

LF 8 – Datenbanken

Kapitel 0 | [Vorbereitung](#)

Kapitel 1 | [Warum eine Datenbank?](#)

Kapitel 2 | [Eigenschaften](#)

Kapitel 3 | [Mathematische Operationen](#)

Kapitel 4 | [ER-Modell](#)

Kapitel 5 | [Normalisierung](#)

Kapitel 6 | [Relationenmodell](#)

Kapitel 7 | [SQLite](#)

Kapitel 8 | [Die Sprache SQL – Erstellung einer DB/Tabellen](#)

Kapitel 9 | [Die Sprache SQL – Grundlagen der Abfrage](#)

Kapitel 10 | [Views](#)

Kapitel 11 | [Sicherheit](#)



Kapitel 12 | [Trigger](#)

Kapitel 13 | [Transaktionen](#)

Kapitel 3 | [Übungsaufgaben](#)

Kapitel 4 | [Übungsaufgaben](#)

Kapitel 5 | [Übungsaufgaben](#)

Kapitel 8 | [Übungsaufgaben](#)

Kapitel 9 | [Übungsaufgaben1](#), [Übungsaufgabe2](#)

Kapitel 11 | [Übungsaufgaben](#), [Übungsaufgaben2](#)

LF 8 – Datenbanken

Kapitel 14 | [Optimierung](#)


Kapitel X1 | [Kundenprojekt](#)

Kapitel X2 | [Privatprojekt](#)

Kapitel X1 | [!!! Auftrag !!!](#)

Kapitel X2 | [!!! Auftrag !!!](#)

Hinweis(e):

1. Im Skript können Fehler enthalten sein. Wenn ihr einen Fehler findet, schreibt mir bitte eine E-Mail an:
joshua.schumacher@bbs1-mainz.de
2. Kapitel, die mit einem  versehen sind, existieren gegenwärtig noch nicht oder bedürfen einer Überarbeitung!
3. Gerade in der Informatik werden viele englischsprachige Begriffe verwendet. Je nach Kontext solltest du diese lernen, da auch die **Schlüsselwörter** entsprechend englisch sind (z. B. select, where, as, ...).
4. Das Skript ersetzt kein Buch oder Internetseite, die das Thema „vollständig“ behandelt. Es dient lediglich als roter Faden für den Unterricht. Weiterführende Literatur **muss in jedem Fall** gelesen/angesehen werden!
5. Lösungsvorschläge sind unter folgendem Repo zu finden (die Lösungen wurden von den Schülern erstellt und können in vielen Fällen noch optimiert werden):
https://github.com/JoshuaSchumacherGER/Datenbanken_Loesungen.git

Kapitel 0 | Vorbereitungen

Für den Unterricht werden folgende Programme benötigt:

- DB Browser für SQLite (<https://sqlitebrowser.org/>)
 - Wird ab Kapitel 7 benötigt. Vereinfacht die Erstellung der Datenbank.

Wenn wir mit Daten umgehen, möchten wir verschiedene **Operationen** durchführen können.
Dazu gehört ...

- Das Erstellen (**C**reate)
- Das Lesen (**R**ead)
- Das aktualisieren (**U**ppdate)
- Das Löschen (**D**elete)

.. von Dateien. Im ersten Moment klingt das simpel und primitiv. Schauen wir uns allerdings folgendes Beispiel an, wird schnell ersichtlich, warum diese grundlegenden Operationen nicht immer reibungslos ablaufen.

Beispiel:

Wir sind für das IT-System eines mittelgroßen Sportvereins mit 140 Mitgliedern zuständig. Dieser besteht aus mehreren Abteilungen (Abteilung Bogen, Abteilung Feuerwaffen, Abteilung Luftdruck). Aufgrund der rechtlichen Vorgaben (Lärm, Bebauung, usw.) liegt die Abteilung Feuerwaffen außerhalb am Waldrand.

Wenn wir jetzt die Mitglieder inkl. Beiträge verwalten wollten, wir könnten wir das tun?

Beispiel:

Wir sind für das IT-System eines mittelgroßen Sportvereins mit 140 Mitgliedern zuständig. Dieser besteht aus mehreren Abteilungen (Abteilung Bogen, Abteilung Feuerwaffen, Abteilung Luftdruck). Aufgrund der rechtlichen Vorgaben (Lärm, Bebauung, usw.) liegt die Abteilung Feuerwaffen außerhalb am Waldrand. Aufgrund der Größe existiert in jeder Abteilung ein Abteilungsleiter, der seine Mitglieder einpflegt und aktualisiert.

Wenn wir jetzt die Mitglieder inkl. Beiträge verwalten wollten, wir könnten wir das tun?

Wir könnten die Informationen in eine simple Textdatei schreiben (z. B. .txt). Ein Eintrag könnte folgende Form beinhalten

<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>

Frage: Welchen Großen Nachteil hat diese Vorgehensweise?

Wir könnten die Informationen in eine simple Textdatei schreiben (z. B. .txt). Ein Eintrag könnte folgende Form beinhalten

<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>

Einige Probleme:

- Es existiert in jeder Abteilung die oben genannte Datei. Ergibt sich eine Änderung, muss diese Änderung den anderen Abteilungen (irgendwie) mitgeteilt werden. Ein Problem entsteht jetzt, wenn ein Mitglied in mehreren Abteilungen registriert ist und sich dessen Beitrag ändert, z. B. passives Mitglied. Hier entstehen schnell **Inkonsistenzen**, die fatal sind.

Frage: Was wäre eine Lösungsmöglichkeit für dieses Problem ?

Ein erster Lösungsansatz wäre, lediglich eine Datei anzulegen, auf die die Abteilungsleiter Zugriff haben. Jeder Abteilungsleiter trägt nun seine Zeile in die Datei ein. Das Ergebnis wäre folgendes:

```
<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>  
1#Hans#Müller#Musterweg 1, 55411 Bingen#Luftdruck#50€  
2#Hans#Müller#Muster weg1, 55411 Bingen#Feuerwaffen#50€
```

Nun entstehen neue Probleme. Wo liegen hier die Schwachpunkte?

Das erste Problem liegt darin, dass jeder Abteilungsleiter sein Mitglied anlegt und dadurch bestimmte Mitglieder mehrere Mitgliedsnummern erhalten! => **Inkonsistenz!**

```
<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>  
1#Hans#Müller#Musterweg 1, 55411 Bingen#Luftdruck#50€  
2#Hans#Müller#Musterweg 1, 55411 Bingen#Feuerwaffen#50€
```

Außerdem ist die Adresse nicht vollkommen identisch. Obwohl der Unterschied marginal erscheint, ist es eine Inkonsistenz!

Beim ändern (update) einer Zeile müsste das Programm überprüfen, ob das Mitglied mehrfach existiert und entsprechend alle Zeilen aktualisieren. Gefahr einer => **Änderungsanomalie!**

Anomalien sind äußerst gefährlich. Je nach Struktur fallen diese Anomalien nicht direkt auf. Würde beispielsweise die Adresse von Hans Müller lediglich in einer Zeile geändert (Umzug), könnte es passieren, dass nach mehreren Monaten wichtige Post an die alte Adresse geschickt wird!

<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>

1#Hans#Müller#Musterweg 1, 55411 Bingen#Luftdruck#50€

2#Hans#Müller#Musterweg 1, 55411 Bingen#Feuerwaffen#50€

Wird hingegen ein Mitglied gelöscht (delete), tritt ein ähnliches Problem auf. Werden nicht betroffene Zeilen gelöscht, wird dem Mitglied weiterhin ein Beitrag abgebucht (was zu unschönen Telefonanrufen führen wird) =>

Löschanomalie!

Ebenfalls tritt ein Problem auf, wenn nicht alle Daten vorliegen. Es könnte durchaus vorkommen, dass eine Person Mitglied im Verein werden möchte, aber sich noch nicht auf eine Abteilung festlegt. Im oben gezeigten Design wäre dieser Fall problematisch, da die Zeile entsprechend lückenhaft wäre => **Einfügeanomalie!**

Das nächste Problem ist die **Datenredundanz**, die in diesem Beispiel auftritt! Auch wenn Festplatten immer günstiger werden, ist Datenredundanz (bis auf wenige Ausnahmen) zu vermeiden!

Wir sehen also, dass diese Vorgehensweise nicht zu empfehlen ist. Gleiche Problematik trifft auch zu, wenn wir ein Tabellenkalkulationsprogramm (z. B. Excel) verwenden!

Natürlich gibt es Vereine, die ihre Struktur über eine Exceldatei abbilden. Oben genannte Probleme lassen sich dadurch kaum vermeiden.

Kapitel 1 | Warum eine Datenbank?

```
<Nr>#<Vorname>#<Nachname>#<Adresse>#<Abteilung>#<Beitrag>  
1#Hans#Müller#Musterweg 1, 55411 Bingen#Luftdruck#50€  
2#Hans#Müller#Musterweg 1, 55411 Bingen#Feuerwaffen#50€
```

Frage: Wie könnten wir obiges Design verbessern?

Die Aufteilung in mehrere Tabellen wäre ein Anfang:

<Mnr>#<Vorname>#<Nachname>#<Adresse>

<Abtnr>#<Abteilung>

<Mnr>#<Beitrag>

In diesem Entwurf wäre Hans Müller lediglich einmal in der Datenbank vorhanden. Ein update/delete wäre dadurch schneller und fehlerresistenter!

Auch der Mitgliedsbeitrag ist lediglich einmal vorhanden. Dieser wird durch die Mnr (Mitgliedsnummer) aus der ersten Tabelle eindeutig!

Nun besteht die Problematik, die Tabellen in eine **Relation** zu bringen.

Diese Struktur bietet noch weiteres Optimierungspotenzial, welches erst einmal noch keine Rolle spielt. Der Grundgedanke sollte hier klar geworden sein!

Die Daten müssen demnach strukturiert werden. Folgende Ziele sollte die Datenorganisation verfolgen:

- **Datenunabhängigkeit**
Unser neues System sollte nicht von einem speziellen Programm abhängig, sondern anwendungsneutral gespeichert werden.
Außerdem sollten die CRUD-Operationen im Hintergrund ablaufen. Der Benutzer muss lediglich die Datenstruktur kennen. Alles andere soll für den Benutzer transparent ablaufen.
- **Benutzerfreundlichkeit**
Das neue System sollte sowohl für den Entwickler, als auch für den Benutzer einfach in der Bedienung sein
- **Multi-User-Support**
Mehrere Benutzer sollten „zeitgleich“ am System arbeiten können!
- **Flexibilität**
Die Daten müssen in beliebiger Form verknüpfbar sein. Das heißt, der Zugriff soll wahlfrei funktionieren!
- **Effizienz**
Das System soll effizient arbeiten, also Abfragen sofort, ohne große Wartezeit ausführen.
- **Datensicherheit**
Nach einem Systemausfall soll es möglich sein, die Daten möglichst einfach wiederherzustellen.
- **Datenintegrität**
Die Daten sollen frei von Anomalien sein (also vollständig, korrekt, widerspruchsfrei!).

