

Berufliche Schulen ZPG-Mitteilungen

Zentrale Projektgruppe - Computertechnik / Informatik
Kaufmännische Schulen

*Innovatives
Bildungsservice*

Beilage:

Karsten Scheider

Transaktionen, Benutzer, Rechte, Trigger und Stored Procedures in relationalen Datenbanken am Beispiel von MySQL



Landesinstitut
für Schulentwicklung

www.lis-bw.de
best@lis.kv.bwl.de

Qualitätsentwicklung
und Evaluation

Schulentwicklung
und empirische
Bildungsforschung

Bildungspläne

Stuttgart ■ Nr. 34 - Juni 2009

Transaktionen, Benutzer, Rechte, Trigger und Stored Procedures in relationalen Datenbanken am Beispiel von MySQL

Einführung

Das Thema „Relationale Datenbanken“ hat im Informatikunterricht an kaufmännischen Schulen seit ca. 20 Jahren einen hohen Stellenwert.

Leider wurden in den letzten Jahren aufgrund der Produktbindung an Microsoft – Access wichtige traditionelle aber auch aktuelle Funktionen nicht mehr vermittelt, da das Produkt diese nicht beinhaltet. Mit dem Einsatz der neuen Versionen (ab 5.0) von MySQL ist es möglich, Themen wie Transaktionskontrolle, Benutzer- und Rechteverwaltung, Trigger oder Stored Procedures wieder bzw. neu und in einfacher Form in den Focus zu rücken.

Transaktionskontrolle

Für die Datensicherheit der Datenbank in einer Mehrbenutzerumgebung ist es entscheidend, wie sicher die einzelnen Abfragen, hauptsächlich Manipulations – (DML) und Datendefinitionsabfragen (DDL) verarbeitet werden. Dies soll an kleinen, wenig komplexen Aufgabestellungen erläutert werden:

Beispiele:

In einem Schreibwarengroßhandel arbeiten die Vertriebsfachbearbeiter Mayer, Steeb und Nolde. Während Mayer und Steeb Aufträge bearbeiten, kümmert sich Nolde um die Preisgestaltung.

Zu Arbeitsbeginn erfasst Nolde mittels SQL – Abfrage eine Reihe von Aufträgen. Mit INSERT – Befehlen will er diese Änderungen durchführen. Im Augenblick des Absendens der Befehle fährt der MySQL - Datenbankserver herunter. Nach Wiederherstellung der Systemumgebung ist die Auftragsituation verfahren. Einzelne der gerade bearbeiteten Aufträge fehlen ganz, für andere sind nur die Köpfe vorhanden.

Nachdem das System wieder bereit ist, bucht Mayer eine größere Menge eines Artikels aus dem Lager Nord in das Lager Süd um. Daraufhin soll ein Umlagerungsauftrag an den Spediteur erstellt werden. Zur gleichen Zeit greift auch Steeb auf den Artikelbestand im Lager Nord zu, um einem Kunden die gesamte Menge zuzusagen. In der Folge streiten die beiden um den Lagerbestand.

Lösung:

Um die durch die Beispiele auftretenden Probleme zu vermeiden, verfügt ein echtes Datenbankmanagementsystem (DBMS) über eine Transaktionskontrolle.

Unter Transaktion versteht man dabei den Übergang der Datenbank von einem (konsistenten) Zustand in einen anderen. Eine Transaktion kann eine Reihe von Abfragen umfassen. Dabei ist z. B. das ACID – Prinzip zu beachten:

Atomicity:

Für eine Funktion eines Geschäftsprozesses, die eine nicht teilbare Einheit mehrerer Abfragen darstellt, muss sicher gestellt werden, dass alle Abfragen ausgeführt werden oder keine. Bei unserem Umlagerungsbeispiel müssen Auslagerung, Umlagerung und Speditionsauftrag ausgeführt werden oder die ganze Abwicklung sofort rückgängig gemacht werden.

Consistency

Die Datenbank muss von einem konsistenten Zustand in einen anderen überführt werden. In unserem Beispiel erfolgt die Erfassung der Auftragsköpfe und – positionen zwangsweise nacheinander und führt damit bei Systemabsturz zu inkonsistenten Zuständen, da zu einem bestimmten Zeitpunkt eben kurz Auftragsköpfe ohne – positionen existieren. Eng verbunden damit ist die Forderung nach referentieller Integrität, d.h. Fremdschlüssel in abhängigen Tabellen dürfen nur existieren, wenn auch der zugrunde liegende Primärschlüsselsatz in der Ausgangstabelle existent ist.

