SQL / MySQL: SELECT

- **Verknüpfung von Tabellen**
  - Wir machen ein Select über zwei Tabellen (Professor, Vorlesung)
    - Ziel: wir wollen den Dozenten-Namen zu jeder Vorlesung sehen

```
> SELECT *
  FROM   Professor , Vorlesung ;
+--------+----------+--------+-------+--------+
| PersNr | Name     | VorlNr | Titel | PersNr |
+--------+----------+--------+-------+--------+
|     12 | Wirth    |   5001 | ET    |     15 |
|     15 | Tesla    |   5001 | ET    |     15 |
|     20 | Urlauber |   5001 | ET    |     15 |
|     12 | Wirth    |   5022 | IT    |     12 |
|     15 | Tesla    |   5022 | IT    |     12 |
|     20 | Urlauber |   5022 | IT    |     12 |
|     12 | Wirth    |   5045 | DB    |     12 |
|     15 | Tesla    |   5045 | DB    |     12 |
|     20 | Urlauber |   5045 | DB    |     12 |
+--------+----------+--------+-------+--------+
```

    - Offensichtlich werden einfach alle (3*3 = 9) Kombinationen gebildet
    - Sinnvoll sind aber nur die, bei denen PersNr übereinstimmt

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (mit Bedingungen)
  - Wir wollen ja nur bestimmte Datensätze → WHERE-Klausel
    - Idee: WHERE Professor.PersNr = Vorlesung.PersNr

```
> SELECT *
  FROM   Professor , Vorlesung
  WHERE  Professor.PersNr = Vorlesung.PersNr;
+--------+-------+--------+-------+--------+
| PersNr | Name  | VorlNr | Titel | PersNr |
+--------+-------+--------+-------+--------+
|     15 | Tesla |   5001 | ET    |     15 |
|     12 | Wirth |   5022 | IT    |     12 |
|     12 | Wirth |   5045 | DB    |     12 |
+--------+-------+--------+-------+--------+
```

    - Das sind die gewünschten Datensätze.
    - Die Spalte PersNr ist allerdings doppelt. (Frage: Warum?)

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen**
  - Jetzt lassen wir noch die unwichtigen Spalten weg

```
> SELECT Vorlesung.Titel, Professor.Name
  FROM   Professor , Vorlesung
  WHERE  Professor.PersNr = Vorlesung.PersNr;
+-------+-------+
| Titel | Name  |
+-------+-------+
| ET    | Tesla |
| IT    | Wirth |
| DB    | Wirth |
+-------+-------+
```

    - Das Verknüpfungs-Attribut PersNr ist übrigens gar nicht mehr in der Ausgabe.
    - Verständnisfrage: Warum ist die Verknüpfung anhand dieses Elements trotzdem möglich?

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (inner Join)
  - Diese Verknüpfung nennt man einen **inner Join** der Tabellen
    - Es wird das Kreuzprodukt über beide Tabellen gebildet.
    - Dann werden die Kombinationen, die die Join-Bedingung nicht erfüllen, ausgefiltert (siehe WHERE-Bedingung oben)
  - Das besondere am inner Join ist, dass Datensätze beider Tabellen die keinen passenden Join-Partner haben, nicht auftauchen
    - Beispiel: Dozent (20, „Urlauber“) hat keine Vorlesung
  - Es gibt dafür auch ein explizites Konstrukt:
    - FROM … **INNER JOIN** … **ON** …

```
> SELECT Vorlesung.Titel, Professor.Name
  FROM Professor INNER JOIN Vorlesung
       ON Professor.PersNr = Vorlesung.PersNr;
```

    - Das Ergebenis ist das selbe wie zuvor.

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (USING)
  - Werden nur gleichnamige Attribute verglichen,
    so kann im Join anstatt ON auch **USING** verwendet werden
    - Anstatt ON …

```
> SELECT *
  FROM Professor INNER JOIN Vorlesung
       ON Professor.PersNr = Vorlesung.PersNr;
```

    - ... kann man auch USING verwenden:

```
> SELECT *
  FROM Professor INNER JOIN Vorlesung
       USING (PersNr);
```

    - Im letzteren Fall ist zudem das in USING angegebene Attribut nicht mehr doppelt vorhanden:

```
> SELECT * FROM Professor INNER JOIN Vorlesung USING (PersNr);
+--------+-------+--------+-------+
| PersNr | Name  | VorlNr | Titel |
+--------+-------+--------+-------+
|     15 | Tesla |   5001 | ET    |
     ...   .....     .....  ...
```

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (outer Join)
  - Es gibt auch einen **outer Join** zwischen Tabellen
  - Besonders nützlich ist der **left outer Join**
    - Es wird analog zum inner Join vorgegangen
    - Datensätze der linken Tabelle, die keinen Join-Partner haben, werden mit NULL-Werten verknüpft in die Ergebnismenge aufgenommen
      - Bsp.: Professor (20, Urlauber) wird mit NULL-Vorlesungsattributen gelistet

```
> SELECT Vorlesung.Titel, Professor.Name
  FROM Professor LEFT OUTER JOIN Vorlesung
       USING (PersNr);
+-------+----------+
| Titel | Name     |
+-------+----------+
| IT    | Wirth    |
| DB    | Wirth    |
| ET    | Tesla    |
| NULL  | Urlauber |
+-------+----------+
```

    - Beim outer Join (ohne „*left*“) geschieht dies auf beiden Seiten so.

