Doing Web Apps

Datenbankdesign und SQL

Autor: Rüdiger Marwein
Letzte Änderung: 2009-05-08
Version: 0.7

Inhaltsverzeichnis

1.Einleitung

Mit Datenbanken organisiert man Daten. Es werden Tabellen verwendet, um Datensätze zu speichern.

Es ist wichtig, dass die Daten kontrolliert gespeichert und ausgelesen werden können. Um **Redundanzen** (mehrere überflüssigerweise gleiche Einträge) zu vermeiden, erstellt man Datenbanken in der Regel mit Hilfe der 3. Normalform.

Wir verwenden aufgrund der Popularität MySQL für Beispiele, weitere von PHP unterstützte Datenbanken sind z.B. Oracle, PostgreSQL, SQLite, Firebird, dBase, Access und viele mehr.

Dieses Dokument ist etwas umfangreicher, da man viel mit Beispielen argumentiert wird. Man lasse sich vom Umfang nicht stressen. Das Prinzip ist nicht schwierig.

2.Die Normalformen

Eine Normalform beschreibt, wie man Daten in Tabellen aufteilt, so dass eine bestimmte Stufe der Eindeutigkeit der Daten vorliegt.

Es gibt insgesamt 5 Normalformen, wobei alle aufeinander aufbauen und die 4. und 5. zwar „schön“ aber inperformant (also zuviel Aufwand und im Endeffekt zu langsam) sind.

Die am meisten angewandte Normalform ist die dritte Normalform. Diese baut auf der zweiten Normalform auf und diese wiederum auf der ersten.

Da man beim Entwerfen nur die Feldnamen vor sich hat, entgehen einem leicht die Stellen, an denen Redundanzen auftreten werden. Das ist eine Sache der Erfahrung.

Wir betrachten einen ersten Entwurf einer Datenbank-Tabelle zum Verwalten von

Rechnungen

ReNr	*ReDatum*	*ReBetrag*	*KuNr*	*KuName*	*KuOrt*	*ArtNr*	*ArtName*	*ArtPreis*
1	13.08.1999	1370	4	Meier	Hamburg	4, 2, 6	Prozessor, Speicher, Monitor	350, 120, 900
2	13.08.1999	900	1	Schulze	Berlin	6	Monitor	900
3	14.08.1999	500	5	Fischer	Hamburg	1, 4	Gehäuse, Prozessor	150, 350

Tabelle 1 Erster Tabellenentwurf

Die *Idee* ist also hier die bestellten Artikel **mit Komma getrennt** in das Feld ArtNr (Atrikelnummer) zu schreiben und in gleicher Reihenfolge ebenfalls kommagetrennt den ArtName (Artikelname) und dazu dann die Preise in ArtPreis. So ***kann*** man dieses Problem angehen... sollte man aber nicht. Das **entspricht keiner Normalform** und ist (wenn man ein bisschen mit SQL rumprobiert) außerdem ziemlich lästig auszulesen.

2.1.Die erste Normalform

Hier gilt: **Die Feldinhalte müssen atomar (einfach vorkommend) sein**. Das bedeutet für unsere Bestell-Tabelle, dass die kommagetrennte Idee verworfen werden muss.

Eine Tabelle, die der ersten Normalform entspricht würde so aussehen:

ReNr	*ReDatum*	*ReBetrag*	*KuNr*	*KuName*	*KuOrt*	*ArtNr*	*ArtName*	*ArtPreis*
1	13.08.1999	1370	4	Meier	Hamburg	4	Prozessor	350
1	13.08.1999	1370	4	Meier	Hamburg	2	Speicher	120
1	13.08.1999	1370	4	Meier	Hamburg	6	Monitor	900
2	13.08.1999	900	1	Schulze	Berlin	6	Monitor	900
3	14.08.1999	500	5	Fischer	Hamburg	1	Gehäuse	150
3	14.08.1999	500	5	Fischer	Hamburg	4	Prozessor	350

Tabelle 2 Erste Normalform

Jetzt sind die Daten pro Zeile atomar. Aber da fällt uns schon das nächste auf. Wir haben jetzt **Redundanzen** bei den Rechnungs- und Kundendaten, womit wir zum nächsten Schritt kommen.

2.2.Die zweite Normalform

Hier gilt außerdem: **Jedes Feld muss funktional vom Schlüsselfeld abhängig sein.**

Die Bestandteile, die sich oben wiederholen sind im Prinzip in 3 einzelne Datensätze aufteilbar. Nämlich Rechnungen, Kunden und Rechnungs-Artikel.