Isolation

Greifen Abfragen auf bestimmte Datenbereiche zu, dürfen diese nicht gleichzeitig für Abfragen anderer Benutzer verwendet werden. In unserem Beispiel dürfte Steeb gar nicht auf den Artikelbestand im Lager Nord zugreifen, da dieser bereits von Mayer bearbeitet wird.

Durability

Wird eine Transaktion abgeschlossen, sind ihre Ergebnisse dauerhaft und nicht umkehrbar. Dies ergibt sich auch aus der Atomicity. Sind alle Abfragen einer Transaktion erfolgreich ausgeführt ist sie unumkehrbar und für alle Benutzer verbindlich.

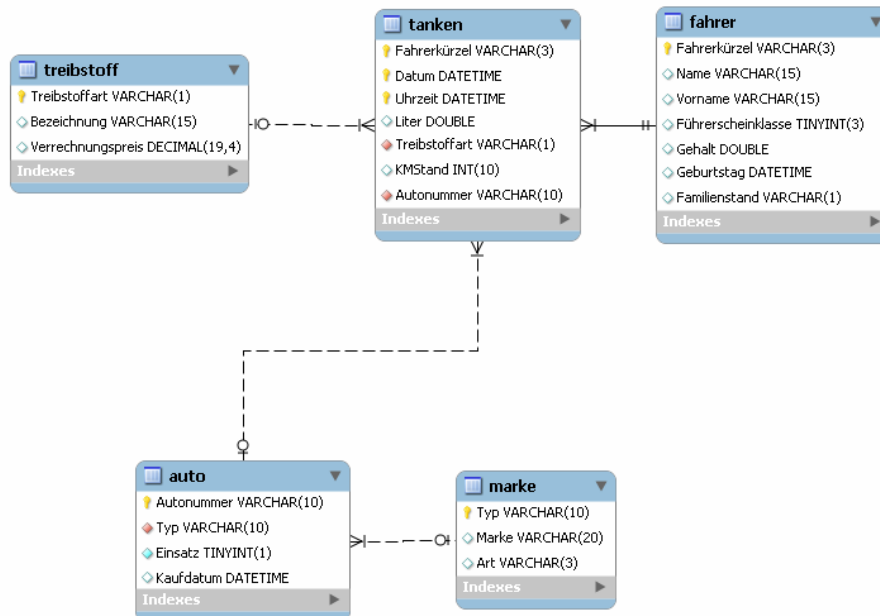
Selbstverständlich kann es immer wieder zu kritischen Situation in einer Datenbank kommen. Diese muss dann aber in der Lage sein, den stabilen Ausgangszustand wieder herzustellen. Dies erfolgt z. B. mittels ROLLBACK, SAVEPOINTS oder SNAPSHOTS

Diesen Anforderungen genügen weder MS – ACCESS noch die früheren Versionen von MySQL. Microsoft reagierte mit der Entwicklung des professionellen SQL – Servers. In seiner aktuellen Version 5 verfolgt auch MySQL dieses Prinzip, allerdings nur bei Verwendung des Datenbanktyps InnoDB.

Erweiterte Funktionen von MySQL

Datenbank

Die folgenden Ausführungen gelten für die Datenbank Fuhrpark mit folgender Struktur, dargestellt mit der MySQL – Workbench.



Beachten Sie; dass für die weiteren Arbeiten im Mehrbenutzerbetrieb in der Firewall Ihres Rechners Port 3306 für MySQL frei geschaltet ist.

Transaktionskontrolle

SQL hält für die Transaktionskontrolle die Befehle „**START TRANSACTION**“ für den Anfang und „**COMMIT**“ für den Abschluss einer Transaktion bereit. Vor dem Ende der Transaktion kann jederzeit mit „**ROLLBACK**“ der Ausgangszustand wieder hergestellt werden. Nach Abschluss mit „**COMMIT**“ ist dies nicht mehr möglich.

Zur Vorbereitung muss das voreingestellte AUTOCOMMIT=1, das automatisch nach jeder Abfrage ein COMMIT absetzt, mit „0“ abgeschaltet werden.