Die Daten müssen demnach strukturiert werden. Folgende Ziele sollte die Datenorganisation verfolgen:

- Redundanzfreiheit
I.d.R. sollte jedes Element lediglich einmal vorliegen!

Die Forderungen sind bewusst mit dem Wort „soll“ beschrieben. Es gibt keine perfekte Datenorganisation. Umso höher die Effizienz, umso niedriger die Redundanz, usw.

Kapitel 2 | Eigenschaften

Eine Datenbank erstellen wir demnach erst, wenn wir uns einige grundlegende Gedanken zum Design gemacht haben!



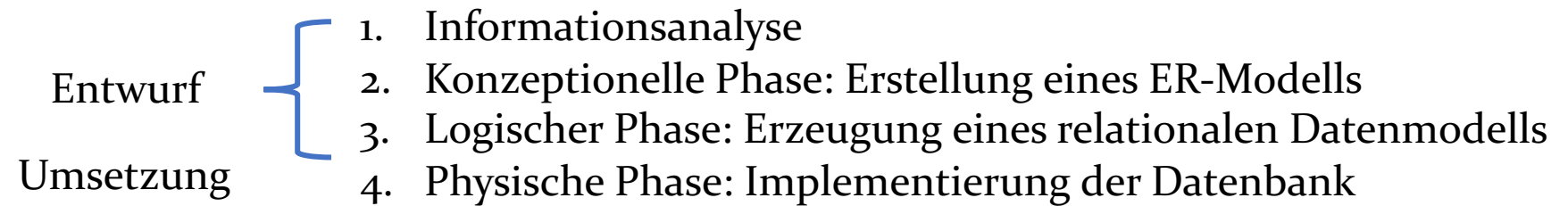
[2]

Möchten wir ein neues Datenbankprojekt erstellen, können wir uns an folgende Checkliste halten:

- Definition von Benutzergruppen
- Sammlung und Analyse der Daten
- Datenbankdesign
- Auswahl eines geeigneten DBMS
- Anwendungsdesign
- Erstellung eines Prototypen
- Test des Prototyps und eventuelle Korrekturmaßnahmen
- Implementierung der Datenbank
- Datenerfassung, Test und Korrektur

Vgl: <https://www.datenbanken-verstehen.de/datenbankdesign/lebensstadien/>

Phasen der Datenbankmodellierung:



Hier wird sehr schnell ersichtlich, dass die Implementierung der Datenbank lediglich einer von vier Phasen darstellt!

Das Anwendungsdesign bezieht sich auf (unter anderem) auf folgende Punkte:

- Usability!
- Einheitliche Verwendung einzelner Felder
- Ansprechendes Layout!
- Einheitliche Terminologie
- Einheitliche/s Farbmarkierung und Farbschema
- Einfache Fehlerkorrektur !!! (sehr wichtig!)
- Klare Trennung zwischen Pflicht- und freiwilligen Angaben
- Hilfsfunktionen/Tooltips für einzelne Felder

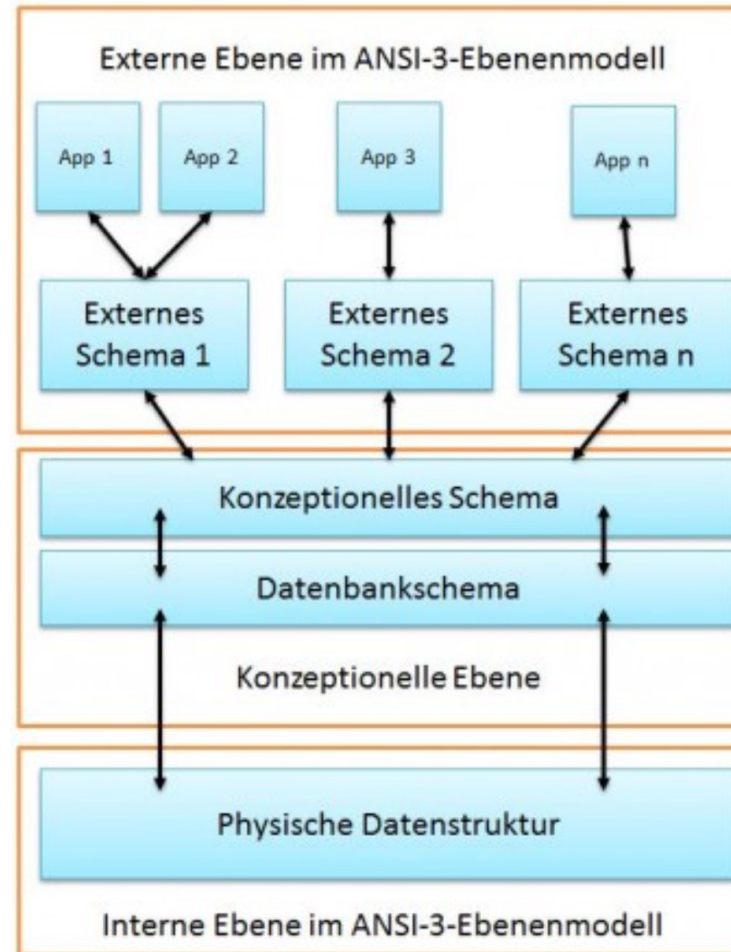
Vgl: <https://www.datenbanken-verstehen.de/datenbankdesign/anwendungsdesign/>

Ein gutes Design ist immer stringent und trennt Schichten klar voneinander ab! Entsprechende Regeln zwischen Front- und Backend müssen existieren und eingehalten werden!

Wer ist beispielsweise dafür verantwortlich, wenn ein Feld leer gelassen wird? Der Frontend-Entwickler, der die fehlerhafte Eingabe in der Anwendungssoftware auffängt oder der Backend-Entwickler? Beide Bereiche besitzen Möglichkeiten, mit derartigen Eingaben umzugehen. Es muss allerdings vorher die Frage geklärt sein, wer für diese Art reagiert!

Ein chaotisches Design führt zu einem schwer wartbaren Produkt! Daher sollten diese Fragestellungen unbedingt vor der **Implementierungsphase** geklärt worden sein! Ein trial-and-error-Verfahren mag bis zu einem gewissen Punkt funktionieren, führt im Bereich der Datenbanken zu größeren Problemen und sollte entsprechend vermieden werden!

ANSI-3-Ebenenmodell



Ausschnitt aus der
Realität



ER-Modell
Relationales Modell



Frage: Welche Vorteile
bietet die Trennung in
verschiedene Schichten?

Vgl: <https://www.datenbanken-verstehen.de/lexikon/ansi-drei-ebenenmodell/>

Ein Datenbanksystem besteht aus einer **Datenbasis** (d. h. die Tabellen (z. B. Mitglieder, Abteilungen usw.)) und dem **Datenbankmanagementsystem** (DBMS). Bekannte DBMS sind z. B. Microsoft SQL Server, MySQL, MongoDB, PostgreSQL, ...

Für welches DBMS man sich entscheidet hängt von einigen Faktoren ab:

- Kosten
- Geschwindigkeit
- Skalierbarkeit
- Integrierbarkeit
- Szenario
- ...

Die falsche Auswahl eines DBMS kann ebenfalls zu größeren Problemen führen.
Beispiel: File size limitation MS Access ...

Arbeitsauftrag: Informiert euch über folgende DBMS und listet die Unterschiede bzw. die jeweiligen Vor- und Nachteile auf:

MySQL
MS SQL
PostgreSQL
MS Access
MongoDB

Kapitel 2 | Eigenschaften

Noch einige Worte zu der Operation **delete/update**. Updates können bei einem undurchdachten Datenbankdesign zu schwerwiegenden Konsequenzen führen. Nehmen wir folgendes Beispiel:

Ihr möchtet eine Schülerverwaltung entwickeln ... Welche Informationen sollte diese Tabelle beinhalten?

Noch einige Worte zu der Operation **delete/update**. Updates können bei einem undurchdachten Datenbankdesign zu schwerwiegenden Konsequenzen führen. Nehmen wir folgendes Beispiel:

Ihr möchtet eine Schülerverwaltung entwickeln ... Welche Informationen sollte diese Tabelle beinhalten?

// Lösung:

Wir benötigen folgende Mindestinformationen:
Schüler, Lehrer, Klasse, Räume und Noten.

Was passiert, wenn der Schüler sitzen bleibt?

Die Operation **delete** bedarf ebenfalls einiger Worte. Datenbanken können (ebenfalls abhängig von Szenario und Design) sehr groß werden. Vor einigen Jahrzehnten galt die Regel:

„nicht benötigte Datensätze werden gelöscht, um Speicherplatz zu sparen!“

Allerdings verlieren wir durch ein delete Informationen, auf die wir vllt. in der Zukunft noch zugreifen möchten. Wenn beispielsweise ein Kunde bei uns ein Abonnement abschließt und dieses kündigt, sollten wir diesen Kunden dann vollständig aus unserer Datenbank löschen?

Allerdings verlieren wir durch ein delete Informationen, auf die wir vllt. in der Zukunft noch zugreifen möchten. Wenn beispielsweise ein Kunde bei uns ein Abo abschließt und dieses kündigt, sollten wir diesen Kunden dann vollständig aus unserer Datenbank löschen?

Es wäre in diesem Fall oftmals sinnvoller, den Datensatz mit einem Flag zu markieren und lediglich aus der View des benutzers auszublenden. Damit haben wir im Fall der Fälle die Möglichkeit, auf die Informationen weiterhin zuzugreifen!!!

Natürlich belegt dieses Vorgehen Speicherplatz! Hier muss wieder eine Abwägung getroffen werden, welches Vorgehen sinnvoller ist.

Ihr seht, wie viele Gedanken wir uns machen sollten/müssen, **bevor** eine Datenbank implementiert wird!

Kapitel 3 | Mathematische Operationen

In einer relationalen Datenbank können Daten auf Basis von mathematischen Operationen gehandhabt werden.

Ein grundlegendes Verständnis der folgenden Operationen ist unerlässlich, damit eine spätere Optimierung gelingen kann!

Vereinigung

Bei der Vereinigung werden die Werte aus beiden Tabellen entnommen und zusammengefügt.
Dubletten entstehen hier nicht!

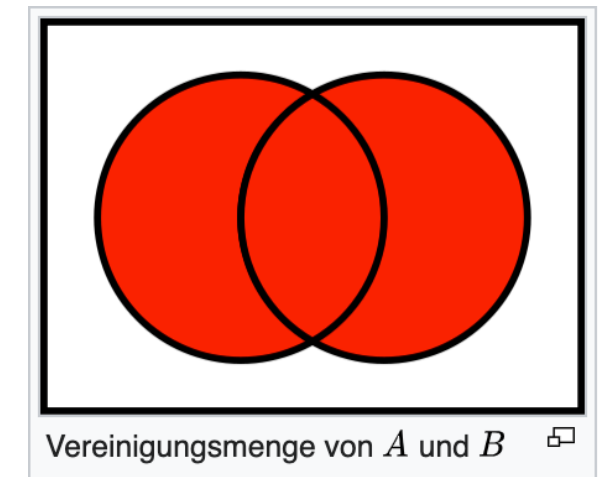
Beispiel:

Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro
Dübel	2 Euro

Produkt	Kosten
Hammer	25 Euro
Säge	15 Euro
Dübel	2 Euro



Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro
Hammer	25 Euro
Säge	15 Euro
Dübel	2 Euro



[3] [https://de.wikipedia.org/wiki/Menge_\(Mathematik\)#Vereinigung_\(Vereinigungsmenge\)](https://de.wikipedia.org/wiki/Menge_(Mathematik)#Vereinigung_(Vereinigungsmenge))

Differenz

Hier werden Daten entweder aus Tabelle 1 oder 2 übernommen, je nachdem von welcher Tabelle subtrahiert wird!

