Web 2.0 Technologien 2

Dr. Joachim Thees

joachim.thees@cs.rptu.de

FB Informatik

RPTU Kaiserslautern / Landau

INF-00-32-V-3 (2V+1Ü) @ SS 2026

Themen und Ziele

- **Web 2.0 Technologien 1** *(im **WiSe**)*

  - Kenntnis der Techniken, Schnittstellen, Protokolle des „*Web 2.0*“
    - HTTP, HTML 5, CSS 3, Javascript, ...
  - Fähigkeit grundlegende und aufbauende Techniken zu verstehen
    - Standards lesen und verstehen können, Techniken selbst erarbeiten können
  - Grundlegende Fähigkeit zur Realisierung von Web-Services
    - Webserver, PHP, (HTML 5, CSS 3, Javascript), …

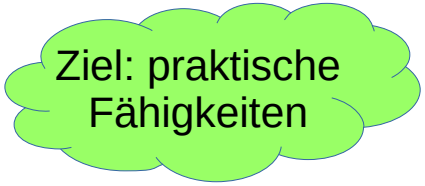
- **Web 2.0 Technologien 2** *(im **SoSe**)*

  - Kenntnisse fortgeschrittener Techniken, Frameworks
    - Applikationsstrukturen, Datenmodelle, Informationssysteme, Zuverlässigkeit
    - Server-Frameworks, Client-Frameworks, Abstraktion, Reaktivität, …
  - Sensibilisierung für Security / Angriffswege, Datenschutz, Privacy
    - Angriffe, Verteidigungstechniken, Authentifizierung, Tracking, …
  - Kompetenz sichere und robuste Webapplikationen zu entwickeln

Struktur der Lehrveranstaltung

- **Vorlesung**
  - Einführung von Konzepten und Techniken
    - „Was?“, „Wozu?“
  - Demonstration von Techniken anhand von Beispielen
    - „Wie?“, „Womit?“, technische Randbedingungen

- **Übung** (als Gruppenarbeit)
  - Praktische Übungsaufgaben vorab (oft am Rechner) bearbeiten
    - z.B. Realisierung einer Funktionalität auf dem Übungs-Webserver
  - Fragestunde (freiwillig)
    - Demonstration der einzelnen Lösungen, Diskussion
  - Aktive Teilnahme ist Voraussetzung für Prüfungszulassung!

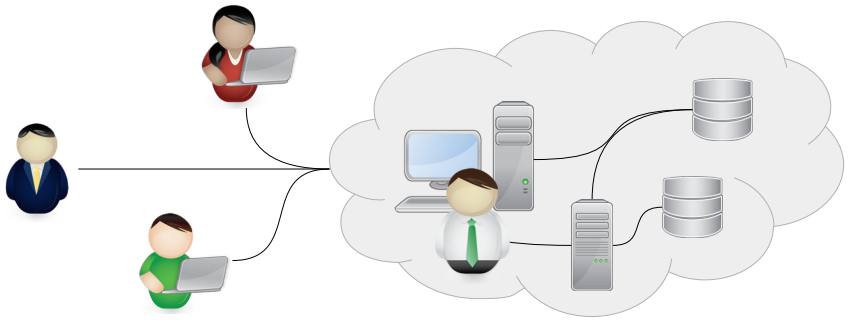


Kapitel 1:

Informationssysteme: ERM + DBM + SQL

Informationssysteme

- „Informationssystem“ (IS)

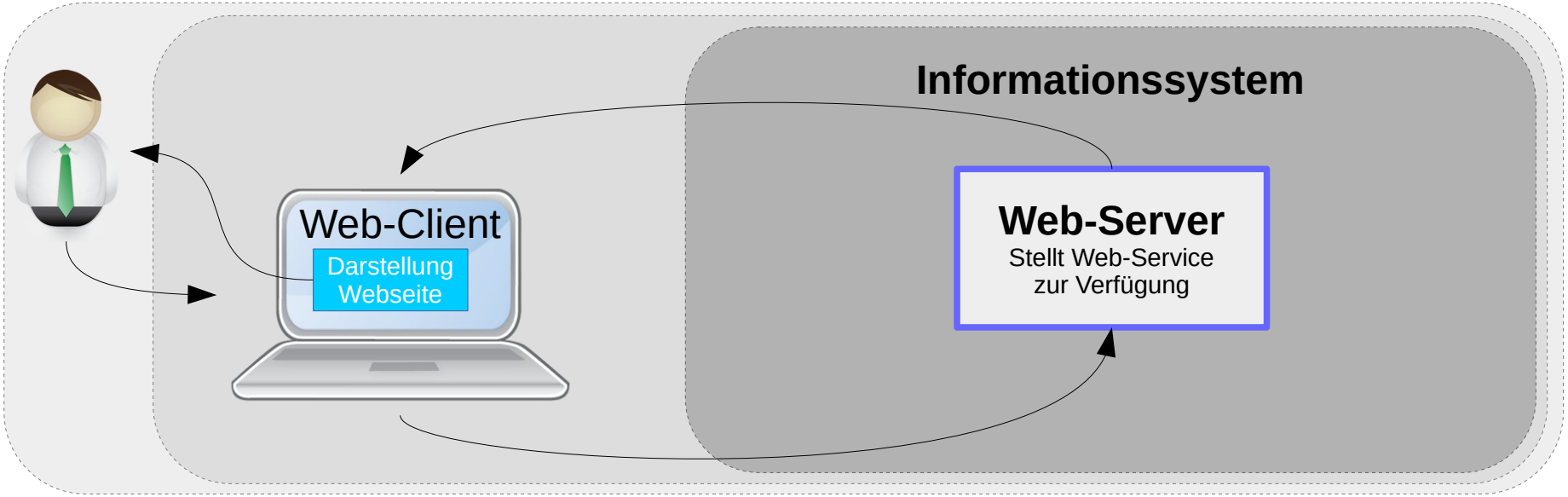


  - Eigenschaften und Ziele
    - produziert, beschafft, verteilt und verarbeitet **Daten** (bzw. **Informationen**)
    - Ziel: Deckung von Informationsnachfrage
    - **Mensch-** / **Aufgabe-** / **Technik-**System (soziotechnisches System)
  - Technische Aspekte
    - Dienste, Server, Kommunikation, Protokolle, Programmiersprachen, …
  - Rolle des Mensch
    - Nutzer von Diensten (innerhalb oder außerhalb des Systems)
    - Funktionsträger (Anbieter von Diensten, Verarbeiter von Daten, …)
  - Abgrenzung des Systembegriffs ist unscharf

Informationssysteme

- **Beispiel: Webservices (bzw. Webserver)**
  - Liefern Informationen auf Basis von eigenen Daten
    - z.B. GET-Request „*gib mir die Liste der Vorlesungen*“
  - Verarbeiten Informationen, die sie aus Requests erhalten
    - z.B. POST-Request „*melde mich für die Vorlesung XYZ an*“
  - Informationen werden „*produziert, beschafft, verteilt und verarbeitet*“

→ ***Frage**: Was ist hier ein IS?* → *Viele Sichtweisen!*

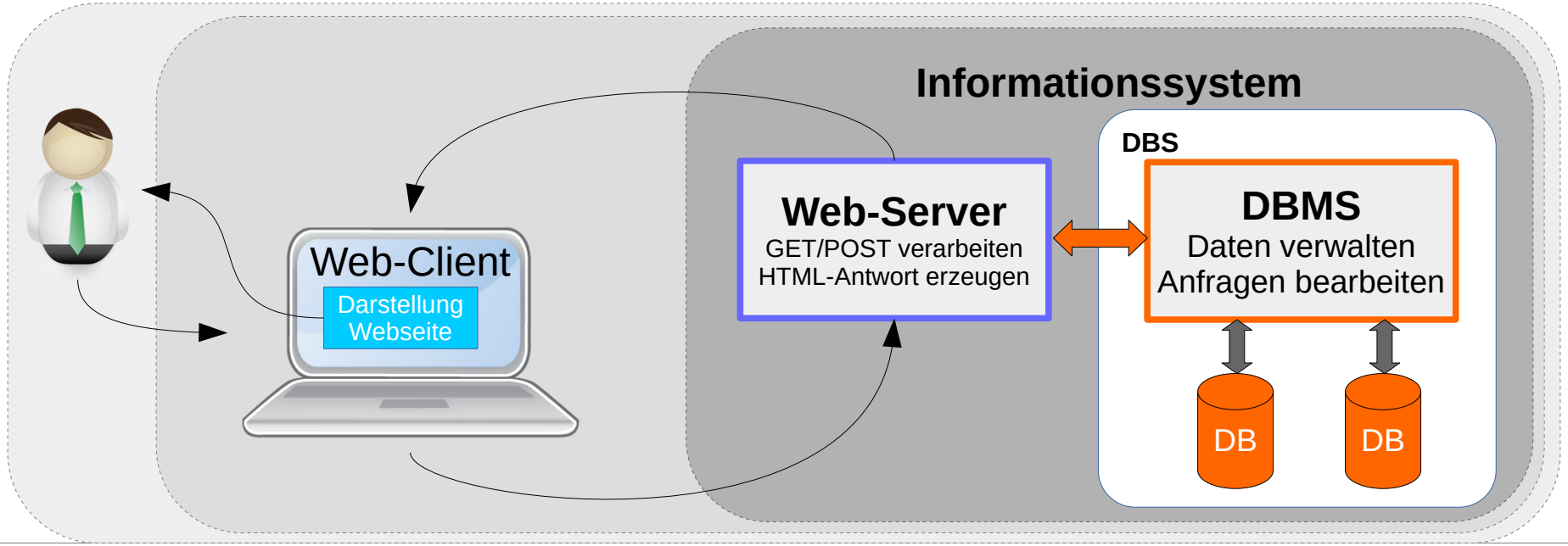


Informationssysteme

- **Webservices**
  - Verarbeiten also **Daten**
    - liefern Informationen, speichern Änderungen, ...
  - Webserver sind aber doch **zustandslos** *(HTTP)*

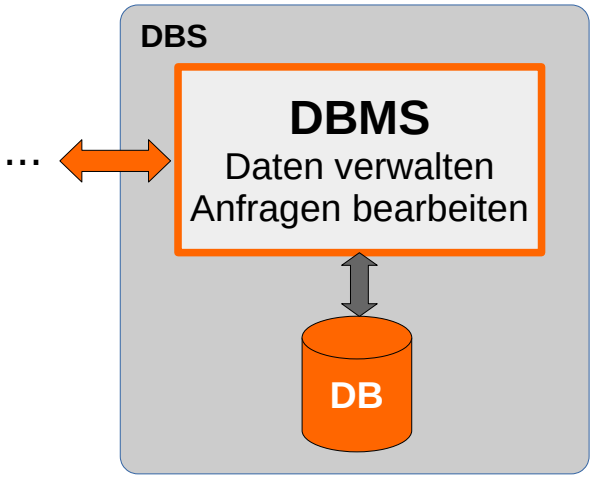
    → Wo kommen die Informationen her?

    → Wo gehen die Änderungen der Informationen hin?



