Stored Procedures in MySQL 5

Inhalt:

1. Einleitung

Diese Tutorial zeigt den Einsatz von "**Stored Procedures**" und "**Stored Functions**", die bei MySQL ab der Version 5.0.0 zur Verfügung stehen. Vereinfacht gesagt kann man verschiedene SQL-Kommandos in einer SQL-Prozedur oder einer SQL-Funktion zusammenfassen. Dadurch, dass diese Prozedur bzw. Funktion auf dem Datenbank-Server gespeichert wird, genügt künftig ein einfacher Aufruf der Prozedur, ohne jedes Mal alle Kommandos neu eingeben zu müssen. Endlich braucht man nicht mehr leidig in Richtung PostgreSQL schielen, das zwar ebenfalls frei erhältlich ist, aber leider auf wenigen Servern zur Verfügung steht. Meine Testplattform war MySQL 5.0.0alpha unter Slackware 9.1.
Dieses Tutorial ist gedacht für bereits geübtere MySQL-Anwender, die ihr Wissen auf professionelle Techniken erweitern wollen als auch für solche, die SQL-Procedures bereits von anderen DBMS her kennen und sich schnell in die MySQL-Besonderheiten einlesen wollen. Das hier erworbene Wissen ist grundsätzlich auch auf andere DBMS übertragbar, sofern man deren Eigenheiten berücksichtigt.
Wann macht eine Verwendung von Stored Procedures eigentlich Sinn?

- Durch "Stored Procedures" kann ein *Gewinn an Performance* erreicht werden. Da die Prozedur oder Funktion wie gesagt auf dem Datenbank-Server gespeichert wird, verringert sich der Traffic zwischen dem Client oder Aufrufer und dem Verlust durch Syntax-Checks. Andererseits erhöht sich die Belastung des Datenbank-Servers, wenn in den Procedures Berechnungen durchgeführt werden, die sonst von einer darüber liegenden Programmiersprache wie PHP, Java oder C++ erledigt würden. Hier muss man sorgfältig Kosten und Nutzen abwägen.
- Man erreicht eine größere *Unabhängigkeit von der Plattform und der Programmiersprache*. Nehmen wir an, in einem Unternehmen wird eine Datenbank von mehreren Anwendungen benutzt: ein Webserver mit PHP unter Linux, eine Eingabemaske in C++ unter Windows, ein Kontrollskript des Administrators in Perl unter Linux. Wenn in diesen Programmen jeweils die selbe Abfrageoperation vorhanden sein soll, könnte man entweder drei Mal die selbe Funktion schreiben (was zudem fehleranfällig ist) oder eine SQL-Prozedur anlegen und diese von den Programmen aufrufen lassen. Der

Vorteil wird noch evidenter, wenn eine solche Prozedur geändert werden soll oder häufig weitere hinzukommen sollen.
- Die *Sicherheit wird erhöht*. Durch "Stored Procedures" kann vor dem Client einiges versteckt werden, z.B. Tabellen- und Spaltennamen. Außerdem kann erreicht werden, dass weniger (sensible) Daten zwischen Client und Server hin und her gesendet werden.

2. Stored Procedures

Um unter MySQL 5 eine Prozedur anzulegen, dient das SQL-Statement **CREATE PROCEDURE**. Es hat folgenden Aufbau:

```
CREATE PROCEDURE procedure_name ([parameter]) [characteristics]
BEGIN
... diverse Anweisungen ...
END
```

Eine Prozedur kann keinen, einen oder mehrere Parameter übernehmen. Die Syntax für jeden Parameter lautet so:

```
[IN / OUT / INOUT] parameter_name DATENTYP
```

Wird eine Parameter als **IN** deklariert, liefert er nur Input für die Prozedur, liefert aber keinen Output zurück. Im Gegensatz dazu geben **OUT**-Parameter keinen Input, holen aber ein etwaiges Ergebnis aus der Prozedur zurück. Default-Wert ist IN. DATENTYP kann jeder gültige MySQL-Datentyp sein, z.B. INT, FLOAT, DECIMAL, TIMESTAMP, BLOB, TEXT, VARCHAR(50) usw.

