eine ganz andere Relevanz zugewiesen bekommen.

Ein FULLTEXT − Index kann auch innerhalb sehr großer Tabellen angelegt werden. Das kann u.U. nur sehr lange dauern. Bei einer 500 Megabyte großen Tabelle kann das durchaus zwischen einer und zwanzig Stunden dauern. Die Sache ist natürlich auch abhängig von der Hardware des verwendeten Rechners (v.a. viel RAM und eine schnelle Festplatte sind hier wichtig).

Bemerkungen zum Abfragen eines FULLTEXT − Index
Wie oben schon angedeutet kann man sich die Relevanz der zurückgegebenen Datensätze bei einer Abfrage mit ausgeben lassen. Man muß es aber nicht.

In der Ergebnismenge werden die Datensätze am höchsten gewichtet, bei denen die meisten Übereinstimmungen gefunden wurden. Wurde z.B. nach 'Teller' und 'Haustür' gesucht, so werden sowohl Datensätze gelistet in denen nur der Begriff 'Teller' gefunden wurde als auch Datensätze in denen nur der Begriff 'Haustür' gefunden wurde. Am höchsten gewichtet werden die Datensätze, in den beide Begriffe gefunden wurden. Ausnahme: Die Datensätze in denen z.B. die Begriffe 'Teller' und 'Haustür' zu häufig vorkommen (siehe dazu auch weiter oben).

Der FULLTEXT − Index eignet sich nur zur Indizierung von (wirklich) großen Textmengen. Bei sehr kleinen Mengen an Text kann eine Abfrage auf den INDEX zu sehr verwunderlichen Ergebnissen führen.

Die Abfrage selbst von sehr großen Tabellen ist später auch auf normaler PC−Hardware sehr schnell. Die Zeiten bei einer 500 Megabyte großen Tabelle liegen zum Teil unter 0,1 Sekunden (AMD Athlon 1000, 256 MB SDRAM, HD mit 5400 U). Fast immer aber unter einer Sekunde. Allerdings hängt die Geschwindigkeit scheinbar sehr stark von der Menge der Ergebniszeilen ab: Die Angabe eines LIMIT kann die Abfrage ungemein beschleunigen. Die verwendete Hardware spielt natürlich ebenfalls eine große Rolle.

Die zweifache Angabe der MATCH−Funktion (in der Feldliste und in der WHERE−Bedingung) bedeutet nicht, daß die Abfrage doppelt so lange dauert als wenn die Funktion nur einmal im Statement enthalten wäre. MySQL merkt sich das Ergebnis der zuerst ausgeführten. Einen Geschwindigkeitsverlust soll es laut Manual nicht geben.

Zuküftige Features
In den Versionen nach 4.0 soll − neben einer höheren Geschwindigkeit − ein zusätzliches Feature implementiert werden: Die Möglichkeit der Verwendung von boolschen Operatoren und Klammerungen innerhalb der Suchanfrage wie man es von vielen Suchmaschinen her kennt.

Beispiel:

```
SELECT
feld1,
feld2,
MATCH(feld3, feld4, feld6, feld8)
AGAINST ('+suchbegriff1 suchbegriff2 −suchbegriff3'
IN BOOLEAN MODE) AS relevanz
FROM tabellenName
MATCH(feld3, feld4, feld6, feld8)
AGAINST ('+suchbegriff1 suchbegriff2 −suchbegriff3'
IN BOOLEAN MODE)
```

Bei solchen Abfragen ist dann der Zusatz 'IN BOOLEAN MODE' innerhalb der Klammer nötig. Daran erkennt MySQL, wie es die Suchanfrage zu interpretieren hat.

Begriffe denen ein Plus−Zeichen vorangestellt ist müssen im Ergebnis enthalten sein. Begriffe mit einem vorangestellten Minus dürfen nicht im Ergebnis enthalten sein. Wenn ein Begriff ohne Vorzeichen wie im Beispiel der Suchbegriff 'suchbegriff2' im Ergebnis enthalten ist, so wird dem entsprechenden Datensatz eine höhere Relevanz zugewiesen als einem in dem er nicht vorkommt.