Für ein INSERT auf die Autotabelle könnten die Befehle so aussehen:

```
SET AUTOCOMMIT=0 ;
USE Fuhrpark;
START TRANSACTION;
INSERT INTO Auto (Autonummer, Typ, Einsatz, Kaufdatum);
VALUES ( 'FR-X 34', 'Golf', true, '2009-01-02' );
SELECT * FROM Auto WHERE typ='Golf';
ROLLBACK;
SELECT * FROM Auto WHERE typ='Golf';
```

Beim ersten SELECT ist das Fahrzeug erfasst, nach dem ROLLBACK, beim 2. SELECT ist das Fahrzeug verschwunden. Würde vor ROLLBACK ein COMMIT abgesendet, wäre der erfasste Satz nicht mehr rückgängig zu machen. Eine vollständige Transaktion würde die folgenden Befehle enthalten:

```

START TRANSACTION;
INSERT INTO Auto (Autonummer, Typ, Einsatz,Kaufdatum);
VALUES ('FR-X 34', 'Golf',true,'2009-01-02');
COMMIT;

```

Bei größeren Transaktionen können auch SAVEPOINTS (keine Ziffern!) gesetzt werden. Damit kann ein ROLLBACK gezielt an eine bestimmte Abfragestelle gesetzt werden:

```

SET AUTOCOMMIT=0;
USE Fuhrpark;
START TRANSACTION;
INSERT INTO Auto (Autonummer, Typ, Einsatz,Kaufdatum)
VALUES ('FR-X 44', 'Golf',true,'2009-01-02');
SAVEPOINT a;
INSERT INTO Auto (Autonummer, Typ, Einsatz,Kaufdatum)
VALUES ('FR-X 45', 'Golf',true,'2009-01-02');
SAVEPOINT b;
INSERT INTO auto (Autonummer, Typ, Einsatz,Kaufdatum)
VALUES ('FR-X 46', 'Golf',true,'2009-01-02');
SELECT * FROM Auto WHERE typ='Golf';
ROLLBACK TO SAVEPOINT b;
SELECT * FROM Auto WHERE typ='Golf';
ROLLBACK TO SAVEPOINT a;
SELECT * FROM Auto WHERE typ='Golf';
ROLLBACK;
SELECT * FROM Auto WHERE typ='Golf';

```

Dieses Beispiel macht die INSERT – Befehle Schritt für Schritt rückgängig.

Sperren von Tabellen

Damit während einer Transaktion keine anderen Benutzer Daten verändern bzw. nicht mehr aktuelle Daten angezeigt erhalten kann man Tabellen, die von der Transaktion betroffen sind, mit LOCK sperren bzw. mit UNLOCK entsperren.

Voraussetzung ist allerdings, dass in der Konfigurationsdatei „my.cnf“ die Eigenschaft skip-locking mit # auskommentiert ist.

Es gibt in MySQL zwei Arten von LOCK. Der READ – Lock erlaubt anderen Benutzern noch das Lesen der Sätze, während der WRITE – Lock auch dies unterbindet.

Darstellen kann man den LOCK mit 2 getrennt laufenden Query – Browsern.

Im ersten Browser sperrt man die Tabelle, im zweiten versucht man, einen Satz hinzuzufügen:

Query - Browser 1

Query - Browser 2

READ – Lock:

```

1 set autocommit=0;
2 use fuhrpark;
3 START TRANSACTION;
4 lock table auto a read;
5
    
```

```

1 INSERT INTO Auto (Autonummer, Typ, Einsatz, Kaufdatum)
2 VALUES ('FR-X 544', 'Golf', true, '2009-01-02');
3
    
```

Folgende Fehlermeldung erscheint und das System wartet:

„Abfrage wird ausgeführt ...“ und nach einiger Zeit:

!	Fehlernr.	Beschreibung
!	1205	Lock wait timeout exceeded; try restarting transaction

Der Lock wird aufgehoben durch:

```

1 unlock table;
2
    
```

1 Zeile von der letzten Anweisung betroffen, keine Ergebnismenge
2: 19

Der Befehl wurde ausgeführt ...

WRITE – Lock:

```

1 set autocommit=0;
2 use fuhrpark;
3 START TRANSACTION;
4 lock table auto as a WRITE;
5
    
```

```

1 select * from auto;
    
```

Folgende Meldung erscheint und das System wartet:

Abfrage wird ausgeführt...		SQL
1:	1	

Der Lock wird aufgehoben durch:

```

1 unlock table;
2
    
```

Autonummer
C-AL-22
C-DC 331
C-EW-454

Der Befehl wurde ausgeführt ...

