Schülerscript 2. DB-Schulaufgabe

Variablen in MySQL

- Variablen dienen zur temporären Speicherung von Werten innerhalb von SQL-Sitzungen, Stored Procedures oder Funktionen.

Typen

Lokale Variablen

- Nur innerhalb eines BEGIN...END-Blocks gültig (z.B. in Stored Procedures / Functions)
- Müssen mit DECLARE deklariert werden
- Existieren nur während der Ausführung der Routine

```
DECLARE variablen_name INT DEFAULT 0;

 SET variablen_name = 42;
```

Session Variablen

- Gültig für die gesamte aktuelle Datenbankverbindung (Session)
- Beginnen immer mit @
- Keine Deklaration notwendig
- Werden automatisch gelöscht, wenn die Session endet

```
SET @meine_variable = 100;

SELECT @meine_variable;
```

Globale System-Variablen

- Gelten für den gesamten MySQL-Server (alle Sessions)
- Beginnen mit @@

- Können nur mit ausreichenden Berechtigungen geändert werden
- Änderungen gelten sofort, aber nicht dauerhaft (nach Neustart zurückgesetzt)

```
SET GLOBAL event_scheduler = ON;
SELECT @@autocommit;
```

Verwendungsbeispiel

```
DELIMITER //
CREATE PROCEDURE var_beispiel()
BEGIN
    -- Lokale Variable
    DECLARE lokal INT DEFAULT 0;
    SET lokal = 10;

    -- Session-Variable setzen SET
    @session_var = lokal * 2;
END // DELIMITER ;

CALL var_beispiel();

SELECT @session_var; -- Ergebnis: 20
```

Stored Procedures

- Gespeicherte Programme auf dem Server, die mehrere SQL-Befehle ausführen Gespeicherte Programme auf dem Server (Datenbank)

- Führen mehrere SQL-Befehle als eine einzige Einheit aus
- Können Parameter akzeptieren und Werte zurückgeben
- Werden zur Kapselung von Geschäftslogik und zur Verbesserung der Leistung/Sicherheit verwendet

Erstellen

DELIMITER //

```
CREATE PROCEDURE proc_name(in param1 int, out
param2 int) BEGIN
  SELECT
    Count(*)
    INTO
     param2

  FROM   users

  WHERE age > param1;
END //
delimiter ;
```

Aufrufen

```
CALL proc_name(18,
@result); SELECT @result;
```

Parameter

Diese Modi legen die **Richtung des Datenflusses** fest.

1. Input (IN)

- Datenfluss **zur** aufgerufenen Einheit (Lesen).
- Übergabe eines Anfangswertes.
- Der Wert in der aufrufenden Einheit bleibt **unverändert**.

2. Output (OUT)

- Datenfluss **von** der aufgerufenen Einheit **zurück** an die aufrufende Einheit.
- Rückgabe eines oder mehrerer Ergebnisse.
- Der zugewiesene Wert **ersetzt** den ursprünglichen Wert beim Aufrufer.

3. Input/Output (INOUT)

- Datenfluss **zur** und **von** der aufgerufenen Einheit (Lesen und Schreiben).
- Dient zur Modifikation eines existierenden Wertes.
- Der **modifizierte** Wert **ersetzt** den ursprünglichen Wert beim Aufrufer.

Löschen

```
DROP PROCEDURE proc_name;
```

Stored Functions

- **Definition:** Gespeicherte SQL-Routinen in der DB. Führen als Einheit aus und **geben immer genau einen Wert zurück**. Benötigen **RETURNS**-Typ. Optional **DETERMINISTIC**.
- **Einsatz:** Kapselung/Wiederverwendung von Logik/Berechnungen. Vereinfachen komplexe `SELECT`-Abfragen. Sichern Datenkonsistenz/Integrität. Performance-Verbesserung (vorkompiliert).