Beispiel:

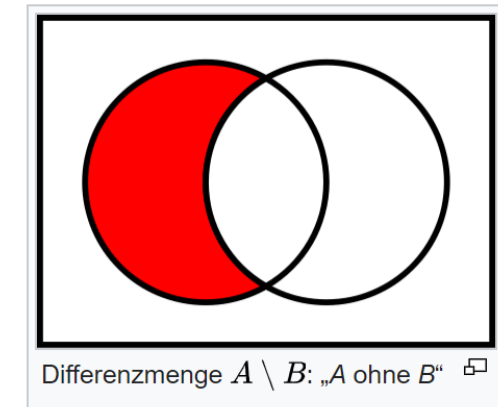
Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro
Dübel	2 Euro

Produkt	Kosten
Hammer	25 Euro
Säge	15 Euro
Dübel	2 Euro



Für A ohne B

Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro



Differenzmenge $A \setminus B$: „A ohne B“

[3] [https://de.wikipedia.org/wiki/Menge_\(Mathematik\)#Vereinigung_\(Vereinigungsmenge\)](https://de.wikipedia.org/wiki/Menge_(Mathematik)#Vereinigung_(Vereinigungsmenge))

Kapitel 3 | Mathematische Operationen

Schnittmenge

Es werden lediglich Daten entnommen, die in beiden Tabellen vorhanden sind!

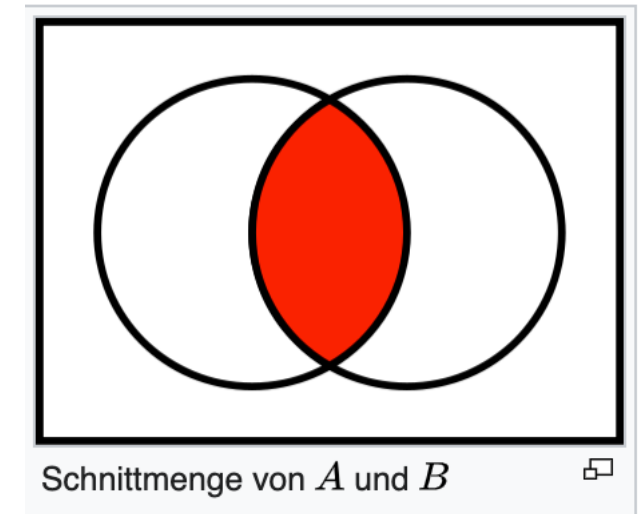
Beispiel:

Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro
Dübel	2 Euro

Produkt	Kosten
Hammer	25 Euro
Säge	15 Euro
Dübel	2 Euro



Produkt	Kosten
Dübel	2 Euro



[3] [https://de.wikipedia.org/wiki/Menge_\(Mathematik\)#Vereinigung_\(Vereinigungsmenge\)](https://de.wikipedia.org/wiki/Menge_(Mathematik)#Vereinigung_(Vereinigungsmenge))

Kartesisches Produkt

Das kartesische Produkt kombiniert alle Zeilen beider Tabellen!

Beispiel:

Produkt	Kosten
Nägel	5 Euro
Schrauben	4 Euro

Produkt	Kosten
Hammer	25 Euro
Säge	15 Euro



Produkt	Kosten	Produkt	Kosten
Nägel	5 Euro	Hammer	25 Euro
Nägel	4 Euro	Säge	15 Euro
Schrauben	4 Euro	Hammer	25 Euro
Schrauben	4 Euro	Säge	15 Euro

Die vorgenannten Mengenoperationen müssen wir um hilfreiche Operationen erweitern, die typischerweise in relationalen Datenbanken Verwendung finden!

Möchte man Beispielsweise eine einzelne **Spalte** einer Tabelle ausgeben, verwenden wir hierzu die **Projektion**!

Möchten wir hingegen eine bestimmte **Zeile** aus einer Tabelle ausgeben, verwenden wir die **Selektion**!

Kapitel 3 | Mathematische Operationen

Eine der wichtigsten Operationen überhaupt ist der **Join**! Schauen wir uns folgendes Beispiel an:

Produktnummer	Name	Preis pro Einheit
10	Salamipizza	10
11	Cola	1,80

Datum	Produktnummer	Menge
11.11	10	1
11.11	11	1



Join

Produktnummer	Name	Preis pro Einheit	Datum	Menge
10	Salamipizza	10	11.11	1
11	Cola	1,80	11.11	1

Kapitel 3 | Mathematische Operationen

Bei der Division werden zuerst die Zeilen extrahiert, die in beiden Tabellen vorkommen, um danach die Spalten zu löschen, die in der rechten Tabelle enthalten sind. Beispiel:

Produktnummer	Name	Datum
10	Salamipizza	11.11
11	Cola	12.11.

Produktnummer	Name
10	Salamipizza



Division

Datum
11.11

Kapitel 3 | Übungsaufgaben

Aufgabe 1: Wie nennt man den Schlüssel, der eine Spalte eindeutig identifiziert?

Aufgabe 2: Wie nennt man den Schlüssel, der auf eine Spalte einer anderen Tabelle verweist?

Aufgabe 3: Erstellen Sie ein eigenes Beispiel, bei der man einzelne Zeilen entnehmen kann und benennen Sie diese Vorgehensweise.

Aufgabe 4: Erstellen Sie zwei Tabellen und vereinigen Sie beide Tabellen miteinander.

Mit einem Entity-Relationship-Modell plant man die Struktur einer Datenbank, bevor diese erstellt wird.

Hierbei ist eine Entität ein identifizierbares Objekt oder Sachverhalt aus der realen Welt. Z. b. Der Lehrer Hobelsberger, der Schüler Müller aus der BSFI20, usw.



Entität

Entitäten werden als Box dargestellt.



Schwache
Entität

Neben der normalen Entität gibt es noch schwache Entitäten. Diese sind durch die doppelte Umrandung erkennbar. Eine schwache Entität kann nur existieren, wenn die dazugehörige Entität existiert. Ein Klassenraum kann nur existieren, wenn das Gebäude Schule existiert. Ohne Schule, kein Klassenraum ; Ohne Stuhl kein Stuhlbein, ...

Jede Entität wird durch Attribute beschrieben. Auch Beziehungen zwischen Entitäten können Attribute besitzen. Folgende Attributstypen sind zu unterscheiden:

Normales
Attribut

z. B. besitzt ein Lehrer (Entität) einen Vornamen, Nachnamen, Alter, ...

Mehrwertige
s Attribut

sind Attribute, die mehrere Werte annehmen können. Z. B. das Attribut Fächer eines Lehrers. Es gibt Lehrer mit einem Fach, zwei Fächern, ...

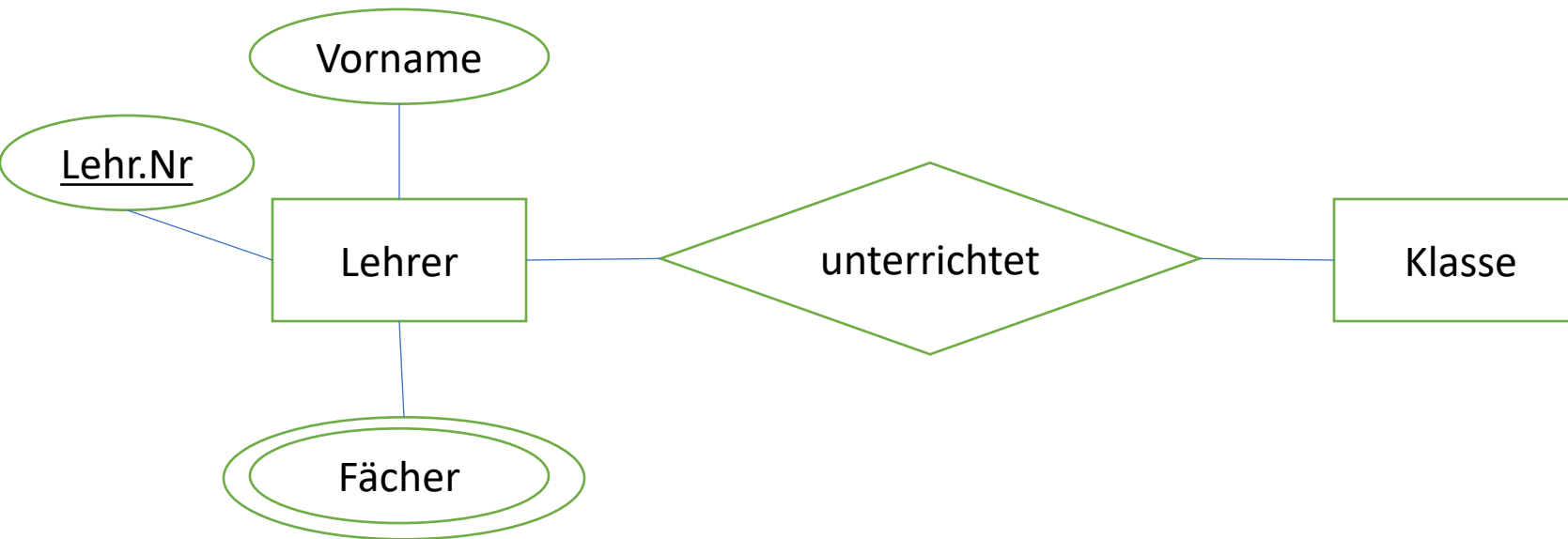
Abgeleitetes
Attribut

sind Attribute, die durch andere hergeleitet werden können. Durch die Klasse und die Klassenbezeichnung, kann der Jahrgang ermitteln.

Schlüssel

Unterstrichene Attribute sind Primärschlüssel (identifizieren das Objekt eindeutig!)

Beziehungen stellen die Verbindung zwischen den einzelnen Entitäten her. Hierbei ist die „Leserichtung“ zweitrangig. Betrachten wir uns folgendes Beispiel:



Hier steht der Lehrer über die Beziehung „unterrichtet“ mit der Klasse in Verbindung. Allerdings sagt die obige Grafik noch nichts über die **Ordinalität/Kardinalität** aus...