Mit den Befehlen lassen sich die in den Beispielen genannten Probleme während der Transaktionsverarbeitung verhindern. Allerdings sollte gerade mit dem LOCK vorsichtig verfahren werden, da ein verzögerter UNLOCK, d.h. ein längeres Sperren von Datenbankbereichen, diese zum Stehen bringen kann.

Benutzer- und Rechteverwaltung

In den Standardeinstellungen von XAMPP wird MySQL mit dem root – User ohne Password betrieben. Dies ist in einer Mehrbenutzerumgebung natürlich nicht sinnvoll, da so kein Zugriff entfernter User auf den MySQL – Server möglich ist und die Datenbank von jedem Nutzer unkontrolliert bearbeitet werden kann.

Es soll daher zunächst ein neuer Nutzer „admin“ mit Kennwort „admin“ angelegt werden, der über alle Rechte verfügt. Zusätzlich ist der Benutzer „gast“ mit Kennwort „gast“ anzulegen, der nur über eingeschränkte Rechte verfügt.

Anlegen von Benutzern

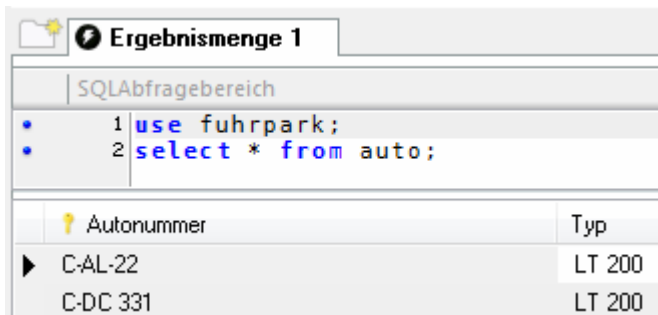
Folgende Befehle sind zu beachten:

Befehl	Funktion	Beispiel
CREATE USER <Name>@<Host>>	Anlegen des Benutzers Das Prozentzeichen bedingt, dass von beliebigem Host angemeldet werden kann.	CREATE USER 'admin'@'%'; CREATE USER 'gast'@'%';
SET PASSWORD FOR <Name>@<Host> PASSWORD(<Kennwort>);	Setzen des Kennwortes für den Benutzer	SET PASSWORD FOR 'admin'@'%'= PASSWORD('admin'); SET PASSWORD FOR 'gast'@'%'= PASSWORD('gast');
GRANT ALL PRIVILEGES ON *.* TO <Name>@<Host>;'	Gewährung aller Rechte auf alle Tabellen für „admin“ und „hugo“	GRANT ALL PRIVILEGES ON *.* TO 'admin'@'%' GRANT ALL PRIVILEGES ON *.* TO 'Hugo'@'%'
REVOKE ALL PRIVILEGES ON *.* FROM <Name>@<Host>;'	Entzug aller Rechte auf alle Tabellen für „hugo“	REVOKE ALL PRIVILEGES ON *.* FROM 'Hugo'@'%'

Anmerkung: „%“ gibt „localhost“ als Server an. Alternativ kann auch die IP – Adresse angegeben werden

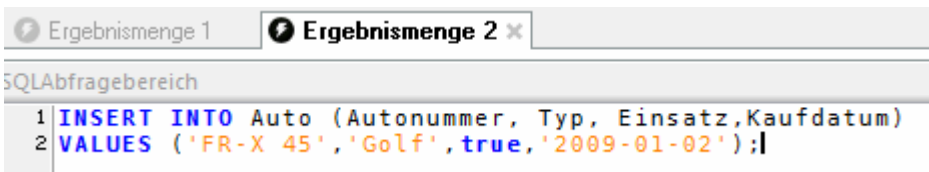


Startet man nun von einem entfernten Rechner den MySQL – Query – Browser, so wählt man sich mit dem neuen Benutzer `gast` unter Angabe des Servernamens bzw. der IP – Adresse ein.

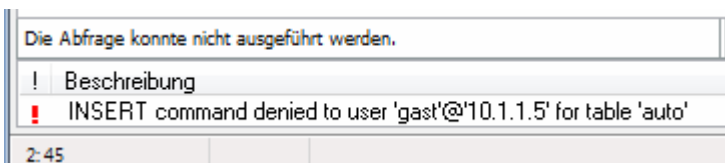


Der SELECT – Befehl funktioniert einwandfrei.