Erstellen

```
DELIMITER //

CREATE FUNCTION
get_user_count(min_age INT) RETURNS
INT

DETERMINISTIC
BEGIN

    DECLARE total INT;

    SELECT COUNT(*) INTO total FROM users WHERE age >=
    min_age; RETURN total;

END // DELIMITER

;
```

Aufruf

```
SELECT get_user_count(18);
```

Löschen

```
DROP FUNCTION get_user_count;
```

Transaktionen

Eine **Transaktion** bündelt mehrere SQL-Befehle zu einer sicheren, logischen Einheit und erfüllt die **ACID**-Prinzipien.

Sie wird immer dann verwendet, wenn:

1. **Mehrere Aktionen zusammengehören und nur als Ganzes gültig sind** (z.B. die Überweisung von Geld, bei der sowohl das Abbuchen als auch das Gutschreiben erfolgreich sein muss).
2. **Die Konsistenz der Datenbank zu jedem Zeitpunkt garantiert werden muss**, auch wenn mehrere Benutzer gleichzeitig Änderungen vornehmen.
3. **Änderungen bei Fehlern oder bewusstem Abbruch rückgängig gemacht werden müssen.**

Beispiel

```
START TRANSACTION;


UPDATE accounts SET balance = balance - 100 WHERE id
= 1; UPDATE accounts SET balance = balance + 100
WHERE id = 2;


COMMIT;
```

Rollback

```
ROLLBACK;
```

Autocommit

```
SELECT @@autocommit;
SET autocommit = 0;
```

Multi-Session

Wenn mehrere Benutzer (Sessions) gleichzeitig versuchen, dieselben Daten zu ändern, können Probleme der Datenkonsistenz und **Datenintegrität** entstehen.

- **Problem:** Ohne Mechanismen zur Steuerung des gleichzeitigen Zugriffs könnten sich Transaktionen gegenseitig beeinflussen und zu falschen Ergebnissen führen (z.B. *Lost Update*, *Dirty Read*).
- **Lösung:** Datenbanken verwenden **Locking** und **Isolation Levels** (siehe unten), um konkurrierenden Zugriff zu regeln. Locks stellen sicher, dass eine Ressource (z.B. eine Zeile in einer Tabelle) während der Bearbeitung durch eine Transaktion für andere Transaktionen gesperrt ist.

Levels

Die **Isolationsebene (Isolation Level)** bestimmt, wie stark sich gleichzeitige Transaktionen gegenseitig beeinflussen dürfen. Sie ist ein Kompromiss zwischen Datenkonsistenz und Performance.

MySQL unterstützt (standardmäßig) vier Isolation Levels, die unterschiedliche Probleme verhindern:

- **READ UNCOMMITTED**
  - **Beschreibung:** Niedrigste Isolation. Eine Transaktion sieht unbestätigte (ungecommittete) Änderungen anderer Transaktionen.
  - **Probleme:** Erlaubt *Dirty Read*, *Non-Repeatable Read* und *Phantom Read*.
- **READ COMMITTED**
  - **Beschreibung:** Eine Transaktion sieht nur Änderungen, die committed wurden.
  - **Probleme:** Verhindert *Dirty Read*, erlaubt aber *Non-Repeatable Read* und *Phantom Read*.
- **REPEATABLE READ**
  - **Beschreibung:** MySQL-Standard. Eine Transaktion erhält bei wiederholtem Lesen dieselben Daten, solange die Transaktion läuft.
  - **Probleme:** Verhindert *Dirty Read* und *Non-Repeatable Read*, erlaubt aber *Phantom Read*.
- **SERIALIZABLE**
  - **Beschreibung:** Höchste Isolation. Die Transaktionen werden seriell ausgeführt, um größtmögliche Konsistenz zu gewährleisten.

- **Probleme:** Verhindert *Dirty Read*, *Non-Repeatable Read* und *Phantom Read*.

Ändern des Isolation Levels:

Eine Transaktion: `SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`

Gesamte Session: `SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;`

ACID

- **Atomicity:** Eine Transaktion ist eine unteilbare Einheit, die entweder komplett ausgeführt wird (*commit*) oder komplett rückgängig gemacht wird (*abort/rollback*). Es gibt keinen "halbfertigen" Zustand.
- **Consistency:** Eine Transaktion überführt die Datenbank von einem gültigen Zustand in einen anderen gültigen Zustand und hält dabei alle definierten Regeln und Integritätsbedingungen ein.
- **Isolation:** Gleichzeitig ablaufende Transaktionen beeinflussen sich nicht gegenseitig.