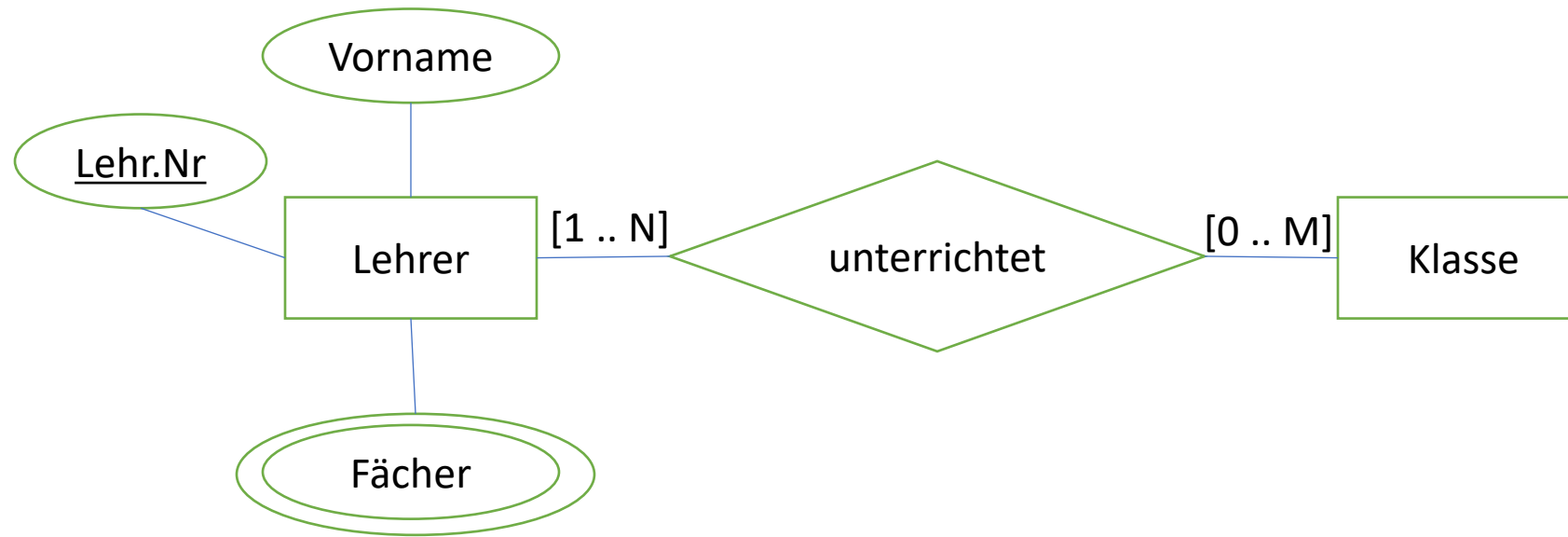
Wir können folgende Kardinalitäten (Beziehungsformen) unterscheiden:

1 zu 1 Beziehung: Hier steht jede Entität mit genau einer Entität in Beziehung.

1 zu N Beziehung: Eine Entität steht mit beliebig vielen Entitäten in Beziehung.

N zu M Beziehung: Jede Entität steht mit beliebig vielen Entitäten in Beziehung.

Diese Grunddefinition lässt allerdings kein Minimum/Maximum zu. Daher erweitern wir die Schreibweise um Anzahl der Entitäten in **[Min .. Max]**. Sehen wir uns hierfür unser Beispiel erneut an:



Eine Klasse wird von mindestens einem Lehrer unterrichtet. Wie viele Lehrer eine Klasse unterrichten ist beliebig (daher $1 \dots N$). Ein Lehrer muss nicht zwingend eine Klasse unterrichten (evtl. Freistellung). Der Normalfall ist allerdings, dass ein Lehrer mehrere Klassen unterrichtet.

WICHTIG: Je nach Vereinbarung kann die Kardinalitätsangaben auch vertauscht werden. Das führt dann zur Verwirrung. Daher muss genau darauf geachtet werden, welche Variante der Autor verwendet. Wir verwenden die Angabe oben!

Das Thema ER-Modell wurde hier nur skizziert. Daher sollten folgende Übungsaufgaben gewissenhaft erledigt werden:

Aufgabe 1: Informiere dich im Internet über das Thema ER-Modell. Was sind schwache Beziehungen?

Aufgabe 2: Wie kann man einen zusammengesetzten Schlüssel im ER-Modell darstellen?

Aufgabe 3: Beziehungen können auch zwischen den Entitäten desselben Typs entstehen (rekursive Beziehungen). Skizziere ein Beispiel hierfür.

Aufgabe 4: Was ist ein Fremdschlüssel?

Aufgabe 5: Über den Vor- und Nachnamen könnte ein zusammengesetzter Schlüssel gebildet werden. Warum ist das keine gute Idee?

Aufgabe 6: Setzen Sie folgenden Text in ein ER-Modell um:

“In unserer Beispielfirma werden diverse Projekte realisiert. Ein Projekt wird von maximal einem Mitarbeiter geleitet. Allerdings können beliebig viele Mitarbeiter einem Projekt zugeordnet sein. Der leitende Mitarbeiter kann ebenfalls beliebig viele Projekte leiten. Ein Projekt besteht aus einem Namen, einem Beginn und Ende (Datum) und einer Beschreibung. Der Mitarbeiter besteht aus einer Personalnummer und einem Vor- und Nachnamen.“

Nachdem wir das ER-Modell erstellt haben, könnten wir (nach einigen Überlegungen bezüglich der Kardinalitätsabbildungen usw.) die Datenbank direkt erstellen.

Sinnvoller ist es häufig, die Datenbank in eine für uns ausreichende **Normalform** zu bringen. Normalformen reduzieren die Datenredundanz und vermeiden Anomalien!

Es gibt eine Vielzahl von Normalformen, die jeweils einen anderen Zweck verfolgen. Wir werden uns lediglich mit den ersten **drei** Normalformen beschäftigen, die (normalerweise) auch ausreichend sind.

Die Normalformen bauen aufeinander auf! Ist eine Datenbank bereits in der dritten Normalform, so sind alle Bedingungen der zweiten und ersten Normalform zwingend erfüllt!

Die erste Normalform:

Definition: Eine Relation befindet sich in der ersten Normalform, wenn alle Attribute atomar vorliegen!

Negativbeispiel:

Adresse
Hans Hubert, Burgstr. 1, 55431 Wichtelhausen

Die obige Information ist **NICHT** atomar. Atomar bedeutet, dass das Attribut nicht weiter aufgeteilt werden kann, z. B. „PLZ“.

Die zweite Normalform:

Definition: Eine Relation befindet sich in der zweiten Normalform, wenn die erste Normalform erfüllt ist und jedes Nicht-Schlüsselattribut von jedem Schlüsselattribut vollständig funktional abhängig ist.

Vollständig funktional bedeutet hierbei, dass ein Nicht-Schlüsselattribut von allen Teilen eines zusammengesetzten Schlüssels abhängig sein muss, nicht nur von einem Teil!

Vereinfacht ausgedrückt: Ein Datensatz sollte einen Sachverhalt abbilden. Werden mehrere abgebildet, sollten diese in separate Tabellen ausgegliedert werden.

Die zweite Normalform:

Beispiel:

IDSchueler	Nachname	IDLehrer	Lehrer	Note
1	Mueller	1	Schumacher	4

Die obige Tabelle ist in der ersten Normalform (Informationen liegen atomar vor!).
Die Tabelle ist allerdings nicht in der **zweiten** Normalform!

Warum?

Die zweite Normalform:

Beispiel:

IDSchueler	Nachname	IDLehrer	Lehrer	Note
1	Mueller	1	Schumacher	4

Die obige Tabelle ist in der ersten Normalform (Informationen liegen atomar vor!).
Die Tabelle ist allerdings nicht in der **zweiten** Normalform!

Antwort: Der Lehrer ist funktional abhängig von IDLehrer. Allerdings ist Lehrer nicht voll funktional abhängig von IDSchueler. Hier liegen zwei Sachverhalte vor.

Lösung?

Die zweite Normalform:

Lösung:

Tabelle: Schüler

IDSchueler	Nachname
1	Müller

Tabelle: Lehrer

IDLehrer	Lehrer
1	Schumacher

Tabelle: Noten

IDSchüler	IDLehrer	Note
1	1	4

*Problematischer Sonderfall: Falls eine Lehrkraft mehrere Lernfelder unterrichtet, entsteht bei dieser Aufteilung ein Problem!
Wie sieht eine robustere Lösung aus?*

Die zweite Normalform:

Zweites Beispiel: Folgende Tabelle ist gegeben:

IDSchüler	IDFach	Fachname
1	1	Sport
1	4	Geschichte
2	1	Sport

Hier wird ein zusammengesetzter Schlüssel benötigt, um eine eindeutige Identifizierung vorzunehmen (IDSchüler, IDFach).

Warum?

Der Fachname ist allerdings lediglich von einem Teil des zusammengesetzten Schlüssels abhängig (IDFach). Damit ist die Regel für die zweite Normalform verletzt!!!

Lösung?

Die zweite Normalform:

Zweites Beispiel: Folgende Tabelle ist gegeben:

IDSchüler	IDFach	Fachname
1	1	Sport
1	4	Geschichte
2	1	Sport

Hier wird ein zusammengesetzter Schlüssel benötigt, um eine eindeutige Identifizierung vorzunehmen (IDSchüler, IDFach).

Warum?

Der Fachname ist allerdings lediglich von einem Teil des zusammengesetzten Schlüssels abhängig (IDFach). Damit ist die Regel für die zweite Normalform verletzt!!!

Lösung: Fachname als Spalte löschen. Die Information muss bereits in einer Tabelle Fächer vorhanden sein!

Die dritte Normalform:

Definition: Eine Relation ist in der dritten Normalform, wenn sie sich in der zweiten Normalform befindet und es keine transitive Abhängigkeit von Nicht-Schlüsselattribut und Schlüssel existiert.

Transitiv Abhängigkeit bedeutet folgendes: Wenn Y von Z abhängig ist und X von Y, dann muss X von Z abhängig sein. (Grafisch: $X \rightarrow Y \rightarrow Z$).

Tabelle: Schüler

Negativbeispiel:

IDSchueler	Name	PLZ	Ort
1	Müller	55421	Wichtelhausen

Die obige Tabelle ist in der zweiten Normalform, allerdings nicht in der dritten.

Warum?

Die dritte Normalform:

Antwort: Mit IDSchueler kann ich PLZ ermitteln und mit PLZ wiederum den Ort (Grafisch: X -> Y -> Z).

Verbesserung:

Tabelle: Schüler




IDSchueler	Name	PLZ
1	Mueller	55421

Tabelle: PLZ

PLZ	Ort
55421	Wichtelhausen

Problematischer Sonderfall: Mehrere Orte können dieselbe PLZ haben. Lösung?

Roadmap einer Datenbankerstellung (für kleinere Projekte):

1. Entitäten, Attribute und Beziehungen klären. 
2. ER-Modell skizzieren (optional: hilft allerdings enorm, gedankliche Fehler zu finden!) 
3. Normalisierung bis Grad 3 durchführen. 
4. Relationenmodell erstellen
5. DB erstellen!
6. Test-Datensätze einspielen! *(auf die Sprache SQL und die technischen Feinheiten gehen wir noch in den folgenden Kapiteln ein!)*
7. Grundlegende DB-Operationen alle testen.