So kann einem Kunden eine Nummer zugewiesen werden. In verweisenden Tabellen müssen wir nur diese Nummer angeben, um uns auf einen Kunden zu beziehen.
Ebenso kann ein man einer Rechnung eine Nummer zuweisen und in anderen Tabellen dann analog nur die Rechnungsnummer angeben.
Diese eindeutigen Schlüssel um einen bestimmten Datensatz zu identifizieren, heißen **Primärschlüssel**. Primärschlüssel werden im folgenden in Tabellen immer mit einem  ausgezeichnet.

KuNr	KuName	KuOrt
1	Schulze	Berlin
2	Carstens	München
3	Schröder	Bremen
4	Meier	Hamburg
5	Fischer	Hamburg

Tabelle 3 Zweite Normalform - Kunden

ReNr	KuNr	ReDatum	ReBetrag
1	4	13.08.1999	1370
2	1	13.08.1999	900
3	5	14.08.1999	230

Tabelle 4 Zweite Normalform - Rechnungen

ReNr	ArtNr	ArtName	ArtPreis
1	4	Prozessor	350
1	2	Speicher	120
1	6	Monitor	900
2	6	Monitor	900
3	1	Gehäuse	150
3	4	Prozessor	250

Tabelle 5 Zweite Normalform - Rechnungsdaten

Als Profis erkennen wir natürlich sofort, dass der Artikelname mehrfach Produkte mit der selben Artikelnummer beschreibt und sich das mit an Sicherheit grenzender Wahrscheinlichkeit **noch weiter normalisiert** lässt. Korrekt. Damit sind wir dann beim nächsten Schritt.

2.3.Die dritte Normalform

Hier gilt: **Es dürfen keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Feldern existieren**. Die Tabelle darf somit **keine transitiven Abhängigkeiten** aufweisen.

In unserem konkreten Beispiel betrifft dies nun die Tabelle mit den redundanten Produktnamen. Wir erstellen eine Produkte-Tabelle und eine Verbindungstabelle zwischen Rechnung und Produkten.

KuNr	KuName	KuOrt
1	Schulze	Berlin
2	Carstens	München
3	Schröder	Bremen
4	Meier	Hamburg
5	Fischer	Hamburg

Tabelle 6 Dritte Normalform - Kunden

ArtNr	ArtName	ArtPreis
4	Prozessor	350
2	Speicher	120
6	Monitor	900
1	Gehäuse	150

Tabelle 7 Dritte Normalform - Produkte

ReNr	KuNr	ReDatum	ReBetrag
1	4	13.08.1999	1370
2	1	13.08.1999	900
3	5	14.08.1999	230

Tabelle 8 Dritte Normalform – Rechnungen

ReNr	ArtNr
1	4
1	2
1	6
2	6
3	1
3	4

Tabelle 9 Dritte Normalform – Rechungsdaten

3.Beziehungen zwischen Tabellen

In der Tabelle, wo auf eine Rechnungsnummer (mehrere Zeilen enthalten die selbe) mehrere Produktnummern kommen, spricht man von einer **n zu m Beziehung (n:m)**, wenn diese in einer Beziehung in einer separaten Verbindungstabelle abgebildet wird. So wie in der Rechnungsdaten-Tabelle der dritten Normalform.

Dieselbe Beziehung aus der zweiten Normalform würde den Titel **eins zu n Beziehung (1:n)** bekommen. Einfach deshalb, weil für eine Rechnungsnummer mehrere Zeilen (die die Rechnungsnummer enthalten) in der Rechnungsdaten-Tabelle vorhanden sein können.

Gibt es zu einem Datensatz in einer anderen Tabelle genau ein Gegenstück, so handelt es sich um eine **eins zu eins Beziehung (1:1)**.

Oder anders formuliert:

Die Tabelle in der ein Satz anhand des Primärschlüssels eindeutig in der Tabelle vorkommt (genau einmal), erhält die **1**, zeigt er auf eine Tabelle wo dieser Schlüssel mehrmals vorkommt, erhält diese andere das **n**.