Zwischen Parameterliste und BEGIN können noch eine oder mehrere Charakteristiken angegeben werden (dies ist optional):
- **LANGUAGE SQL**
spezifiziert die Sprache, in der die Prozedur geschrieben wird. Bis jetzt steht nur SQL zur Verfügung. Später wird es auch möglich sein, Prozeduren in anderen Sprachen zu erstellen, z.B. in PHP.
- **SQL SECURITY {DEFINER / INVOKER}**
bestimmt, wessen Berechtigungen (z.B. zum Abfragen oder Ändern einer Tabelle) für die Ausführung der Prozedur berücksichtigt werden soll: dessen, der sie angelegt hat (DEFINER) oder dessen, der sie aufrufen will (INVOKER). Da MySQL noch nicht das **GRANT EXECUTE**-Statement kennt, kann die Prozedur nur ausgeführt werden, wenn die entsprechenden Rechte für alle Tabellen, die in der Routine einbezogen werden, vorhanden sind. Default-Wert ist DEFINER.
- **COMMENT**
hiermit kann ein Kommentar einer Prozedur hinzugefügt werden. Dies ist eine Erweiterung speziell von MySQL und gehört nicht zum SQL-Standard.

Sehen wir uns nun ein einfaches Beispiel an (dazu starten wir die MySQL-Konsole und verbinden uns mit der Datenbank test):

```
mysql> DROP TABLE IF EXISTS testusers;
mysql> DROP PROCEDURE IF EXISTS logincheck;
mysql> CREATE TABLE testusers (username VARCHAR(20) NOT NULL,
userpassword CHAR(40) NOT NULL, PRIMARY KEY(username));
mysql> INSERT INTO testusers VALUES ('Peter', SHA1('abcdefg'));
mysql> INSERT INTO testusers VALUES ('Thomas', SHA1('zyxzyx'));

mysql> delimiter |
mysql> CREATE PROCEDURE logincheck (uname VARCHAR(20), upwd
CHAR(40), OUT isvalid INT)
    -> SQL SECURITY INVOKER
    -> COMMENT 'ueberprueft login'
    -> BEGIN
    ->   SELECT COUNT(*) INTO isvalid FROM testusers WHERE username =
uname AND userpassword = upwd;
    -> END|
```

Für dieses Beispiel legen wir eine Tabelle "testusers" an, das der Einfachheit halber nur 3 Spalten hat: eine ID, einen Benutzernamen und das zugehörige Passwort als SHA-Wert. Danach fügen wir zwei einfache Zeilen ein. Zuletzt wird eine Prozedur definiert: sie soll testen, ob Name und Passwort, bspw. für einen Login, korrekt sind. Die Prozedur hat drei Parameter: zum Einen Benutzername und Passwort, zum Anderen eine Variable, die das Abfrageergebnis (die gefundenen Zeilen zu uname und upwd) aufnehmen soll. Da dieser letzte Parameter keine Eingabe für die Prozedur ist, sondern ein Ergebnis zurück liefert, definieren wir: **OUT isvalid INT**. Weiterhin legen wir fest, dass die Prozedur nur ausgeführt werden darf, wenn der aufrufende Benutzer Leserechte auf der Tabelle testusers hat, und fügen einen Kommentar hinzu. Die eigentliche Routine steht zwischen **BEGIN** und **END** und umfasst in diesem sehr einfachen Beispiel nur einen Query. Hier lernen wie auch gleich einen neuen Befehl kennen: das **SELECT INTO**-Statement.

```
SELECT spalte[,...] INTO variable[,...] FROM tabellenname [bedingungen]
```

Diese Abfrage nimmt den Inhalt einer oder mehrerer Spalten und schreibt sie direkt in eine oder mehrere Variablen, mit denen dann weiter gearbeitet werden kann. Logischerweise kann dieser Query nur die Werte der ersten Zeile abfangen. Sollen nacheinander alle Zeilen in Variablen gespeichert werden, ist ein Cursor notwendig (dazu später mehr).
Hinweis: Da Prozeduren und Funktionen auf dem Server ähnlich wie Tabellen gespeichert werden, dürfen sie nur einmal angelegt werden. Ein weiterer Versuch, eine Prozedur mit gleichem Namen zu erzeugen, würde mit einer Fehlermeldung abbrechen. Deshalb löschen wir vorher eine eventuell vorhandene Prozedur. Das geht, analog dem DROP TABLE-Statement, mit **DROP PROCEDURE**. **IF EXISTS** prüft, ob bereits eine solche Prozedur gespeichert ist, und führt nur in diesem Fall den DROP-Befehl aus (es ist eine MySQL-Erweiterung und gehört nicht zum SQL-Standard).
Näherer Erklärung wird die Zeile "**delimiter |**" bedürfen: da wir in der Prozedur einen vollständigen Query nach wie vor mit ; abschließen müssen, die Eingabe der Prozedur aber damit nicht zu Ende ist, müssen wir einen anderen Delimiter zum Abschließen der Eingabe wählen, bspw. "|". Legen wir eine Procedure von einer darüber liegenden Skriptsprache wie PHP an, fassen wir die Procedure in einem String zusammen und haben dann dieses Problem nicht:

```
$proc1 = "CREATE PROCEDURE logincheck (uname VARCHAR(20), upwd
CHAR(40), OUT isvalid INT) " .
        "SQL SECURITY INVOKER " .
          "COMMENT 'ueberprueft login' " .
          "BEGIN " .
          "SELECT COUNT(*) INTO isvalid FROM testusers WHERE username =
uname AND userpassword = upwd; " .
          "END";
mysql_query($proc1, $conn) or die(mysql_error());
```