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (Subqueries)
  - Man kann in Queries auch auf das Ergebnis von eingeschachtelten Anfragen Bezug nehmen (**Subqueries**)

```
> SELECT Titel,
         (SELECT Name FROM Professor
                      WHERE Vorlesung.PersNr=Professor.PersNr
         ) AS Name
  FROM Vorlesung;
```

    - Für jeden Datensatz der äußeren Anfrage (über Vorlesung) wird die innere Anfrage (über Professor) einmal ausgeführt.

```
+-------+-------+
| Titel | Name  |
+-------+-------+
| ET    | Tesla |
| IT    | Wirth |
| DB    | Wirth |
+-------+-------+
```

    - Da das ineffizient ist vermeidet man das, wenn auch ein Join möglich ist.
    - Frage: Ist das ein Inner- oder ein Outer Join?

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (Subqueries)
  - **Subqueries** können auch als Bedingungen (in WHERE-Klauseln) wie Werte benutzt werden.

```
> SELECT Titel
  FROM Vorlesung
  WHERE (SELECT Name
         FROM   Professor
         WHERE  Vorlesung.PersNr=Professor.PersNr
        ) = 'Wirth';

+-------+
| Titel |
+-------+
| IT    |
| DB    |
+-------+
```

    - Frage: Was bedeutet diese Anfrage?
  - Sie dürfen dort meist nur einen Wert (Datensatz) liefern
    - Frage: Kann das hier schief gehen?

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen** (Subqueries)
  - **Subqueries** können auch als Datenquellen (in FROM-Klauseln) benutzt werden.
    - Primitives Beispiel:

```
> SELECT *
  FROM  (SELECT Titel FROM Vorlesung);
+-------+
| Titel |
+-------+
| ET    |
| IT    |
| DB    |
+-------+
```

  - Das kann bei komplexeren Anfragen zur klareren Strukturierung dienen.
    - Hier darf der Subquery natürlich mehrere Datensätze liefern.
    - Übung: Geben Sie ein Beispiel an, bei dem das sinnvoll genutzt wird.

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - Wir betrachten nun, wie die Datenstrukturen der Beispiel-Datenbank aus dem obigen Beispiel in SQL erzeugt werden.
    - Vorzugsweise bereitet man die Generierung als SQL-Datei vor und importiert diese dann per Eingabeumleitung auf Kommandozeilenebene in den mysql-Client.

```
[~] mysql < create-schema.sql
```

    - Damit Erzeugung der Datenbank zum Testen immer erneut erfolgen kann, *löschen* wir zuallererst möglicherweise noch existierende frühere Fassungen

```
> DROP   DATABASE IF EXISTS wikipedia_sql_example;
> CREATE DATABASE           wikipedia_sql_example;
> USE                       wikipedia_sql_example;
```

    - Nun existiert die leere Datenbank „*wikipedia_sql_example*“ und ist aktuelle Datenbank.

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - „**CREATE TABLE ...**“ erzeugt eine neue Tabelle
    - **Definition der Spalten** (Name, Typ, Eigenschaften)
    - Angabe von **Tabelleneigenschaften** (Primärschlüssel, …)
  - Beispiel

```
CREATE TABLE Student (
               MatrNr     INT(10)  NOT NULL,
               Name       CHAR(64) NOT NULL,

               PRIMARY KEY (MatrNr)
 );
```

Spaltendefinitionen

Tabelleneigenschaft

    - Die Spalte MatNr ist ein Integer mit maximal 10 Stellen
    - Die Spalte Name ist ein Character-String mit max. 64 Zeichen
    - Primärschlüssel ist die Spalte MatNr
  - Siehe https://dev.mysql.com/doc/refman/8.0/en/create-table.html

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas: Datentypen (Auszug)**
  - Ganze Zahlen: **INT, INTEGER**
    - INT[(length)] [UNSIGNED]
  - Fließkommazahlen: **FLOAT**
    - FLOAT [(length,decimals)] [UNSIGNED]
  - Strings mit fester / begrenzter Länge: **CHAR, VARCHAR**
    - CHAR[(length)] (Strings werden mit Leerzeichen aufgefüllt)
    - VARCHAR(length) (Strings werden mit exakter Länge gespeichert)
    - Optional Angabe von Encoding / Collation
      - ... [CHARACTER SET charset_name] [COLLATE collation_name]
  - Strings mit variabler Länge: **TEXT**
    - Optinal mit Encoding / Collation

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas: Datentypen (Auszug)**

  Viele weitere Datentypen, z.B. ...

  - Zeit/Datum: **DATE, TIME, DATETIME**
  - Aufzählungstypen: **ENUM**
    - ENUM(value1,value2,value3,...)
      - Beispiel: ENUM('yes', 'no', 'perhaps')
  - Binäre Objekte: **BLOB**
    - „**Binary Large Object**“, werden uninterpretiert gespeichert
  - Viele **Typ-Varianten**
    - z.B. zu INT, INTEGER: **TINYINT, SMALLINT, MEDIUMINT, BIGINT**
    - Haben meist unterschiedliche Wertebereiche

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas: Spalten-Eigenschaften**

  Jede Spalte kann weitere Eigenschaften haben, z.B. ...

  - Ist ***NULL*** als Wert erlaubt *(default: ja)*: **NULL**, **NOT NULL**
  - Einen Default-Wert angeben: **DEFAULT** value
    - Beispiel: comment TEXT DEFAULT 'no comment'
  - Werte der Spalte müssen sich unterscheiden: **UNIQUE**
    - NULL-Werte (wenn erlaubt) dürfen mehrfach vorkommen
  - Spalte ist Primärschlüssel: [**PRIMARY**] **KEY**
    - Impliziert NOT NULL und UNIQUE
      - Besser trotzdem explizit angeben!
  - Automatisch neuen Wert setzen: **AUTO_INCREMENT**
    - Wenn beim Einfügen kein Wert angegeben wird, wird ein noch nie genutzter Wert (der zudem größer ist als der maximal vorhandene) eingesetzt.
    - Verständnisfrage: Wozu braucht man das?

