

Name:
 Klasse:
 Datum:
 Schuljahr: 2022/23

Verwendete DB-Version:



Version MariaDB	Version gm3
10.9.2-MariaDB-1:10.9.2+maria~ubu2204	2022-08-12

Prozeduren und Funktionen

MariaDB ermöglicht es, benutzerdefinierte Funktionen bzw. Prozeduren zu erstellen.

In wie weit und im welchem Umfang man Funktionen/Prozeduren nutzt, muss man sich gut überlegen, gerade wenn hier aufwendige Programmierungen vorgenommen werden. Das Debuggen ist hier nicht wirklich einfach und die Notwendigkeit, dies auf SQL-Ebene zu machen, anstelle eine moderne Programmiersprache in der Applikation zu verwenden, die sehr komfortabel mit dem Debugger kombiniert werden kann, muss immer für den Einzelfall entschieden werden.

Eine Funktion wird mit CREATE FUNCTION, eine Prozedur mit CREATE PROCEDURE erstellt. Dabei sieht der vereinfachte Aufruf wie folgt aus:

```
CREATE OR REPLACE PROCEDURE name ([parameter[,...]])
  [characteristic ...] routine_body

CREATE OR REPLACE FUNCTION name ([parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body
```

CREATE ... name Erstellt eine neue Funktion oder Prozedur. Der wesentliche Unterschied dabei ist, dass eine Prozedur keinen Rückgabewert hat. Dabei wird die Funktion bzw. Prozedur in der Standarddatenbank gespeichert und steht allen zur Verfügung. Soll diese nur in einer Datenbank verwendet werden, so muss der DB-Name vorangesetzt werden (dbname.fktname).

parameter Legt die Parameter fest. Wenn keine Parameter übergeben werden, muss eine leere Parameterliste mit Klammern () verwendet werden. Die Parameter bei Prozeduren können als IN (Standard), OUT und INOUT definiert werden. Funktionsparameter werden immer als IN-Parameter übergeben.

characteristic Legt die Art des Codes fest, beispielsweise LANGUAGE SQL. Mehr dazu im MariaDB-Handbuch.

routine_body Der eigentliche Code. Besteht dieser aus mehr als einer Anweisung, so muss der Code mit BEGIN und END eingeschlossen werden.

Arbeitet man auf der Konsole:

Da die äußere Definition mit ';' abgeschlossen wird und die innere Definition im Normalfall auch, führt dies zu Problemen. Daher muss der Befehls-Trenner beispielsweise mit DELIMITER// umdefiniert und am Ende wieder auf den Standardwert gesetzt werden. Mehr dazu unter Delimiters.

Wird die Prozedure beispielsweise über Java (JDBC) erstellt, ist dies nicht notwendig.



```

DELIMITER //
CREATE PROCEDURE myprocedure ()
BEGIN
    SELECT 1+1;
END
//
DELIMITER ;

```

Eine gespeicherte Funktion bzw. Prozedur wird mit ALTER PROCEDURE bzw. ALTER FUNCTION geändert. Gelöscht mit DROP PROCEDURE bzw. DROP FUNCTION. Der Status wird mit SHOW PROCEDURE STATUS bzw. SHOW FUNCTION STATUS angezeigt. Der dahinterliegende Code wird mit SHOW PROCEDURE CODE bzw. SHOW FUNCTION CODE¹.

Nachfolgendes Beispiel definiert eine Prozedur, die den ältesten Mitarbeiter ausgibt.²

```

USE gm3;
CREATE OR REPLACE PROCEDURE aeltesterMitarbeiter()
BEGIN
    SELECT * FROM mitarbeiter ORDER BY gebdat ASC LIMIT 1;
END

```

Ob die Prozedur auch erzeugt worden ist, kann man wie folgt überprüfen:

```

USE gm3;
SHOW PROCEDURE STATUS LIKE '%aeltesterMitarbeiter%'\G

```

```

***** 1. row *****
      Db: gm3
      Name: aeltesterMitarbeiter
      Type: PROCEDURE
      Definer: test@localhost
      Modified: 2022-08-09 18:18:26
      Created: 2022-08-09 18:18:26
      Security_type: DEFINER
      Comment:
character_set_client: utf8mb4
collation_connection: utf8mb4_general_ci
Database Collation: utf8_general_ci