  Jede Transaktion sieht es so, als wäre sie die einzige, die auf der Datenbank arbeitet.
- **Durability:** Sobald eine Transaktion erfolgreich abgeschlossen (*committed*) wurde, sind ihre Änderungen dauerhaft gespeichert und überleben Systemausfälle wie Stromausfälle oder Abstürze.

Events

- **Reaktion auf Aktionen:** Events ermöglichen es, dass verschiedene Teile einer Anwendung auf bestimmte Vorkommnisse (z.B. Mausklick, Datenänderung) reagieren, ohne direkt miteinander gekoppelt zu sein.
- **Entkopplung und Modularität:** Sie fördern lose Kopplung, da der Auslöser eines Events (Publisher) nicht wissen muss, wer darauf reagiert (Subscriber). Das verbessert die Wartbarkeit und Erweiterbarkeit.
- **Asynchrone Kommunikation:** Events können für die asynchrone Kommunikation verwendet werden, um zeitintensive Aufgaben in den Hintergrund zu verlagern und die Hauptanwendung nicht zu blockieren.

Aktivieren

```
SET GLOBAL event_scheduler = ON;
```

Einmaliges Event

```
CREATE EVENT one_time_event
ON SCHEDULE AT '2026-01-01 00:00:00' DO
  INSERT INTO test VALUES ('Happy New Year');
```

Wiederkehrendes Event

```
CREATE EVENT my_event
ON SCHEDULE EVERY 1
DAY DO
  DELETE FROM logs WHERE created_at < NOW() - INTERVAL 30 DAY;
```

Löschen

```
DROP EVENT my_event;
```

Triggers

  - **Automatisierung:** Automatische Ausführung von Aktionen bei Datenänderungen (INSERT, UPDATE, DELETE).
  - **Datenintegrität/Validierung:** Sicherstellen komplexer Geschäftsregeln und Datenkonsistenz.

- **Protokollierung/Audit:** Nachverfolgen und Aufzeichnen von Änderungen an Daten.

Erstellen

```
DELIMITER //


CREATE TRIGGER before_user_insert
BEFORE INSERT ON users

FOR EACH ROW
BEGIN

    SET NEW.created_at =
NOW(); END //


DELIMITER ;
```

Typen

- **BEFORE INSERT**
- **AFTER INSERT**
- **BEFORE UPDATE**
- **AFTER UPDATE**
- **BEFORE DELETE**
- **AFTER DELETE**

Zugriff

- `NEW.column`
- `OLD.column`

Beispiel

```
DELIMITER //
CREATE TRIGGER
after_delete_user AFTER
DELETE ON users

FOR EACH ROW
BEGIN
    INSERT INTO deleted_users_log(user_id,
    msg) VALUES (OLD.id,

    CONCAT("User gelöscht: ",
OLD.name)); END //

DELIMITER ;
```

Löschen

```
DROP TRIGGER trigger_name;
```

Beispiele aus dem Unterricht

Procedure

```
DELIMITER @@
    CREATE PROCEDURE
       proc_OrtEinfuegen( IN id INT,

    IN ort VARCHAR(255),

    IN schulname VARCHAR(255)
)
BEGIN
    INSERT INTO kurse.ort (ortnr, ort, schule)
    VALUES (id, ort, schulname);

END @@ DELIMITER ;

CALL proc_OrtEinfuegen(71, 'Erlangen', 'Baumschule');
```

Functions

```
DELIMITER $$
CREATE FUNCTION fakultaet(n INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE ergebnis INT DEFAULT 1;
    DECLARE i INT DEFAULT 1;
    IF n < 0 THEN
        RETURN NULL; -- negative Fakultät existiert nicht
    END IF;
    WHILE i <= n DO
        SET ergebnis = ergebnis * i;
        SET i = i + 1;

    END WHILE;
    RETURN ergebnis;
END$$

DELIMITER ;
SELECT fakultaet(5) AS result;
```