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas: Tabellen-Eigenschaften**

  Die Tabelle kann ebenfalls Eigenschaften haben, z.B. ...

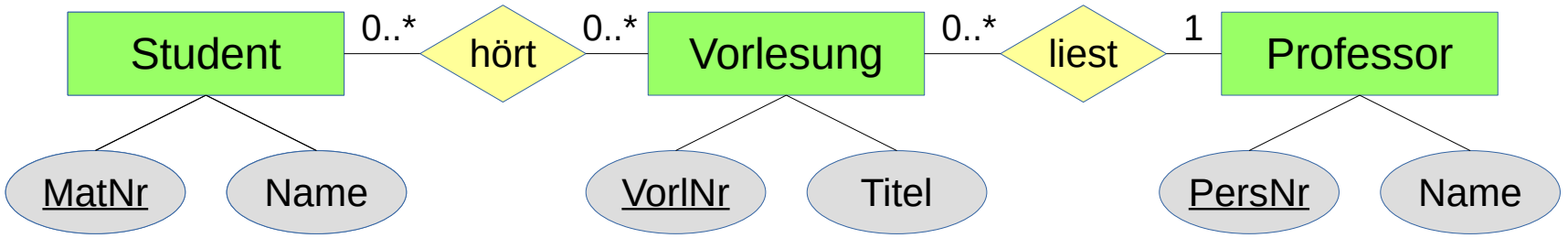
  - Mehreren Spalten müssen sich als Tupel unterscheiden: **UNIQUE**
    - UNIQUE [KEY](index_col_name, …)
    - Wenn mehrere Spalten nicht die selbe Wertkombination haben dürfen
  - Mehrere Spalten sind Primärschlüssel: **[PRIMARY] KEY**
    - PRIMARY KEY (index_col_name,...)
    - Beispiel: PRIMARY KEY (MatrNr, VorlNr)
  - Spalten sind Fremdschlüssel: **FOREIGN KEY**
    - FOREIGN KEY (index_col_name,...) *reference_definition*

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas: Tabellen-Eigenschaften**
  - **FOREIGN KEY** (index_col_name,...) reference_definition
    - **reference_definition**: REFERENCES tbl_name (index_col_name,...)
      [ON DELETE reference_option]
      [ON UPDATE reference_option]
    - **reference_option**: RESTRICT | CASCADE | SET NULL | ...
  - Zur Erinnerung: **Referentielle Integrität**
    - Referenzierte Objekte müssen existieren!
    - Was passiert, wenn der referenzierte Schlüssel **verändert** / **gelöscht** wird?
      - RESTRICT: Das ist nicht erlaubt, so lange es Referenzen gibt
      - CASCADE: Ändere den Fremdschlüssel ebenfalls ab
      - SET NULL: lösche den Fremdschlüssel (auf NULL setzen)
      - SET DEFAULT: Fremdschlüssel auf angegebenen Wert setzen
      - NO ACTION: Erst mal erlauben, am Ende der **Transaktion** (s.u.) prüfen
    - Siehe auch http://en.wikipedia.org/wiki/Foreign_key

SQL / MySQL: Schemaerzeugung

- **Anwendungsbeispiel (SQL-Schema)**
  - ER-Schema



  - Beispiel-Daten

Student

MatNr	Name
26120	Fichte
25403	Jonas
27103	Fauler

hört

MatNr	VorlNr
25403	5001
26120	5001
26120	5045

Vorlesung

VorlNr	Titel	PersNr
26120	ET	15
25403	IT	12
27103	DB	12

Professor

PersNr	Name
12	Wirth
15	Tesla
20	Urlauber

  - Quelle: Deutsche Wikipedia-Seite zu SQL
    - http://de.wikipedia.org/wiki/SQL

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - **Tabelle Student** anlegen

```
CREATE TABLE Student (
                MatrNr      INT(10)UNSIGNED
                              UNIQUE
                              NOT NULL
                              AUTO_INCREMENT,
                Name        CHAR(64)    NOT NULL,

                PRIMARY KEY (MatrNr)
 );
```

- Die MatrNr ist ein vorzeichenloser Integer mit max 10 Dezimalstellen.
  - Da sie **Primärschlüssel** sein soll, ist sie
    - UNIQUE (die Werte dafür müssen verschieden sein, sofern sie nicht NULL sind)
    - NOT NULL (es müssen konkrete Werte benutzt werden, NULL ist verboten)
    - AUTO_INCREMENT (es werden beim Einfügen ggf. automatisch Werte vergeben)
- Der Name ist ein String mit max. 64 Zeichen
    - NOT NULL (es müssen konkrete Werte benutzt werden, NULL ist verboten)
- PRIMARY KEY (MatNr): MatNr ist Primärschlüssel der Tabelle

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - **Tabelle Professor** anlegen

```
CREATE TABLE Professor (
                PersNr      INT(10)UNSIGNED
                              UNIQUE
                              NOT NULL
                              AUTO_INCREMENT,
                Name        CHAR(64)    NOT NULL,

                PRIMARY KEY (PersNr)
 );
```

    - völlig analog zu oben

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - **Tabelle Vorlesung** anlegen

```
CREATE TABLE Vorlesung (
              VorlNr     INT(10)    UNSIGNED
                           UNIQUE
                           NOT NULL
                           AUTO_INCREMENT,
              Titel      CHAR(64)
                           NOT NULL,
              PersNr     INT(10)    UNSIGNED
                           NULL,

              PRIMARY KEY (VorlNr),
              FOREIGN KEY (PersNr) REFERENCES Professor(PersNr)
                        ON DELETE SET NULL
                        ON UPDATE CASCADE
);
```