```

Der Aufruf erfolgt mit CALL:

```

USE gm3;
CALL aeltesterMitarbeiter();

```

id	name	vorname	gebdat	strasse	oid	aid	fid	eingestellt
16	Hellmeister	Sepp	1959-08-01	Von-Hünefeld-Weg 41A	7939	4	6	2002-09-15

Notiz:

- ¹ Diese Funktion muss aus Sicherheitsgründen im MariaDB-Server aktiviert sein!
- ² Da der SQL-Code hier über Java (JDBC) an die MariaDB „geschickt“ wird, muss hier der DELIMITER nicht umgestellt werden. Dies ist nur auf der Konsole notwendig.



Arbeiten mit Parametern

Parameter können per IN, OUT und INOUT definiert werden. Diese werden wie normale Variablen verwendet. Bei IN gibt es keine Möglichkeit, Werte zurückzugeben, bei den anderen beiden schon.

Um die billigsten Produkte zu ermitteln, kann man mit einem Subselect arbeiten.

```
USE gm3;
SELECT id, bez, vpreis
  FROM produkt
 WHERE vpreis=(SELECT MIN(vpreis) FROM produkt);
```

```
+-----+-----+-----+
| id | bez | vpreis |
+-----+-----+-----+
| 121 | Mineralquelle Q4 Classic | 5.25 |
| 141 | Mineralquelle Q4 Medium | 5.25 |
+-----+-----+-----+
```

Wird der Wert für den billigsten Artikel öfters gebraucht, macht es Sinn, diesen in einer Variablen zu speichern.

Mit nachfolgender Definition wird der billigste Preis in der Variablen `bilPreis` gespeichert. Dazu wird `INTO` im `SELECT`-Statement verwendet.

```
USE gm3;
CREATE OR REPLACE PROCEDURE billigsterPreis(OUT bilPreis DECIMAL(4,2))
BEGIN
  SELECT MIN(vpreis) INTO bilPreis FROM produkt;
END
```

Die so definierte Variable kann dann wie folgt verwendet werden:

```
USE gm3;
CALL billigsterPreis(@billigsterPreis);
SELECT @billigsterPreis;
```

```
+-----+
| @billigsterPreis |
+-----+
| 5.25 |
+-----+
```

Jetzt kann man den Subselect durch die Variable ersetzen.

```
USE gm3;
CALL billigsterPreis(@billigsterPreis);
SELECT id, bez, vpreis
  FROM produkt
 WHERE vpreis=@billigsterPreis;
```

```
+-----+-----+-----+
| id | bez | vpreis |
+-----+-----+-----+
| 121 | Mineralquelle Q4 Classic | 5.25 |
| 141 | Mineralquelle Q4 Medium | 5.25 |
+-----+-----+-----+
```



Zeilen nummerieren

Man kann aber auch die Variable verwenden, um die Ausgabe zu nummerieren. Variablen werden dabei mit SET definiert und bekommen mit := den Wert zugewiesen.

```
USE gm3;
SET @counter:=0;
SELECT (@counter:=@counter+1) AS 'Position', id, bez, vpreis
FROM produkt LIMIT 5;
```

Position	id	bez	vpreis
1	1	Binding Export	15.45
2	2	Dachsenfranz Kellerbier Bügelflasche	12.45
3	3	Eichbaum Export	15.80
4	4	Heidelberger Export	16.45
5	5	Kurpfalz Bräu Kellerbier	19.45

Notiz:



Arbeiten mit lokalen Variablen und Kontrollstrukturen in Prozeduren und Funktionen

Alle nachfolgenden Anweisung lassen sich nur in Prozeduren bzw. Funktion verwenden!

