Stored Procedures in MariaDB/MySQL

Vorab: Transaction/Transaktionen

Transaktionen fassen eine Gruppe von Datenbank-Operationen zusammen. Diese werden als „Block“ ausgeführt. Erst wenn alle Operationen erfolgreich durchgeführt wurden, ist die Transaktion beendet. Ein einzelner Fehlschlag bei den Operationen führt dazu, dass die gesamte Transaktion fehlschlägt!

Transaktionen laufen nach dem ACID-Prinzip ab:

* Atomicity: Jeder einzelne Befehl in der Transaktion kann erfolgreich sein oder nicht; bei Nichterfolg wird die ganze Transaktion abgebrochen!
* Consistency: Die Datenbank wird erst als letzter Schritt upgedatet; alle Änderungen werden berücksichtigt.
* Isolation: Bewahrt die Unabhängigkeit von den Transaktionen
* Durability: Dauerhaftigkeit – sichert den Datenbestand gegen Abstürze etc.

Syntax(Bsp.):

START TRANSACTION;

SELECT …;

UPDATE …;

Bis hierhin wird noch nichts an der Datenbank verändert. Erst mit einem abschließendem

COMMIT;

Werden die Änderungen in der Datenbank tatsächlich umgesetzt.

Falls ein Fehler bei z.B. den obigen Befehlen auftritt (egal, ob es logische, Syntax – oder interne Fehler sind), kann die Ausführung der Transaktion mit

ROLLBACK;

Rückgängig gemacht werden.

Sinn machen Transaktionen vor Allem bei vielen gleichzeitig zugreifenden Usern; dort kann man sicherstellen, dass die Daten-Konsistenz gewahrt bleibt.

Vorab: LOCK TABLES

Exklusiv-Zugriff auf Tabellen kann in MariaDB/MySQL auch erreicht werden, indem man die zu bearbeitenden Tabellen für Lese- oder auch Schreib-Zugriff sperrt.

Syntax:

LOCK TABLES table\_name1 [**READ** | WRITE],

table\_name2 [**READ** | WRITE],…;

wobei READ ein „lesender“ Lock ist. Der User, der diesen READ-Lock initiiert, hat eben diesen lesenden Zugriff und kann nicht in die Tabelle schreiben; für andere User gilt ebenfalls ein Schreib-Verbot, solange diese Tabelle gesperrt ist.  
WRITE – Locks gewähren Lese- und Schreibzugriff auf die entsprechende Tabelle; andere User können weder schreibend noch lesend zugreifen!

Bei Beenden der Datenbankverbindung werden diese LOCKS implizit aufgehoben; man sollte allerding die gemachten LOCKS sauber und per Hand mit

UNLOCK TABLES;

Wieder aufheben, damit andere Leute wieder ordentlichen Zugriff haben.

Befehle zum Sperren von Tabellen etc. können nur von Benutzern ausgeführt werden, denen das auch in der MariaDB/MySQL-Benutzerverwaltung gewährt wurde! Siehe hierzu auch die entsprechende Dokumentation.

MariaDB/MySQL Stored Procedures

Stored Procedures in SQL sind eine Möglichkeit, gewisse Arbeitsabläufte zu automatisieren. Dabei geht die Flexibilität noch sehr viel weiter als z.B. bei den Views (man muss sich nicht nur auf SELECT – Befehle beschränken und kann durch Übergabe-Parameter relativ allgemeingültige Prozeduren erstellen).

Weiter Vorteile von SPs:

* Reduzierter Netzwerk-Traffic: Stored Procedures (SPs) werden auf dem DB-Server ausgeführt; es müssen nicht erst alle Daten zum Client übertragen werden.
* Zentralisieren der DB-Logik: Alle SPs werden auf dem Server gespeichert und sind damit (mit den entsprechenden Berechtigungen) für alle DB-Nutzer zugänglich.
* Durch die DB-Berechtigungen lässt sich eine Datenbank gut absichern, indem man z.B. nur bestimmten Leuten Ausführungsrecht für gewisse SPs gewährt
* (früher: Vorteile in der Ausführungsgeschwindigkeit, wenn SPs auf dem (leistungsstarken?) Server liegen und ausgeführt werden)

Dagegen gibt es auch ein paar Nachteile:

* Ressourcen können bei umfangreichen Prozeduren vielleicht trotz leistungsstarkem Server knapp werden – v.a. wenn viele Leute gleichzeitig an der DB arbeiten
* MariaDB/MySQL liefern wenig bis keine Debug-Möglichkeiten, um Probleme in Stored Procedures zu beheben
* „Spezielle Syntax“: MariaDB/MySQL benötigen für SPs eine spezielle Syntax, die in anderen SQL-Versionen (SQL Server, Oracle) substantiell unterschiedlich sein kann (wie auch die allgemeine SQL-Syntax unter den verschiedenen Versionen nicht ganz kompatibel ist!)