Sehen wir uns nun an, wie wir die Stored Procedure aufrufen können:

```
mysql> delimiter ;
mysql> CALL logincheck('Thomas', SHA1('zyxzyx'), @res);
mysql> SELECT @res AS passed;
```

Wie wir in diesem Beispiel sehen, wird die Stored Procedure mit dem SQL-Befehl **CALL** aufgerufen. Das Ergebnis der Operation wird in der Variable **@res** gespeichert (Variablen in SQL werden durch vorangestellten Klammeraffen gekennzeichnet; diese gelten global). Der Wert der Variable kann mit einem einfachen SELECT-Statement abgefragt werden.

3. Stored Functions

Um eine Funktion anzulegen, dient das SQL-Statement **CREATE FUNCTION**. Es hat folgenden Aufbau:

```
CREATE FUNCTION function_name ([parameter])
RETURNS [return_type]
[characteristics]
BEGIN
... diverse Anweisungen ...
END
```

Eine Funktion kann wie eine Prozedur keinen, einen oder mehrere Parameter übernehmen. Der einzige Unterschied besteht darin, dass es bei Funktionen keine IN-, OUT- oder INOUT-Parameter gibt. Alle Parameter dienen nämlich grundsätzlich nur als Input. Nach der Parameterliste wird der Datentyp der Rückgabe mit dem Statement **RETURNS** spezifiziert: wird aus der Funktion eine Integer-Zahl zurückgegeben, schreibt man bspw. RETURNS INT. Soll ein Datum zurückgegeben werden, schreibt man RETURNS DATE usw. Alle gültigen MySQL-Datentypen sind erlaubt. Vor der eigentlichen Funktion zwischen BEGIN und END können wieder optional Charakteristiken angegeben werden. Hier gibt es außer den drei schon bei Prozeduren genannten Statements zusätzlich die Spezifikation **DETERMINISTIC**. Dieses Statement kann man angeben, wenn man weiß, dass zu einem bestimmten Input jedes Mal dasselbe Ergebnis zurückgegeben wird. Diese Angabe dient zur weiteren Optimierung der Funktion, hat aber derzeit noch keine Auswirkungen (MySQL 5.0.0 alpha). Entsprechend gibt es auch noch die Angabe **NOT DETERMINISTIC**.
Wichtig: In einer Funktion sind *keine Abfragen oder Änderungen an Tabellen erlaubt* (also kein SELECT, INSERT, DROP oder Artverwandtes)!

Sehen wir uns wieder ein einfaches Beispiel an. Diesmal wollen wir eine Funktion "encipher" definieren. Sie soll einen Text anhand eines 16-Byte langen Schlüssels mit AES verschlüsseln und als Hex-String formatiert zurück liefern (dazu verwenden wir das Statement **AES_ENCRYPT**).

```
mysql> delimiter |
mysql> DROP FUNCTION IF EXISTS encipher;
mysql> CREATE FUNCTION encipher (plaintext TEXT, userkey CHAR(16))
    -> RETURNS TEXT
    -> BEGIN
    ->   RETURN HEX(AES_ENCRYPT(plaintext, userkey));
    -> END|
```

Zunächst wird die Funktion encipher, sofern sie schon existiert, gelöscht. Dies geschieht, analog zu Prozeduren, mit dem SQL-Statement **DROP FUNCTION**. Die Funktion selbst dürfte nach unserem jetzigen Wissenstand bereits selbsterklärend sein. Deshalb soll der Hinweis genügen, dass das Ergebnis der Funktion - wie zu erwarten war - mit **RETURN** zurückgegeben wird.
Die Funktion wird übrigens nicht mit CALL, sondern mit SELECT aufgerufen, da sie direkt ein Ergebnis zurück liefert. Siehe dazu dieses kleine Beispiel:

```
mysql> delimiter ;
mysql> SELECT encipher('hallo...', 'unsecure userkey') AS ciphertext;
```