DECLARE-Anweisung

Mit DECLARE lassen sich Variablen definieren.

```
DECLARE var_name [, var_name] ... [[ROW] TYPE OF]] type [DEFAULT value]
```

Beispiel:

```
DECLARE mycount      int;  
DECLARE tmp          TYPE OF table.col;
```

Notiz:

IF-Anweisung

Mit IF wird ein einfaches Bedingungskonstrukt zur Verfügung gestellt.

```
IF bedingung THEN  
    sql-list  
[ELSEIF bedingung THEN  
    sql-list  
] ...  
[ELSE  
    sql-list  
]  
END IF
```

Wenn die Bedingung zutrifft, wird die zugehörige SQL-Anweisungsliste (ein bzw. mehrere Befehle) ausgeführt. Trifft keine Bedingung zu, wird die Anweisungsliste aus der ELSE-Klausel ausgeführt.

Beispiel:

```
IF mycount = 0 THEN  
    -- true ....  
ELSE  
    -- else ...  
END IF;
```

Notiz:



CASE-Anweisung

Die CASE-Anweisung stellt eine Mehrfachauswahl zur Verfügung.

```
CASE variable
WHEN wert1 THEN
    sql-list
[WHEN wert2 THEN
    sql-list
] ...
[ELSE
    sql-list
]
END CASE
```

Oder:

```
CASE
WHEN bedingung THEN
    sql-list
[WHEN bedingung THEN
    sql-list
] ...
[ELSE
    sql-list
]
END CASE
```

Es werden die Anweisungen ausgeführt, bei denen der Wert dem Inhalt der Variablen entspricht bzw. die Bedingung zutrifft. Trifft nichts zu, werden die SQL-Anweisungen bei ELSE ausgeführt.

Beispiel:

Nachfolgendes Beispiel zeigt, ob es zu warm, zu kalt, ... ist.

```
USE gm3;
CREATE OR REPLACE FUNCTION XLevel ( temperatur INT ) RETURNS VARCHAR(10)
BEGIN

    DECLARE level VARCHAR(10);

    CASE
        WHEN temperatur <= 15 THEN
            SET level = 'kalt';

        WHEN temperatur > 15 AND temperatur <= 25 THEN
            SET level = 'warm';

        ELSE
            SET level = 'zu warm';
    END CASE;

    RETURN level;
END
```

```
USE gm3;
SELECT XLevel(16);
```

```
+-----+
| XLevel(16) |
+-----+
| warm      |
+-----+
```



LOOP-Schleife

Mit LOOP lässt sich eine einfache Schleife realisieren. Die Schleife wird solange durchlaufen, bis sie mit LEAVE verlassen wird. Mit ITERATE wird die entsprechende Schleife von oben erneut durchlaufen. Optional kann die LOOP-Anweisung auch beschriftet werden. Werden beide Label verwendet, so müssen diese identisch sein.

```
[begin_label:] LOOP
  sql-list
  IF condition THEN
    LEAVE [label];
  END IF;
END LOOP [end_label]
```

Beispiel:

```
USE gm3;
CREATE OR REPLACE FUNCTION XLoop ( endval INT ) RETURNS VARCHAR(100)
BEGIN
  DECLARE i INT;
  DECLARE str VARCHAR(100);
  SET i = 1;
  SET str = '';

  iterateX: LOOP
    IF i > endval THEN
      LEAVE iterateX;
    END IF;
    SET i = i + 1;
    IF (i mod 2) THEN
      ITERATE iterateX;
    ELSE
      SET str = CONCAT(str,i,' ');
    END IF;
  END LOOP;
  RETURN str;
END
```

```
USE gm3;
SELECT XLoop(16);
```

```
+-----+
| XLoop(16) |
+-----+
| 2 4 6 8 10 12 14 16 |
+-----+
```

Notiz:



WHILE-Schleife

Eine WHILE-Schleife wird so lange wiederholt, wie die Bedingung wahr ist.