Der INSERT – Befehl



produziert eine Fehlermeldung:



Entzogen werden die Rechte mit REVOKE ... FROM. Der Befehl folgt der gleichen Syntax wie GRANT ... TO, also z. B.

```
REVOKE ALL PRIVILEGES ON *.* FROM 'gast'@'%';
```

Die Beschränkung wirkt. Anstelle des „*.*“ können auch einzelne Tabellen einer Datenbank angegeben werden, z. B. „Fuhrpark.Auto“, also z. B.

```
GRANT SELECT ON Fuhrpark.Auto TO 'gast'@'%';
```

Führt man diese beiden Befehle nacheinander durch, so lässt sich für ‚gast‘ nur noch ein SELECT auf die Tabelle Auto absetzen.

Weitere Rechte lassen sich aus folgender Tabelle entnehmen
(Auszug aus der MySQL - Dokumentation):

Berechtigung	Bedeutung
ALL [PRIVILEGES]	Setzt alle einfachen Berechtigungen außer GRANT OPTION .
ALTER	Erlaubt die Verwendung von ALTER TABLE .
ALTER ROUTINE	Erlaubt die Änderung oder Löschung gespeicherter Routinen.
CREATE	Erlaubt die Verwendung von CREATE TABLE .
CREATE USER	Erlaubt die Verwendung von CREATE USER , DROP USER , RENAME USER und REVOKE ALL PRIVILEGES .
CREATE VIEW	Erlaubt die Verwendung von CREATE VIEW .
DELETE	Erlaubt die Verwendung von DELETE .
DROP	Erlaubt die Verwendung von DROP TABLE .
INSERT	Erlaubt die Verwendung von INSERT .
LOCK TABLES	Erlaubt die Verwendung von LOCK TABLES für Tabellen, für die Sie die Berechtigung SELECT haben.
SELECT	Erlaubt die Verwendung von SELECT .
SHOW DATABASES	SHOW DATABASES zeigt alle Datenbanken.
SHOW VIEW	Erlaubt die Verwendung von SHOW CREATE VIEW .
SUPER	Erlaubt die Verwendung der Anweisungen CHANGE MASTER , KILL , PURGE MASTER LOGS und SET GLOBAL und des Befehls mysqladmin debug . Erlaubt ferner die (einmalige) Verbindungsherstellung auch in dem Fall, dass der Wert für max_connections erreicht wurde.
UPDATE	Erlaubt die Verwendung von UPDATE .
GRANT OPTION	Erlaubt die Gewährung von Berechtigungen.

Trigger

Trigger werden in einer Datenbank verwendet, um beim Eintreten von Datenbankänderungen bestimmte Aktionen auszulösen. Dabei prüft der Trigger bei jeder Datenbankänderung, ob er aktiv werden muss und greift in diesem Fall ein.

Prinzipiell bieten sich alle schreibenden SQL – Befehle wie **INSERT**, **UPDATE** und **DELETE** an. Ereignisses sind dabei jeweils **BEFORE** und **AFTER**.

Ein einfaches Beispiel hierfür wäre eine Änderungsverwaltung. Wird ein Datensatz durch **UPDATE** geändert soll der Zeitpunkt der Änderung erfasst werden und der bis dato gültige Eintrag für die letzte Änderung gesichert werden:

Am Beispiel der Tabelle Treibstoff der Datenbank Fuhrpark soll dies verdeutlicht werden:

Zunächst ist die Struktur der Tabelle Treibstoff um die Felder „aktuell“ für die letzte und „vorher“ für die vorletzte Änderung zu ergänzen:



Ziel ist es nun, einen Trigger zu erstellen, der bei Aktualisierung des Verrechnungspreises das aktuelle Datum in das Feld „Aktuell“ schreibt und vorher den alten Datumswert in das Feld „Vorher“ sichert.

Zuvor muss allerdings die Datenbasis konsolidiert werden. Die folgenden Befehle ergänzen die Tabelle „Treibstoff“ und initialisieren die Werte:

```
ALTER TABLE treibstoff ADD aktuell date;
ALTER TABLE treibstoff ADD vorher date;
UPDATE treibstoff SET aktuell='2009-01-02', vorher=null;
```

Mit „SELECT * FROM treibstoff;“ wird der Datenstand geprüft:

...	Bezeichnung	Verrechnungspr...	aktuell	vorher
B	verbleit	2.6654	2009-01-02	NULL
D	Diesel	2.5112	2009-01-02	NULL
N	Normal bleifrei	2.3958	2009-01-02	NULL
S	Super bleifrei	2.4706	2009-01-02	NULL

Anschließend wird mit „SHOW TRIGGERS;“ zunächst geprüft, ob keine Trigger vorhanden sind.