(Statement) Delimiter

Die meisten SQL-Oberflächen sehen ein SQL-Statement als abgeschlossen an, sobald sie auf den ersten, sogenannten Delimiter treffen (Default: Semikolon „;“).

Dies führt zu Schwierigkeiten bei Stored Procedures, da in der Regel mehrere Statements in einer Prozedur zusammengefasst werden. Somit wäre für den SQL-Interpreter die SP nach dem ersten „;“ beendet und führt dann mit großer Wahrscheinlichkeit zu Fehlermeldungen!

Lösung: Man stellt vorab den „Delimiter“ um auf ein Zeichen, welches man in der nachfolgenden Prozedur nicht benutzt:

DELIMITER| / DELIMITER $$ / DELIMITER §§ etc.

Nach dem Ende der eigentlichen SP sollte man den „Delimiter“ wieder auf den Ursprungswert zurücksetzen, um die normale Funktionalität wieder herzustellen:

DELIMITER ;

Bemerkung: Am Ende der SP will man natürlich die Befehlskette abschließen. An dieser Stelle muss man also einmalig das veränderte Delimiter-Symbol verwenden (siehe unten).

Erstellen der eigentlichen Stored Procedure

Nach der Zeile, die den Delimiter verändert:

CREATE PROCEDURE ProcedureName (Parameter1, Parameter2, …)

BEGIN

-\*SQL-Statements\*-

END |

In den runden Klammern können Parameter verschiedener Art „übergeben“ werden:

* IN Parameter – Input-Parameter (Default, d.h. „IN“ muss nicht angegeben werden). Input-Parameter sind an die SP übergebene Werte; pro Parameter muss natürlich auch ein Datentyp angegeben werden. (vgl. Übergabewerte an Funktionen in anderen Programmiersprachen)

Bsp.:

IN Warenwert DECIMAL(6,2)

* OUT Parameter – Output-Parameter. Damit können Werte aus der Prozedur „exportiert“ und ggf. weiterverwendet werden. Dazu muss man beim Aufruf der Prozedur eine lokale Variable mit übergeben; lokale Variablen müssen mit einem vorangestelltem „@“ gekennzeichnet sein.

Bsp.:

CREATE PROCEDURE Beispiel01 (OUT Ausgabe1 INT)

…

Aufruf:

CALL Beispiel101 (@aus1);

SELECT @aus1;

* INOUT Parameter – Kombination aus Input- und Output-Parameter. Dieser muss vorab schon mit z.B.

SET @bspinout = 42;

Deklariert werden.

Allgemein werden in SPs die Codeblöcke, die in anderen Sprachen z.B. mit Klammern ({}) eingeschlossen werden, hier mit BEGIN und END eingeschlossen.

Dabei ist allerdings darauf zu achten, dass der Haupt-Codeblock (entsprechend einer main()-Funktion in anderen Sprachen) dann tatsächlich einmalig mit dem anfangs veränderten „Delimiter“ abgeschlossen werden muss, denn an der Stelle will man ja tatsächlich die SP abschließen! (END |), siehe oben!

„Abspeichern“ und Löschen von Stored Procedures

Abspeichern soll in dem Zusammenhang bedeuten, dass man den Code seiner SP in SQL einmal ausführen/interpretieren lässt (in DBeaver: Alt-X zum Ausführen von mehrzeiligen Befehlen). Anschließend ist die SP in der Datenbank gespeichert und zum Aufruf bereit.

Gelöscht werden können SPs von der Datenbank mittels:

DROP PROCEDURE [IF EXISTS] sp\_name;

Aufruf einer SP

CALL sp\_name (parameter1, parameter2, …);

Wichtig ist, dass im Falle von INOUT-Parametern der übergebene Parameter vorab schon deklariert werden muss!

Variablen innerhalb der SP

Beliebige Variablen können mit

DECLARE var\_name Datentyp [DEFAULT default\_value];

Deklariert werden. Dabei kann optional auch ein Startwert mit DEFAULT … übergeben werden.