4. Procedures ändern und anzeigen lassen

Zum Ändern einer Stored Procedure dient das **ALTER PROCEDURE** bzw. das **ALTER FUNCTION**-Statement. Mit ihr kann man eine Prozedur bzw. Funktion umbenennen und/oder die Sicherheitseinstellung ändern und/oder einen neuen Kommentar setzen. SQL-Syntax:

```
ALTER PROCEDURE procname [NAME 'name_neu' | SQL SECURITY {INVOKER
| DEFINER} | COMMENT 'neuer Kommentar'];

ALTER FUNCTION funcname [NAME 'name_neu' | SQL SECURITY {INVOKER |
DEFINER} | COMMENT 'neuer Kommentar'];
```

Als Beispiel benennen wir die Funktion "encipher" um in "aes_encipher" und fügen den Kommentar "AES-Verschluesselung" hinzu:

```
mysql> ALTER FUNCTION encipher NAME aes_encipher COMMENT 'AES-
Verschluesselung';
```

Um sich noch mal anzeigen zu lassen, wie die Prozedur bzw. Funktion genau angelegt wurde, dient das **SHOW CREATE PROCEDURE** bzw. **SHOW CREATE FUNCTION**-Statement. Dies ist eine MySQL-Erweiterung und nicht Bestandteil des SQL-Standards. SQL-Syntax:

```
SHOW CREATE PROCEDURE procname;
SHOW CREATE FUNCTION funcname;
```

Beispiel für unsere Prozedur "logincheck":

```
mysql> SHOW CREATE PROCEDURE logincheck;
```

Die Statements **SHOW PROCEDURE STATUS** und **SHOW FUNCTION STATUS** liefern eine Auflistung alle Prozeduren bzw. Funktionen samt folgender Angaben: Name, Typ (FUNCTION oder PROCEDURE), Name des Erstellers, Zeitpunkt der letzten Änderung, Zeitpunkt der Erstellung, Security-Type (DEFINER oder INVOKER), Kommentar. Dies ist ebenfalls eine MySQL-Erweiterung und gehört nicht zum SQL-Standard. SQL-Syntax:

```
SHOW PROCEDURE STATUS [LIKE pattern];
SHOW FUNCTION STATUS [LIKE pattern];
```

Mit **LIKE** lässt sich die Anzeige auf bestimmte Namen eingrenzen. Als Beispiel lassen wir uns alle Prozeduren anzeigen, außerdem alle Funktionen, deren Namen auf "encipher" endet:

```
mysql> SHOW PROCEDURE STATUS;
mysql> SHOW FUNCTION STATUS LIKE '%encipher';
```

5. Cursor, Ablaufsteuerung, DECLARE-Statements

Bevor wir zur eigentlichen Aufgabe kommen, leisten wir einige kleine Vorarbeiten, die wir für unser Beispiel brauchen, und legen drei Tabellen an (items1, items2 und items3), die jeweils dieselben Artikel enthalten, allerdings zu unterschiedlichen Preisen:

```
mysql> CREATE TABLE items1 (itemname VARCHAR(100) NOT NULL, price
FLOAT NOT NULL, PRIMARY KEY(itemname));
mysql> INSERT INTO items1 VALUES('wine', 3.99);
mysql> INSERT INTO items1 VALUES('cheese', 2.35);
mysql> INSERT INTO items1 VALUES('pork', 4.09);

mysql> CREATE TABLE items2 (itemname VARCHAR(100) NOT NULL, price
FLOAT NOT NULL, PRIMARY KEY(itemname));
mysql> INSERT INTO items2 VALUES('wine', 3.49);
mysql> INSERT INTO items2 VALUES('cheese', 2.60);
mysql> INSERT INTO items2 VALUES('pork', 4.49);

mysql> CREATE TABLE items3 (itemname VARCHAR(100) NOT NULL, price
FLOAT NOT NULL, PRIMARY KEY(itemname));
mysql> INSERT INTO items3 VALUES('wine', 4.19);
mysql> INSERT INTO items3 VALUES('cheese', 2.40);
mysql> INSERT INTO items3 VALUES('pork', 3.79);

mysql> CREATE TABLE cheapitems (itemname VARCHAR(100) NOT NULL, price
FLOAT NOT NULL, PRIMARY KEY(itemname));
```

Da das folgende Beispiel viele neue Konstrukte erhält, wollen wir es schrittweise entwickeln und erst zum Schluss in MySQL eingeben.