Da im Trigger zwei Statements zur Aktualisierung benötigt werden, würde ein „;“ das System verwirren. Daher wird ein „DELIMITER |;“ gesetzt, der dem System kenntlich macht, dass nach „|“ der Befehl zu Ende ist. Anschließend wird mit „DELIMITER;“ neutralisiert.

Der Trigger wird eingegeben:

<pre>DELIMITER CREATE TRIGGER geaendert BEFORE update on treibstoff FOR EACH ROW BEGIN SET NEW.vorher = NEW.aktuell; SET NEW.aktuell=now(); end; DELIMITER;</pre>	<p>Nach „ “ ist der Befehl beendet Name des Triggers Kontrollereignis des Triggers Umfang des Triggers, hier: alle aktualisierten Zeilen</p> <p>Start der Triggeraktionen</p> <p>Der Wert aus „aktuell“ wird in „vorher“ gesichert. Die Angabe der Tabelle wie bei SQL – Update ist nicht nötig, da nur „Treibstoff“ infrage kommt</p> <p>„aktuell“ wird mit dem Tagesdatum als Änderungsdatum versehen</p> <p>Ende der Triggeraktionen Setzen des DELIMITERS Aufheben des DELIMITERS</p>
---	---

Mit „SHOW TRIGGERS;“ wird der Trigger aufgelistet.

Zur Kontrolle wird ein Update auf den Verrechnungspreis durchgeführt:

```
UPDATE treibstoff SET verrechnungspreis=verrechnungspreis*1.1
where treibstoffart='D';
```

Mit „SELECT * FROM treibstoff“ wird das Ergebnis kontrolliert:

?	...	Bezeichnung	Verrechnungspr...	aktuell	vorher
▶	B	verbleit	2.6654	2009-01-02	NULL
	D	Diesel	2.7623	2009-01-06	2009-01-02
	N	Normal bleifrei	2.3958	2009-01-02	NULL
	S	Super bleifrei	2.4706	2009-01-02	NULL

Ein weiterer Trigger könnte beim Erfassen einer Treibstoffart das Feld aktuell automatisch mit dem Tagesdatum füllen: Da es dabei nur um einen Befehl geht, wird weder DELIMITER noch BEGIN/END benötigt. Das Ereignis

```
CREATE TRIGGER erfasst
BEFORE INSERT on treibstoff
FOR EACH ROW
SET NEW.aktuell = NOW();
```

Erfasst man nun mit

```
insert into treibstoff(treibstoffart,bezeichnung,verrechnungspreis)
values ('X','Extra',1.95);
```

einen neuen Datensatz, so setzt der Trigger automatisch das Tagesdatum in „aktuell“:

?	...	Bezeichnung	Verrechnungspr...	aktuell	vorher
▶	B	verbleit	2.6654	2009-01-02	NULL
	D	Diesel	2.7623	2009-01-06	2009-01-02
	N	Normal bleifrei	2.3958	2009-01-02	NULL
	S	Super bleifrei	2.4706	2009-01-02	NULL
	X	Extra	1.9500	2009-01-06	NULL

Stored Procedures

Stored Procedures haben die Aufgabe, mehrere hintereinander geschaltete Abfragen automatisiert auszuführen. Dabei können auch lokale Variablen und algorithmische Grundstrukturen eingesetzt werden.

Verwendung finden Stored Procedures vor allem in Verbindung mit Skriptsprachen wie PHP, um den Code effizienter zu gestalten. Durch die Übergabe von Parameter – Werten kann die Abarbeitung und Kontrolle z. B. von UPDATE - und INSERT – Befehlen in die Datenbank verlegt werden.