Events

```
use bank;

create event test_event_03ALTER

on schedule every 1 minute ends '2026-02-03
20:31:00' do

insert into messages (message, created_at)
values ('test', NOW();
```

Trigger

```
DELIMITER //
CREATE TRIGGER
trg_auth_before_insert BEFORE INSERT
ON auth

FOR EACH ROW
BEGIN
    -- Entferne alle Leerzeichen aus dem neuen Usernamen
    SET NEW.user = REPLACE(NEW.user, ' ', '');

    -- Trage das Ereignis ins Log ein
    INSERT INTO log (wann, wer, was)
    VALUES (

        NOW(),
        USER(),
        CONCAT('Neuer User angelegt: ', NEW.user)
    );
END // DELIMITER ;
```

Transactions

```
set autocommit = OFF;
select * from kunde;
begin;

update kunde set saldo = saldo -500 where id = 1;
update kunde set saldo = saldo +500 where id = 2;

commit; # erst mit dem commit werden die neuen saldos
gespeichert # mit Rollback kann die Änderung also vor dem
commit rückgängig gemacht werden
```

```
set autocommit = ON;
begin;

update kunde set saldo = saldo -500 where id = 1;
update kunde set saldo = saldo +500 where id = 2;

commit; # durch autocommit on hat diese Zeile keinen
Einfluss # Rollback hat keinen Einfluss
```

START TRANSACTION;

```
INSERT INTO tab2 VALUES (NULL, 'Hello', 1);
INSERT INTO tab2 VALUES (NULL, 'World', 2);
INSERT INTO tab2 VALUES (NULL, 'in SQL', 3);
COMMIT;
```

1. Views (Sichten)

Ein View ist eine **gespeicherte Abfrage, die sich wie eine echte Tabelle verhält** (eine virtuelle Tabelle).

- **Wann benutzen?** Wenn du diesen riesigen Triple-Join von vorhin nicht jedes Mal neu tippen willst, oder wenn du bestimmten Benutzern nur einen Teil der Daten zeigen darfst (z.B. eine Tabelle mit Mitarbeitern, aber ohne die Spalte "Gehalt").

Beispiel:
SQL
-- View erstellen
CREATE VIEW kunden_kaeufe AS
SELECT k.name, p.titel FROM kunden k JOIN bestellungen b ON ...;

-- View abfragen (genau wie eine normale Tabelle!)
SELECT * FROM kunden_kaeufe;

-

2. Indizes (Indexes)

Ein Index ist wie das **Inhaltsverzeichnis in einem Buch**. Ohne Index muss MySQL bei der Suche nach "Max Mustermann" jede einzelne Zeile der Tabelle lesen (Full Table Scan). Mit Index springt MySQL direkt zur richtigen Seite.

- **Wann benutzen?** Für Spalten, nach denen du oft suchst (`WHERE`), sortierst (`ORDER BY`) oder die du verknüpfst (`JOIN`).
- **Achtung:** Indizes machen das *Lesen* extrem schnell, aber das *Schreiben* (`INSERT`, `UPDATE`) minimal langsamer, da das Inhaltsverzeichnis aktualisiert werden muss.

Beispiel:
SQL
CREATE INDEX idx_kunden_nachname ON kunden(nachname);

-

3. Constraints (Einschränkungen & Datenintegrität)

Du kennst schon `PRIMARY KEY`. Aber es gibt noch weitere harte Regeln, die du beim Erstellen einer Tabelle definieren kannst, damit kein "Müll" in der Datenbank landet.

- **FOREIGN KEY (Fremdschlüssel):** Garantiert, dass du z.B. keine Bestellung für eine `kunden_id` anlegen kannst, die es in der Kundentabelle gar nicht gibt (Referentielle Integrität).

- **UNIQUE:** Stellt sicher, dass ein Wert in der Spalte nur einmal vorkommt (z.B. E-Mail-Adresse).
- **CHECK:** Prüft eine Bedingung beim Einfügen (z.B. `CHECK (alter >= 18)`).
- **NOT NULL:** Zwingt den Nutzer, einen Wert einzutragen.