Unsere Aufgabe soll sein, eine Procedure zu schreiben, die Folgendes leistet:
- Zu jedem Artikel soll das Billigste der drei Angebote ermittelt werden.
- Dieses wird dann in eine neue Tabelle "cheapitems" geschrieben, aber nur, wenn es mehr als 3,50 Euro kostet.
Wir wollen die Procedure "mergelists" nennen. Der Rumpf unserer Prozedur:

```
CREATE PROCEDURE mergelists()
BEGIN

END
```

5.1. DECLARE-Statements

DECLARE-Statements müssen ganz am Anfang einer Prozedur, also unmittelbar nach BEGIN angegeben werden. Mit ihnen lässt sich Folgendes bewerkstelligen:

- Deklaration lokaler Variablen
- Deklaration von Bedingungen (Conditions) und Handlern zur Behandlung von Error-Codes, Warnungen usw.
- Deklaration von Cursors

Überlegen wir uns zuerst, welche lokalen Variablen wir brauchen. Wir wissen, dass wir die drei Preise der verschiedenen Anbieter vergleichen müssen, welche alle vom Typ FLOAT sind. Des Weiteren soll das Ergebnis dieses Vergleichs in einer Variable "bestprice" gespeichert werden. Wir brauchen also folgende Variablen:

- Den Namen des Guts: "iname" mit Typ VARCHAR(100)
- Die einzelnen Preise: "price1", "price2", "price3" und "bestprice", jeweils vom Typ FLOAT.

Das Anlegen von lokalen Variablen ist sehr einfach. SQL-Syntax:

```
DECLARE variablenname1 [, variablenname2, ...] Typ [DEFAULT wert];
```

Für unser Beispiel also:

```
DECLARE iname VARCHAR(100);
DECLARE price1, price2, price3, bestprice FLOAT;
```

Als Nächstes brauchen wir Cursors, um - vereinfacht gesagt - sequenziell auf die entsprechenden Daten in jeder Tabelle zugreifen zu können. Die SQL-Syntax:

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

An die Stelle von sql-statement wird eine einfache Datenbankabfrage gesetzt, die uns die Daten liefern würde, auf die wir hier mittels Cursor zugreifen wollen. Für unser Beispiel brauchen wir drei Cursor, wobei sie jeweils die Spalte "price" abfragen sollen (und der Erste zusätzlich den Namen des Guts, welcher ja auch in der neuen Tabelle gespeichert werden soll):

```
DECLARE cur1 CURSOR FOR SELECT itemname, price FROM items1;
DECLARE cur2 CURSOR FOR SELECT price FROM items2;
DECLARE cur3 CURSOR FOR SELECT price FROM items3;
```

An dieser Stelle greife ich gleich einmal vorweg: Wenn die Cursor an das Ende der Tabellen stoßen, wird ein SQL-Status vom Typ **NOT FOUND** ausgelöst (der genaue Code ist '02000'). Wenn dies der Fall ist, ist unsere Aufgabe beendet und die Cursor müssen wieder geschlossen werden.
Dazu deklarieren wir eine Variable "done" vom Typ BOOLEAN, die wir mit FALSE initiieren und auf TRUE setzen, sobald dieser Fall eintritt, und die Schleife, die wir nachher noch einsetzen müssen, abbricht. Um die Variable genau beim SQL-Status '02000' auf TRUE zu setzen, definieren wir einen Handler. SQL-Syntax:

```
DECLARE handlertyp HANDLER FOR condition sql-statement;
```

An der Stelle von "handlertyp" kann stehen:

- **CONTINUE**: Ein Continue-Handler führt zwar den unter sql-statement stehenden Befehl aus, bricht aber die Prozedur nicht ab
- **EXIT**: Die Prozedur wird sofort abgebrochen
- **UNDO**: Wird momentan noch nicht unterstützt und verhält sich wie ein Continue-Handler

An der Stelle von "condition" kann stehen:

- **SQLSTATE [VALUE]** 'nummer': Handler wird ausgeführt, wenn ein SQL-Status mit der entsprechenden Nummer eintritt
- **SQLWARNING**: Fängt alles SQL-States ab, die mit 01 beginnen
- **NOT FOUND**: Fängt alle SQL-States ab, die mit 02 beginnen
- **SQLEXCEPTION**: Fängt alle übrigen SQL-States ab