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

Beispiel:

```
USE gm3;
CREATE OR REPLACE PROCEDURE XWhile ( endval INT )
BEGIN
    SET @i=0;
    WHILE @i < endval DO SET @i = @i + 1; END WHILE;
END
```

```
USE gm3;
CALL XWhile(16);
SELECT @i;
```

```
+----+
| @i |
+----+
| 16 |
+----+
```

Notiz:



REPEAT LOOP-Schleife

Eine REPEAT LOOP verhält sich sehr ähnlich, wie eine LOOP-Schleife, wird jedoch mindestens einmal durchlaufen.

```
[begin_label:] REPEAT
  statement_list
UNTIL search_condition
END REPEAT [end_label]
```

Beispiel:

```
USE gm3;
CREATE OR REPLACE PROCEDURE XRLoop ( endval INT )
BEGIN
  SET @i=0;
  REPEAT SET @i = @i + 1; UNTIL @i > endval END REPEAT;
END
```

```
USE gm3;
CALL XRLoop(16);
SELECT @i;
```

```
+-----+
| @i |
+-----+
| 17 |
+-----+
```

Notiz:



FOR-Schleife

FOR-Schleifen ermöglichen es, Code eine bestimmte Anzahl von Malen auszuführen.

```
[begin_label:]  
FOR var_name IN [ REVERSE ] lower_bound .. upper_bound  
DO statement_list  
END FOR [ end_label ]
```

Beispiel:

```
USE gm3;  
CREATE OR REPLACE TABLE test (a int);  
CREATE OR REPLACE PROCEDURE bspfor()  
BEGIN  
  FOR i IN 1..3  
  DO  
    INSERT INTO test VALUES (i);  
  END FOR;  
END
```

```
USE gm3;  
CALL bspfor();  
SELECT * FROM test;
```

```
+---+  
| a |  
+---+  
| 1 |  
| 2 |  
| 3 |  
+---+
```

Notiz:



Cursor

Ein Cursor ist eine Art Schleife, die Datensätze sequentiell durchläuft. Grundlage ist dabei das Ergebnis eines SELECT-Befehls.

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

Wenn der Cursor an das Ende des Abfrageergebnisses kommt, wird der Status `NOT FOUND` (genaugenommen der Code `'02000'`) gesetzt, der Durchlauf ist somit beendet und der Cursor kann wieder geschlossen werden. Dazu wird die boolesche Variable `done` definiert. Erreicht der Cursor das Ende der Abfrage, kann über einen `HANDLER` festgelegt werden, dass die Variable `done` auf `TRUE` gesetzt wird.

Die allgemeine Syntax lautet dabei: `DECLARE ... HANDLER`

```
DECLARE handler_action HANDLER
  FOR condition_value [, condition_value] ...
  sql-statement
```

Für `handler_action` gilt:

CONTINUE Der Handler führt das SQL-Statement aus, bricht aber die Prozedur nicht ab.

EXIT Der Handler bricht die Prozedur sofort ab.

UNDO (wird nicht unterstützt)

Für `condition_value` gilt:

SQLSTATE Der Handler wird ausgeführt, wenn ein entsprechender Status eintritt. Mehr dazu im MariaDB-Handbuch.

SQLWARNING Der Handler wird bei Warnungen ausgeführt.

NOT FOUND Der Handler wird ausgeführt, wenn Objekte etc. nicht gefunden werden.

SQLEXCEPTION Der Handler wird bei Fehlern ausgeführt.

Für obiges Beispiel gilt dann:

```
DECLARE done BOOLEAN DEFAULT FALSE;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=TRUE;
oder besser:
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=TRUE;
```

Im nächsten Schritt muss der Cursor geöffnet und nach Verwendung wieder geschlossen werden:

```
OPEN cursorname;
...
CLOSE cursorname;
```

Die einzelnen Einträge werden über `FETCH` ermittelt.

```
FETCH cursorname INTO var_name [, var_name] ...
```

Dies holt die nächste Zeile aus dem Cursor (falls vorhanden), speichert die Spalteneinträge in entsprechenden Variablen und setzt dann den Cursor um eins weiter.