  - Neu ist hier die **Fremdschlüssel-Definition**
    - PersNr ist **Fremdschlüssel** zum Attribut PersNr in der Tabelle Professor
    - Beim Löschen des referenzierten Professors wir der Verweis auf NULL gesetzt
    - Beim Ändern der Personalnummer des Professors wird die neue übernommen

SQL / MySQL: Schemaerzeugung

- **Erzeugung des Datenschemas**
  - **Beziehungs-Tabelle „hört“** anlegen

```
CREATE TABLE hört (
               MatrNr     INT(10) UNSIGNED,
               VorlNr     INT(10) UNSIGNED,

               UNIQUE  KEY (MatrNr, VorlNr),
               PRIMARY KEY (MatrNr, VorlNr),
               FOREIGN KEY (MatrNr) REFERENCES Student(MatrNr)
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
               FOREIGN KEY (VorlNr) REFERENCES Vorlesung(VorlNr)
                          ON DELETE CASCADE
                          ON UPDATE CASCADE
);
```

    - Neu ist hier der **zweiteilige Primärschlüssel**
      - (MatNr, VorlNr) bilden **gemeinsam** den **Primärschlüssel**
        - Sie sind daher **gemeinsam UNIQUE**
      - MatNr und VorlNr sind einzeln Fremdschlüssel zu Student bzw. Vorlesung
      - Beim Löschen des referenzierten Professors oder der referenzierten Vorlesung wird der betroffen „hört“-Datensatz auch gelöscht.

SQL / MySQL: Daten-Einpflege

- **Einfügen von Datensätzen**
  - für jede Tabelle werden die vorgegebenen Datensätze eingefügt

```
INSERT INTO Student VALUES
              (26120, 'Fichte'),
              (25403, 'Jonas' ),
              (27103, 'Fauler');

INSERT INTO Professor VALUES
              (12, 'Wirth'),
              (15, 'Tesla'),
              (20, 'Urlauber');

INSERT INTO Vorlesung VALUES
              (5001, 'ET', 15),
              (5022, 'IT', 12),
              (5045, 'DB', 12);

INSERT INTO hört VALUES
              (25403, 5001),
              (26120, 5001),
              (26120, 5045);
```

SQL / MySQL: Daten-Einpflege

- **Hinzufügen von (partiellen) Datensätzen**
  - Einen Studenten mit Namen „Neumann“ einfügen.

```
INSERT INTO Student (Name) VALUES
                ('Neumann');
```

    - Es wurde kein Wert für MatrNr angegeben, obwohl es Primärschlüssel ist.

```
SELECT * FROM Student;
+--------+---------+
| MatrNr | Name    |
+--------+---------+
|  25403 | Jonas   |
|  26120 | Fichte  |
|  27103 | Fauler  |
|  27104 | Neumann |
+--------+---------+
```

    - Erklärung: MatrNr hatte ja Eigenschaft „AUTO_INCREMENT“

```
CREATE TABLE Student (
    MatrNr   INT(10) UNSIGNED ... AUTO_INCREMENT, ...
```

SQL / MySQL: Daten-Einpflege

- **Hinzufügen von (partiellen) Datensätzen**
  - Einen weiteren neuen Studenten mit Namen „Fauler“ einfügen.

```
INSERT INTO Student (Name) VALUES
                ('Fauler');
```

```
SELECT * FROM Student;
+--------+---------+
| MatrNr | Name    |
+--------+---------+
|  25403 | Jonas   |
|  26120 | Fichte  |
|  27103 | Fauler  |
|  27104 | Neumann |
|  27105 | Fauler  |
+--------+---------+
```

- Wie finde ich den Primärschlüssel heraus? Der Name genügt ja hier nicht.
- Die Funktion LAST_INSERT_ID() liefert diese Information:

```
SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|            27105 |
+------------------+
```

SQL / MySQL: Daten-Einpflege

- **Löschen von Datensätzen**
  - Alle Professoren mit dem Namen „Urlauber“ werden gelöscht

```
DELETE FROM Professor
              WHERE Name = 'Urlauber';
```

    - Achtung: Name ist in Professor nicht UNIQUE (und kein Primärschlüssel)
      - Es könnten mehrere Datensätze gelöscht werden

- **Ändern von Datensätzen**
  - Dem Professor mit PersNr = 20 einen neuen Namen geben

```
UPDATE Professor
              SET   Name = 'Schaffer'
              WHERE PersNr = 20;
```

    - PersNr ist Primärschlüssel, d.h. kann hier nur ein Datensatz betroffen sein
      - Vorsicht: Ohne WHERE-Klausel Würden alle Datensätze modifiziert werden!
  - Siehe https://dev.mysql.com/doc/refman/8.0/en/sql-data-manipulation-statements.html

SQL / MySQL: Daten-Einpflege

- **Ändern von Tabellen (Datenbank-Schema Modifikation)**
  - Den Studenten ein Attribut „Vorname“ hinzufügen

```
ALTER TABLE Student
              ADD COLUMN Vorname CHAR(64) NULL;
```

    - Tipp: Da hier kein Default-Wert angegeben ist, wird das Attribut beim Anlegen der Spalte erst mal in allen Datensätzen auf NULL gesetzt. Deshalb muss das auch (zunächst) erlaubt sein (kann ggf. später wieder entfernt werden).

      Alternative: **Transaktionen** (s.u.).