Außerdem kann an dieser Stelle die Zahl eines MySQL-Error-Codes stehen oder der Name einer selbst definierten Condition. Letztere wird ebenso über ein DECLARE-Statement angelegt. SQL-Syntax:

```
DECLARE condition_name CONDITION FOR condition;
```

Unser Handler soll die Prozedur nicht abbrechen und beim SQL-State '02000' die Variable "done" auf TRUE setzen. Es ergibt sich:

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

Mit einer selbst definierten Condition könnte man auch schreiben (Ergebnis ist absolut gleich):

```
DECLARE my_condition CONDITION FOR SQLSTATE '02000';
DECLARE CONTINUE HANDLER FOR my_condition SET done = TRUE;
```

Fassen wir alles zusammen, sieht unsere Procedure jetzt so aus (die Einrückung dient allein der besseren Lesbarkeit):

```
CREATE PROCEDURE mergelists()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = TRUE;
  DECLARE cur1 CURSOR FOR SELECT itemname, price FROM items1;
  DECLARE cur2 CURSOR FOR SELECT price FROM items2;
```

```
  DECLARE cur3 CURSOR FOR SELECT price FROM items3;
  DECLARE iname VARCHAR(100);
  DECLARE price1, price2, price3, bestprice FLOAT;
END
```

5.2. Einsatz des Cursors

Bevor wir mit dem Cursor Daten aus den einzelnen Tabellen lesen können, muss er geöffnet werden. SQL-Syntax:

```
OPEN cursorname;
```

Ebenso muss er nach Gebrauch wieder geschlossen werden. SQL-Syntax:

```
CLOSE cursorname;
```

Nun fehlt uns eigentlich nur noch das Statement, mit dem wir den nächsten Datensatz auslesen und den Cursor-Zeiger einen Datensatz weiter setzen können. Dies geschieht über ein **FETCH**-Statement. SQL-Syntax:

```
FETCH cursorname INTO variavlenname1 [, variablenname2, ...];
```

Es wird schnell klar, dass die FETCH-Statements so oft ausgeführt werden müssen, wie Datensätze in den Tabellen vorhanden sind. Dazu stellt uns SQL Schleifen zur Ablaufsteuerung zur Verfügung. Jetzt wollen wir in unser Beispiel eine while-Schleife einbauen (welche so lange durchlaufen werden soll, wie "done" den Wert FALSE hat), zusätzlich zur bereits erlernten Cursor-Steuerung:

```
CREATE PROCEDURE mergelists()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = TRUE;
  DECLARE cur1 CURSOR FOR SELECT itemname, price FROM items1;
  DECLARE cur2 CURSOR FOR SELECT price FROM items2;
  DECLARE cur3 CURSOR FOR SELECT price FROM items3;
  DECLARE iname VARCHAR(100);
  DECLARE price1, price2, price3, bestprice FLOAT;
  OPEN cur1;
  OPEN cur2;
  OPEN cur3;

  WHILE NOT done DO
    FETCH cur1 INTO iname, price1;
    FETCH cur2 INTO price2;
    FETCH cur3 INTO price3;
  END WHILE;

  CLOSE cur1;
  CLOSE cur2;
  CLOSE cur3;
END
```

Die while-Schleife ähnelt sehr bekannten Programmiersprachen. SQL-Syntax:

```
[labelname:] WHILE bedingung DO
  sql-statements
END WHILE [labelname];
```

Was uns jetzt noch fehlt, ist die Ermittlung des günstigsten Preises sowie das Schreiben eines neuen Datensatzes in "cheapitems", falls er weniger als 3,50 Euro beträgt. Während wir den geringsten Preis einfach über das **LEAST**-Statement ermitteln können, brauchen wir zur Überprüfung der 3,50-Grenze ein weiteres Element der Ablaufsteuerung: das **IF**-Statement. Es ähnelt wiederum den Konstrukten bekannter Programmiersprachen. SQL-Syntax:

```
IF bedingung THEN
  sql-statements
[ESLEIF bedingung THEN sql-statements]
[ELSE sql-statements]
END IF;
```