Kapitel 5 | Normalisierungen

Aufgabe 1: Warum sollten wir Normalisierungen verwenden?

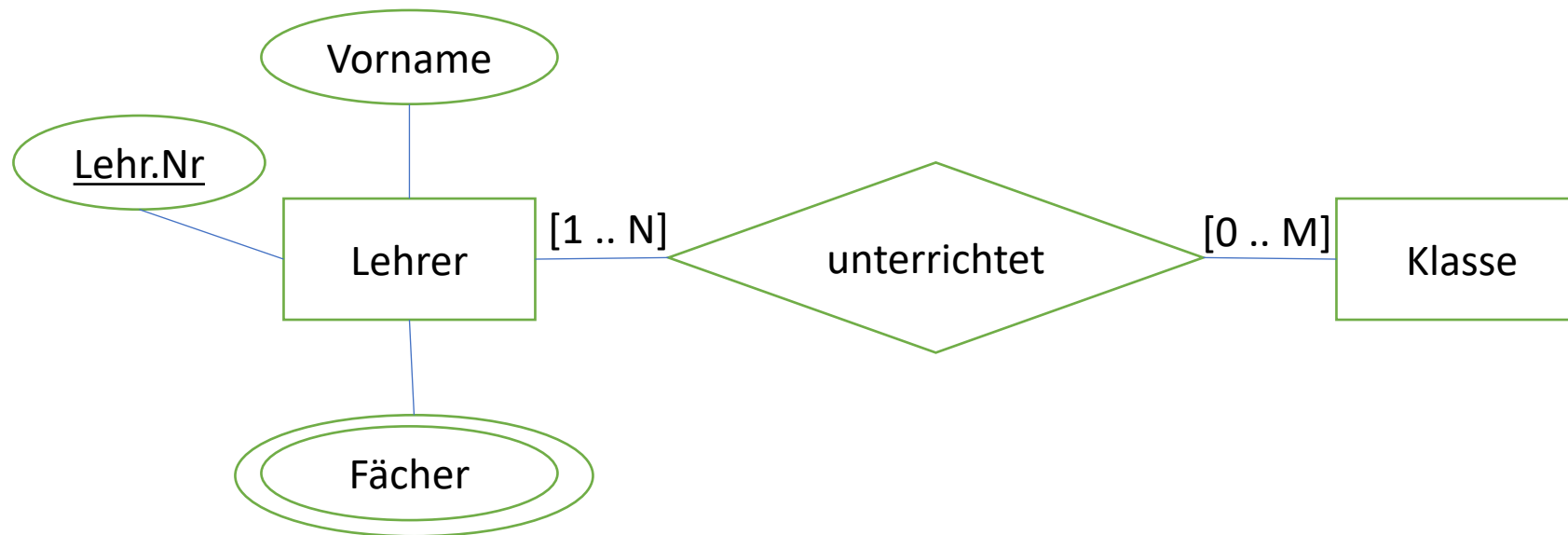
Aufgabe 2: In welcher Normalform befindet sich ein Relationsschema (i. d. R., also häufig), wenn zuvor das ER-Modell angewendet wurde!

Aufgabe 3: Definieren Sie alle drei Normalformen schriftlich in eigenen Worten.

Aufgabe 4: Erstellen Sie zu jeder Normalform ein eigenes Beispiel. Schön wäre es, wenn es sich um ein Szenario handelt!

Kapitel 6 | Relationenmodell

Normalerweise sollte jetzt spätestens der Wunsch aufkommen, endlich die Datenbank zu erstellen. Allerdings fehlt uns hierfür noch ein kleiner Zwischenschritt, ohne den die Erstellung (gerade für Anfänger) schwierig wäre. Schau dir hierzu noch einmal folgendes Beispiel an:



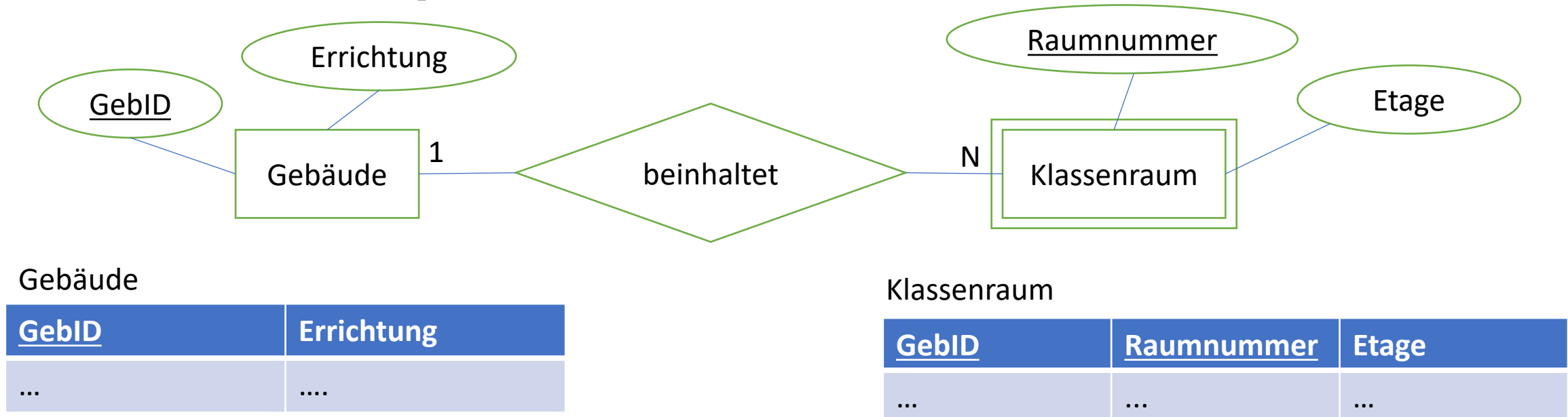
Zum aktuellen Zeitpunkt ist noch nicht jedem klar, wie er das ER-Modell sinnvoll umsetzen könnte. Wie können wir beispielsweise eine 1:N oder N:M-Beziehung abbilden?

Für diese Fragestellungen benötigen wir das **Relationenmodell**!

Für das Relationenmodell gelten folgende Regeln:

1. Jede Entität des ER-Modells entspricht einem Relationsschema!
2. Attribute einer Entität entsprechen Attributen des Relationsschemas!
3. Schlüssel der Entität entsprechen Schlüssel des Relationsschemas!
4. Attribute einer “weak Entity” (schwachen Entität) werden um den Schlüssel der Hauptentität ergänzt!

Schauen wir uns hierfür ein Beispiel an:



Für zweiwertige Beziehungen gelten folgende Regeln:

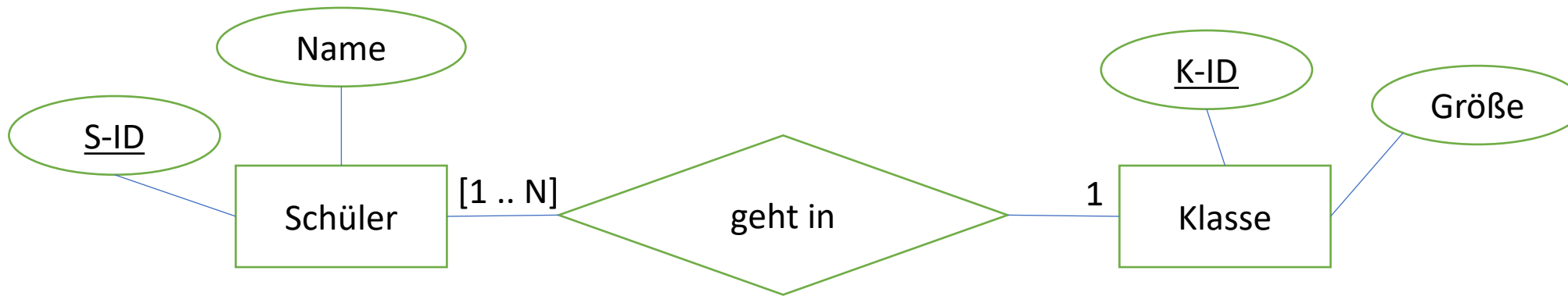
1:1 = keine zusätzliche Relation! Eine der beiden Relationen wird um den Fremdschlüssel der anderen erweitert

1:n = Relation auf der N-Seite wird um den Schlüssel der 1-Seite erweitert!

n:m = Es wird eine **neue** Relation erzeugt. Diese beinhaltet die beiden Primärschlüssel aus den zugrundeliegenden Entitäten!

Die obigen Regeln gelten als Grundgerüst und beinhalten keine Optimierungen. Je nach Anwendungsszenario sind durchaus Variationen denkbar! Generell stellen die Folien im Skript lediglich einen Kurzüberblick da. Für eine umfassende Einarbeitung in das Thema **MUSS** zusätzliche Literatur gelesen werden!

Beispielhaft die Übersetzung einer **1:n**-Beziehung:

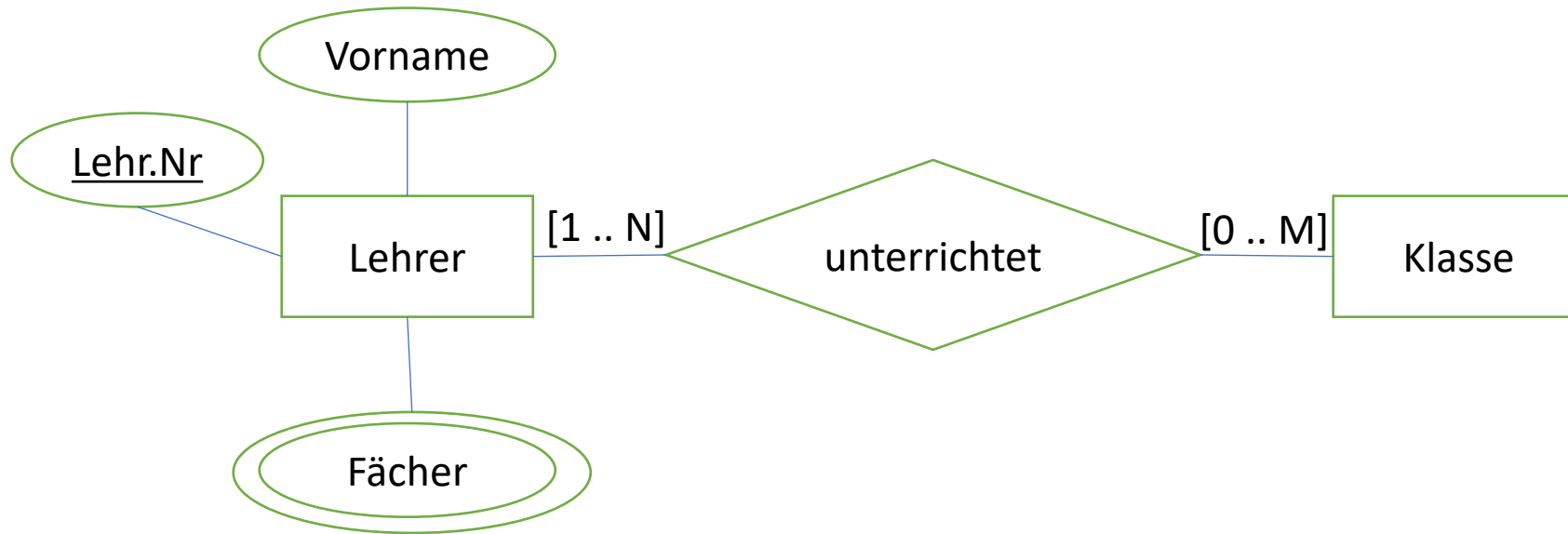


<u>S-ID</u>	K-ID	Name
...