Variablen können auch explizit gesetzt werden mit

SET var\_name = value;

Oder auch „interaktiv“ mittels SELECT-Befehl ausgelesen werden:

SELECT … INTO var\_name FROM… WHERE…;

Labels für Code-Blöcke

Code-Blöcke werden, wie schon erwähnt, mit BEGIN und END eingegrenzt. Bei bestimmten Konstrukten ist es günstig, solchen Blöcken einen Namen zu geben, um sie explizit anzusprechen und insbesondere auch beenden zu können:

[label:]BEGIN

-\*Code\*-

END;

Dann kann man einen so benannten Code-Block auch z.B. explizit verlassen mit:

LEAVE label;

Einfache Programmierkonstrukte in MariaDB/MySQL:

* IF Statement:

IF Bedingung THEN

-\*Code\*-

ELSEIF BEDINGUNG THEN

-\*Code\*-

ELSE

-\*Code\*-

END IF;

* CASE Statement:

CASE case\_value

WHEN value1 THEN -\*Code\*-

WHEN value2 THEN -\*Code\*-

…

[ELSE -\*Code\*-]

END CASE;

* LOOP Schleife: funktioniert ähnlich wie [While True] in anderen Sprachen; es wird also eine Schleife mehrfach durchlaufen. Um eine solche LOOP-Schleife zu steuern, gibt es die ITERATE und LEAVE-Statements.

[loop\_label:]LOOP

-\*Code\*-

END LOOP;

Mit ITERATE kann der Schleifendurchlauf von vorne gestartet werden; mit LEAVE kann der Loop verlassen werden.

* WHILE-Schleife: Solange eine Bedingung wahr ist, wird die Schleife immer wieder durchlaufen.

[while\_label:]WHILE bedingung DO

-\*Code\*-

END WHILE;

Hier kann auch wieder, wie in allen Schleifen, mittels ITERATE und LEAVE genauer gesteuert werden.

* REPEAT-Loop: Ähnlich der WHILE-Schleife, jedoch wird die Bedingung erst am Ende getestet. Effektiv heißt das, dass die Schleife *mindestens* einmal durchlaufen wird, bevor zum ersten Mal die Bedingung getestet wird.

[repeat\_label:]REPEAT

-\*Code\*-

UNTIL bedingung

END REPEAT;

SQL Cursors

Mit Cursors kann nicht nur ein einzelnes Suchergebnis bearbeitet werden, sondern ein ganzer Satz an Ergebnissen durchlaufen und weiter bearbeitet werden.

Cursors sind read-only, müssen sukzessive abgearbeitet werden (d.h. nacheinander, ohne Sprünge oder Rückwärtsschritte). Außerdem sind Cursors „asensitiv“, d.h. sie lesen die tatsächlichen Daten aus der DB und machen nicht erst eine Arbeitskopie davon. Dies bringt einen kleinen Geschwindigkeitsvorteil. Nachteil ist, dass etwaige parallele Schreibzugriffe die Cursor-Daten beeinflussen können.

Deklarieren und Öffnen eines Cursors:

DECLARE cursor\_name CURSOR FOR SELECT …;

Die Cursor-Deklaration muss *nach* allen anderen Variablen-Deklarationen stattfinden.

Wenn der Cursor dann gebraucht/ausgelesen werden soll, dann „öffnet“ man ihn:

OPEN cursor\_name;

Um dann tatsächlich, z.B. innerhalb einer Schleife, sukzessive die Daten des Cursors auslesen zu können, benutzt man:

FETCH cursor\_name INTO var1, var2,…;

Damit wird also pro Schleifendurchlauf immer die nächste Zeile „geholt“. Wenn dann keine weiteren Datensätze gefunden werden (also man am Ende seiner Selektion angekommen ist), muss man einen „Handler“ für diesen Fall deklarieren (am Besten direkt unter die Cursor-Deklaration):

DECLARE CONTINUE HANDLER FOR NOT FOUND

-\*Code, z.B. finished-Variable auf 1 setzen etc.\*-;

Der Cursor wird normalerweise automatisch geschlossen, sobald die Verbindung zur DB abreißt. Des guten Tones halber sollte man ihn aber nach Beendigung der Arbeit explizit schließen:

CLOSE cursor\_name;

Error-Handling in MariaDB/MySQL

Error-Handling, also das Abfangen von Fehlern sowie auch das Erzeugen von Fehlerbedingungen, ist auch hier möglich. Da dies bei kürzeren SPs eventuell nicht so sehr wichtig ist, wird dieses Thema hier nur erwähnt; es sei auf die nachfolgenden Beispiele zu Stored Procedures hingewiesen!