Das ER-Diagramm zu unserem Beispiel könnte z.B. so aussehen

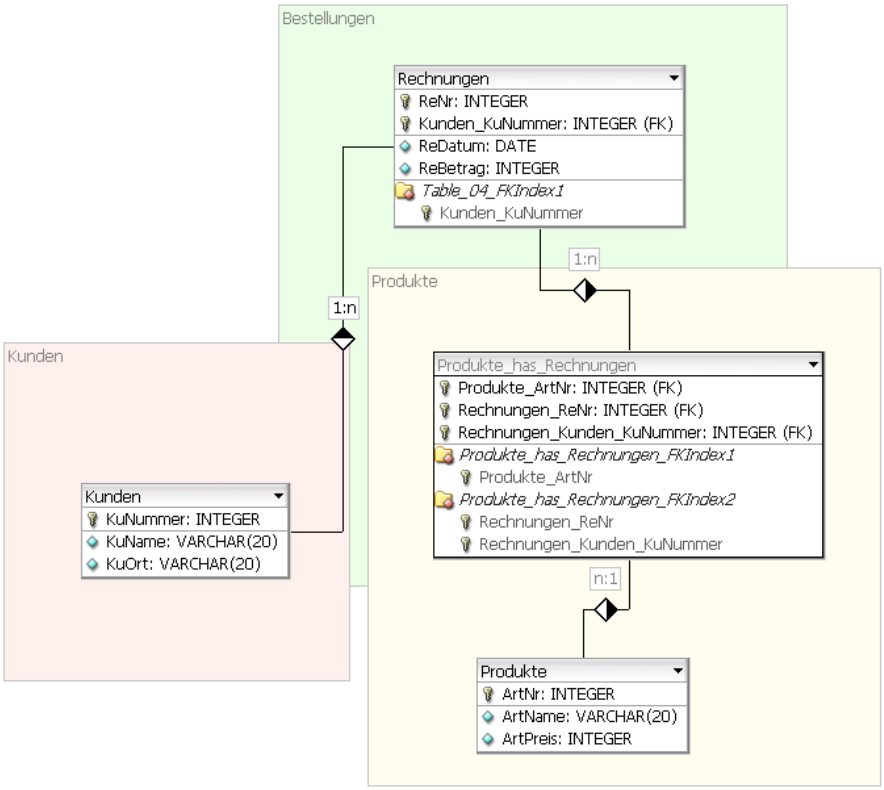

Abbildung 1 ER Datenbank-Diagramm

Wir sehen also, es gehört durchaus etwas Gehirnschmalz zum Entwurf eines Datenbank-

Schemas.

In der Regel verwendet man die dritte Normalform. Es bietet sich also an, beurteilen zu können, in welcher Normalform sich eine Tabelle befindet, um gegebenenfalls zu optimieren.

4.SQL

In einer Datenbank werden ja bekanntlich Daten gespeichert. Aber

a) wie kommen die da hin,

b) wie greift man drauf zu und

c) wie kriegt man sie wieder weg ?

Fragen die die Welt bewegen...

Wir werden hier Zugriffsmechanismen beschreiben, das Erstellen von Datenbanken und Tabellen überlassen wir vorerst den zahllosen Tools, die uns das abnehmen. Die Syntax zu erstellen und ändern (CREATE / DROP / ALTER) sind im Anhang zu finden.

Hinweis: Die Syntax kann von Datenbank zu Datenbank etwas variieren, der Grundaufbau bleibt jedoch meist bestehen. Je nachdem welche Datebank eingestezt wird, macht es natürlich Sinn, sich über die besonderen Fähigkeiten zu informieren und diese gegebenenfalls zu verwenden.

Die formelle Spezifikation der Kommandos ist im Anhang zu finden.

4.1.INSERT

Ein Insert **fügt eine neue Zeile in eine Tabelle ein**. Er folgt im Großen und Ganzen immer der Syntax

```
INSERT INTO Tabelle (Feld1,Feld2) VALUES ('Wert1','Wert2');
```

Also ausformuliert etwa: **Mach eine neue Zeile in Tabelle x, und zwar für folgende Felder die folgenden Werte.**

So könnten wir in unsere Kundentabelle jetzt einen neuen Eintrag machen:

```
INSERT INTO Kunden
    (KuNr,KuName,KuOrt)
VALUES
    (6,'Marwein','Reutlingen');
```

4.2.UPDATE

Mit einem Update werden **Datensätze aktualisiert**, die ein bestimmtes Kriterium erfüllen. Konkret wird Inhalt geprüft und bei zutreffen wird die Aktion ausgeführt. Er folgt im Großen und Ganzen immer der Syntax

```
UPDATE Tabelle SET Feld1='Wert1',Feld2='Wert2' WHERE
Feld1='Wert_1' AND Feld2='Wert_2';
```

Also ausformuliert etwa: **Aktualisiere alle Zeilen, wobei Feld1 auf 'Wert1' und Feld2**

auf 'Wert2' zu setzen ist... aber nur dort, wo Feld1 momentan 'Wert_1' beinhaltet und Feld2 momentan 'Wert_2' beinhaltet.