4. Benutzerrechte (DCL - Data Control Language)

In der Realität loggt man sich selten als allmächtiger `root`-User ein. Man erstellt Accounts mit beschränkten Rechten.

- **GRANT (Rechte vergeben):** `GRANT SELECT, INSERT ON shop_db.* TO 'azubi'@'localhost';`
- **REVOKE (Rechte entziehen):** `REVOKE INSERT ON shop_db.* FROM 'azubi'@'localhost';`

Privileg	Bedeutung	Ebene
SELECT	Erlaubt das Lesen von Daten aus Tabellen.	Tabelle / Spalte
INSERT	Erlaubt das Hinzufügen neuer Datensätze.	Tabelle / Spalte
UPDATE	Erlaubt das Ändern bestehender Datensätze.	Tabelle / Spalte
DELETE	Erlaubt das Löschen von Datensätzen.	Tabelle
CREATE	Erlaubt das Erstellen neuer Datenbanken oder Tabellen.	DB / Tabelle

DROP	Erlaubt das Löschen kompletter Tabellen oder Datenbanken.	DB / Tabelle
ALTER	Erlaubt das Ändern der Tabellenstruktur (z.B. Spalte hinzufügen).	Tabelle
INDEX	Erlaubt das Erstellen und Löschen von Indizes.	Tabelle
ALL PRIVILEGES	Gibt dem Nutzer alle verfügbaren Rechte (außer Grant Option).	Global / DB
GRANT OPTION	Erlaubt dem Nutzer, seine eigenen Rechte an andere weiterzugeben.	Global / DB

GRANT [RECHTE] ON [DATENBANK].[TABELLE] TO '[USER]'@'[HOST]';

- **Rechte:** Kommagetrennte Liste (z.B. `SELECT, INSERT`).
- **Ebenen (`ON`):**
  - `*.*` : Überall auf dem ganzen Server (Global).
  - `shop_db.*` : Auf alle Tabellen einer bestimmten Datenbank.
  - `shop_db.kunden` : Nur auf eine spezifische Tabelle.
- **User & Host (`TO`):**
  - `'max'@'localhost'` : Max darf sich nur vom selben Rechner einloggen.
  - `'max'@'%'` : Max darf sich von jedem beliebigen Rechner einloggen.

Schritt 3: Rechte prüfen Man kann jederzeit nachsehen, was ein User darf:

SQL
SHOW GRANTS FOR 'support_azubi'@'localhost';

Daten abfragen und manipulieren (DQL & DML)

Befehl	Verwendung (Wann benutzt man das?)	Beispiel
SELECT	Um Daten aus einer oder mehreren Tabellen zu lesen.	`SELECT name, email FROM users;`
DISTINCT	Um doppelte Werte in der Ergebnismenge zu vermeiden.	`SELECT DISTINCT stadt FROM kunden;`
INSERT	Um neue Datensätze in eine Tabelle einzufügen.	`INSERT INTO users (name) VALUES ('Max');`
UPDATE	Um bestehende Werte zu ändern. **Wichtig:** Immer `WHERE` nutzen, sonst änderst du alles!	`UPDATE users SET status = 'aktiv' WHERE id = 5;`
DELETE	Um Datensätze dauerhaft zu löschen.	`DELETE FROM logs WHERE created_at < '2023-01-01';`
WHERE	Um Ergebnisse nach bestimmten Kriterien zu filtern.	`SELECT * FROM produkte WHERE preis > 100;`
ORDER BY	Um die Ausgabe zu sortieren (`ASC` aufsteigend, `DESC` absteigend).	`SELECT * FROM users ORDER BY name ASC;`

LIMIT	Um die Anzahl der ausgegebenen Zeilen zu begrenzen.	SELECT * FROM artikel LIMIT 5;
AS	**Tabellen-Alias:** `FROM kunden AS k` – Ab jetzt kannst du einfach `k.name` statt `kunden.name` schreiben. **Spalten-Alias:** `SELECT name AS Kundenname` – In der Ergebnistabelle steht oben im Header "Kundenname" statt nur "name".	