<u>K-ID</u>	Größer
...	...

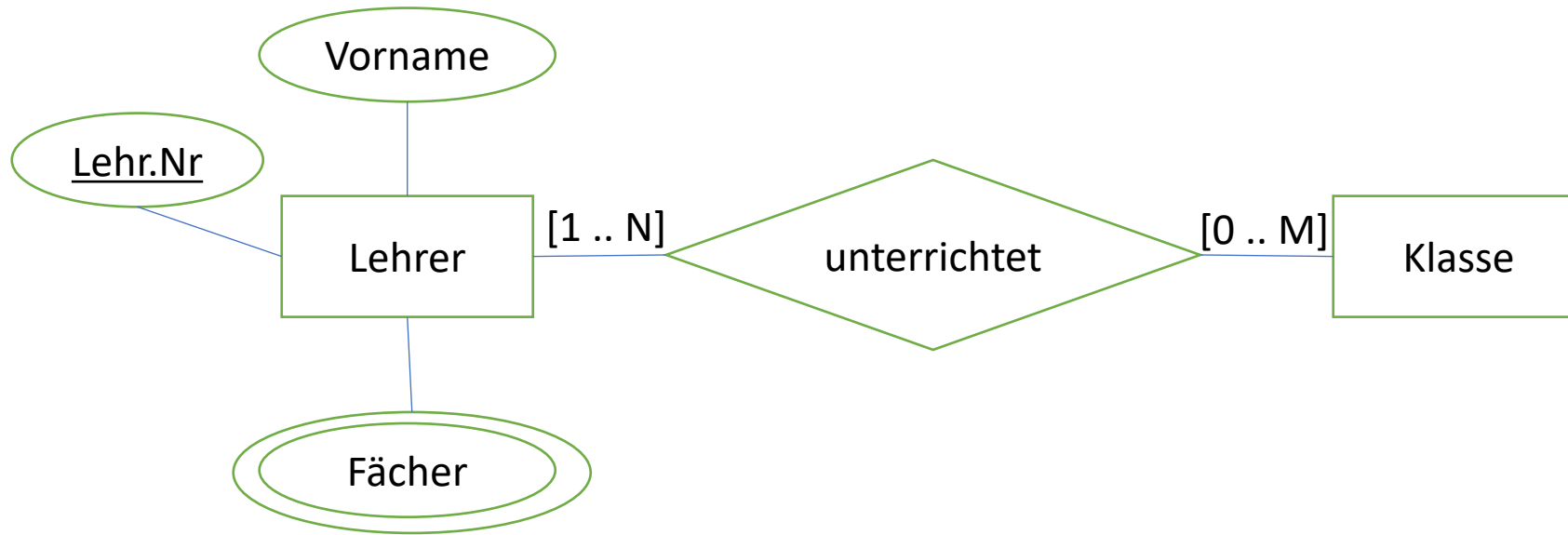
Alternativ wäre es möglich, unabhängig der **Beziehungsform** drei Tabellen zu erstellen. Aus Optimierungsgründen ist davon allerdings abzuraten!

Schauen wir uns zum Schluss noch einmal unser Ausgangsbeispiel an:



Frage: Wie können wir das mehrwertige Attribut Fächer übersetzen?

Schauen wir uns zum Schluss noch einmal unser Ausgangsbeispiel an:



Antwort: Für das Attribut Fächer muss eine zusätzliche Relation - mit Bezug zur Lehrer-Entität - erstellt werden!

Die Schreibweise für die Darstellung im Relationenmodell lautet wie folgt:

Relation(primary_key,attribut2,attribut3,attributN).

Beispiel:

Wir wollen die Entität Lehrer mit den Attributen L-ID, Vorname, Nachname überführen:

Lehrer(L-ID,Vorname,Nachname)

Regeln:

1. Die Notation erfolgt in Tupelschreibweise
2. Der Primärschlüssel wird immer als erstes angegeben!
3. Der Primärschlüssel wird unterstrichen
4. Die restliche Reihenfolge ist beliebig

Durch diese sehr kompakte Schreibweise können wir auch größere Datenbankstrukturen übersichtlich darstellen. Beachte: Das Relationenmodell liefert uns **keine** Hinweise darauf, welche zusätzlichen Einschränkungen gesetzt werden müssen/können!

Bsp: Lehrer(L-ID, Vorname, Nachname)

Beispielsweise macht es wenig Sinn, einen Lehrer ohne Nachnamen in die Datenbank eintragen zu können! SQL bietet uns die Möglichkeit, solche Fälle bei der Erstellung auszuschließen. Allerdings sollte hier die Frage sein, wer dafür verantwortlich ist, dass eine leere Spalte (Nachname) nicht eingefügt werden kann.

Frage: Ist es eher die Pflicht des **Frontend** oder **Backend**-Entwicklers, solche Eingaben abzufangen?

Kapitel 6 | Relationenmodell

Frage: Ist es eher die Pflicht des **Frontend** oder **Backend**-Entwicklers, solche Eingaben abzufangen?

Antwort: It depends ... Es gibt Szenarien, bei denen es eher Sinn macht, dass die Eingabe im Frontend bereits abgefangen und korrigiert wird. Unter Umständen ist auch eine doppelte Sicherung notwendig.

Wichtig ist die **Absprache** zwischen Frontend und Backend!!!

Folgendes Szenario zeigt wie schnell (nicht vorhandene) Absprachen zu Problemen führen können:

Eine Mitgliederverwaltung X ermöglicht die Wahl des Geschlechts (m oder w). Die Eingabe wurde bisher sowohl im Front- als auch Backend geprüft. Werte außerhalb von m oder w wurden im Frontend mit einem Fehler quittiert.

Nach neuer Gesetzeslage wurde die Auswahl allerdings erweitert. Der Frontend-Entwickler passt entsprechend seinen Bereich derart an, dass nun auch Werte außerhalb von m oder w valide sind. Der Backend-Entwickler hingegen nimmt keine Veränderungen vor.

Dadurch entstehen (je nach Aufbau) plötzlich Fehler(Meldungen). Der Frontend-Entwickler sucht den Fehler in seinem Bereich (Zeitaufwand -> Kostenaufwand, da er diese Änderung in das System gesetzt hat).

Der Backend-Entwickler ist sich hier keiner Schuld bewusst, da er an der Codebasis keine Änderung vornahm und sich entsprechend nicht um den Fehler kümmert, der im Bugtracker vorliegt. Dieser würde zuerst an den Frontend-Entwickler zugewiesen werden!

Kapitel 6 | Relationenmodell

Das Szenario offenbart einige Baustellen sowohl in der Kommunikation, als auch im Zuständigkeitsbereich und kommen wesentlich häufiger vor, als man zunächst glaubt.

Daher ist es für euer Projekt auch wichtig genau abzugrenzen, welche Person für welchen Bereich zuständig ist!

Die Wahl des DBMS ist abhängig von vielen verschiedenen Faktoren. Darunter:

- Skalierbarkeit
- Ressourcenverbrauch
- Kosten
- MultiUser-Fähigkeit
- Anbindungsmöglichkeiten
- Paradigma
- ...

Auf dem Markt existiert eine Vielzahl verschiedener DBMS. Wir verwenden für unser Lernfeld Sqlite. Sqlite ist keine Server-Client-Engine im klassischen Sinn. Das bietet einige Vorteile:

- Es muss keine Serverinstanz gestartet werden
- Dementsprechend muss auch kein Server installiert und gewartet werden
- Die Sqlite-DB kann als einzelne Datei mit der Applikation mitgeliefert werden
- Dadurch ist ein **Backup/Restore** auch für einen Benutzer leicht zu bewerkstelligen

Sqlite ist zudem kostenlos und Open Source. Es wird keine Lizenz benötigt. Das Kundenprojekt kann daher problemlos mit dieser Lösung umgesetzt werden. Natürlich steht es frei, eine andere DBMS-Lösung für das Kundenprojekt zu verwenden.

Für die kommenden Aufgaben werden wir das Programm DBBrowser verwenden (siehe Eingangsfolien). Damit können wir einfach SQL-Statements absetzen und Manipulationen der Datenbank vornehmen.

Kapitel 8 | SQL

SQL (Structured Query Language) ist die Standardsprache im Bereich der Datenbanken. Wir werden in den kommenden Stunden die Grundlagen der Sprache erarbeiten. Aufgrund der geringen Kontaktzeit werden lediglich die absoluten **Basics** behandelt!

Es sei jedem angeraten, sich selbstständig mit dem Thema zu befassen und kleinere (datenbankgestützte) Projekte zu realisieren. Nur so verfestigen sich die grundlegenden Konzepte!

Das Skript unterteilt SQL in die Bereiche “Datenbank erstellen“ und „Datenmanipulation“. Zuerst benötigen wir eine Datenbank und Tabellen, damit wir Abfragen über den DBBrowser durchführen können.

Die Erstellung der Sqlite Datenbank kann auf 2 Arten erfolgen.

Möglichkeit 1: Erstellung der Datenbank über die grafische Oberfläche des DB-Browsers. Diese Variante ist simple und schnell. Allerdings sind wir dann von der Software abhängig. Verwenden wir eine andere Software/Plugin (z. B. Phpmyadmin), erfordert dies eine neue Einarbeitungszeit.

Möglichkeit 2: Erstellung via Statement.

Die Syntax lautet (je nach DBMS leichte Variationen möglich):

sqlite: sqlite3 DatenbankName.db

MySQL: CREATE DATABASE DatenbankName;

// Hier sollte vorher ein neuer DB-User angelegt werden. Z. B mit (testweise!)
// GRANT ALL PRIVILEGES ON *.* TO 'db_user'@'localhost' IDENTIFIED BY 'password';
// mysql -u db_user -p

Nach der Erstellung der Datenbank können die Tabellen angelegt werden.
Hier gibt es wieder 2 Möglichkeiten. Zusammenklicken im DBBrowser oder via Statement anlegen. Ihr solltet Variante 2 wählen!

Die Syntax lautet (je nach DBMS unterschiedliche Syntax!):

```
CREATE TABLE tabellen_name (  
    spalte1 datentyp,  
    spalte2 datentyp,  
    spalte3 datentyp,  
    ...  
);
```

Aufgrund der großen Anzahl verschiedener Datentypen, verweise ich hier auf:

https://www.w3schools.com/sql/sql_datatypes.asp

Dort können die entsprechenden Datentypen mit Wertebereich nachgesehen werden.