Technisch fährt das Update-Kommando einmal vom Anfang der Tabelle bis ans Ende, prüft jeweils die Kriterien und ändern bei einem Treffer die Werte.

Wir könnten jetzt also ein Update machen auf unsere Produkte-Tabelle und alle "Monitor"-Einträge zu "TFT-Monitor"-Einträgen abzuändern.

```
UPDATE
     Produkte
SET
     ArtName='TFT-Monitor'
WHERE
     ArtName='Monitor';
```

4.3.SELECT

Mit einem Select werden Daten aus der Datenbank ausgelesen. Dabei werden wieder Kriterien abgefragt und nur übereinstimmende Einträge werden ausgegeben. Er folgt im Großen und Ganzen immer der Syntax

```
SELECT Feld1,Feld2 FROM Tabelle WHERE Feld1='Wert1';
```

Oder für einfach alle Felder mit Sternchen:

```
SELECT * FROM Tabelle WHERE Feld1='Wert1';
```

Wir könnten uns also alle Kunden herausfinden lassen, die in Hamburg wohnen.

```
SELECT
     *
FROM
     Kunden
WHERE
     KuOrt='Hamburg';
```

Die Ausgabe würde wir folgt aussehen:

```
KuNr |KuName  |KuOrt
=====|========|=======
   4 |Maier   |Hamburg
   5 |Fischer |Hamburg
```

4.4.DELETE

Ein Delete arbeitet ähnlich wie ein Update, allerdings löscht er die den Kriterien entsprechenden Zeilen. Er folgt im Großen und Ganzen immer der Syntax

```
DELETE FROM Tabelle WHERE Feld1='Wert1';
```

Wir könnten also alle Monitore aus dem Produktangebot herausschmeißen anhand der

Artikel-Nummer 6

```
DELETE FROM
     Produkte
WHERE
     ArtNr=6;
```

5.Mit mehreren Tabellen

Es ist ja jetzt ganz toll, dass wir aus einer Tabelle Daten herausholen können. Blöderweise haben wir aber – wie es sich gehört - alle logischen Bestandteile wunderschön in einzelne Tabellen aufgeteilt. Um also an Daten zu kommen, die im Zusammenhang Sinn machen gibt es eine handvoll Möglichkeiten.

Wenn wir z.B. wissen wollen, aus welcher Stadt Kunden kommen, die Monitore gekauft haben, könnten wir natürlich zuerst aus den **Produkten** die Produktnummer eines Monitors ermitteln, danach die Rechnungsnummer dazu aus den **Rechnungsdaten**, dann anhand der Rechnungsnummer die Kundennummer aus den **Rechnungen** und schließlich die Städte zur Kundennummer aus der **Kundentabelle**. Auf dem Weg müssen wir uns die Werte mit denen weitergearbeitet wird irgendwie und irgendwo merken. Mit einer Programmiersprache kein Problem, jedoch steigt dadurch zum einen die Menge der Code-Zeilen und zum anderen sind so viele Selects hintereinander nicht unbedingt datebankschonend.

Hier sind jetzt 4 Tabellen mit einbezogen. Da das ein bisschen zu heftig wäre, denken wir uns, wir wissen, dass Monitore die Artikelnummer 6 haben.

Man kann aus mehreren Tabellen gleichzeitig auslesen.

Wenn mehrere Tabellen abgefragt werden, nennt man das einen Join (Verbindung).

Das Beispiel verdeutlicht, wie es im Prinzip von statten geht:

```
SELECT
     k.KuOrt
FROM
     Kunden AS k, Rechnungen AS r, Rechnungsdaten AS rd
WHERE
     rd.ArtNr  = 6 AND
     r.ReNr    = rd.ReNr AND
     k.KuNr    = r.KuNr
```

Dieses Beispiel beinhaltet jetzt eine ganze Menge Elemente. Zum einen sehen wir, dass die WHERE-Bedingung mit mehreren Bedingungen und logischen Verknüpfungen umgehen kann. Diese sind in jedem Fall AND, OR und NOT. Des weiteren kann aus Gleichheit mittels = und != die Gleichheit geprüft werden, mit >, <, >=, <= können Vergleiche angestellt werden.

Die Ausgabe für unser Beispiel ist im übrigen

```
KuOrt    |
=========|=
Hamburg  |
Berlin   |
```

Bei einem solchen Join muss alles gegengeprüft werden, was gegenprüfbar ist. Vergisst

man einen Vergleich, kommen zu viele[1] und u.U. Falsche Daten zurück.