Befehlssyntax:

```
DELIMITER |  
CREATE PROCEDURE <Name>(IN <Parameter1>, IN <Parameter2>, ...)  
BEGIN
```

<Befehlsfolge>

```
END;  
|  
DELIMITER;
```

Der Aufruf der Prozedur erfolgt dann mit: CALL <Name>(<Wert1>, <Wert2>...);

In einem Beispiel soll die Datenbank um den Sachverhalt der Wartung der Fahrzeuge erweitert werden. Zunächst wird dafür eine neue Tabelle „Werkstatt“ mit WNR (BIGINT) als Schlüssel und dem Werkstattnamen (VARCHAR(20)) als Attribut erstellt.

```
CREATE TABLE Werkstatt(WNR bigint primary key, Werkstattname  
varchar(25));
```

Die Stored Procedure beinhaltet zunächst die CREATE PROCEDURE – Anweisung, die den Übergabeparameter des Werkstattnamens enthält (IN Name varchar(25)).

Dann erfolgt die Festlegung einer Speichervariablen „zaehler“ vom Typ „BIGINT“ mit „DECLARE“.

Anschließend wird der bisher maximale Wert der „WNR“ mit SELECT MAX(WNR); ermittelt und in die Variable „zaehler“ geschrieben (INTO zaehler). Anders als im Kommandozeilen – SQL schreibt INTO nicht in eine Tabelle!

Leider ist bei leerer Tabelle „Werkstatt“ das Maximum = NULL, nicht 0. Daher geht hier ein Zählen nicht. In diesem Fall wird „zaehler“ direkt auf 1 gesetzt. Man benötigt also auch eine Verzweigung.

Der Insert – Befehl wird mit dem Parameter abgesetzt, wobei „zaehler“ um 1 erhöht wird.

```
DELIMITER //  
CREATE PROCEDURE werkstatt_anlegen(IN name varchar(25))  
BEGIN  
    DECLARE zaehler BIGINT DEFAULT 0;  
    SELECT MAX(WNR) INTO zaehler FROM werkstatt;  
    IF zaehler>=1 THEN  
        INSERT INTO werkstatt (WNR,Werkstattname) VALUES  
            (zaehler+1,Name);
```

```

ELSE
  INSERT INTO werkstatt (WNR,Werkstattname) VALUES (1,name);
END IF;
END;
//
DELIMITER ;

```

Fügt man nun z.B. drei Datensätze mit:

```

call werkstatt_anlegen('Müller OHG');
call werkstatt_anlegen('Mayer OHG');
call werkstatt_anlegen('Beck OHG');

```

ein, erhält man mit SELECT folgende Ausgabe:

	WNR	Werkstattname
▶	1	Müller OHG
	2	Meyer OHG
	3	Beck OHG

Aufgrund der vielfältigen Möglichkeiten lassen sich auch komplexe Prozeduren formulieren. Möglich sind auch Funktionen, die den gleichen Aufbau haben wie Prozeduren, allerdings einen Rückgabewert liefern.

Als Beispiel soll eine einfache Dollarumrechnung dienen:

Der Funktion „dollar“ werden zwei Parameter „euro“ und „kurs“ zugeordnet. Mit „return“ wird der Rückgabewert festgelegt.

```

delimiter //
create function dollar(euro double,kurs double) RETURNS double
  BEGIN
    return euro*kurs;
  end;
//
delimiter ;

```

Mit einem SELECT kann die Funktion geprüft werden:

```

select treibstoffart, verrechnungspreis,
dollar(verrechnungspreis,1.35) as Dollarbetrag
from treibstoff;

```

	treibstoffart	verrechnungspreis	Dollarbetrag
▶	B	2.6654	3.59829
	D	2.7623	3.729105
	N	2.3958	3.23433
	S	2.4706	3.33531
	X	1.9500	2.6325

Fazit

Die Ausführungen sollten gezeigt haben, welches Potential Relationale Datenbanken bieten und warum sie insbesondere auch im Bereich Internet, wo riesige Datenmengen auftreten, eine immer größere Rolle spielen.

Die genannten Funktionen gehören schon seit langer Zeit zum Standardrepertoire professioneller Datenbanksysteme, wie ORACLE, INFORMIX, POSTGRES oder MS SQL SERVER.

Mit der Version 5.1 unter Verwendung des Datenbanktyps „InnoDB“ ist MySQL auf dem Weg zu einer professionellen Datenbank. Leider besteht auch heute noch die Möglichkeit, MySQL ohne Verwendung der professionellen Einstellungen und Funktionen zu verwenden, was immer wieder zu Schwierigkeiten in Praxis und Unterricht führt.

Quelle aller Abbildungen:
MySQL Query Browser (Open Source)