Datenbanksysteme

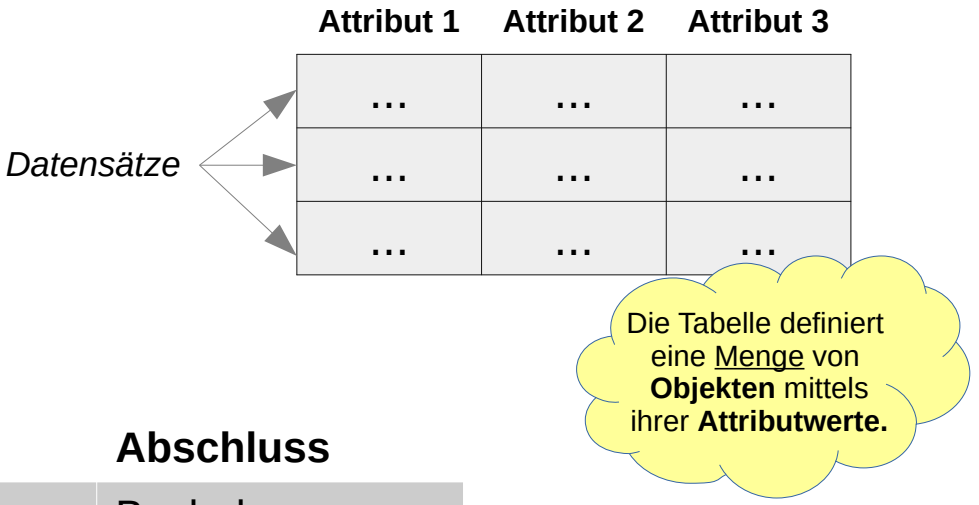
- **Datenbanksystem (DBS)**
  - System zur elektronischen Datenverwaltung
  - Zwei Teile:
    - **Datenbank** (**DB**)
      - also der zu verwaltende Datenbestand
    - **Datenbankmanagementsystem** (**DBMS**)
      - Hard- und Software zur Datenverwaltung
  - Aufgaben:
    - große Datenmengen ***speichern***
      - effizient, konsistent und dauerhaft
    - Daten für Nutzung in der benötigten Form ***bereitstellen***
      - ggf. auch (örtlich) verteilt (effizient, konsistent)



Datenbankmanagementsysteme

- **Wesentliche Funktionen eines DBMS**
  - **Speicherung** / **Änderung** / **Löschen** von Daten
  - Effiziente **Bereitstellung** von Daten über **Anfragesprachen**
    - Bei relationalen Datenbanken meist **SQL** (s.u.)
  - **Datensicherheit** / **Datenschutz** sichern
    - Daten dürfen nicht verloren gehen oder (ungewollt / unberechtigt) verändert werden (Datensicherheit)
    - Daten dürfen nicht „*in falsche Hände*“ geraten (Datenschutz)
      - was sind dann „*die richtigen Hände*“? → benötigt **Autorisationsmodell**
  - **Datenintegrität** sichern
    - Daten müssen „zueinander passen“ (Integritätsbedingungen erfüllen)
      - z.B. keine Einschreibung in einen Studiengang, der gar nicht existiert
  - **Mehrbenutzerbetrieb** und ggf. **örtliche Verteilung** ermöglichen
    - Es sollen mehrere Anfragen zeitgleich ausgeführt werden können, ohne dass es zu unerwarteten Wechselwirkungen kommt

- **Relationale Datenbanken**
  - basieren auf **Tabellen**
    - Spalten = **Attribute**
    - Zeile = **Datensätze** ⇔ Objekte



  - **Beispiel**: Tabelle „Einschreibung“

Matrikelnummer	Studiengang	Abschluss
123456	Informatik	Bachelor
121212	Mathematik	Master
133333	Mathematik	Bachelor

    - Jede Zeile beschreibt einen **Datensatz** (also ein Objekt)
    - Jeder Datensatz hat die angegebenen **Attributwerte**
      - Attributwerte können u.U. auch **NULL** sein (also explizit keinen Wert haben)
    - Die Namen der Spalten und die Typen der Attribute sind Teil der **Metadaten**.
      - Metadaten beschreiben Strukturen und Eigenschaften der Daten.
  - Die Tabellen repräsentieren **Relationen** zwischen den Attributen

Relationale Datenbanken

- **Schlüssel**
  - Ein **Schlüssel** einer Tabelle ist eine **Menge von Attributen**, mit der jeder Datensatz eindeutig identifiziert werden kann.
    - Häufig wird auch gefordert, dass diese Attributmenge minimal sein muss
  - **Beispiel**: Tabelle „Studierende“

Matrikelnummer	Vorname	Nachname	Emailadresse
123456	Peter	Müller	pm@gmail.de
121212	Karin	Müller	km@web.de
133333	Peter	Schmitt	ps@gmx.net

    - { Matrikelnummer } und { Emailadresse } sind jeweils (minimale) Schlüssel
    - { Matrikelnummer, Nachname } ist ein Schlüssel (aber nicht minimal)
    - Ungeeignet als Schlüssel ist z.B. { Nachname } *(warum?)*
    - **Frage**: Kann { Vorname, Nachname } hier ein Schlüssel sein?
      - Warum ja? Warum nein? Wenn ja: Ist das eine gute Idee?

Relationale Datenbanken

- **Primärschlüssel**
  - Einer der Schlüssel einer Tabelle muss als **Primärschlüssel** (*Primary Key*, ***PK***) gewählt werden.
    - Um einen Datensatz zu identifizieren, geben wir meist den PK an.
      - Man kann aber auch jeden anderen Schlüssel der Tabelle nutzen
    - Der PK sollte **gut zu handhaben** sein (kurz, möglichst nur ein Attribut)
    - Der PK sollte sich nur möglichst **selten ändern**

Wir unterstreichen die Attribut-Namen des Primärschlüssels

  - **Beispiel**: Tabelle „Studierende“

Matrikelnummer	**Vorname**	**Nachname**	**Emailadresse**
123456	Peter	Müller	pm@gmail.de
121212	Karin	Müller	km@web.de
133333	Peter	Schmitt	ps@gmx.net

    - { Matrikelnummer } ist hier ein günstiger Primärschlüssel
    - { Emailadresse } ist zwar Schlüssel, ändert sich aber möglicherweise
      - Auch: Eindeutigkeit der Schreibweise, z.B. Ist Groß-Kleinschreibung eindeutig bei Emailaddr.?

Relationale Datenbanken

- **Natürliche Schlüssel** (sprechende Schlüssel)
  - Ergeben sich aus den Attributen des modellierten Objekts
    - z.B. um ein Buch zu identifizieren genügt vielleicht { Autor, Buchtitel, Jahr }
  - Beispiel: Personenliste

Vorname	Nachname	Geburtstag	Geburtsort	...
Peter	Müller	28.02.2000	Berlin	
Karin	Schmitt	31.12.1999	München	
Peter	Müller	28.02.2000	Kaiserslautern	

    - Genügt { Vorname, Nachname } als Schlüssel?
    - Genügt { Vorname, Nachname, Geburtstag, Geburtsort } als Schlüssel?
    - Selbst wenn: Ist dieser Schlüssel **handlich**?

Relationale Datenbanken

- **Künstliche Schlüssel**
  - Zusätzliches Attribut (z.B. „id“) mit **künstlicher** „Durchnummerierung“ um eindeutige Schlüssel zu erhalten
    - z.B. aufsteigende ganze Zahlen (id = 1, 2, 3, …)
      - Kann aber jede (kollisionsfreie) Sequenz von beliebigen Werten sein
      - z.B.: **UUID** (siehe Wikipedia), z.B. „`69cdd0f2-10f3-4104-aa39-0bd449cf68a0`“
    - **unvermeidlich**, wenn es keine natürlichen Schlüssel gibt
    - manchmal **sinnvoll**, wenn vorhandene natürliche Schlüssel zu unhandlich
  - Beispiel:

id	Vorname	Nachname	Geburtstag	Geburtsort	...
1	Peter	Müller	28.02.2000	Berlin	
2	Karin	Schmitt	31.12.1999	München	
42	Peter	Müller	28.02.2000	Kaiserslautern	

- **Frage:** Ist die { Matrikelnummer } in der früher betrachteten Tabelle „Studierende“ prinzipiell ein *natürlicher* oder ein *künstlicher* Schlüssel? (Oder beides?)

Relationale Datenbanken

- **Beziehungen zwischen Tabellen**
  - Eine Tabelle kann auf eine andere Tabelle **verweisen**
    - Der Verweis besteht aus den Attributenwerten ihres (Primär-) Schlüssels.
  - **Beispiel**: Tabellen „Studierende“ und „Fachstudium“

Matrikel-nummer	**Vor-name**	**Nach-name**
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

Matrikel-nummer	**Studien-gang**	**Ab-schluss**
123456	Informatik	Bachelor
123456	Mathematik	Bachelor
133333	Mathematik	Master

    - (Primär-) Schlüssel in der Tabelle „Studierende“ ist { Matrikelnummer }
    - Die Tabelle „Fachstudium“ verweist mit dem Attribut { Matrikelnummer } auf die Tabelle „Studierende“
    - Diese Datensätze mit Matrikelnummer = „123456“ stehen also in Beziehung:
      - „Peter Müller“ studiert „Informatik“ mit dem Abschluss „Bachelor“
      - „Peter Müller“ studiert „Mathematik“ mit dem Abschluss „Bachelor“
    - *Achtung: Die verweisenden Attribute **müssen nicht** die selben Namen haben.*

- **Fremdschlüssel**
  - Verweist eine Tabelle auf den (Primär-) Schlüssel einer anderen, so nennt man diese Attribute **Fremdschlüssel** (Foreign Key)
    - Beispiel: Leiht eine Studentin ein Buch aus, so könnte in der Tabelle „Ausleihe“ durch den Fremdschlüssel { Matrikelnummer } auf den entsprechenden Studierenden-Datensatz verwiesen werden.
  - Eine Tabelle kann auch Fremdschlüssel auf sich selbst enthalten
    - **Beispiel**: In der Tabelle „Personal“ mit dem Primärschlüssel { Personalnummer } gibt es das Attribut { Vorgesetzter }, das auf die Personalnummer der selben Tabelle verweist.

Personalnummer	**Vorname**	**Nachname**	**Vorgesetzter**
101	Peter	Obermotz	NULL
121	Karin	Mittelmeyer	101
133	Peter	Kleinschmidt	101

      - Herr Obermotz ist also Vorgesetzter von Frau Mittelmeyer und Herrn Kleinschmidt
      - Herr Obermotz selbst hat keinen Vorgesetzten.

Relationale Datenbanken

- **Integritätsbedingungen (1)**
  - **Integritätsbedingungen** definieren zulässige Zustände der DB
  - **Fremdschlüssel** unterliegen einer strikten Integritätsbedingung:
    - Jeder referenzierte Datensatz muss existieren („**Referentielle Integrität**“)

      ⇒ Wird ein referenzierter Datensatz z.B. **entfernt**, so dürfen die darauf verweisenden (referenzierenden) Fremdschlüssel nicht bestehen bleiben.
    - Es gibt verschiedene Optionen zur **Lösung** des Problems:
      - Fremdschlüssel auf **NULL** setzen (falls das erlaubt ist)
      - Fremdschlüssel auf einen anderen (existierenden) Datensatz umsetzen
      - Den referenzierenden Datensatz (der den Fremdschlüssel enthält) ebenfalls löschen
    - Wird keine Lösung (s.u.) gefunden, so wird die auslösende Löschung des referenzierten Objekts **rückgängig** gemacht (Rollback, Transaktions-Abbruch, s.u.).
  - Es gibt noch diverse andere Integritätsbedingungen, z.B.
    - Datentypen und Bereichsbeschränkungen bei Attributen
    - Die Eindeutigkeit mancher Attribute oder Attributmengen (z.B. Primärschlüssel)
  - **Das DBMS garantiert die Einhaltung der Integritätsbedingungen**

Relationale Datenbanken

- **Integritätsbedingungen (2)**
  - Beispiel: „Studierende“ und „Fachstudium“ (s.o.)

Matrikel-nummer	Vor-name	Nach-name
~~123456~~	~~Peter~~	~~Müller~~
121212	Karin	Müller
133333	Peter	Schmitt

Matrikel-nummer	Studien-gang	Ab-schluss
123456	Informatik	Bachelor
123456	Mathematik	Bachelor
133333	Mathematik	Master

Löscht man den Studenten „123456“ (Peter Müller) aus der Tabelle „Studierende“, so können die beiden Datensätze in „Fachstudium“, die sich auf ihn beziehen, nicht weiter darauf verweisen.