Angenommen wir hätten die letzte Zeile weggelassen, würden wir folgende Ausgabe erhalten

```
KuOrt    |
=========|=
Berlin   |
München  |
Bremen   |
Hamburg  |
Hamburg  |
Berlin   |
München  |
Bremen   |
Hamburg  |
Hamburg  |
```

Man kann sich vorstellen, dass SQL über jede Tabelle einzeln läuft und jeweils jeden Datensatz mit jedem Datensatz in Relation setzt indem es die WHERE-Clause anwendet. Die Kundentabelle wird auch durchlaufen, jedoch greift bei ihr keine einzige Regel. Somit erhalten wir die vollen Sätze. Fehler.

Man mag sich fragen warum man überhaupt normalisiert, wenn das doch den Aufwand so exorbitant steigert. Mit Recht. Jedoch sollte man bedenken, dass, wenn man die Tabellen in einer ganz bestimmten Form aufbaut, die genau eine Sicht abdeckt, man bei Abwandlungen der Sicht schon in eine Sackgasse laufen kann. Mittels Joins kann man sehr **viele beliebige Relationen** herstellen. Und das ist mit Tabellen der **dritten Normalform** am angenehmsten zu realisieren.

6.Weitere Funktionen

Mit SQL lässt sich so einiges anstellen. Die wichtigsten Funktionen sind wahrscheinlich die Ergebnis-Gruppierung, die Anzahl der Ergebniszeilen und die Anordnung der Ergebniszeilen.

6.1.Wildcards

Oft werden recht große Zeichenketten gespeichert, in denen nach dem Vorkommen einer bestimmten Phrase gesucht werden soll. Das kann man mit SQL mit Hilfe des LIKE-Vergleichs machen.

```
SELECT
    Feld1
```

1 Das liegt daran, dass beim Verbinden mehrerer Tabellen mittels Join das kartesische Produkt gebildet wird, also erstmal alles mit allem kombiniert wird. Erst die Bedingungen schaffen wieder Klarheit.

```
FROM
     Tabelle
WHERE
     Feld2 LIKE '%Wert_1%'
GROUP BY
     Feld2;
```

Das **%** und **_** sind Wildcards. % steht für beliebig viele Zeichen und _ für ein beliebiges Zeichen. In unserem Beispiel heißt das, dass die Zeichenkette irgendwo „Wert_1“ stehen haben muss. Das % am Anfang und nicht am Ende bedeutet, die Zeichenkette muss mit dieser Phrase aufhören. Umgekehrt gilt das selbe. Mit dem Unterstrich finden wir sowohl „Mein Wert 123“ als auch „Wertx1“ und so weiter. Sind keine Wildcards enthalten, verhält sich das LIKE nahezu wie eine Gleichheitsvergleich. Manche Datenbanken führen den LIKE-Befehl ohne Beachtung von Groß- und Kleinschreibung aus. Ist die Konfiguration der Datenbank nicht bekannt, sollte man jedoch mit diesem Wunsch folgendermaßen verfahren:

```
LOWER(Feld2) LIKE LOWER('%Wert1%')
```

6.2.Gruppieren

Zu gruppieren macht dann Sinn, wenn viele Zeilen den selben Wert enthalten und die volle Anzahl nicht relevant ist.

Das Grundprinzip ist folgendes

```
SELECT
     COUNT(Feld1), Feld2
FROM
     Tabelle
WHERE
     Feld1='Wert1'
GROUP BY
     Feld2;
```

Dieses Beispiel holt alle eindeutigen Feld2-Inhalte und deren Anzahl für Datensätze, bei denen Feld1 den Wert „Wert1“ hat.

6.3.Sortieren der Ergebniszeilen

Die Ergebniszeilen kommen bei den meisten Datenbanken unsortiert, d.h. in der undefinierten Reihenfolge, zurück. Um sie gleich von vorneweg zu sortieren (alphabetisch, numerisch, nach Datum) verfährt man folgendermaßen

```
SELECT * FROM Kunden ORDER BY KuName ASC
```

ASC bedeutet aufsteigend (a->z oder 0->9). DESC würde absteigend bedeuten (z->a oder 9->0).

6.4.Anzahl der Zeilen

Zur einfachen Selektion von statistischen Daten (wieviele Kunden kommen aus Hamburg?) kann man diese einfach via SQL zählen lassen.

```
SELECT COUNT(*) FROM Kunden WHERE KuOrt='Hamburg';
```

6.5.Noch mehr Funktionen

Jeder SQL-Dialekt bringt eigene Funktionen, die die Hersteller für besonders hilfreich erachten, mit. Für die Funktionen von MySQL lohnt sich ein Blick in deren Online-Doku, die unter

http://dev.mysql.com/doc/refman/5.1/de/

zu finden ist.