Mit diesem Wissen ergänzen wir unser Beispiel wie folgt:

```
CREATE PROCEDURE mergelists()
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = TRUE;
  DECLARE cur1 CURSOR FOR SELECT itemname, price FROM items1;
  DECLARE cur2 CURSOR FOR SELECT price FROM items2;
  DECLARE cur3 CURSOR FOR SELECT price FROM items3;
  DECLARE iname VARCHAR(100);
  DECLARE price1, price2, price3, bestprice FLOAT;
  OPEN cur1;
  OPEN cur2;
  OPEN cur3;

  WHILE NOT done DO
    FETCH cur1 INTO iname, price1;
    FETCH cur2 INTO price2;
    FETCH cur3 INTO price3;
    SET bestprice = LEAST(price1, price2, price3);
    IF bestprice > 3.50 THEN
      INSERT INTO cheapitems VALUES(iname, bestprice);
    END IF;
  END WHILE;

  CLOSE cur1;
  CLOSE cur2;
  CLOSE cur3;
END
```

Bevor wir die Prozedur endgültig in SQL eingeben, sei noch auf eine Falle hingewiesen. Nehmen wir an, wir haben mit den Cursors den jeweils letzten Datensatz ausgelesen, der SQL-Status ist aber noch nicht auf NOT FOUND gestellt, da der Pointer der Cursor ja noch nicht über das Ende der Datensätze hinaus

bewegt wurde. Dies bedeutet, dass die **WHILE**-Schleife *noch einmal* passiert wird, die Cursor jetzt keinen Datensatz mehr finden und *jetzt erst* der SQL-Status auf '02000' gesetzt wird (und damit die Variable "done" auf TRUE). Da die Variablen iname, price1, price2 und price3 keinen neuen Wert enthalten, sondern den des vorherigen Datensatzes, muss das Schreiben in "cheapitems" übersprungen werden. Wir hätten eine *Verletzung des Primärschlüssels*, was zu einem Abbruch mit dem Error-Code 1062 (SQL-State '23000') führt. Wir kapseln ergo die Berechnung des günstigsten Preises und des Schreibens des Datensatzes in einen weiteren **IF**-Block, der den Wert von "done" überprüft. Jetzt ist die Prozedur auch fertig zur Eingabe:

```
mysql> delimiter |
mysql> CREATE PROCEDURE mergelists()
    -> BEGIN
    ->   DECLARE done BOOLEAN DEFAULT FALSE;
    ->   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done =
TRUE;
    ->   DECLARE cur1 CURSOR FOR SELECT itemname, price FROM items1;
    ->   DECLARE cur2 CURSOR FOR SELECT price FROM items2;
    ->   DECLARE cur3 CURSOR FOR SELECT price FROM items3;
    ->   DECLARE iname VARCHAR(100);
    ->   DECLARE price1, price2, price3, bestprice FLOAT;
    ->   OPEN cur1;
    ->   OPEN cur2;
    ->   OPEN cur3;
    ->
    ->   WHILE NOT done DO
    ->     FETCH cur1 INTO iname, price1;
    ->     FETCH cur2 INTO price2;
    ->     FETCH cur3 INTO price3;
    ->     IF NOT done THEN
    ->       SET bestprice = LEAST(price1, price2, price3);
    ->       IF bestprice > 3.50 THEN
    ->         INSERT INTO cheapitems VALUES(iname, bestprice);
    ->       END IF;
    ->     END IF;
    ->   END WHILE;
    ->
    ->   CLOSE cur1;
    ->   CLOSE cur2;
    ->   CLOSE cur3;
    -> END|

mysql> delimiter ;
mysql> CALL mergelists();
mysql> SELECT * FROM cheapitems;
```

5.3. Weitere Konstrukte zur Ablaufsteuerung

Zum Abschluss dieses Tutorials sollen neben den WHILE- und IF-Statements noch weitere Anweisungen zur Ablaufsteuerung gezeigt werden. Als Beispiel nehmen wir wieder die Procedure "mergelists" und schreiben jeweils die WHILE-Schleife um.

Als Ersatz für IF-Statements bietet sich das **CASE**-Statement an. Es liegt in zwei Varianten vor. SQL-Syntax:

```
CASE variable
  WHEN ein_wert THEN
    sql-statements
  [WHEN anderer_wert THEN sql-statements]
  [ELSE sql-statements]
END CASE;

CASE
  WHEN bedingung THEN
    sql-statements
  [WHEN andere_bedingung THEN sql-statements]
  [ELSE sql-statements]
END CASE;
```