1. Lösung: Die beiden betroffenen „Fachstudium“-Datensätze löschen.

2. Lösung: Die Fremdschlüssel auf **NULL** (also undefiniert) setzten.

3. Lösung: Den Fremdschlüssel auf einen anderen (erlaubten) Wert setzen

4. Lösung: Das Löschen wird verweigert.

  - *Frage: Was ist hier sinnvoll? (Welche Folgen hat es jeweils?)*

- **Integritätsbedingungen (3)**
  - Beispiel: „Personal“ (s.o.)

Personalnummer	**Vorname**	**Nachname**	**Vorgesetzter**
~~101~~	~~Peter~~	~~Obermotz~~	~~NULL~~
121	Karin	Mittelmeyer	101
133	Peter	Kleinschmidt	101

Wird der Mitarbeiter „101“ (Peter Obermotz) aus „Personal“ gelöscht, so können die beiden anderen Datensätze, die sich mit dem Fremdschlüssel „Vorgesetzter“ auf ihn beziehen, nicht weiter darauf verweisen.

1. Lösung: Die beiden betroffenen „Personal“-Datensätze löschen.

2. Lösung: Die Fremdschlüssel auf **NULL** (also undefiniert) setzen.

3. Lösung: Den Fremdschlüssel auf einen anderen (erlaubten) Wert setzen

4. Lösung: Das Löschen wird verweigert.

  - *Frage: Was ist hier sinnvoll? (Welche Folgen hat es jeweils?)*

Relationale Datenbanken

- **Modellierung von Tabellenstrukturen (1)**
  - Oft sind für gegebene Probleme mehrere Modellierungen möglich
  - Beispiel: „Studierende“ und „Fachstudium“ (s.o.)

Matrikel-nummer	Vor-name	Nach-name
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

Matrikel-nummer	Studien-gang	Ab-schluss
123456	Informatik	Bachelor
123456	Mathematik	Bachelor
133333	Mathematik	Master

    - Statt der beiden Tabellen wäre auch eine große vorstellbar:

Matrikelnummer	Vorname	Nachname	Studiengang	Abschluss
123456	Peter	Müller	Informatik	Bachelor
123456	Peter	Müller	Mathematik	Bachelor
121212	Karin	Müller		
133333	Peter	Schmitt	Mathematik	Master

    - **Nachteile**: Gefahr von Wiederholungen (Redundanz, Änderungen aufwändig, Inkonsistenzen), riesige Tabellen, viele leere Felder, ...

- **Modellierung von Tabellenstrukturen (2)**
  - Beispiel: „Studierende“ (s.o.)

Matrikel-nummer	**Vor-name**	**Nach-name**
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

  - Man könnte die Tabellen aber auch weiter aufteilen:

Matrikel-nummer	**Vor-name**
123456	Peter
121212	Karin
133333	Peter

Matrikel-nummer	**Nach-name**
123456	Müller
121212	Müller
133333	Schmitt

  - Nachteile: Sehr viele Tabellen, Einsammeln von Attributen ggf. sehr mühsam und wenig performant (viele Verknüpfungen und Leseoperationen nötig um alle Attribute zu erhalten)

- **Modellierung von Tabellenstrukturen (3)**
  - Es gilt hier zu einen Kompromiss zwischen gegenläufigen Anforderungen zu finden.
  - Es gibt im Bereich der Relationalen Datenbanken dazu eine eigene Theorie: Die **Normalformenlehre**
    - Das Erreichen der 5 (oder 6) Normalformen ist ein inkrementeller Prozess. Die Tabellenstruktur wird dabei immer abstrakter und redundanzärmer.
      - **1. Normalform**: Attribute aufteilen bis sie atomar sind
      - **2. bis 5. Normalform**: Tabellen immer weiter aufspalten um Wiederholungen zu vermeiden und Abhängigkeiten zwischen Attributen zu verringern
    - Die unteren Normalform-Stufen sind fast immer sinnvoll anzuwenden
    - Die höheren Normalform-Stufen zersplittern die Tabellenstruktur oft über das sinnvolle Maß hinaus.

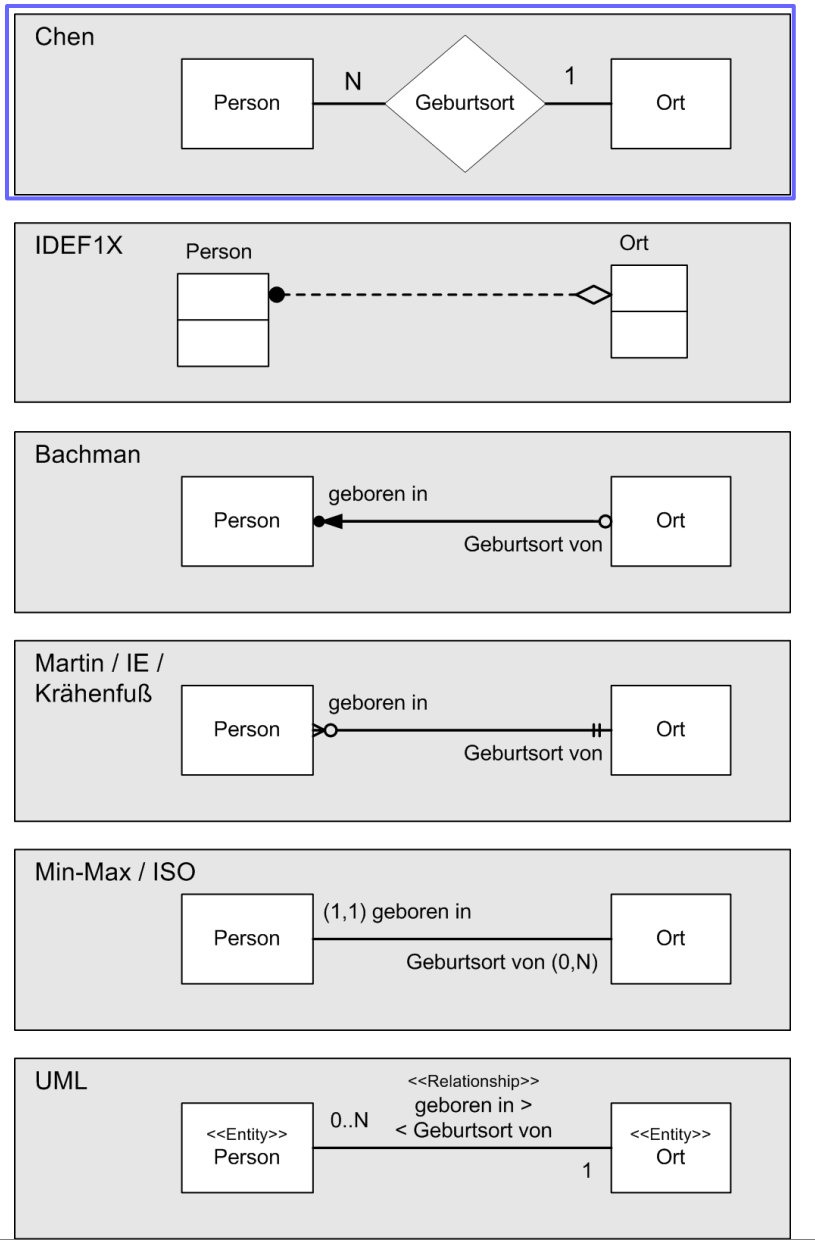
    - siehe Wikipedia: Artikel "Normalisierung (Datenbank)"
    - *Empfehlung: Schauen sie sich zumindest einmal die Beispiele im Wikipedia-Artikel an, um einen Eindruck zu erhalten.*

Entity-Relationship-Modell

- **Das Entity-Relationship-Modell (ERM)**
  - Grafische Notation zur Datenmodellierung (Chen, 1976)
  - De-facto-Standard bei der Modellierung von relationalen Modellen
  - **Gegenstände**:
    - **Entity** (Entität) = (konkreter) Gegenstand der realen Welt
      - z.B. Student „Peter Müller“, Hörsaal „42-105“, ...
    - **Relationship** (Beziehung) = Beziehung zwischen 2 oder mehr Entitäten
      - z.B. „Student ABC besucht Vorlesung XYZ in 42-105“,
    - **Eigenschaft** = Eigenschaft einer konkreten Entität (z.B. Vorname)
  - Darauf Aufbauende **Typen** (Klassen / Mengen)
    - **Entitätstyp** (z.B. Studierende, Hörsäle, Vorlesungen, …)
    - **Beziehungstyp** = Beziehung zwischen den (Elementen der) Entitätstypen
      - z.B. „besucht Vorlesung“
    - **Attribut** = Typisierung der Eigenschaft eines Entitätstyps (z.B. Vorname)

Entity-Relationship-Modell

- **ER-Diagramm (ERD)**
  - Verschiedene Grafische Darstellungen möglich
  - Ziele:
    - Entitätstypen und Beziehungstypen übersichtlich darzustellen
    - Rollennamen für Beziehungen zu vergeben
    - Kardinalitäten zu vergeben
  - Wir benutzen in der Vorlesung eine spezielle *Chen*-Notation
    - Mit modifizierten Kardinalitäten
    - Alternativen: Siehe Wikipedia-Artikel zum ER-Modell



Entity-Relationship-Modell

- **Beispiel für ER-Diagramm:** Personen und Geburtsorte

  - Das Diagramm beschreibt die Beziehung „Geburtsort“ zwischen den **Entitäts-Typen** (Entitäten-Mengen) „Person“ und „Ort“
    - Idee: *„Jede Person wird **verknüpft** mit genau einem Ort (aus der Menge aller Orte). Diese Verknüpfung beschreibt den Geburtsort der Person.“*
  - Die Zahlenangaben sind **Kardinalitäten**
    - „1“ = „Es gibt zu jeder Person genau 1 Ort, an dem sie geboren ist.“
    - „0..*“ oder „0..N“ = „Zu jedem Ort gibt es eine beliebige Anzahl von Personen, die dort geboren sind (einschließlich 0).“
    - **Beachten Sie die Anordnung:** „1“ steht bei „Ort“, „0..*“ bei „Person“.
      - Idee: Aus Sicht der Person gibt es „1“ Orte,
        aus Sicht des Ortes „0..*“ (*beliebig viele*) Personen

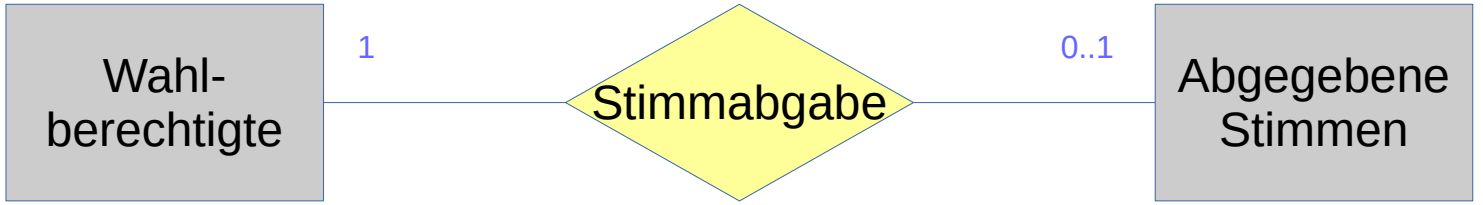
Entity-Relationship-Modell

- **Weitere Kardinalitäten**

  - Es sind auch andere Kardinalitäten möglich
    - „1..*“ oder „1..N“ = „Zu jedem Ort gibt es eine beliebige Anzahl von Personen, die dort geboren sind, mindestens jedoch einen.“
      - z.B. sinnvoll, wenn ich nur Orte in der Menge haben will, die auch als Geburtsort verwendet werden.
    - „0..1“ = „Es gibt zu jeder Person höchstens 1 Ort, an dem sie geboren ist.“
      - z.B. hier sinnvoll, wenn ich nicht zu jeder Person den Geburtsort kenne
  - Man bezeichnet die Kardinalitäten außerhalb der Diagramme **vereinfacht** (**abstrahiert**) als „**1:1**“, „**1:n**“ oder „**n:m**“
    - Die „1“ bedeutet in „1:1“ bzw. „1:n“ ein „*höchstens 1*“, schließt also 0 mit ein
    - „n“ und „m“ sind (wie oben „N“ und „*“) keine konkreten Werte (= „*beliebig*“)