  - Den Studenten das Attribut „Vorname“ wieder entfernen

```
ALTER TABLE Student
              DROP COLUMN Vorname;
```

  - Praktisch alle Aspekte der Tabellendefinition können nachträglich geändert werden
    - z.B. Typ, Primärschlüssel, NULL / NOT NULL, UNIQUE …
    - Siehe **Syntax**: https://dev.mysql.com/doc/refman/8.0/en/alter-table.html
      Siehe **Beispiele**: https://dev.mysql.com/doc/refman/8.0/en/alter-table-examples.html

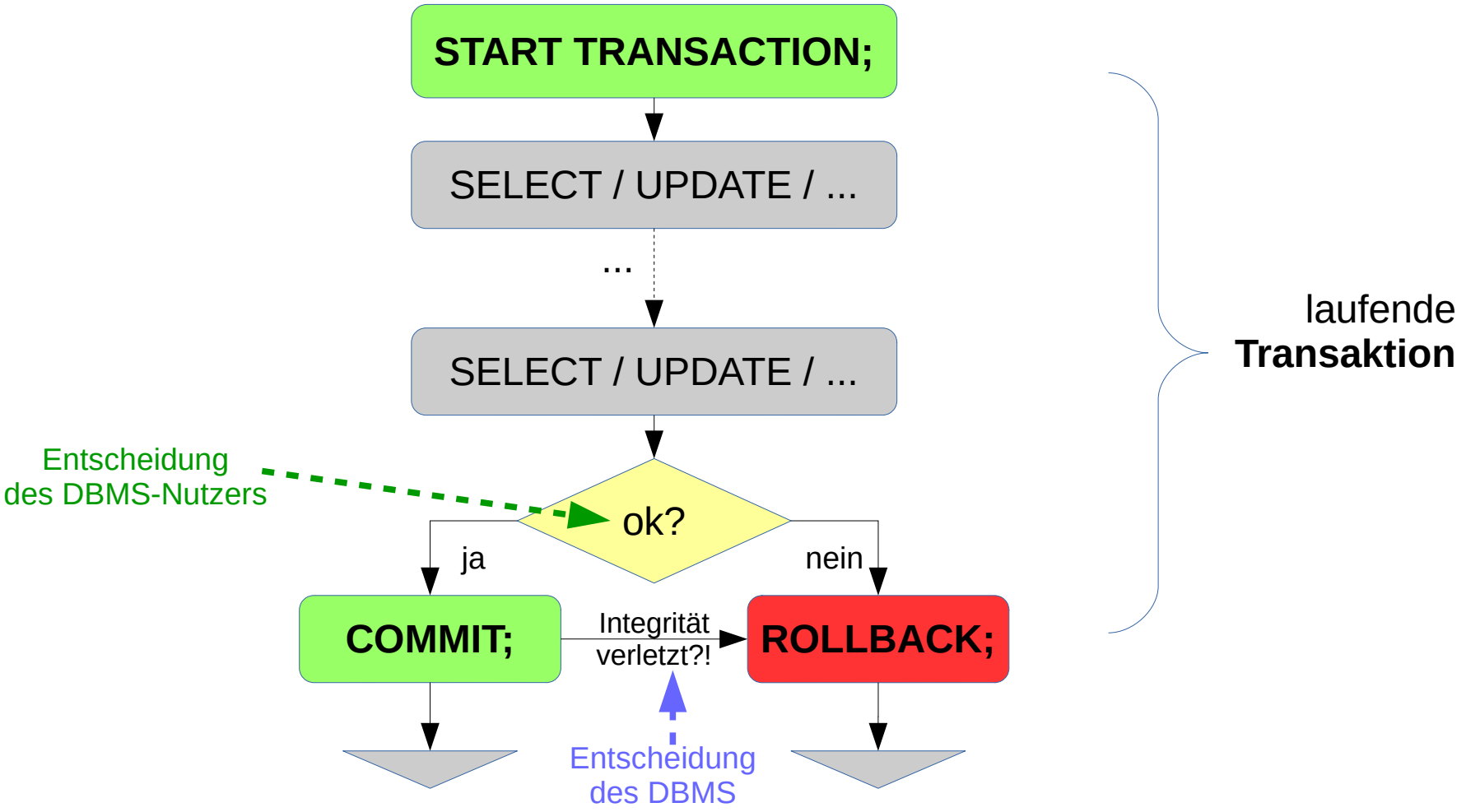
SQL / MySQL: Integrität

- **Datenbankintegrität: Komplexe Operationen**
  - Datenbanken garantieren Integritätsbedingungen
    - Verletzt eine Operation die Integrität, wird ihre Ausführung verweigert
  - Um komplexere Operationen durchführen zu können, müssen Integritätsbedingungen kurzfristig auch einmal verletzt werden.
    - Am Ende der (komplexen) Operation müssen sie aber wieder gelten!
    - Beispiel: **Bank-Überweisung** Summe X von Konto A nach Konto B
      - Summe X von Konto A abbuchen
      - Summe X auf Konto B hinzubuchen
      - Logbuch der durchgeführten Überweisungen ergänzen

      Danach muss z.B. die Summe aller Kontostände wieder gleich sein.
  - Das DBMS muss dazu wissen, …
    - welche Operationen logisch zusammen gehören
    - wann die Integritätsbedingungen dann wieder gelten sollen
    - was es tun soll, wenn sie dann immer noch verletzt ist
  - Dazu brauchen wir eine „**semantische Klammer**“ für Operationen

SQL / MySQL: Integrität

- **Datenbank-Transaktionen**
  - Eine solche „semantische Klammer“ für Operationen nennt man **Transaktion**



SQL / MySQL: Integrität

- **Explizite Datenbank-Transaktionen**
  - Transaktion beginnen: **START TRANSACTION** oder **BEGIN**
    - Startet neue Transaktion
  - Transaktion erfolgreich beenden: **COMMIT**
    - Die Konsistenz wird geprüft.
    - Bei Erfolg werden die Änderungen seit Transaktionsbeginn abgespeichert.
    - Bei verletzten Konsistenzbedingungen wird ROLLBACK ausgeführt.
  - Transaktion abbrechen: **ROLLBACK**
    - Alle Änderungen seit Transaktionsbeginn werden rückgängig gemacht.
- **Automatische Transaktionen: Autocommit**
  - Autocommit bedeutet, dass jede Aktion sofort Committed wird.
  - Steuerbar mit „SET autocommit = {0 | 1}“
    - Außerhalb von Transaktionen normalerweise auf 1
    - Innerhalb von Transaktionen auf 0