Ein konkretes Beispiel wäre folgendes:

```
CREATE TABLE Member (  
    MemberID int,  
    Vorname varchar(255),  
    Nachname varchar(255),  
    Alter int,  
    Stadt varchar(255),  
);
```

Frage: In diesem Beispiel gibt es einige Verbesserungsmöglichkeiten. Was fehlt im oben genannten Beispiel?

Über **Constraints** können Einschränkungen vorgenommen werden, um zu verhindern, dass invalide Eingaben getätigt werden. Verbessern wir das Beispiel:

```
CREATE TABLE Member (  
    MemberID int PRIMARY KEY,  
    Vorname varchar(255) NOT NULL,  
    Nachname varchar(255) NOT NULL UNIQUE, // hier problematisch!  
    Alter int,  
    Stadt varchar(255),  
);
```

// Auch hier sind durchaus weitere Constraints und Optionen möglich, um die Datenbank vor Fehleinträgen zu schützen.

Wichtig ist hier zu beachten, dass die Syntax je nach verwendeter DBMS unterschiedlich sein kann. Schauen wir uns hierzu folgendes Beispiel an.

Sqlite:

```
CREATE TABLE Member (  
    MemberID int PRIMARY KEY,  
    Vorname varchar(255) NOT NULL,  
    Nachname varchar(255) NOT NULL,  
    Alter int,  
    Stadt varchar(255),  
);
```

MySQL:

```
CREATE TABLE Member (  
    MemberID int,  
    Vorname varchar(255) NOT NULL,  
    Nachname varchar(255) NOT NULL,  
    Alter int,  
    Stadt varchar(255),  
    PRIMARY KEY(MemberID)  
);
```

Kapitel 8 | SQL

Durch diese kleinen Unterschiede kann es (gerade am Anfang) durchaus vorkommen, dass Statements nicht funktionieren. Hier sollte als erstes überprüft werden, ob die angegebene Syntax zum entsprechenden System passt!

Fremdschlüssel werden wie folgt definiert:

```
CREATE TABLE Member (  
    MemberID int PRIMARY KEY,  
    Vorname varchar(255) NOT NULL,  
    Nachname varchar(255) NOT NULL,  
    Alter int,  
    Stadt varchar(255),  
    GroupID int,  
    FOREIGN KEY (GroupID) REFERENCES Group(GroupID)  
);
```

Nachdem die Datenbank inkl. der Tabellen angelegt ist, können wir diese mit Inhalten füllen.

Die Syntax lautet:

INSERT INTO tabellenname (spalte1, spalte2, ...) **VALUES** (wert1, wert2, ...);

Sollte jede Spalte durch einen Eintrag befüllt werden, kann das Statement vereinfacht werden:

INSERT INTO tabellenname **VALUES** (wert1, wert2, ...);

Wichtig ist hier die **Reihenfolge**!

Beispiel:

INSERT INTO Mitarbeiter(vorname,nachname,alter) **VALUES** ('Hans','Herbert',25);

Frage: Hier fehlt scheinbar der Primary Key. Ist das ein Fehler?

Die Tabellen sind jetzt mit Inhalt befüllt. Wollen wir diesen Inhalt aus der Datenbank lesen, müssen wir uns der Sprache SQL bedienen. Vielleicht ist dir aufgefallen, dass die **Keywords** groß geschrieben wurden (**INSERT INTO**). SQL ist **nicht** case-sensitive! Ein **insert into** würde ebenfalls funktionieren. Historisch hat sich die Capslockschreibweise eingebürgert.

Die wichtigsten Keywords innerhalb der Sprache SQL lauten:

SELECT
UPDATE
DELETE
INSERT INTO
CREATE DATABASE
ALTER DATABASE
CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE INDEX
DROP INDEX

Der wohl am häufigsten verwendete Befehl in SQL lautet SELECT. Mit diesem können Datensätze aus einer Tabelle gelesen werden. Nehmen wir folgende Beispieltabelle mit dem Namen „SomeTable“

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50

Möchten wir beispielsweise alle Spielernamen aus der Datenbank auflisten, verwenden wir das Statement: **SELECT** Name **FROM** SomeTable;

Möchten wir hingegen die gesamte Tabelle wiedergeben, können wir Wildcards verwenden. In diesem Fall **SELECT * FROM** SomeTable;

Klasse und Gold mit einem Query:
SELECT Class, GoldAmount **FROM** SomeTable;

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50
...

Gehen wir in unserer Beispieltabelle davon aus, dass diese aus mehreren hundert Einträgen besteht. Vielleicht möchten wir wissen, welche Klassen aktuell gespielt werden. Ein

SELECT Class **FROM** SomeTable;

Wäre ungünstig, da wir Duplikate bekommen werden.

Hier bietet sich das Keyword **DISTINCT** an.

SELECT DISTINCT Class **FROM** SomeTable;

Kapitel 9 | SQL

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50
...

Vielleicht möchten wir lediglich die reichen Spieler auflisten. Also Personen, die mehr als 1000 Goldstücke besitzen. Hierfür bietet sich WHERE an. Die allgemeine Syntax lautet:

SELECT spalte1, spalte2, ...
FROM Tabellename
WHERE Bedingung;

In unserem Fall:

SELECT Name
FROM SomeTable
WHERE GoldAmount > 1000;

Folgende Operatoren sind innerhalb der WHERE Klausel gültig

=
>
<
>=
<=
<> -- not equal!
BETWEEN
LIKE
IN
OR, AND, NOT

Beispiel:

```
SELECT Name  
FROM SomeTable  
WHERE Class IN ('Rogue','Warrior');
```

Beispiel:

```
SELECT Name  
FROM SomeTable  
WHERE Class IN ('Rogue','Warrior');
```

ODER

```
SELECT Name  
FROM SomeTable  
WHERE Class='Rogue' OR Class='Warrior'
```

Häufig möchten wir die Ergebnisse direkt sortieren. Die allgemeine Syntax lautet

```
SELECT spalte1,spalte2, ...  
FROM tabellenname  
ORDER BY spalte1, spalte2, ... ASC/DEC;
```

Beispiel:

```
Select * From SomeTable  
ORDER BY Name ASC;
```

Möchte man einzelne Einträge ändern, verwendet man hierzu **UPDATE**. Hier sollte man jedoch äußerst vorsichtig vorgehen. Die Syntax lautet:

UPDATE Tabellename
SET *spalte1* = wert1, Spalte2 = wert, ...
WHERE Bedingung;

Beispiel:

UPDATE SomeTable
SET Name = 'HolyMinister'
WHERE CharID = 1;

Frage: Was würde passieren, wenn die WHERE Klausel vergessen wird?

Ähnliches gilt für das Löschen (**DELETE**) von Einträgen. Die Syntax lautet:

DELETE FROM Tabellenname **WHERE** Bedingung;

Wird die WHERE Bedingung vergessen, werden ALLE Einträge in der Tabelle gelöscht!

Kapitel 9 | SQL

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50
...

Es wäre interessant zu erfahren, welcher Spieler das meiste Gold besitzt.
Hierfür bietet sich die MIN bzw. MAX Funktion an.

Syntax:

```
SELECT MAX(GoldAmount), Name  
FROM SomeTable;
```

Kapitel 9 | SQL

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50
...

Wie viele Spieler haben wir in unserer Datenbank?

Syntax:

```
SELECT COUNT(CharID)
FROM SomeTable;
```


Kapitel 9 | SQL

CharID	Name	Class	GoldAmount
1	WatchYaBack	Rogue	1000
2	HolyFist	Priest	50
...

Wieviel Gold haben die Spieler im Durchschnitt?

Syntax:

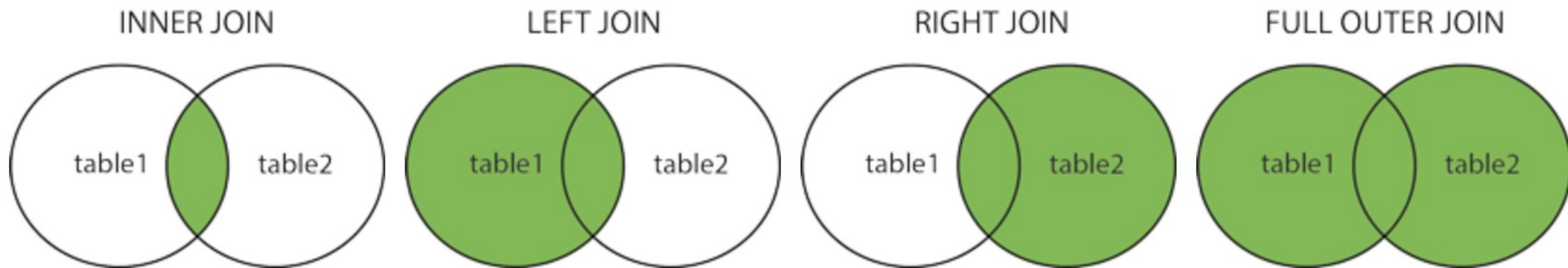
```
SELECT AVG(GoldAmount)  
FROM SomeTable;
```

Nachfolgend erhalten Sie die Tabelle SomeTable. Auf der nächsten Folie sind eine Reihe Fragen abgebildet. Bestimmen Sie das jeweilige Statement zur Frage!

CharID	Name	Class	GoldAmount	PlayedTime	Premium	Banned	Deleted
1	WatchYaBack	Rogue	1000	102	Y	N	N
2	HarzFear	Priest	500	29	N	Y	Y
3	Gnomery	Mage	350	60	N	N	Y
...

- 1: Wie viele Spieler sind auf dem Server?
- 2: Welcher Spieler hat das meiste Gold?
- 3: Nennen Sie die drei reichsten Spieler!
- 4: Welche Spieler haben einen Premiumaccount?
- 5: Welche Spieler haben ihren Account gelöscht?
- 6: Welche Spielernamen beginnen mit einem G?
- 7: Welche Spieler spielen das Spiel seit mindestens 20 und längstens 80 Tagen?
- 8: Geben Sie Frage 3 aufsteigend aus.
- 9: Fügen Sie einen neuen Eintrag mit folgendem Inhalt hinzu
„HansMasterbeard,Warrior,o,o,Y,N,N“
- 10: Gnomery hat gegen die Richtlinien verstoßen. Löschen Sie den Eintrag aus der Tabelle!