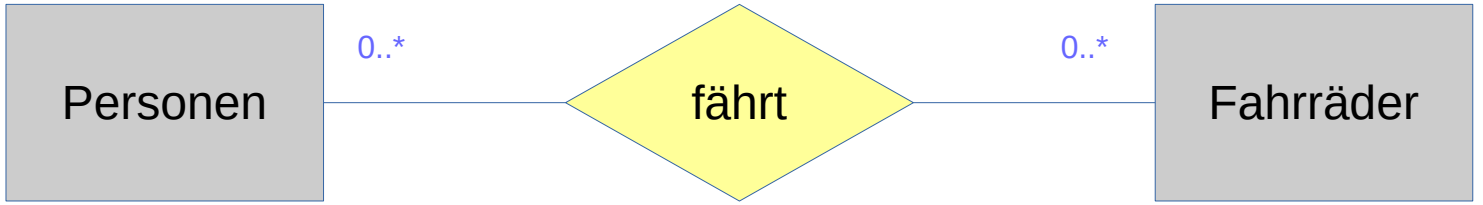
Entity-Relationship-Modell

- **Abstrahierte Kardinalitäten** (charakterisierend)
  - **1:1** („One man, one vote“ – man muss aber nicht wählen)



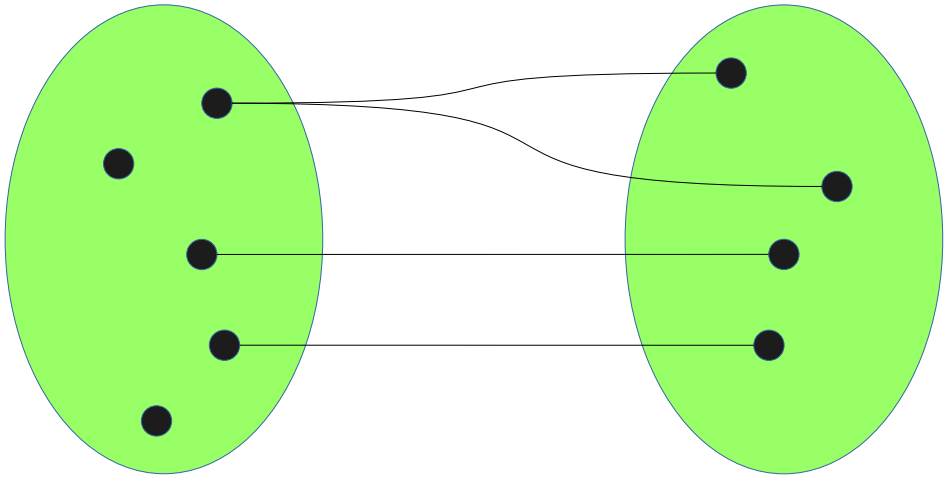
  - **1:n** (Jedes KFZ hat einen Halter, aber nicht jeder hat ein KFZ)

  - **n:m** (Beliebig viele Fahrer pro Fahrrad und umgekehrt)



Entity-Relationship-Modell

- **Es gibt auch Diagramme für Entities**
  - Diese sind nützlich um Kardinalitäten klar zu machen
- **Beispiel:**
  - **1:n**



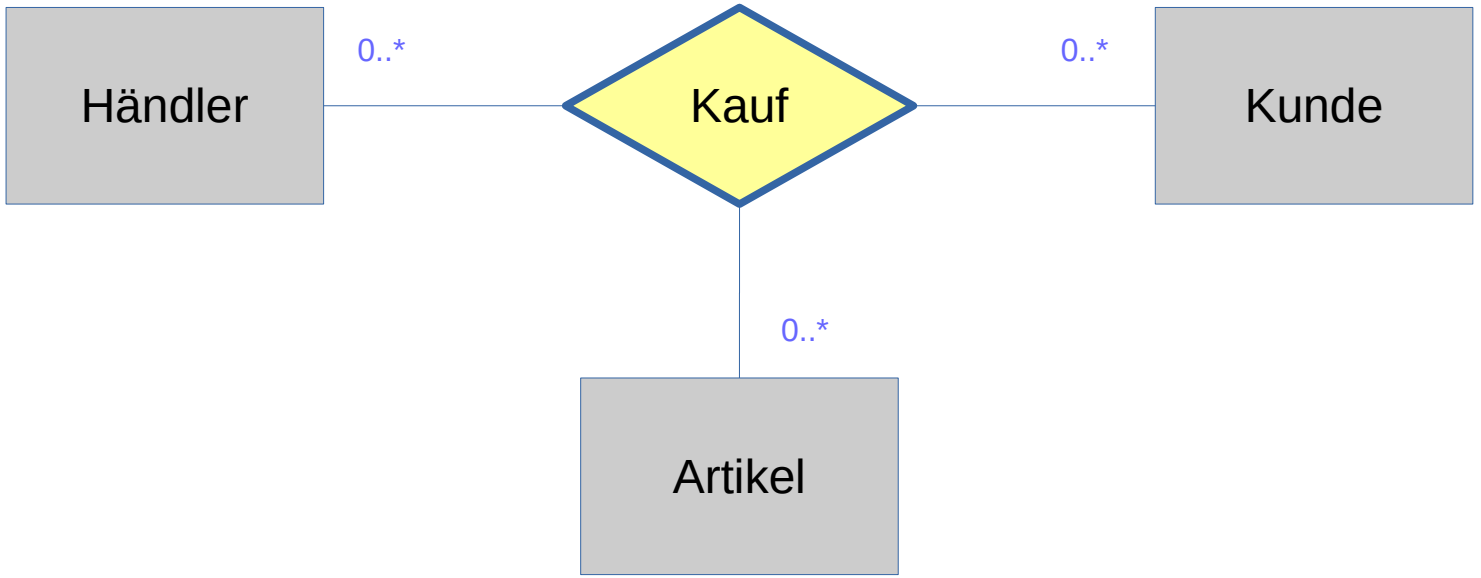
Jeder Punkt repräsentiert beispielhaft ein **Entity**, also ein **Objekt**

- Zwei Entity-Mengen (Person, KFZ),
- jeweils mehrere Entities (Punkte) und ihre Beziehungen (Linien)