6.6.Einer geht noch

Die verschiedenen COUNT, WHERE, GROUP und ORDER-Clauses kann man natürlich kombinieren. Und damit wir auch ein sinnvolles Beispiel benutzen können, haben wir damit bis zum Schluss gewartet.

Was ist das Ergebnis der folgenden Zeilen SQL?

```
SELECT
     r.ReNr, r.ReDatum, r.ReBetrag,
     k.KuNr, k.KuName, k.KuOrt,
     p.ArtNr, p.ArtName, p.ArtPreis
FROM
     Rechnungen AS r, Rechnungsdaten AS rd,
     Kunden AS k, Produkte AS p
WHERE
     r.KuNr=k.KuNr AND
     rd.ReNr=r.ReNr AND
     rd.ArtNr=p.ArtNr
ORDER BY
     r.ReNr ASC, r.ReDatum ASC, r.ReBetrag DESC;
```

Das kommt uns irgendwie bekannt vor, oder? **Das ergibt unsere erste Normalform**. Dies könnte eine Übersichtstabelle für die Buchhaltung in einer Webapplikation sein.

Wer hat die meisten Produkte gekauft - das Ranking?

```
SELECT k.KuName, COUNT( p.ArtNr ) AS Anzahl
FROM kunden k, produkte p, rechnungen r, rechnungsdaten rd
WHERE rd.ReNr = r.ReNr AND rd.ArtNr = p.ArtNr AND r.KuNr = k.KuNr
```

```
GROUP BY k.KuName
```

```
KuName    | Anzahl |
==========|========|
Fischer   | 2      |
Meier     | 4      |
Schulze   | 1      |
```

Das Schlüsselwort AS um einen Alias zu bilden, kann auch weggelassen werden.

Was ist das meist verkaufte Produkt?

```
SELECT COUNT( rd.ArtNr ) AS Anzahl , p.ArtName
FROM rechnungsdaten rd, produkte p
WHERE p.ArtNr = rd.ArtNr
GROUP  BY p.ArtNr
ORDER BY Anzahl DESC
```

Eine Spalte im Ergebnis wird absteigend sortiert.

```
Anzahl |  ArtName |
=======|==========|
3      | Prozessor|
2      | Monitor  |
1      | Speicher |
1      | Gehäuse  |
```

6.7.Primärschlüssel, Fremdschlüssel und Indizes

Die Schlüssel, die wir mit MySQL vergeben können sind Primärschlüssel und zusammengefasste Primärschlüssel. Diese sind offensichtlich, da von Hand angelegt.

Es gibt aber auch Schlüssel, die durch das Verweisen auf andere Tabellen entstehen. Wenn also ein Schlüssel aus einer anderen Tabelle verwendet wird, um eine Verbindung herzustellen, nennt man selbigen **Fremdschlüssel**.

Der **Primärschlüssel** muss eindeutig sein. Ist es ein Zusammengefasster, so muss die Kombination eindeutig sein.

Primärschlüssel sind automatisch mit einem **Index** versehen (indiziert). Das bedeutet, dass nach diesen Feldern **schneller gesucht** werden kann, als nach anderen. Man kann Indizes auch von Hand anlegen, wenn man weiß, dass ein Feld einer Tabelle ein häufiges Vergleichskriterium ist.

6.8.Views

Es könnte sein, dass wir die Ergebnisse von oben sehr häufig brauchen. Dann wäre es blöd, wenn wir sie ständig neu beschaffen müssten. Dazu gibt es Views, die **virtuelle Tabellen** generieren, auf denen man dann wieder Selects absetzen kann. Aktualisiert

man eine Quell-Tabelle, wird der View automatisch aktualisiert.

Dieses Feature gibt es bei MySQL seit **der Version 5**. Andere Datenbanken implementieren diese Funktion schon länger.

6.9.Sub-Selects

So nennt man Selects innerhalb von Selects. Dabei wird das Ergebnis eines Selects als Quelltabelle eines anderen Selects behandelt. Z.B.

```
SELECT name FROM (SELECT COUNT(autos) as autos, name WHERE
autos>10) GROUP BY name
```

oder es wird bei Existenz eines anderen Datensatzes selektiert.

```
SELECT * FROM nutzer n WHERE EXISTS (SELECT id FROM accounts a
WHERE a.id = n.id)
```

Dieses Feature ist in MySQL seit Version 4.1 verfügbar.