SQL / MySQL: Integrität

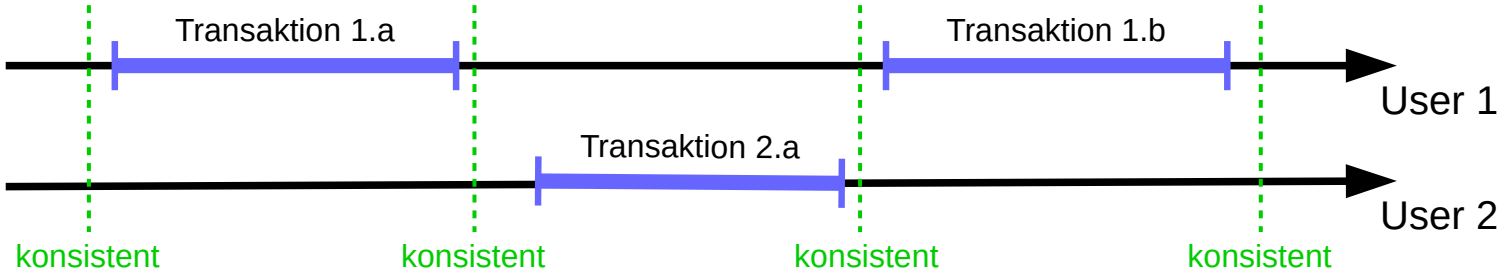
- **Konzept hinter Transaktionen und Integrität: „ACID“**

  Siehe auch http://de.wikipedia.org/wiki/ACID

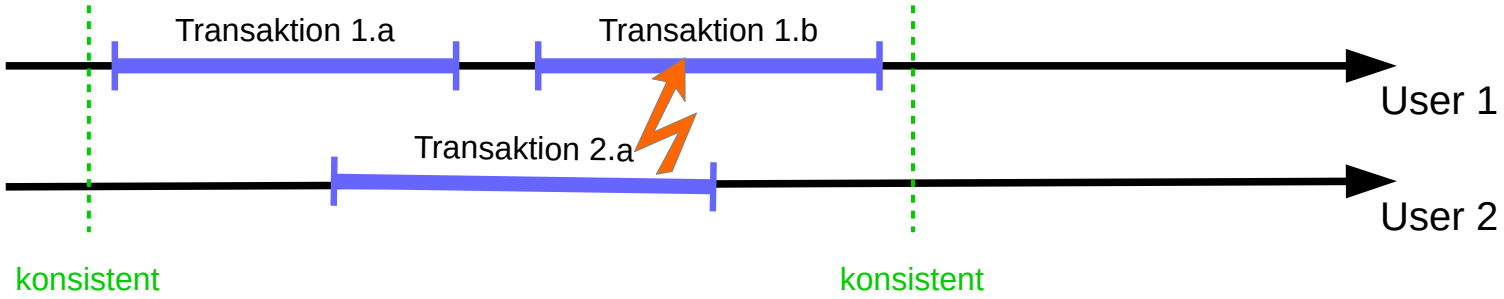
  - „A“ = **Atomarität** (engl. „Atomicity“, Abgeschlossenheit)
    - Jede Transaktion wird **ganz oder gar nicht** ausgeführt
      - z.B. die Bank-Überweisung von oben darf nicht halb ausgeführt werden, sonst geht im Saldo Geld verloren oder entsteht
  - „C“ = **Konsistenzerhaltung** (engl. „Consistency“)
    - Nach jeder Transaktion müssen **alle Konsistenzbedingungen erfüllt** sein
      - z.B. durch Fremdschlüssel referenzierte Objekte müssen existieren
  - „I“ = **Isolation** (engl. „Isolation“, **logischer Einbenutzerbetrieb**)
    - Nebenläufig ausgeführte Transaktionen führen nicht zu Ergebnissen, die nicht auch durch eine sequentielle Ausführung erklärbar ist.
      - Hier gibt es aus Performancegründen auch Abschwächungen der Isolation (s.u.)
  - „D“ = **Dauerhaftigkeit** (engl. „Durability“)
    - Nach einem erfolgreichen Commit gehen keine der Änderungen mehr verloren
      - Auch nicht durch einen **Systemabsturz** oder **(tolerierbaren) Hardwarefehler**

SQL / MySQL: Integrität

- **Probleme der semantischen „Isolation“**
  - Nur perfekt, wenn alle Transaktionen strikt serialisiert werden
    - Strikte Serialisation (reduziert Performance)



  - Man kann versuchen, Transaktionen parallel auszuführen
    - Höhere Performance, aber auch **Gefährdung der Isolation**



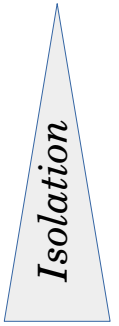
    - **Wunsch**: Es gibt eine strikt sequentielle Ausführung, die äquivalent ist

SQL / MySQL: Integrität

- **Probleme der semantischen „Isolation“**
  - Oft ist perfekte Isolation aber gar nicht erforderlich
  - Eingeschränkte Isolation verursacht „**Read Phenomena**“
    - **Dirty Read**
      - Daten einer noch nicht abgeschlossenen fremden Transaktion werden gelesen
    - **Lost Updates** (eigentlich kein „Read Phenomenon“)
      - Zwei Transaktionen modifizieren parallel denselben Wert / Datensatz.
        Ein Wert setzt sich am Ende durch, der andere geht verloren.
    - **Non-Repeatable Read**
      - Wiederholte Lesevorgänge liefern bei den selben Datensätzen unterschiedliche Attributwerte.
    - **Phantom Read**
      - Wiederholte Lesevorgänge liefern eine andere Menge von Datensätzen.
  - Siehe auch
    - http://en.wikipedia.org/wiki/Isolation_(database_systems)
    - http://de.wikipedia.org/wiki/Isolation_(Datenbank)