- Hier: *Jedes KFZ hat einen Halter, aber nicht jeder hat ein KFZ*

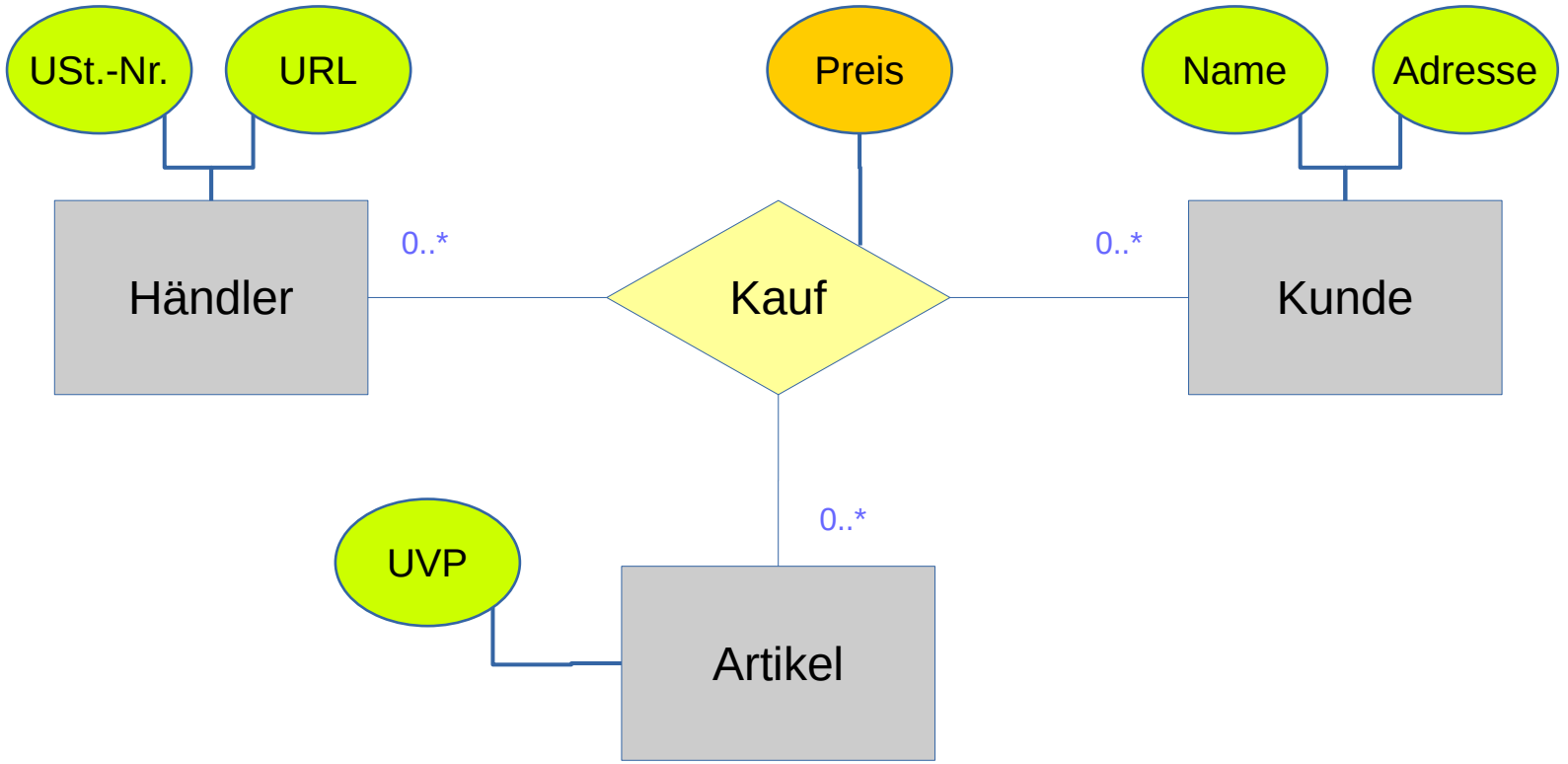
Entity-Relationship-Modell

- **Es gibt neben binären auch n-äre Relationen**
  - Beispiel: n=3



Entity-Relationship-Modell

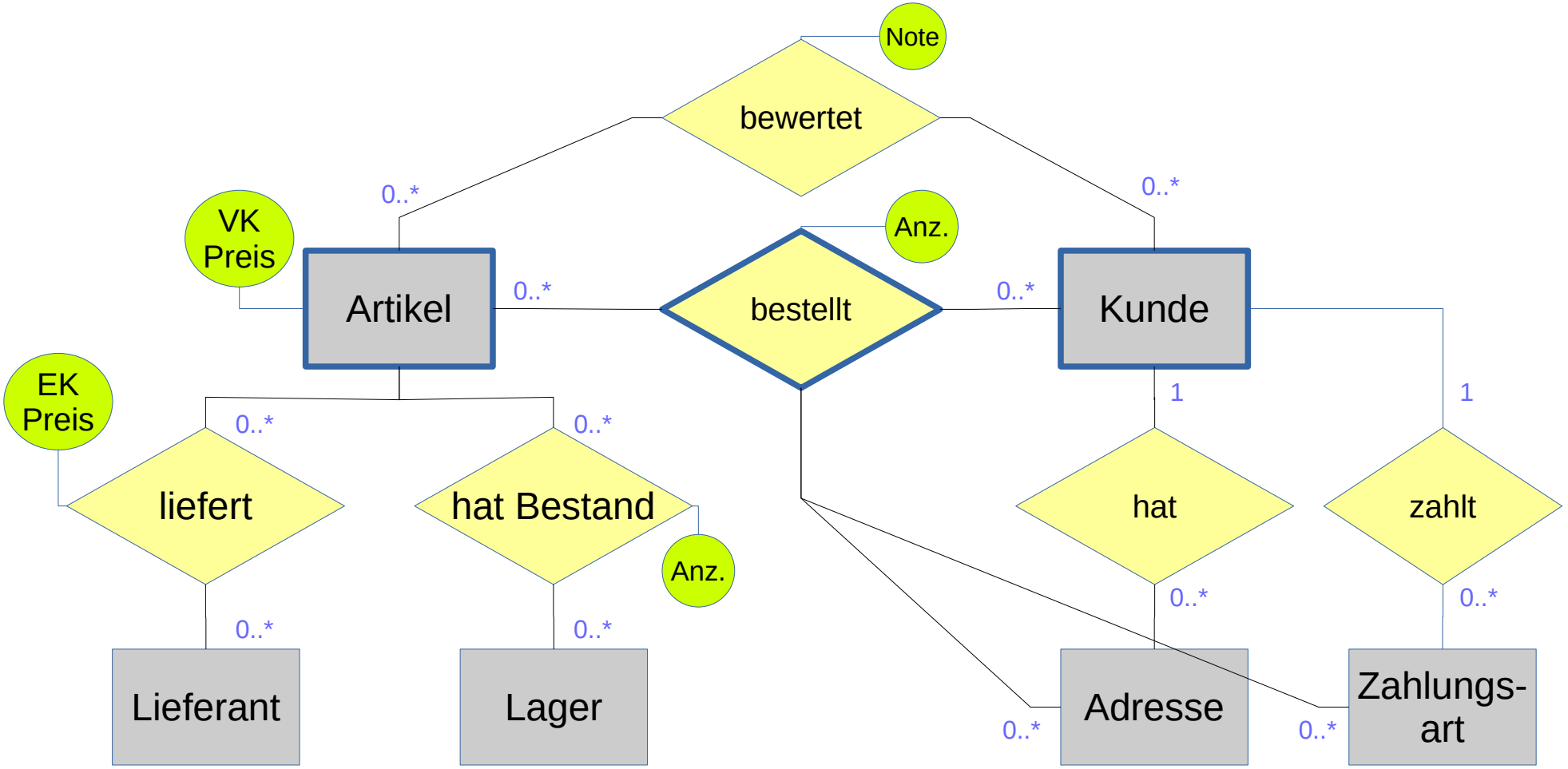
- **Man kann auch Attribute zuordnen**
  - zu Entitätstypen oder auch zu Relationen



  - Das wird aber schnell unübersichtlich

Entity-Relationship-Modell

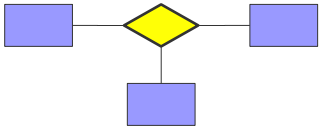
- **Das ER-Modell ist nützlich zur Datenmodellierung**



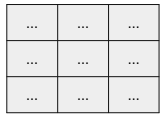
  - Es liefert uns Übersicht bei komplexen Datenstrukturen

Vom ER-Modell zur Relationalen Datenbank

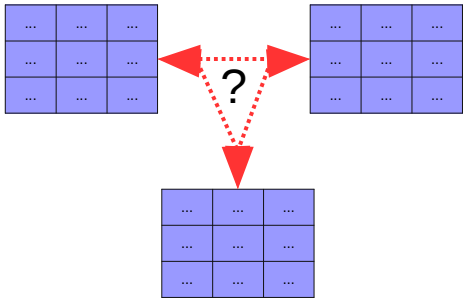
- **Ausgangspunkt: ER-Modell**
  - Entity-Typen, Beziehungen, Kardinalitäten



- **Ziel: Relationale Datenbanken** (arbeiten mit **Tabellen**)
  - **Zeilen** = Datensätze
  - **Spalten** = Attribute
  - Schlüssel bzw. Primärschlüssel sind Attribut-Mengen



- **Frage: Wie modelliert man das?**
  - Entity-Typen → Tabellen
    - Zeilen = Entities
    - Spalten = Attribute
  - Beziehungen → **?**
    - Naheliegend: Verweise auf Datensätze durch **Fremdschlüssel**



Vom ER-Modell zur Relationalen Datenbank

- **Realisierung von 1:n-Relationen**

  - Zwei Tabellen zur Realisierung der beiden **Entity-Typen**

Person

Person-ID	V_name	N_name
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

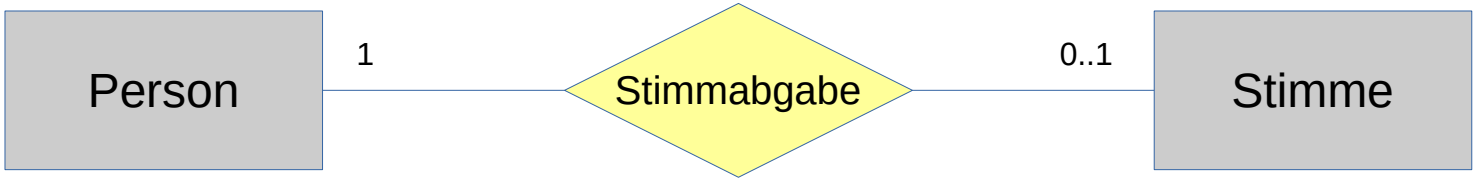
KFZ

Halter	FG-Nr	Hersteller	Baujahr
123456	2341234	Audi	2010
123456	4432333	BMW	1985
133333	8877783	Opel	2014

    - Da es zu einem KFZ nur maximal eine Person in der „ist Halter-Beziehung gibt, können wir diesen Verweis direkt im KFZ-Datensatz eintragen
    - Dazu fügen wir das Attribut „Halter“ als **Fremdschlüssel** zu Tabelle „Person“ (hat PK „Person-ID“) in die Tabelle KFZ ein.
  - **Beziehung** wird als Attribut (d.h. ohne eigene Tabelle) realisiert

Vom ER-Modell zur Relationalen Datenbank

- **Realisierung von 1:1-Relationen**



  - Zwei Tabellen zur Realisierung der beiden **Entity-Typen**

Person

Person-ID	V_name	N_name
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

Stimme

Wähler	Erststimme	Zweitstimme
123456	A-Partei	A-Partei
133333	B-Partei	C-Partei

  - 1:1 ist ein Spezialfall von 1:n, daher die prinzipiell gleiche Lösung
    - Bei dieser Kardinalität (1 zu 0..1), kann der Fremdschlüssel zu Person gleichzeitig Primärschlüssel in Stimme sein.
    - Bei strikten 1:1-Beziehungen kann man beide Tabellen sogar vereinen
      - Frage: Warum? Warum nicht im obigen Beispiel?

Vom ER-Modell zur Relationalen Datenbank

- **Realisierung von n:m-Relationen**

  - Zwei Tabellen zur Realisierung der beiden **Entity-Typen**

Person

Person-ID	V_name	N_name
123456	Peter	Müller
121212	Karin	Müller
133333	Peter	Schmitt

KFZ

FG-Nr	Hersteller	Baujahr
2341234	Audi	2010
4432333	BMW	1985
8877783	Opel	2014

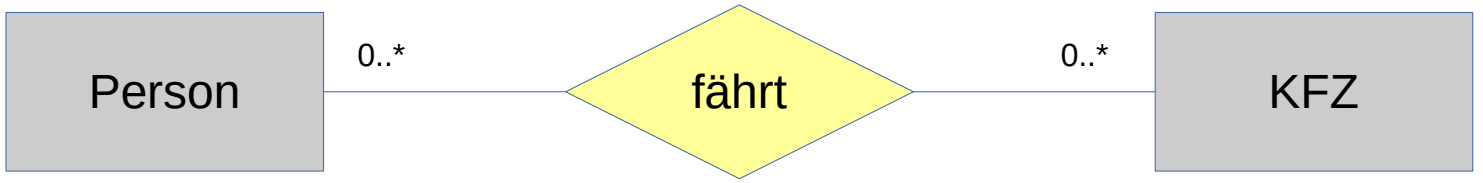
  - **Beziehung** realisiert als **dritte Tabelle** mit zwei Fremdschlüsseln

Fahrt

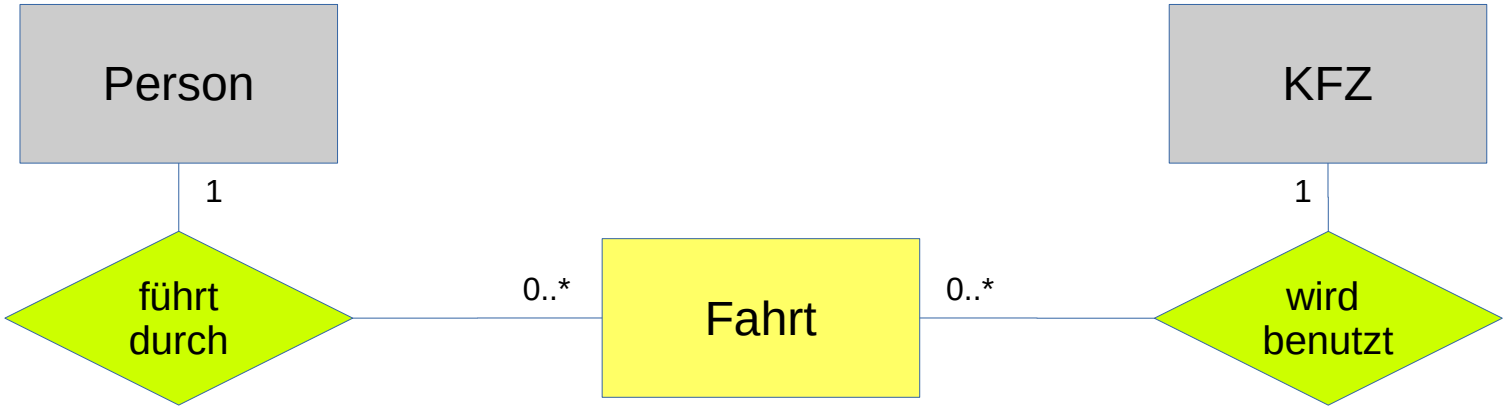
Fahrer	Fahrzeug
123456	2341234
123456	8877783
133333	2341234

Vom ER-Modell zur Relationalen Datenbank

- **Realisierung von n:m-Relationen**



  - Letztlich werden n:m-Relationen also umgewandelt in
    - Ein Hilfs-Entity-Typ, der die Beziehungsobjekte darstellt
    - Zwei **1:n Relationen**
  - Diese Transformation kann schon auf Ebene des ER-Modells erfolgen (konzeptionelles Modell → **Implementierungsmodell**)