Notiz:



Beispiel:

Summiere alle Preise in der Tabelle produkt.

```
USE gm3;
CREATE OR REPLACE FUNCTION bspcursor() RETURNS DECIMAL(11,2)
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE summe DECIMAL(11,2) DEFAULT 0;
  DECLARE value DECIMAL(9,2) DEFAULT 0;

  DECLARE mycursor CURSOR FOR SELECT vpreis FROM produkt;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=TRUE;

  OPEN mycursor;

  read_loop: LOOP
    FETCH mycursor INTO value;

    IF done THEN
      LEAVE read_loop;
    END IF;
    SET summe = summe + value;

  END LOOP;
  CLOSE mycursor;
  RETURN summe;
END
```

```
USE gm3;
SELECT bspcursor();
```

```
+-----+
| bspcursor() |
+-----+
| 3568.65      |
+-----+
```

Zur Kontrolle

```
USE gm3;
SELECT sum(vpreis) FROM produkt;
```

```
+-----+
| sum(vpreis) |
+-----+
| 3568.65      |
+-----+
```



Notiz:



Trigger

Trigger sind SQL-Anweisungen, die bei Tabellen (keine Views oder temporäre Tabellen) bei bestimmten Anweisungen (INSERT, UPDATE oder DELETE) automatisch ausgeführt werden.

Achtung: Auch bei Triggern muss der DELIMITER auf der Konsole angepasst werden.

Der allgemeine Aufruf sieht dabei wie folgt aus:

```
CREATE [OR REPLACE]
  [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
  TRIGGER [IF NOT EXISTS] trigger_name trigger_time trigger_event
  ON tbl_name FOR EACH ROW
  sql-statement
```

DEFINER Legt fest, unter welchen Rechten der Trigger ausgeführt werden soll.

trigger_name Der Name des Triggers.

trigger_time Der Zeitpunkt, wann der Trigger ausgelöst wird. Mit BEFORE wird der Trigger vor der Anweisung und mit AFTER nach der Anweisung ausgeführt.

trigger_event Gibt an, welche Anweisung den Trigger aktiviert.

INSERT Der Trigger wird aktiviert, wenn in der Tabelle eine neue Zeile eingefügt wird, beispielsweise bei INSERT . . . , LOAD DATA und REPLACE.

UPDATE Der Trigger wird aktiviert, wenn eine Zeile geändert wird (nur UPDATE).

DELETE Der Trigger wird aktiviert, wenn eine Zeile gelöscht wird, beispielsweise bei DELETE . . . und REPLACE.

tbl_name Für welche Tabelle der Trigger bestimmt ist.

sql-statement Die SQL-Anweisung(en) (mit BEGIN...END) für den Trigger.

Mit dem Prefix OLD kann man auf Spalten der vorhandenen Zeile zugreifen, bevor diese geändert oder gelöscht worden sind, mit NEW auf die neuen oder geänderten Spalten der Zeile.

BEFORE/AFTER UPDATE Alte Werte mit OLD verfügbar.
Neue Werte mit NEW verfügbar.

BEFORE/AFTER DELETE Alte Werte mit OLD verfügbar.

BEFORE/AFTER INSERT Neue Werte mit NEW verfügbar.

Mit SHOW TRIGGERS lassen sich erstellte Trigger anzeigen, mit DROP TRIGGER wieder löschen.



Beispiel

Im nachfolgenden Beispiel sollen das Änderungsdatum, die Änderungszeit und wer die die Aktion veranlasst hat, protokolliert werden. Dazu wird eine Tabelle erstellt, die die Änderungen aufnimmt.

```
USE gm3;
CREATE OR REPLACE TABLE changed (
  name      VARCHAR(64) NOT NULL,
  id        INT          NOT NULL,
  created   DATETIME,
  modified  DATETIME,
  deleted   DATETIME,
  wer       VARCHAR(64)
);
DESC changed;
```

Field	Type	Null	Key	Default	Extra
name	varchar(64)	NO		null	
id	int(11)	NO		null	
created	datetime	YES		null	
modified	datetime	YES		null	
deleted	datetime	YES		null	
wer	varchar(64)	YES		null	