6.10.Sonstiges in Kürze

- Bei DELETE gibt es kein Sternchen und keine Feldnamen
- Wenn die WHERE-Clause weggelassen wird, „matchen“ alle Zeilen.
- Groß- Kleinschreibung in SQL macht in der Regel keinen Unterschied. Ausnahme: Tabellennamen und Zeichenketten.
- Maximale Zahl ermitteln: `MAX(Feld)`
- Minimale Zahl ermitteln: `MIN(Feld)`
- Zeichenkette klein schreiben: `LOWER(Feld)`
- Zeichenkette groß schreiben: `UPPER(Feld)`

7.MySQL mit PHP

Das Ansprechen einer MySQL Datenbank verläuft mit PHP folgendermaßen:

1. Verbindung zum Server aufbauen
2. Datenbank auswählen
3. Ein SQL-Kommandos ausführen
4. Das Ergebnis (Result-Set) auslesen
5. Das Result freigeben
6. 3-5 beliebig oft wiederholen
7. Verbindung schließen

Und wie sieht das im Quellcode aus? Nehmen wir ein Beispiel zum testen der Datenbankfunktion.

```
<?php
  $link = mysql_connect('localhost','root','myPass');
  mysql_select_db('mysql',$link);
  $res = mysql_query('select * from user',$link);
  if(mysql_fetch_assoc($res)) {
    echo "Datenbank vorhanden und funktionsfähig";
  } else {
    echo "Datenbank nicht vorhanden bzw. nicht funktionsfähig";
  }
?>
```

Zunächst wird mit `mysql_connect` die Verbindung zum Server hergestellt. Man erhält einen Datenbank-Handler mit dessen Hilfe man anderen MySQL-Funktionen sagen kann, welche Server-Verbindung verwendet werden soll.

So zu sehen beim nächsten Befehl, `mysql_select_db`, wo jetzt sowohl der Datenbankname als auch der Datenbank-Handler übergeben wird. Ab jetzt ist alles, was mit unserem Handler gemacht wird automatisch als „für die ausgewählte Datenbank“ zu verstehen.

Da wir jetzt eine Verbindung und eine Datenbank haben, können wir nun mit unserem Datenbank-Handler ein SQL-Statement auf der Datenbank absetzen. In unserem Fall `"select * from user"`.

Info: Die Tabelle user gehört zu MySQL und hat 31 Spalten. Sie enthält die verschiedenen Benutzer-Rechte. Uns interessieren augenblicklich nur die Felder „Host“, „User“ und „password“, so dass wir unseren Select abändern zu `"select Host, User, password from user"`.

Das obige Beispiel prüft jetzt nur noch, ob ein Ergebnis herauszuholen ist und macht in Folge dessen die Aussage über die Funktionalität der Datenbank selbst. Wir wollen ja jetzt aber diese Ergebnis-Zeilen verwenden. Was tun?

```
$res = mysql_query(
     "select Host, User, password from user",
     $link
);
```

Beispiel 1: Ein Array mit den Zeilen befüllen:

```
$alles = Array();
while($row = mysql_fetch_assoc($res)) {
     $alles[] = $row;
}
```

Beispiel 2: Ein Array mit den Zeilen befüllen und die Primärschlüssel als Key benutzen:

```
$alles = Array();
while($row = mysql_fetch_assoc($res)) {
     $alles[ $row['Host'].'_'.$row['User'] ] = $row;
}
```

Beispiel 3: Eine HTML-Tabelle direkt ausgeben:

```
echo '<table border="1">';
echo '<tr><th>HOST</th><th>USER</th><th>PASSWORD</th></tr>';
while($row = mysql_fetch_assoc($res)) {
     echo '<tr>';
     echo '<td>'.$row['Host'].'</td>';
     echo '<td>'.$row['User'].'</td>';
     echo '<td>'.$row['password'].'</td>';
     echo '</tr>';
}
echo '</table>';
```

Und abschließend geben wir jeweils das Result-Set frei und schließen (obwohl wir natürlich noch beliebig viele Kommandos absetzten könnten) jetzt auch gleich die Datenbank-Verbindung:

```
mysql_free_result($res);
mysql_close($link);
```

Manchmal möchte man **2** (oder mehr) **Selects parallel** laufen lassen. Dazu muss man einfach einen anderen Variablennamen für das Result-Set verwenden. Schon kann z.B. in einer geschachtelten Schleife der eine Fetch in der Äußeren und der andere Fetch in der inneren Schleife verarbeitet werden. **Man braucht keine zweite Datenbankverbindung**.