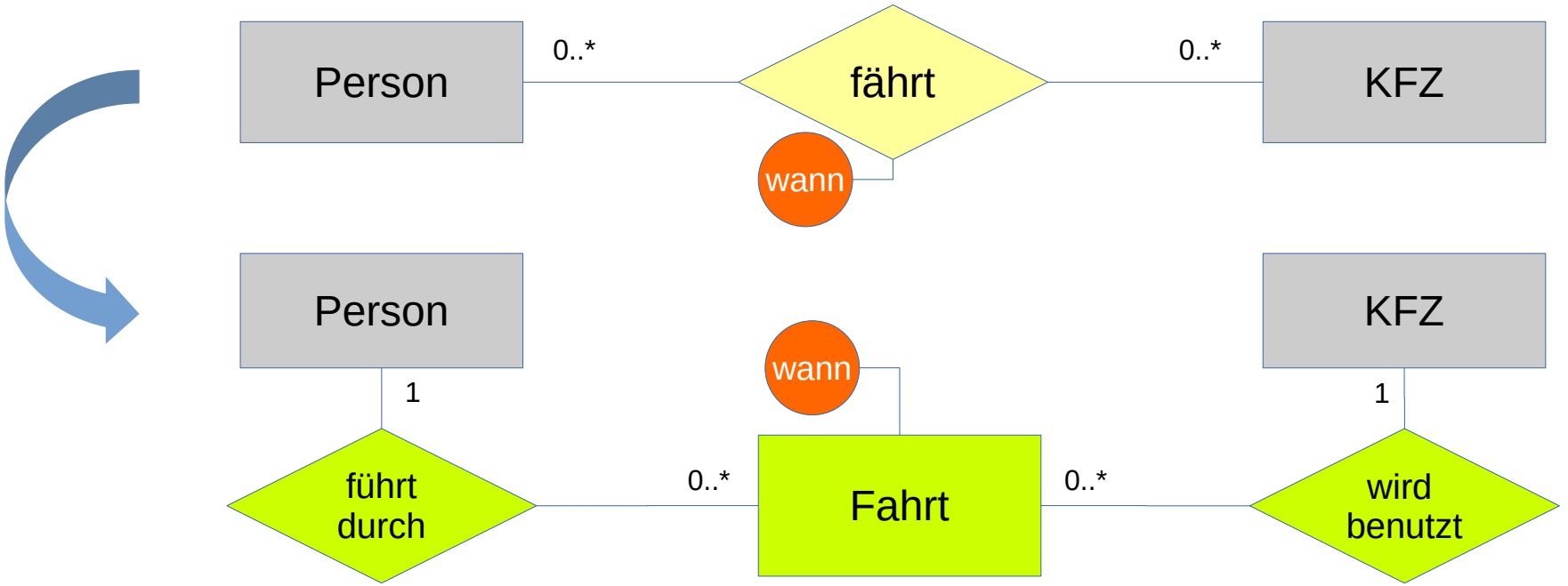


    - Wir haben hier nur noch Kardinalitätskombination 1 zu 0..*

Vom ER-Modell zur Relationalen Datenbank

- **Modellierung von Beziehungs-Attributen**
  - **1:1** bzw. **1:n**
    - Beziehungs-Attribute werden der Tabelle zugeordnet, die den **Fremdschlüssel** erhält *(der Fremdschlüssel repräsentiert ja die Beziehung)*
  - **n:m**
    - Beziehungs-Attribute werden der hinzugefügten **Beziehungs-Tabelle** (bzw. im ER-Modell den Hilfs-Entity-Typen) zugeordnet

z.B. ein „seit“ Attribut der 1:n Relation „*ist Halter*“ (s.o.) käme in die KFZ-Tabelle



Vom ER-Modell zur Relationalen Datenbank

- **Ziel: Von der Datenmodellierung zur Implementierung**
  - Wir können bereits konzeptionelle Datenmodelle entwerfen
    - Entity-Mengen, Beziehungen, Kardinalitäten
  - Wir können diese in eine relationale Datenbankstruktur abbilden
    - Tabellen (Attribute, Datensätze), Primärschlüssel, Fremdschlüssel

- **Wie bildet man das auf SQL ab?**
  - Wie definiert man relationale Modelle in SQL?
    - Tabellen, Attribute, Attributtypen, Primärschlüssel, Fremdschlüssel
  - Wie ändert / ergänzt / löscht man Daten in diesen Modellen
    - u.a. Transaktionsmodell
  - Wie fragt man Daten ab?
    - SQL-Queries, Datensatz-Selektion, Joins

SQL

- **SQL = Structured Query Language**
  - Standard-Anfrage-Sprache für relationale Datenbanken
  - Anfragen und Daten werden als Text (Text-String) übergeben
    - Beispiel:

      SELECT Nachname, Vorname **FROM** Student **WHERE** MatrNr = 123456 **;**

  - Anfragen können ...
    - Daten abfragen
      - Datensätze selektieren, Attribute auswählen, Daten verknüpfen
    - Daten ändern
      - einfügen, löschen, ändern
    - Datenstrukturen ändern
      - Tabellenstruktur, Attributtypen, Restriktionen

SQL

- **Es gibt viele relationale DBMS** (Software-Lösungen)
  - *Open Source*, z.B.
    - **MySQL** (siehe https://dev.mysql.com/doc/refman/8.0/en/)
    - **MariaDB** (siehe https://mariadb.org/) ← Ein „Fork“ vom MySQL
    - **PostgreSQL** (siehe https://www.postgresql.org/docs/)
    - **SQLight** (siehe https://www.sqlite.org/docs.html)
      ← *keine separate DBMS-Instanz, sondern nur Client-Bibliothek*
  - *Kommerziell* (proprietär, meist Closed Source), z.B.
    - **Oracle RDBMS**
    - **DB2** (IBM)
    - **Microsoft SQL Server**
- **Wir betrachten hier MySQL als Beispiel**
  - Typisch: **LAMP-Server** = Linux + Apache + MySQL + PHP
  - Beliebte Lösung (kostengünstig, ressourcensparsam, relativ einfach)
    - Andere DBMS als MySQL können aber oft mehr

SQL

- **DBMS bieten ihren Dienst anderen Programmen**
  - Zugriff in Form von SQL-Queries über **Netzwerk-Schnittstelle**
    - Via TCP → zugreifbar über das Internet
    - Kann aber auch auf Server-interne Zugriffe beschränkt werden (Isolation)
  - Der Zugreifer muss sich beim DBMS **authentifizieren**
    - Bei MySQL: Benutzername + Passwort
    - Auch Programme (z.B. PHP-Scripte) müssen das tun
  - Den einzelnen Nutzern können unterschiedliche Rechte gewährt werden (**Autorisierung**)
    - Zugriff auf bestimmte Datenbanken eines DBMS
    - Zugriff auf bestimmte Tabellen in einer Datenbank
    - Zugriff auf bestimmte Attribute einer Tabelle
    - Zugriff auf bestimmte Datensätze

SQL

- **Zum Zugriff gibt es auf Client-Seite Hilfsmittel**
  - SQL-Client-Tools (Grafisch / Webfrontend)
    - z.B. Web-Admin-Oberfläche PHPmyAdmin
  - SQL-Client-Bibliotheken (Connector)
    - z.B. für Zugriffe aus PHP heraus: http://php.net/manual/de/set.mysqlinfo.php
  - SQL-Client-Tools (Kommandozeile)
    - https://dev.mysql.com/doc/refman/8.0/en/programs-client.html
    - z.B. das **Client-Kommandozeilen-Programm** „**mysql**“
      - damit können SQL-Queries von Hand oder aus Dateien eingegeben werden

SQL

- **Plattform für die Übungen**
  - Übungsserver sind LAMP-Server (scilab-*nnnn*.informatik.uni-kl.de)
    - Jede Übungsgruppe erhält einen eigenen Server
  - MySQL ist nur Server-intern zugreifbar
    - Der Client muss also auf dem Server betrieben werden
  - Zugriff auf den Server per SSH
    - Zugangsdaten für SSH und Datenbank werden ausgegeben
    - richten Sie sich aber gleich einen Public-Key-Authentifizierung ein
  - Wir gehen im Folgenden von einer bestehenden SSH-Sitzung auf den LAMP-Server aus

SQL / MySQL

- **Kommandozeilen-Tool „mysql“ - erste Schritte**

```
[~] mysql
ERROR 1045 (28000): Access denied for user 'lamp'@'localhost'
(using password: NO)
```

  - Nach kurzem Lesen von „`man mysql`“ …

```
[~] mysql -p
Enter password: _
```

  - Der Login in die Datenbank funktioniert nun

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.

mysql> _
```

  - Man kann bei Bedarf Server (`-h`, default ist „localhost“) und Username (`-u`, default ist der Login-Acccount-Name) angeben

```
[~] mysql -h localhost -u lamp -p
Enter password: *******
```

SQL / MySQL

- **Konfigurationsdatei ~/.my.cnf**
  - Hier kann man z.B. das DB-Passwort ablegen

```
[client]
   user = lamp
   password = **********
```

  - Und vielleicht auch den Eingabe-Prompt erweitern

```
[mysql]
   prompt = (\\u@\\h) [\\d]>\\_
```

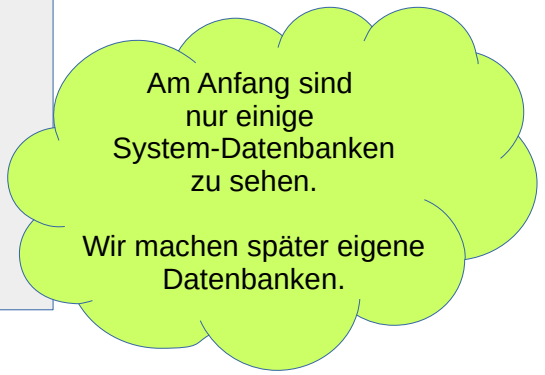
  - Danach funktioniert der DB-Login ohne weitere Angaben

```
[~] mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.

(lamp@localhost) [(none)]> _
```

- **SQL-Queries**
  - Wir können nun **Anfragen** (**Queries**) stellen
    - Anfragen können über mehrere Zeilen gehen
    - Groß-/Kleinschreibung bei **Schlüsselworten** beliebig (**Konvention**: groß), bei **Namen** (Tabellen, Attribute) aber genau wie bei ihrer Definition.
    - Anfragen an den Server enden mit einem **Semikolon**

```
[(none)]> SHOW DATABASES;
+----------------------+
| Database             |
+----------------------+
| information_schema   |
| mysql                |
| performance_schema   |
+----------------------+
```



    - Die Ausgabe zeigt eine **Tabelle**
      - Nur ein Attribut („Database“) als Spalte
      - Drei Datensätze als Zeilen
    - Wir sehen hier also **die Namen von** drei **Datenbanken**
      - Diese speziellen Tabellen enthalten Verwaltungsinformationen (*Metadaten*) des DBMS

- **SQL-Queries**
  - Client-bezogene **Anfragen** (**Queries**)
    - Anfragen, die der Client selbst beantwortet, brauchen kein Semikolon

```
[(none)]> HELP SHOW DATABASES
Name: 'SHOW DATABASES'
Description:
Syntax:
SHOW {DATABASES | SCHEMAS}
    [LIKE 'pattern' | WHERE expr]

SHOW DATABASES lists the databases on the
MySQL server host. [...]
```

    - Die Client-Abfrage „STATUS“ liefert Informationen zu Client und Verbindung

```
[(none)]> STATUS
mysql  Ver 14.14 Distrib 5.5.43

Current database:
Current user:          lamp@localhost
Server characterset:    utf8
Uptime:                    1 day 6 hours 12 min 53 sec
```

- **SQL-Queries: Tabellen auflisten**
  - Wir können nun eine der Datenbanken auswählen
    - Wir schauen uns als Beispiel mal die System-DB „mysql“ an

```
[(none)]> USE mysql
Database changed
[mysql]>
```

Die System-Datenbank “mysql“ ist nur ein erstes Beispiel

Analog zu „**cd** xyz“ bei Verzeichnissen in Dateisystemen

    - Ab jetzt ist in dieser Sitzung „mysql“ die **Default-DB**