SQL / MySQL: Integrität

- **Probleme der semantischen „Isolation“**
  - Isolations-Niveau Steuerbar in SQL:
    - **SET TRANSACTION ISOLATION LEVEL** level
  - Werte für level:
    - READ UNCOMMITTED
    - READ COMMITTED (keine Dirty Reads)
    - REPEATABLE READ (keine Dirty + Non-Repeatable Reads)
    - SERIALIZABLE (keine Read-Phenomena mehr)



  - Das DBMS versucht aber oft, Zugriffe stärker zu parallelisieren.
    - Bei dadurch verursachten Verletzungen der Isolations-Semantik muss das DBMS die betroffene **Transaktion abbrechen**.
    - Diskussion: Was bedeutet das für die betroffenen User?
  - Es gibt hier noch weitere SQL-Mechanismen (z.B. Table-Locking)
    - Siehe auch https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-transactions.html

SQL / MySQL: Integrität

- **Problem bei spekulativer Parallelausführung**
  - Das DBMS soll möglichst viele Aktivitäten parallel ausführen
    - Es „weiß“ zu Beginn einer Nutzer-Transaktion aber nicht, auf was alles zugegriffen wird
  - Kollisionen werden erst spät (während der Transaktion) festgestellt
- **Idee: Vorher sagen, was man vor hat: Table Locking**

```
LOCK TABLES   Student    READ,
              Vorlesung  READ,
              hört       WRITE;
```

→

```
SELECT ... ;
UPDATE ... ;
DELETE ... ;
```

→

```
UNLOCK TABLES;
```

  - Das „`LOCK TABLES`“ *blockiert* (wartet), falls eine Tabelle gerade anderweitig gelockt ist.
    - Lesendes Locking kann parallel erfolgen, schreibendes Locking nicht.
  - Alle Table-Locks müssen auf einmal angefordert werden.
    - *Verständnisfrage: Was könnte sonst passieren?*

SQL / MySQL: Aktive Datenbankoperationen

Wir betrachten zuletzt noch einige **fortgeschrittene Konzepte** von SQL und DBMS

- **SQL ermöglicht es, mit Daten aktiv umzugehen:**
  - Mehr als nur einfaches INSERT, DELETE und UPDATE
    - ON UPDATE und ON DELETE ermöglicht kaskadierte Reaktionen auf Änderungen bei Fremdschlüsseln (s.o.)
    - **Trigger** ermöglichen kaskadierte Reaktionen auf *beliebige* Änderungen
    - **Stored Procedures** ermöglichen komplexe Abläufe im DBMS zu realisieren
  - Mehr als nur statische Daten in Tabellen:
    - **Views** bilden dynamisch berechnete Tabellen
    - **Stored Functions** berechnen dynamisch Daten
  - Dies verlagert einen Teil der Applikationslogik in das DBMS
    - **Vorteile**: Erweiterte Konsistenz-Garantien, Zugriffsschutz, Abstraktion
    - **Nachteile**: Abhängigkeit vom konkreten DBMS steigt, Komplexität im DBMS

SQL / MySQL: Aktive Datenbankoperationen

- **Views**
  - **Views** ermöglichen es, das Ergebnis eines SELECTs in der Datenbank wie eine reale Tabelle darzustellen
    - Beispiel: Professoren als Teilmenge aller Personen (fiktives Schema)

```
CREATE VIEW Professor AS
    SELECT * FROM Person WHERE Statusgruppe = 'PROF';
```

  - Auf die View kann z.B. mit SELECT zugegriffen werden

```
SELECT * FROM Professor WHERE Name = 'Urlauber';
```

  - Views können (fast) beliebige SELECTS beinhalten
    - Z.B. Auch komplexe Queries mit Joins, berechneten Attributen, etc.
  - Siehe auch: https://dev.mysql.com/doc/refman/8.0/en/create-view.html

SQL / MySQL: Aktive Datenbankoperationen

- **Views**
  - Views sind nützlich, um ein **vereinfachtes Datenschema** bereit zu stellen
    - z.B. als **langlebige API** für Fremdprogramme
    - um frühere Tabellenstrukturen (für Legacy-Programme) zu simulieren
  - Es gibt auch Updatable Views und Insertable Views
    - Diese können per „UPDATE“ oder „INSERT“ beschrieben werden
    - Änderungen müssen auf die zugrundeliegenden Tabellen abgebildet werden
    - Dies hat oft eine hohe Komplexität
      - Beispiel: Insert in eine View, die nur einen Teil der Attribute bereithält
      - Beispiel: Insert in eine View, die einen Join darstellt
    - Dazu muss ggf. **benutzerdefinierter Code** im DBMS ausgeführt werden

SQL / MySQL: Aktive Datenbankoperationen

- **Stored Procedures / Functions (Stored Programs)**
  - Im DBMS kann benutzerdefinierter Code abgelegt werden
    - Beispiel (*Skizze*): Eine komplette Überweisung durchführen

```
CREATE PROCEDURE ueberweisung(kto_from INT, kto_to INT, sum FLOAT)
     BEGIN
     START TRANSACTION;
          UPDATE Konto SET saldo = saldo - sum WHERE nummer=kto_from;
          UPDATE Konto SET saldo = saldo + sum WHERE nummer=kto_to;
    COMMIT;
     END
```

  - Der Code wird vom DBMS ausgeführt
    - Durch **expliziten Aufruf** z.B. per CALL (Prozeduren)

```
CALL ueberweisung(123456, 738521, 100.00);
```

    - … oder z.B. durch SELECT (liefert Ergebnis von Funktionen)

```
SELECT passwort_test('mueller', 'geheim2345');
```

Bsp. dazu: s.u.