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Feldtypen in MySQL

In MySQL kann man zwischen drei Kategorien von Feldtypen unterscheiden:

- Numerische Typen
- String−Typen
- Datum− und Zeit−Typen

Numerische−Typen
Bei den numerischen Typen in MySQL kann man zwischen Typen für Ganzzahlen und Typen für Fließkommazahlen unterscheiden. Ganzzahlige Typen können mit negativem Vorzeichen und mit positiven Vorzeichen versehen sein (z.B. 1244, −3, −9808080). Die Typen für Fließkommazahlen können auch Bruchteile von Zahlen darstellen (z.B. 1.46, −0.3125, 2132327878.132387).

Wenn beim Anlegen von Feldern vom Typ 'Ganzzahl' (z.B. 'BIGINT') die Option 'UNSIGNED' mit angegeben wurde so erweitert sich der Bereich der möglichen positiven Zahlen für das Feld um fast 100%. Der Bereich für negative Zahlen dagegen entfällt. Die Option 'UNSIGNED' kann beim Anlegen von Fließkommatypen nicht verwendet werden

Der verwendete Typ hat auch Einfluß auf den benötigten Speicherplatz (wie bei den meisten anderen Feldtypen auch). Der Typ 'BIGINT' belegt zum Beispiel mehr Speicherplatz als der Typ 'INT'. Genaueres dazu und zum Wertebereich steht in der Tabelle unter diesem Text.

Hinweis: Kommas werden in MySQL als Punkt dargestellt!

Übersicht über die Wertebereiche der numerischen Datentypen in MySQL:

Datentyp	Speicherplatz	Wertebereich	Wertebereich (unsigned)
TINYINT	1 Byte	−128 − 127	0 − 255.
SMALLINT	2 Bytes	−32768 − 32767	0 − 65535
MEDIUMINT	3 Bytes	−8388608 − 8388607	0 − 16777215
INT	4 Bytes	−2147483648 − 2147483647	0 − 4294967295
BIGINT	8 Bytes	−9223372036854775808 − 9223372036854775807	0 − 18446744073709551615
FLOAT	4 Bytes	−3.402823466E+38 bis −1.175494351E−38, 0, und 1.175494351E−38 bis 3.402823466E+38	Deklarierung nicht möglich
DOUBLE (Syn. f. REAL)	8 Bytes	−1.7976931348623157E+308 bis −2.2250738585072014E−308, 0, und 2.2250738585072014E−308 bis	Deklarierung nicht möglich

1.7976931348623157E+308

String−Typen
Bei den String−Typen in MySQL kann man zwischen fünf verschiedenen Untertypen unterscheiden. Wenn man will auch mehr oder weniger ;−).

- **CHAR, VARCHAR**
  Dieser Typ eignet sich für kleinere Texte, die nicht mehr als 255 beinhalten. In einer Tabelle kann man immer nur 'VARCHAR' oder nur 'CHAR' verwenden. Eine Verwendung von beiden Typen innerhalb einer Tabelle ist nicht möglich. Nach Möglichkeit sollte man CHAR verwenden. Dieses aber nur dann, wenn die gespeicherten Werte in dem Feld immer ungefähr dieselbe Länge haben. Der Speicherbedarf für 'CHAR' ist etwas geringer da die Länge des enthaltenden Strings nicht mitgespeichert werden muß.
- **TINYTEXT, MEDIUMTEXT, TEXT, LONGTEXT**
  Der Typ TEXT in seinen vier Ausprägungen eignet sich zur Aufnahme von kleineren bis hin zu riesigen Textmengen (4294967296 Zeichen). Bei einer Abfrage eines Feldes mit dem Typ TEXT (oder TINYTEXT, etc. ;−)) wird zwischen Groß− und Kleinschreibung nicht unterschieden. Die Groß− und Kleinschreibung bleibt gleichwohl beim Speichern von Werten in dem Feld erhalten.
- **TINYBLOB, MEDIUMBLOB, BLOB, LONGBLOB**
  Die vier BLOB−Typen unterscheiden sich insofern vom Typ Text als dass sie Daten in binärer Form speichern. Man kann z.B. außer normalem Text auch Bilder in einem BLOB−Feld speichern. Bei der Abfrage von BLOB−Feldern wird im Gegensatz zu TEXT−Feldern zwischen Groß− und Kleinschreibung unterschieden!
- **ENUM**
  Der Typ 'ENUM' kann − wie der Typ 'SET' auch − nur bestimmte Werte aufnehmen. Diese müssen beim CREATE TABLE Statement bereits als kommaseparierte Liste in Klammern mit angegeben werden. Es sind maximal 65535 Werte in der Liste möglich. Ein ENUM−Feld kann − im Gegensatz zum SET−Feld immer nur einen einzigen Wert aufnehmen.
- **SET**
  Der Typ 'SET' kann ebenfalls nur bestimmte Werte aufnehmen. Diese Werteliste muß beim Anlegen mit angegeben werden. Nachträglich kann man diese Liste − wie beim ENUM−Typ auch − per ALTER TABLE noch verändern.