Joins & Aggregation (Verknüpfungen)

Befehl	Verwendung (Wann benutzt man das?)	Beispiel
INNER JOIN	**Verbindet zwei Tabellen, wenn in beiden eine Übereinstimmung existiert.**	**SELECT u.name, o.id FROM users u INNER JOIN orders o ON u.id = o.user_id;**
LEFT JOIN	**Gibt alle Zeilen der linken Tabelle zurück, auch wenn rechts kein Treffer ist.**	**SELECT u.name, o.id FROM users u LEFT JOIN orders o ON u.id = o.user_id;**

JOIN (Multi)	**Wenn Informationen über drei oder mehr Tabellen verteilt sind.**	**`SELECT u.name, p.titel FROM users u JOIN orders o ON u.id = o.user_id JOIN products p ON o.p_id = p.id;`**
GROUP BY	**Gruppiert Daten, um Berechnungen pro Gruppe durchzuführen (z.B. Summe pro Kategorie).**	**`SELECT kategorie, COUNT(*) FROM produkte GROUP BY kategorie;`**
HAVING	**Wie `WHERE`, aber für gefilterte Gruppen nach einem `GROUP BY`.**	**`SELECT kategorie FROM produkte GROUP BY kategorie HAVING COUNT(*) > 10;`**

Befehl / Variante	Verwendung	Praxis-Beispiel
Triple Join (3 Tabellen)	**Wenn Daten über drei Ecken verknüpft sind (z.B. Kunde -> Bestellung -> Produkt).**	**`SELECT k.name, p.titel`** **`FROM kunden AS k`** **`JOIN bestellungen AS b ON k.id = b.kunde_id`** **`JOIN produkte AS p ON b.produkt_id = p.id`** **`WHERE p.preis > 100;`**

Spalten-Alias (AS)	**Um berechnete Werte oder kryptische Namen schöner anzuzeigen.**	**SELECT (preis * 1.19) AS brutto_preis** **FROM produkte;**
Tabellen-Alias	**Um Tipparbeit zu sparen und Eindeutigkeit bei Joins zu schaffen.**	**SELECT u.name FROM users AS u;**

Vollständige Übersicht aller SQL-Varianten

Hier ist die erweiterte Tabelle inklusive der Konzepte aus deinem Cheat-Sheet:

Kategorie	Befehl	Wann benutzen?	Beispiel
Abfrage	**SELECT**	**Daten auslesen.**	**SELECT * FROM auth;**
Abfrage	**JOIN (Multi)**	**Daten aus mehreren Tabellen kombinieren.**	**SELECT k.ort, s.schulname FROM ort k JOIN schule s ON k.id = s.id;**
Abfrage	**WHERE**	**Ergebnisse präzise filtern.**	**WHERE age >= min_age;**
Manipulation	**INSERT**	**Neue Datensätze anlegen.**	**INSERT INTO log (wann, wer) VALUES (NOW(), USER());**

Manipulation	**UPDATE**	**Bestehende Daten korrigieren.**	**UPDATE kunde SET saldo = saldo - 500 WHERE id = 1;**
Manipulation	**DELETE**	**Daten entfernen.**	**DELETE FROM logs WHERE created_at < NOW();**
Logik	**Procedures**	**SQL-Logik auf dem Server bündeln.**	**CALL proc_OrtEinfuegen(71, 'Erlangen', 'Baumschule');**
Logik	**Functions**	**Berechnungen mit Rückgabewert.**	**SELECT fakultaet(5);**
Automatik	**Trigger**	**Automatische Aktion bei Änderung.**	**BEFORE INSERT ON auth FOR EACH ROW...**
Sicherheit	**Transactions**	**Mehrere Schritte sicher bündeln (ACID).**	**START TRANSACTION; ... COMMIT;**
Variablen	**Session (@)**	**Werte innerhalb der Sitzung merken.**	**SET @session_var = 100;**
Variablen	**Global (@@)**	**Systemeinstellungen ändern.**	**SET GLOBAL event_scheduler = ON;**