    - Aufruf auch durch **Trigger-Ereignisse**
  - Siehe auch: https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

SQL / MySQL: Aktive Datenbankoperationen

- **Stored Procedures / Functions (Stored Programs)**
  - Die Berechtigung zum Aufruf kann über das Rechtesystem gesteuert werden
  - Man brauch zum Aufruf aber nicht die unmittelbaren Zugriffsrechte auf die benutzten Tabellen
    - Dadurch kann man Teile der DB vor direktem Zugriff schützen
  - Beispiel: Überweisung
    - Ein Benutzer darf eine Überweisung tätigen (CALL ueberweisung())
    - Er darf aber nicht direkt das Attribut Konto.saldo ändern.
      → **Erweiterte Konsistenz** *(„Geld kann nicht verloren gehen oder entstehen“)*
  - Beispiel: Passwort-Test anhand Benutzer-Datenbank
    - Ein Benutzer (z.B. PHP-Script) darf ein konkretes Passwort auf Korrektheit testen (Vergleich mit Passwort (-Hash) in der DB)
    - Er darf aber nicht die (ggf. gehashten) Passwörter aus der DB lesen
      → **Sicherheit** *(das PHP-Script kennt nur das gerade zu testende Passwort)*

SQL / MySQL: Aktive Datenbankoperationen

- **Trigger**
  - Das DBMS kann aktiv auf Daten-Änderungen reagieren
    - Beispiel: Einen neuen Benutzer-Datensatz vor dem Einfügen vervollständigen (Zeitstempel anlegen, Passwort verschlüsseln)

```
CREATE TRIGGER creating_new_user
     BEFORE INSERT ON Accounts
     FOR EACH ROW
          BEGIN
               SET NEW.TimeStampCreated = NOW(),
                    NEW.Password         = MD5(NEW.Password);
          END
```

  - Der Trigger-Body (BEGIN … END) wird ausgeführt ...
    - … vor („BEFORE“) oder nach („AFTER“) einem ...
    - … „INSERT“, „DELETE“ oder „UPDATE“ auf der angegeben Tabelle.
    - Mit „NEW“ und „OLD“ kann auf den neuen bzw. früheren Datensatz Bezug genommen werden
  - Siehe auch: https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html

SQL / MySQL: Variablen

- **Variablen**
  - Zwischenergebnisse können in User-**Variablen** abgelegt werden
    - Variablen-Namen: @... , z.B. @age
    - Zuweisung: **SET** @var_name = expr , @var_name = expr , ...
    - Beispiel: **SET** @age = 22
  - Variablen können in allen Expressions verwendet werden
    - Beispiel: Ausgabe mit SELECT

```
SET @age = 22;
SELECT @age, @age+1;
+------+--------+
| @age | @age+1 |
+------+--------+
|   22 |     23 |
+------+--------+
```

SQL / MySQL: Variablen

- **Variablen**
  - Die Ergebnisse eines SELECTs können mit SELECT … INTO Variablen zugewiesen werden
    - Beispiel:

```
SELECT UserId, FirstName
     FROM Accounts WHERE LastName = 'Mayer'   LIMIT 1
          INTO @uid, @fn;
```

    - Select-Ergebnis darf nur ein einzelner Record sein (ggf. „LIMIT 1“)
- **Systemvariablen**
  - Systemvariablen steuern Eigenschaften der DB oder Session
    - Zuweisung: **SET** [GLOBAL | SESSION] system_var_name = expr
    - **Beispiel**: **SET** GLOBAL **sql_mode** = 'STRICT_ALL_TABLES';
      - Dies aktiviert eine **strikte Prüfung** von Parametern z.B. bei INSERT
      - Ist ein z.B. Wert zu lang für einen Attribut-Typ, so bewirkt dies einen Fehler)
      - Der Default bei MySQL ist, nur eine Warnung auszugeben (sql_mode = ‘‘)
      - Siehe auch https://dev.mysql.com/doc/refman/8.0/en/sql-mode.html

SQL / MySQL: Import, Export

- **CSV-Daten-Export**
  - CSV = *Comma Separated Values*, z.B. aus Tabellenkalkulation
  - Beispiel:

```
SELECT * FROM test
    INTO OUTFILE '/tmp/test.csv'
    FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
    LINES  TERMINATED BY '\n' STARTING BY '' ;
```

  - Siehe https://dev.mysql.com/doc/refman/8.0/en/select.html
- **CSV-Daten-Import**
  - Beispiel:

```
LOAD DATA INFILE '/tmp/test.csv'
INTO TABLE test
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES  TERMINATED BY '\n' STARTING BY '' ;
```

  - Siehe https://dev.mysql.com/doc/refman/8.0/en/load-data.html

SQL / MySQL: Import, Export

- **SQL-Datenbank-Export** (Backup)
  - Tool „mysqldump“ (auf Unix-Shell)

```
# mysqldump db_name > backup-file.sql
```

  - Sichert komplettes Schema und Daten
    - Da das resultierende SQL-File die Datenbank später komplett neu aufbaut, ist es eine interessante Quelle um SQL zu lernen.
    - Mit Option „`--single-transaction`“ transaktionsgeschütztes Backup

- **CSV-Datenbank-Import** (Restore)
  - Tool „mysql“ (auf Unix-Shell)

```
# mysql db_name < backup-file.sql
```