```
[mysql]> SHOW TABLES;
+---------------------------+
| Tables_in_mysql           |
+---------------------------+
| columns_priv              |
| db                        |
| event                     |
| …                         |
| user                      |
+---------------------------+
```

    - Man kann die Datenbank auch direkt beim `mysql`-Aufruf übergeben
      - **mysql** *datenbankname*

- **SQL-Queries: Tabellenstruktur anzeigen**
  - Mit „DESCRIBE“ erhält man Informationen zur **Tabellenstruktur**

```
[mysql]> DESCRIBE user;
+------------------+------------------+------+-----+---------+-------+
| Field            | Type             | Null | Key | Default | Extra |
+------------------+------------------+------+-----+---------+-------+
| Host             | char(60)         | NO   | PRI |         |       |
| User             | char(16)         | NO   | PRI |         |       |
| Password         | char(41)         | NO   |     |         |       |
| Select_priv      | enum('N','Y')    | NO   |     | N       |       |
  …                  …                  …      …     …         …
+------------------+------------------+------+-----+---------+-------+
42 rows in set (0.01 sec)
```

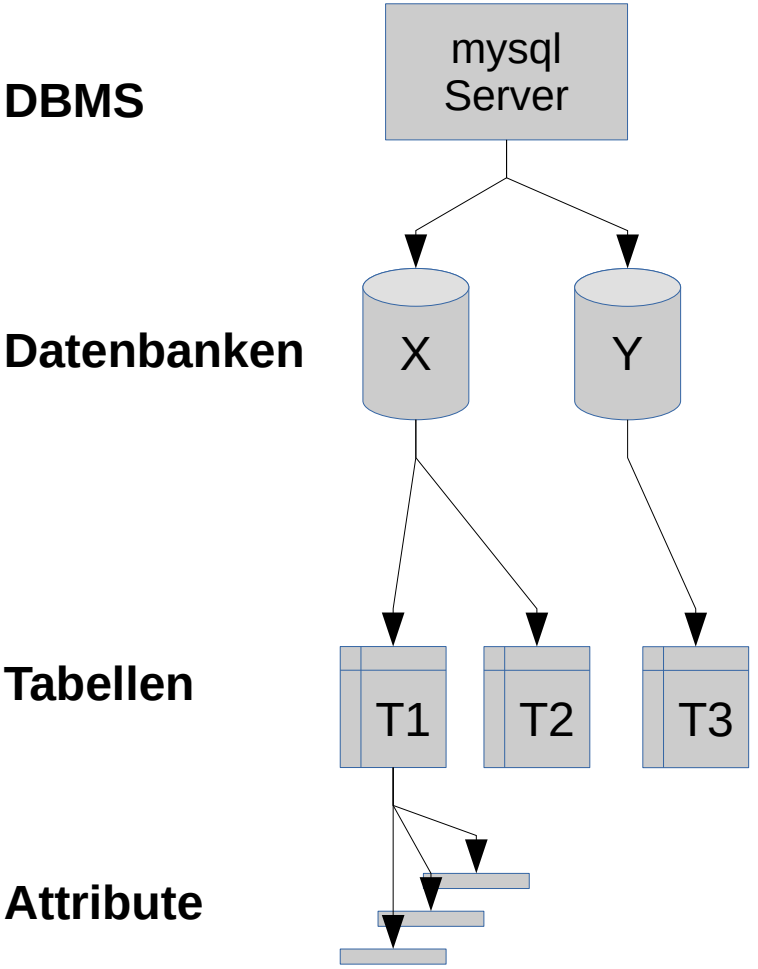
    - Es gibt also 42 Spalten in mysql.user, z.B.
      - Attribut „Host“ vom Typ „char(60)“
      - Attribut „User“ vom Typ „char(16)“
      - Attribut „Password“ vom Typ „char(41)“
      - ...

Wir schauen uns die Systemtabelle „*user*“ hier als mal Beispiel an, weil sie ja schon da ist.

SQL / MySQL

- **Überblick: Schrittweise Untersuchung des Schemas**
  - SHOW DATABASES;
    - zeigt die Namen aller **Datenbanken** an
  - USE database;
    - wechelt aktuelle **Datenbank** auf die angegebe
      - Man kann Tabellen qualifiziert angeben, z.B. „mysql.user“
  - SHOW TABLES;
    - zeigt die Namen der **Tabellen** in der aktuellen Datenbank
  - DESCRIBE tablename;
    - Zeigt die **Attribute** einer Tabelle an (z.B. „`DESCRIBE user;`“)

Hierarchie



SQL / MySQL: SELECT

- **SQL-Queries: Daten anfordern**
  - Mit „**SELECT**“ erhält man Zugriff auf die **Daten** der DB
    - z.B. die Daten der Spalten (→ Attribute) user und host aus der Tabelle user

```
> SELECT user,host FROM user;
+------------------+-----------+
| user             | host      |
+------------------+-----------+
| debian-sys-maint | localhost |
| lamp             | localhost |
| mysql.session    | localhost |
| mysql.sys        | localhost |
| root             | localhost |
+------------------+-----------+
```

    - Mit WHERE kann man die Zeilen (→ Datensätze) selektieren

```
> SELECT user,host FROM user WHERE user = 'lamp';
+------------------+-----------+
| user             | host      |
+------------------+-----------+
| lamp             | localhost |
+------------------+-----------+
```

SQL / MySQL: SELECT

- **SQL-Queries: Daten anfordern**
  - Spaltenwerte der Ausgabe können Duplikate enthalten
    - z.B. die Daten der Spalten user aus der Tabelle user

```
> SELECT host FROM user;
+-----------+
| host      |
+-----------+
| localhost |
| localhost |
| localhost |
| localhost |
| localhost |
+-----------+
```

    - Mit DISTINCT kann man die Duplikate entfernen

```
> SELECT DISTINCT host FROM user;
+-----------+
| host      |
+-----------+
| localhost |
+-----------+
```

SQL / MySQL: SELECT

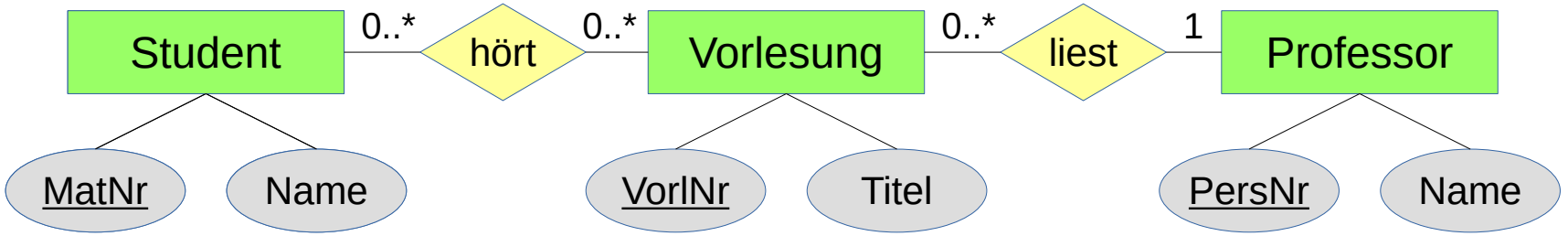
- **SQL-Queries: Daten anfordern**
  - Mit „**SELECT ***“ bekommt man alle Attribute

```
> SELECT * FROM user;
+-------------+-------+-------------------+-------------+-------------- …
| Host        | User  | Password          | Select_priv | Insert_priv  …
+-------------+-------+-------------------+-------------+-------------- …
| localhost   | root  | *E37FC...3DE8060 | Y           | Y             …
  …             …       …                   …             …
+-------------+-------+-------------------+-------------+-------------- …
```

    - Problem: Die Tabelle ist sehr breit (42 Attribute)
    - Einen Datensatz als Zeile (also horizontal) darzustellen ist hier nicht sinnvoll
  - Vertikale Darstellung: „\G“ statt Semikolon

```
> SELECT * FROM user \G
*************************** 1. row ***************************
                  Host: localhost
                  User: root
              Password: *E37FCBC917BA6E5F0EDBC5B6441E532293DE8060
           Select_priv: Y
                     …  …
```

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



  - Beispiel-Daten

Student

MatNr	Name
26120	Fichte
25403	Jonas
27103	Fauler

hört

MatrNr	VorlNr
25403	5001
26120	5001
26120	5045

Vorlesung

VorlNr	Titel	PersNr
5001	ET	15
5022	IT	12
5045	DB	12

Professor

PersNr	Name
12	Wirth
15	Tesla
20	Urlauber

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

Zur Übung: Zeichnen Sie das Implementierungsmodell (ER-Diagramm ohne n:m)

Mehr dazu auf dem ersten Übungsblatt.

HOWTO: SQL-*Schema* und *Daten* zum Anwendungsbeispiel

Um mit dem o.g. Anwendungsbeispiel praktisch arbeiten zu können, stellen wir Ihnen das SQL-Schema und die Daten zur Verfügung.

Auf den **Übungsservern** ist das Schema mit den Daten bereits in der Datenbank `wikipedia_sql_example` installiert.

Falls Sie das Schema und die Daten auf **eigenen Systemen** installieren wollen, können Sie von folgender URL zwei SQL-Scripte herunterladen:

```
https://sci.cs.uni-kl.de/lv/w2t2/download/wikipedia_sql_example
```

Damit können Sie das Schema und die Daten anlegen. Rufen Sie dazu folgende Shell-Kommandos in dem Verzeichnis mit den Dateien auf.

```
mysql < create-schema.sql
mysql < create-data.sql
```

- Die Funktionsweise der SQL-Scripte werden wir später erklären.

SQL / MySQL: SELECT

Zur Übung:
Jeweils erst überlegen,
dann ausprobieren!

- **Anwendungsbeispiele**
  - SELECT * FROM Student;
  - SELECT VorlNr, Titel FROM Vorlesung;
  - SELECT DISTINCT MatrNr FROM hört;
  - SELECT VorlNr, Titel FROM Vorlesung WHERE Titel = 'ET';
  - SELECT VorlNr AS Vorlesungsnummer, Titel FROM Vorlesung;
    - Nur die beiden Spalten anzeigen, dabei die erste Spalte **umbenennen**
  - SELECT Name FROM Student WHERE Name LIKE 'F%';
    - Nur Namen die mit „F“ beginnen
  - SELECT Name FROM Student ORDER BY Name;
    - Alphabetisch sortieren (mit „ORDER BY Name DESC“ umgekehrt)
  - SELECT Name FROM Student LIMIT 5 OFFSET 10;
    - Höchstens 5 Ergebnisse ausgeben, überspringe die ersten 10 vorher
      - Anwendung z.B. Paginator: 3. Seite wenn pro Seite nur 5 angezeigt werden

SQL / MySQL: SELECT

- **Berechnungen und Funktionen in Queries**
  - Im SELECT-Statement können auch berechnete Ergebnisse ausgegeben werden

```
> SELECT 1+2*3;
+-------+
| 1+2*3 |
+-------+
|     7 |
+-------+
```

  - Hier können auch Funktionsergebnisse abgefragt werden
    - z.B. MIN(), MAX(), SUM(), AVG(), COUNT()

```
> SELECT MIN(PersNr), MAX(PersNr), SUM(PersNr), AVG(PersNr), COUNT(*)
  FROM   Vorlesung;
+-------------+-------------+-------------+-------------+---------------+
| MIN(PersNr) | MAX(PersNr) | SUM(PersNr) | AVG(PersNr) | COUNT(*)      |
+-------------+-------------+-------------+-------------+---------------+
|          12 |          15 |          39 |     13.0000 |             3 |
+-------------+-------------+-------------+-------------+---------------+
```

    - COUNT(Attributname) zählt nur Nicht-NULL-Werte, COUNT(*) zählt alle Zeilen
    - COUNT(DISTINCT Attributname) zählt unterschiedliche Nicht-NULL-Werte

SQL / MySQL: SELECT

- **Berechnungen und Funktionen in Queries**
  - Mit GROUP BY können Berechnungen auch für Gruppen von Datensätzen mit gleichen Eigenschaften ausgegeben werden
    - Bsp.: Wir wollen wissen, wie viele Studierende jeweils welche VL hören