Die erste Variante gleicht den switch-case-Blöcken aus Java, C++ usw. Es wird der Wert einer Variable abgefragt und je nach vorgegebenen Werten ein bestimmter Programmabschnitt ausgeführt. Die zweite Variante gleicht mehr einem normalen if-Block: der case-Block ist nicht an eine Variable gebunden, stattdessen steht hinter **WHEN** ein beliebiger logischer Ausdruck, der einen booleschen Wert zurück gibt. Jetzt ersetzen wir in "mergelists" alle IF-Statements durch CASE-Statements:

```
WHILE NOT done DO
  FETCH cur1 INTO iname, price1;
  FETCH cur2 INTO price2;
  FETCH cur3 INTO price3;

  CASE done
    WHEN NOT true THEN
      SET bestprice = LEAST(price1, price2, price3);

      CASE
        WHEN bestprice > 3.50 THEN
          INSERT INTO cheapitems VALUES(iname, bestprice);
      END CASE;

  END CASE;
END WHILE;
```

Selbstverständlich gibt es neben dem WHILE-Statement noch andere Möglichkeiten, eine Schleife zu konstruieren. Im Folgenden sehen wir uns eine Schleife mit dem **REPEAT**-Statement an. SQL-Syntax:

```
[label:] REPEAT
  sql-statements
UNTIL bedingung
END REPEAT;
```

Diese Schleife wird so lange ausgeführt, wie "bedingung" *nicht* erfüllt ist. Tauschen wir in "mergelists" nun die while-Schleife durch eine repeat-until-Schleife aus:

```
REPEAT
  FETCH cur1 INTO iname, price1;
  FETCH cur2 INTO price2;
  FETCH cur3 INTO price3;
  IF NOT done THEN
    SET bestprice = LEAST(price1, price2, price3);
    IF bestprice > 3.50 THEN
      INSERT INTO cheapitems VALUES(iname, bestprice);
    END IF;
  END IF;
UNTIL done
END REPEAT;
```

Als Letztes sehen wir uns Schleifen an, die mit dem **LOOP**-Statement gebildet werden. SQL-Syntax:

```
[label:] LOOP
  sql-statements
END LOOP [label];
```

Um die Schleife steuern zu können, brauchen wir zwei weitere Statements. Um die Schleife erneut aufzurufen, dient das **ITERATE**-Statement. SQL-Syntax:

```
ITERATE label;
```

Und um die Schleife zu verlassen, dient das **LEAVE**-Statement (entspricht in etwa einer break-Anweisung in Java, C++, PHP etc.). SQL-Syntax:

```
LEAVE label;
```

Da sowohl **ITERATE** als auch **LEAVE** einen **Labelnamen** erwarten, müssen wir der loop-Schleife auch ein Label zuweisen (das wir einfach "my_label" nennen). Und wieder schreiben wir "mergelists" um:

```
my_label: LOOP
  FETCH cur1 INTO iname, price1;
  FETCH cur2 INTO price2;
  FETCH cur3 INTO price3;
  IF done THEN
    LEAVE my_label;
  END IF;
  SET bestprice = LEAST(price1, price2, price3);
  IF bestprice > 3.50 THEN
    INSERT INTO cheapitems VALUES(iname, bestprice);
  END IF;
  ITERATE my_label;
END LOOP my_label;
```

Während man mit LEAVE jedwede Konstruktion zur Ablaufsteuerung verlassen kann, kann ITERATE (logischerweise) nur innerhalb LOOP, WHILE und REPEAT-Statements stehen.

6. Fazit

MySQL 5 geht konsequent den Weg weiter, der mit MySQL 4 (u.a. volle Unterstützung von Transaktionen und des InnoDB-Formats) und MySQL 4.1 (u.a. Sub-Queries) eingeschlagen wurde: Den Wünschen auch anspruchsvoller Benutzer zu genügen. Auch wenn noch ein paar Details fehlen, so scheint die Entwicklung für die Zukunft sehr viel versprechend. Künftig sind für komplexe Datenbanken und Anwendungen keine kommerziellen Lösungen mehr notwendig.
Zum Abschluss sei noch darauf hingewiesen, dass meine Beispiele auf einer Alpha-Version getestet wurden. Bis zur stabilen Version von MySQL 5 kann noch etwas Zeit vergehen. Bis dahin sollten aber, eventuell verbunden mit Detailänderungen, die meisten noch auftretenden Bugs beseitigt werden. Wer Trigger (angekündigt für MySQL 5.1) braucht oder aber SQL-Procedures in einer anderen Sprache wie Perl oder Python schreiben will, schaut vorerst noch in die Röhre und muss zu PostgreSQL greifen. Sobald diese Features auch in MySQL verfügbar sind, werde ich dieses Tutorial entsprechend erweitern.

Christoph Bichlmeier
E-Mail: chris 'at' bichlmeier 'dot' info
Website: http://www.bichlmeier.info
erstellt am: 23.10.2004