Übersicht über den benötigten Speicherplatz der String−Datentypen in MySQL:

Datentyp	Speicherplatz
CHAR(M)	M Bytes, 1 <= M <= 255
VARCHAR(M)	L+1 Bytes, wenn L <= M und 1 <= M <= 255
TINYBLOB	L+1 Bytes, wenn L < 2^8
TINYTEXT	L+1 Bytes, wenn L < 2^8
BLOB	L+2 Bytes, wenn L < 2^16
TEXT	L+2 Bytes, wenn L < 2^16
MEDIUMBLOB	L+3 Bytes, wenn L < 2^24
MEDIUMTEXT	L+3 Bytes, wenn L < 2^24
LONGBLOB	L+4 Bytes, wenn L < 2^32
LONGTEXT	L+4 Bytes, wenn L < 2^32

ENUM('wert1','wert2',...)	1 or 2 Bytes, abhängig von der Anzahl der Einträge (max. 65535)
SET('wert1','wert2',...)	1, 2, 3, 4 oder 8 Bytes, abhängig von der Anzahl der Einträge (max. 64)

Datum− und Zeit−Typen
MySQL kennt fünf verschiedene Datum− und Zeittypen. Mit diesen Typen sind auch Vergleiche möglich. Dies ist auch u.a. der Grund dafür, warum man immer ein Datumsfeld einem String−Feld vorziehen sollte wenn man es mit Datum− oder Zeitangaben zu tun hat.

Beispiel:
```
SELECT FROM tabelle WHERE datumsfeld < now()
```

Die Typen im einzelnen:

- **DATE**
  Der Typ 'DATE' nimmt Werte im Format JJJJ−MM−TT auf. Das Jahr kann auch zweistellig angegeben werden. Die Bindestriche können weggelassen werden. Kleinster Wert ist '1000−01−01'.
- **TIME**
  Ein Feld vom Typ 'TIME' kann Werte im Format hh:mm:ss aufnehmen. Die Doppelpunkte können entfallen. 'TIME' kann auch negative Werte aufnehmen. Der Wert in einem TIME−Feld stellt im Ggs. zu 'DATETIME' immer einen Zeitraum dar!
- **DATETIME**
  Der Typ 'DATETIME' ist eine Kombination aus 'DATE' und 'TIME'. Das Standardformat ist 'JJJJ−MM−TT hh:mm:ss. Bindestriche und Punkte können weggelassen werden.
- **TIMESTAMP**
  'TIMESTAMP' nimmt − wie der Name schon sagt − einen Zeitstempel auf. Der Typ ähnelt dem typ 'DATETIME'. Die möglichen Formate sind mit denen von 'DATETIME' identisch. Nur das Standardformat kommt ohne Leerzeichen, Bindestriche und Doppelpunkte aus. Der kleinste Wert ist 19700101000000 (Beginn der UNIX−Epoche im Jahr 1970). Das letzte Jahr der UNIX−Epoche ist − momentan − das Jahr 2037...
- **YEAR**
  'YEAR' nimmt Jahreszahlen auf. Dies tut es sowohl zwei− als auch vierstellig.

Übersicht über den benötigten Speicherplatz der numerischen Typen für Datum und Zeit:

Datentyp	Speicherplatz
DATE	3 Bytes
TIME	3 Bytes
DATETIME	8 Bytes
TIMESTAMP	4 Bytes
YEAR	1 Byte

Letzte Änderung: 17. Mai 2002, 21:26 Uhr

Kontakt

Klaus Becker
Dorfbauerschaft 5
48485 Neuenkirchen

eMail: becker−k@web.de

Letzte Änderung: 17. Mai 2002, 21:26 Uhr