8.Anhang

Die formalen Beschreibungen der Befehle sind durchgängig in der (E)BNF gehalten.

Die Inhalte inklusive Bespiele und ausführlicher Erklärung sind die der MySQL-Dokumentation zu finden

http://dev.mysql.com/doc/refman/5.1/de/

8.1.Formaler Insert

Die **formale Beschreibung** eines Inserts von MySQL hat 3 mögliche Formen (die jeweils selten verwendete Features beinhalten):

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

Oder:

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

Oder:

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    SELECT ...
```

8.2.Formaler Update

Die **formale Beschreibung** eines Updates von MySQL hat 2 mögliche Formen:

Eine-Tabelle Syntax:

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

Mehrere-Tabellen Syntax:

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

8.3.Formaler Select

Die **formale Beschreibung** eines Selects von MySQL gestaltet sich etwas ausführlicher, wobei vieles davon nicht sehr häufig Verwendung findet:

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr, ...
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name']
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

Wir sehen also schon, dass hier einiges möglich ist. In der Praxis lässt man gern die Daten von der Datenbank vorverarbeiten, da diese intern im Vergleich zu einer Scriptsprache wesentlich schneller ist.

Deswegen wird am Ende des Kapitels noch weiter auf dieses Thema eingegangen.

8.4.Formaler Delete

Die **formale Beschreibung** eines Deletes von MySQL hat 3 mögliche Formen

Eine-Tabelle Syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
       [WHERE where_definition]
       [ORDER BY ...]
       [LIMIT row_count]
```

Mehrere-Tabellen Syntax:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
       tbl_name[.*] [, tbl_name[.*] ...]
       FROM table_references
       [WHERE where_condition]
```

Oder:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
       FROM tbl_name[.*] [, tbl_name[.*] ...]
       USING table_references
```

8.5.Formaler Create

Erzeugen einer Datenbank:

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification [, create_specification] ...]

create_specification:
    [DEFAULT] CHARACTER SET charset_name
  | [DEFAULT] COLLATE collation_name
```

Erzeugen einer Tabelle:

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

create_definition:
  spalten_name typ [NOT NULL | NULL] [DEFAULT vorgabe_wert]
    [AUTO_INCREMENT]
         [PRIMARY KEY] [referenz_definition]
  oder   PRIMARY KEY (index_spalten_name,...)
  oder   KEY [index_name] (index_spalten_name,...)
  oder   INDEX [index_name] (index_spalten_name,...)
  oder   UNIQUE [INDEX] [index_name] (index_spalten_name,...)
  oder   FULLTEXT [INDEX] [index_name] (index_spalten_name,...)
  oder   [CONSTRAINT symbol] FOREIGN KEY index_name
           (index_spalten_name,...)
           [referenz_definition]
```

```
  oder   CHECK (ausdruck)
```

```
CREATE [UNIQUE|FULLTEXT] INDEX index_name ON tabelle
(spalten_name[(laenge)],... )
```

8.6.Formaler Drop

```
DROP DATABASE [IF EXISTS] datenbank
```

```
DROP TABLE [IF EXISTS] tabelle [, tabelle,...] [RESTRICT |
CASCADE]
```

```
DROP INDEX index_name ON tabelle
```

8.7.Formaler Alter

```
ALTER [IGNORE] TABLE tabelle aenderungs_angabe [,
aenderungs_angabe ...]


aenderungs_angabe:
        ADD [COLUMN] create_definition [FIRST | AFTER
spalten_name]
  oder    ADD [COLUMN] (create_definition, create_definition,...)
  oder    ADD INDEX [index_name] (index_spalten_name,...)
  oder    ADD PRIMARY KEY (index_spalten_name,...)
  oder    ADD UNIQUE [index_name] (index_spalten_name,...)
  oder    ADD FULLTEXT [index_name] (index_spalten_name,...)
  or ADD [CONSTRAINT symbol] FOREIGN KEY index_name
(index_spalten_name,...)
            [referenz_definition]
  oder    ALTER [COLUMN] spalten_name {SET DEFAULT literal | DROP
DEFAULT}
  oder    CHANGE [COLUMN] alter_spalten_name create_definition
  oder    MODIFY [COLUMN] create_definition
  oder    DROP [COLUMN] spalten_name
  oder    DROP PRIMARY KEY
  oder    DROP INDEX index_name
  oder    DISABLE KEYS
  oder    ENABLE KEYS
  oder    RENAME [TO] neue_tabelle
  oder    ORDER BY spalte
```

```
oder    tabellen_optionen
```