```
> SELECT VorlNr, MatrNr FROM hört;
+--------+--------+
| VorlNr | MatrNr |
+--------+--------+
|   5001 |  25403 |
|   5001 |  26120 |
|   5045 |  26120 |
+--------+--------+
```

    - Idee: Zeilen mit gleicher VorlesungsNummer werden gruppiert

```
> SELECT VorlNr, COUNT(MatrNr) FROM hört GROUP BY VorlNr;
+--------+---------------+
| VorlNr | COUNT(MatrNr) |
+--------+---------------+
|   5001 |             2 |
|   5045 |             1 |
+--------+---------------+
```

SQL / MySQL: SELECT

- **Verknüpfung von Tabellen (Join)**
  - 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. …

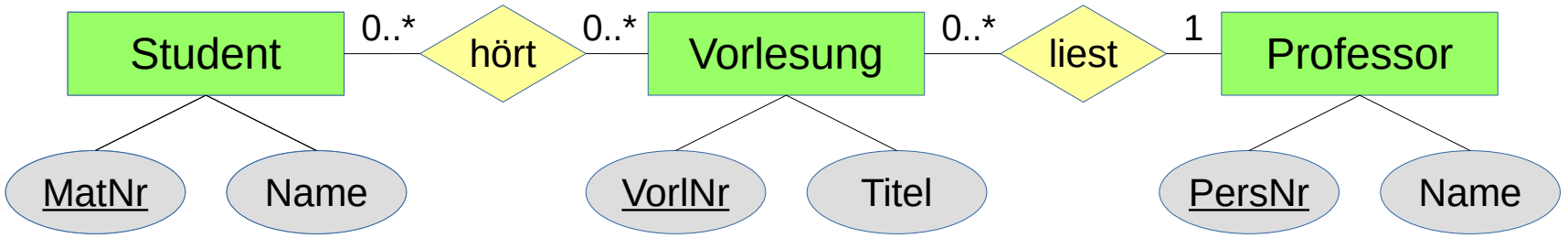
  - Mehreren Spalten müssen sich als Tupel unterscheiden: **UNIQUE**
    - UNIQUE [KEY](index_col_name, …)
    - Wenn mehrere Spalten nicht die selbe Wertkombination haben dürfen
  - Mehrere Spalten sind Primärschlüssel: **[PRIMARY] KEY**
    - PRIMARY KEY (index_col_name,...)
    - Beispiel: PRIMARY KEY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

               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 (implizit) **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 (MatrNr, Name) VALUES
               (26120, 'Fichte'),
               (25403, 'Jonas' ),
               (27103, 'Fauler');

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

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

INSERT INTO hört (MatrNr, VorlNr) 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

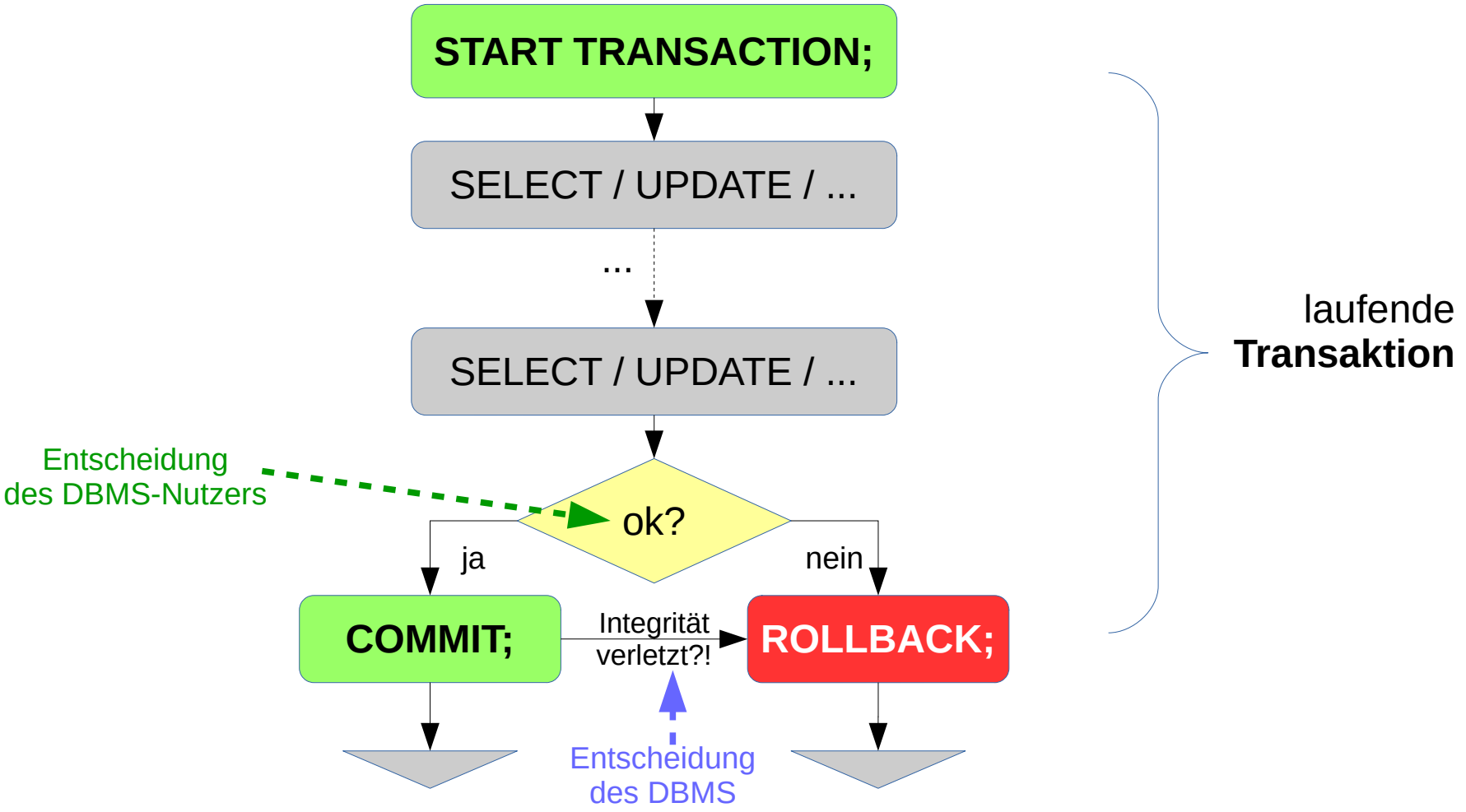
SQL / MySQL: Integrität

- **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

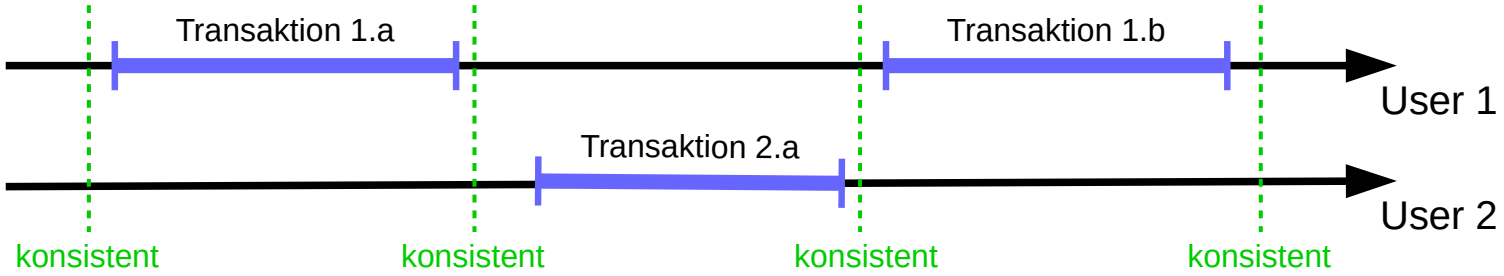
- **Konzept hinter Transaktionen und Integrität: „ACID“**

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

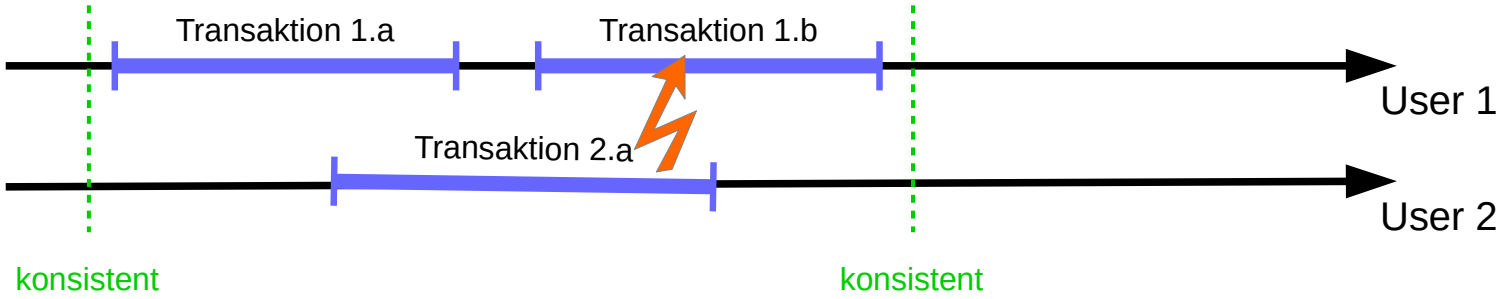
  - „A“ = **Atomarität** (engl. „Atomicity“, Abgeschlossenheit)
    - Jede Transaktion wird **ganz oder gar nicht** ausgeführt
      - z.B. die Bank-Überweisung von oben darf nicht halb ausgeführt werden, sonst geht im Saldo Geld verloren oder entsteht
  - „C“ = **Konsistenzerhaltung** (engl. „Consistency“)
    - 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**



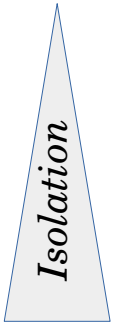
    - **Wunsch**: Es gibt eine strikt sequentielle Ausführung, die äquivalent ist

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

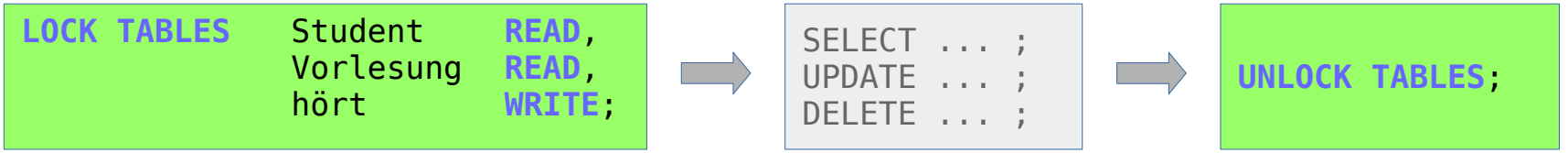
- **Probleme der semantischen „Isolation“**
  - Isolations-Niveau Steuerbar in SQL:
    - **SET TRANSACTION ISOLATION LEVEL** level
  - Werte für **level**:
    - READ UNCOMMITTED
    - READ COMMITTED (keine Dirty Reads)
    - REPEATABLE READ (keine Dirty + Non-Repeatable Reads)
    - SERIALIZABLE (keine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



  - 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';
```

```
SELECT * FROM Person
    WHERE  Name = 'Urlauber'
    AND    Statusgruppe = 'PROF';
```

  - 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
```