Im nächsten Schritt werden drei Trigger für insert, update und delete für die Tabelle mitarbeiter erstellt.

```
USE gm3;
CREATE OR REPLACE TRIGGER mitarbeiter_insert
  AFTER INSERT ON mitarbeiter FOR EACH ROW
  BEGIN
    INSERT INTO changed (name,id,created,wer)
      VALUES ('mitarbeiter', NEW.id, NOW(), CURRENT_USER());
  END
```

```
USE gm3;
SHOW TRIGGERS FROM gm3\G;
***** 1. row *****
      Trigger: mitarbeiter_insert
      Event: INSERT
      Table: mitarbeiter
      Statement: BEGIN
        INSERT INTO changed (name,id,created,wer)
          VALUES ('mitarbeiter', NEW.id, NOW(), CURRENT_USER());
      END
      Timing: AFTER
      Created: 2022-08-12 11:00:17.37
      sql_mode: IGNORE_SPACE, STRICT_TRANS_TABLES, 2
      ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, 2
      NO_ENGINE_SUBSTITUTION
      Definer: test@localhost
character_set_client: utf8mb4
collation_connection: utf8mb4_general_ci
Database Collation: utf8_general_ci
```



update

```
USE gm3;
CREATE OR REPLACE TRIGGER mitarbeiter_update
AFTER UPDATE ON mitarbeiter FOR EACH ROW
BEGIN
    INSERT INTO changed (name,id,modified,wer)
        VALUES ('mitarbeiter', OLD.id, NOW(), CURRENT_USER());
END
```

delete

```
USE gm3;
CREATE OR REPLACE TRIGGER mitarbeiter_delete
AFTER DELETE ON mitarbeiter FOR EACH ROW
BEGIN
    INSERT INTO changed (name,id,deleted,wer)
        VALUES ('mitarbeiter', OLD.id, NOW(), CURRENT_USER());
END
```

Test

```
USE gm3;
INSERT INTO mitarbeiter (name, vorname)
    VALUES ('Schnecke', 'Marion');
UPDATE mitarbeiter SET gebdat=NOW()
    WHERE name='Schnecke' AND vorname='Marion';
DELETE FROM mitarbeiter
    WHERE name='Schnecke' AND vorname='Marion';
SELECT * FROM changed;
```

name	id	created	modified	deleted	wer
mitarbeiter	85	2022-09-18 10:00:05.0	null	null	test@localhost
mitarbeiter	85	null	2022-09-18 10:00:05.0	null	test@localhost
mitarbeiter	85	null	null	2022-09-18 10:00:05.0	test@localhost

Sie sehen, das Erstellen von Triggern kann sehr schnell aufwendig werden und man muss überlegen, ob dies nicht einfacher über die Anwendung gelöst wird.



Prepared Statements

Prepared Statements dienen dazu, SQL-Anweisung mit Platzhaltern vorzubereiten und diese dann nur noch mit entsprechenden Werten aufzurufen. Der Vorteil dabei ist, dass die SQL-Anweisung nur einmal analysiert werden muss. Anstelle eines Prepared Statements lässt sich aber auch eine Prozedur verwenden.

Die Definition lautet dabei:

```
PREPARE stmtname FROM sql-statement
```

Ausgeführt wird das Prepared Statement mit EXECUTE. Ein vorhandenes kann über DEALLOCATE PREPARE bzw. DROP PREPARE wieder gelöscht werden.

Beispiel

Es soll der billigste Lieferant eines Produkts ermittelt werden.

```
USE gm3;
PREPARE produkt_billigLieferant FROM
  'SELECT l.id, l.name, p.id, p.bez, p.vpreis
   FROM lieferant l, lieferung lf, lieferpos lp, produkt p
   WHERE l.id=lf.lid AND lf.id=lp.lid AND l.id AND lp.pid=p.id AND p.id=?
   ORDER BY p.vpreis
   LIMIT 1';
SET @Artikel=30;
EXECUTE produkt_billigLieferant USING @Artikel;
```

id	name	id	bez	vpreis
1	Augustiner Brauerei GmbH	30	Bitburger Pils	16.80

Prepared Statements haben direkt auf SQL-Ebene an Bedeutung verloren. Wichtiger sind diese dagegen in der Anwendung und entsprechenden Programmiersprache z. B. Java mit JDBC, da dadurch SQL-Injection ziemlich ausgehebelt werden.

Prozeduren anwenden [AUF-09-2-1](#)

► [SQL-proc](#)