In der Praxis ist es häufig notwendig, Informationen aus verschiedenen Tabellen zusammenzuführen. Das Zusammenführen von Tabellen geschieht mithilfe eines **JOIN**. Je nachdem welche Daten genau benötigt werden, unterscheidet man 4 JOIN-Typen:



[5]:https://www.w3schools.com/sql/sql_join.asp

Folgendes Tabellen ist gegeben:

Bestellungen

OrderiD	AccountID	OrderDate	Euro	GoldAmount
1029	322	14.03.2021	10	100
739	2	01.04.2013	10	5

Accounts

AccountiD	AccountName	Country
2	Maple	USA
322	Skycrow	Germany

Beispielabfrage für einen INNER JOIN:

```
SELECT Bestellungen.OrderDate, Accounts.AccountName  
FROM Bestellungen INNER JOIN Accounts ON Bestellungen.AccountID = Accounts.AccountID
```

Das Ergebnis:

OrderDate	AccountName
14.03.2021	Skycrow
01.04.2013	Maple

Für JOINS sollten einige Punkte beachtet werden:

1. Erst überlegen, welcher JOIN-Typ benötigt wird!
2. Nachschlagen, ob besagter JOIN-Typ überhaupt unterstützt wird
3. Bei der JOIN-Abfrage die Syntax Tabellename.Spaltenname beibehalten
4. Eventuell können Schlüsselwörter entfallen (z. B JOIN)

Wird beispielsweise ein FULL OUTER JOIN nicht unterstützt, kann dieser durch die Kombination eines RIGHT + LEFT JOINS erreicht werden. Eventuell müssen Unterabfragen gebildet werden.

Weitere Formen (z. B. **NATURAL JOIN**) spielen in diesem Skript keine Rolle. Wichtig ist es, den **INNER JOIN** zu verstehen und anwenden zu können. Andere JOIN-Typen können bei Bedarf nachgeschlagen werden.

Wichtig ist: Ein JOIN ist temporär! Es wird keine neue Tabelle erstellt und angelegt, sondern dient lediglich der kurzfristigen Abfrage. Möchte man dies, so ist die Verwendung eines **Views** die richtige Wahl.

Vgl: https://www.w3schools.com/sql/sql_join.asp

Ein **View** ist eine *virtuelle* Tabelle in einem Datenbanksystem. Die Inhalte der View spiegeln sich in den real existierenden Tabellen wieder. Durch das Bereitstellen geeigneter Views können wir aufwändige SQL-Abfragen vermeiden. Zusätzlich können Views in Verbindung mit einem Rechtesystem verwendet werden. Außerdem können Spaltennamen geändert bzw. die View so angepasst werden, dass sie für Endbenutzer leichter verwendbar werden.

Neben den oben genannten Vorteilen gibt es noch eine ganze Reihe weiterer Vorteile, die man sich bei Interesse ansehen kann.

Die Syntax zum Erstellen einer View lauten:

```
CREATE VIEW name AS  
SELECT spalte1,spalte2, ...  
FROM tabelle  
WHERE bedingung;
```

Vgl: https://www.w3schools.com/sql/sql_view.asp

In unserem Rollenspielbeispiel könnte eine View {Payment} erstellt werden, die es einem Supportmitarbeiter einfacher erlaubt, Details über den Zahlungsstatus zu erfahren, ohne Kenntnisse über die dahinterliegenden Schichten zu haben.

In der Praxis: Innerhalb einer GUI sollten lediglich Views angezeigt werden. Die beschriebenen Vorteile kommen hier besonders zum tragen.

Komplexer wird es, wenn durch die View die dahinterliegenden Tabellen manipuliert werden sollen. Hier muss unterschieden werden ob es sich um eine einzelne Tabelle oder einen Verbund aus Tabellen handelt (die vorher mit einer JOIN-Operation verbunden wurden).

Sicherheit spielt in Datenbanken eine herausragende Rolle. Zuerst muss geklärt werden, was unter dem Begriff „Sicherheit“ zu verstehen ist.

Welche Punkte fallen euch ein, wenn Ihr den Begriff Sicherheit hört?

Sicherheit ist – je nach Definition – in verschiedene Kategorien unterteilbar. Zum einen sollen Daten innerhalb der Datenbank nicht verloren gehen (Ausfallsicherheit). Zum anderen möchten wir Daten vor Manipulation und Zugriff von unbefugten schützen.

Im Unterricht selbst können wir lediglich einige, wenige Punkte exemplarisch anschneiden. Es ist jedem Leser/in angeraten, sich selbstständig - über den Unterricht hinaus – mit dem Thema zu beschäftigen!

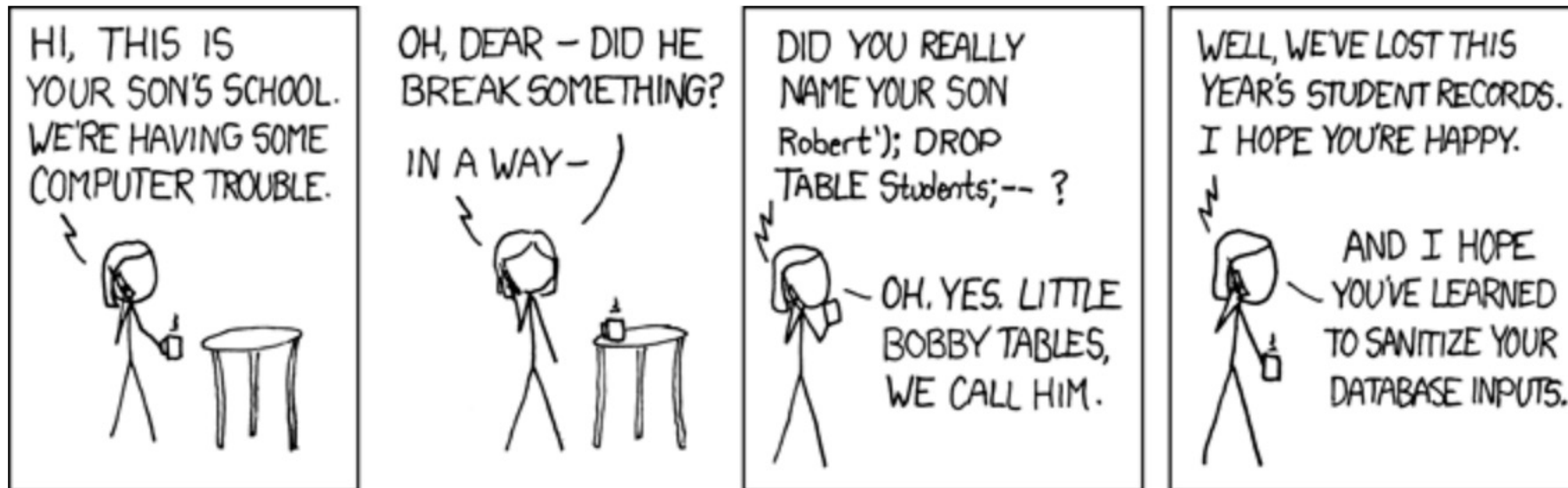
Sicherheit ist – je nach Definition – in verschiedene Kategorien unterteilbar. Zum einen sollen Daten innerhalb der Datenbank nicht verloren gehen (Ausfallsicherheit). Zum anderen möchten wir Daten vor Manipulation und Zugriff von unbefugten schützen.

Hinweis: Maßnahmen zur Ausfallsicherheit werden an anderer Stelle detailliert betrachtet und beleuchtet. Diese spielen daher hier keine Rolle

Im Unterricht selbst können wir lediglich einige, wenige Punkte exemplarisch anschneiden. Es ist jedem Leser/in angeraten, sich selbstständig - über den Unterricht hinaus – mit dem Thema zu beschäftigen!

Kapitel 11 | Sicherheit

Lese dir nachfolgenden Comic durch und gebe ihn mit deinen eigenen Worten wieder. Welche Aussage verfolgt der Autor mit dem Comic?



[6]: <https://xkcd.com/327/>

Aufgabe 1: Erstelle eine Definition zur Angriffstechnik „SQL-Injection“

Aufgabe 2: Erstelle mindestens 2 Beispiele zur SQL-Injection.

Aufgabe 3: Wie kann ein System gegen eine SQL-Injection gehärtet werden? Erstelle hierfür ein geeignetes Negativ- & Positivbeispiel.

Aufgabe 4: Sensible Daten (z. B. Passwörter) sollten immer verschlüsselt in einer Datenbank vorliegen. Hierfür stehen verschiedene Möglichkeiten zur Verfügung:

4a: Beschreibe (technisch) eine Möglichkeit, wie ein Passwort verschlüsselt in eine Datenbank abgelegt UND wieder gelesen werden kann.

4b: Was versteht man unter einer Hash- bzw Einwegfunktion. Wie funktioniert diese?

4c: Beschreibe, wofür man einen „Salt“ in diesem Zusammenhang verwendet. Implementiere ein Beispiel in einer Programmiersprache/Stack deiner Wahl (Klartextspeicherung vs. encoded).

Aufgabe 5: Wir gehen von folgendem Szenario aus. Du administrierst einen MySQL-Datenbank, welche auf einem Linux-Server (Debian) läuft. Du möchtest ein Backup der Datenbank (dump) erstellen. Hierfür gelten folgende Anforderungen:

- 5a: Der Dump beinhaltet das Tagesdatum und die Uhrzeit
- 5b: Der Dump ist zusätzlich verschlüsselt
- 5c: Der Dump ist verschlüsselt und komprimiert
- 5d: Jeden Tag um 01:00 nachts wird der Dump automatisch erstellt und auf den Remoteserver „NAS_1_QS1“ gesichert.
- 5e: Es werden maximal 7 Dumps (Mo-So) vorgehalten. Danach wird der älteste Dump gelöscht.

*Hinweis zur Bearbeitung: Aufgabe 5 a-c sollte ein Shell/Bashskript erstellt werden.
Für Aufgabe 5d den Begriff „cronjob“ googeln, lesen, verstehen und verwenden.*

Zum ausprobieren kann kostenfrei Debian, Ubuntu usw. auf einer virtuellen Maschine installiert und getestet werden. Mit „sudo apt-get install mysql-server“ kann ein MySQL-Server installiert werden. Der Rest kann via Google herausgefunden werden 😊

Kapitel 12 | Trigger

FOLGT....

Eine **Transaktion** ist eine Folge von Aktionen, die in einer oder in mehreren Tabellen ausgeführt.

Ein **Commit** bestätigt/speichert die Änderungen der Transaktion.

Ein **Rollback** hingegen verwirft die Änderungen, die von einer Transaktion(en) vorgenommen wurden und stellt den Zustand zum Zeitpunkt Tx wieder her.

Damit eine Datenbank nach der Ausführung einer Transaktion weiterhin konsistent und verlässlich ist, sollten einige Regeln gelten ...

Welche Regeln könnten dies sein?

Das zweite wichtige Akronym in diesem Skript heißt **ACID**:

A (Atomicity): Das bedeutet, dass eine Transaktion wird entweder vollständig (also inkl. aller Teilaktionen) oder gar nicht ausgeführt. Wurden bereits Teile der Transaktion ausgeführt und es erfolgt ein Abbruch, dann wird in interner Rollback durchgeführt und die Änderungen verworfen.

C (Consistency): Nach der Ausführung einer Transaktion muss die Datenbank weiterhin konsistent sein. Das heißt, es dürften durch die Transaktion keine Anomalien entstanden sein.

I (Isolation): Werden mehrere Transaktionen zeitgleich ausgeführt, so dürfen sich diese nicht gegenseitig beeinflussen

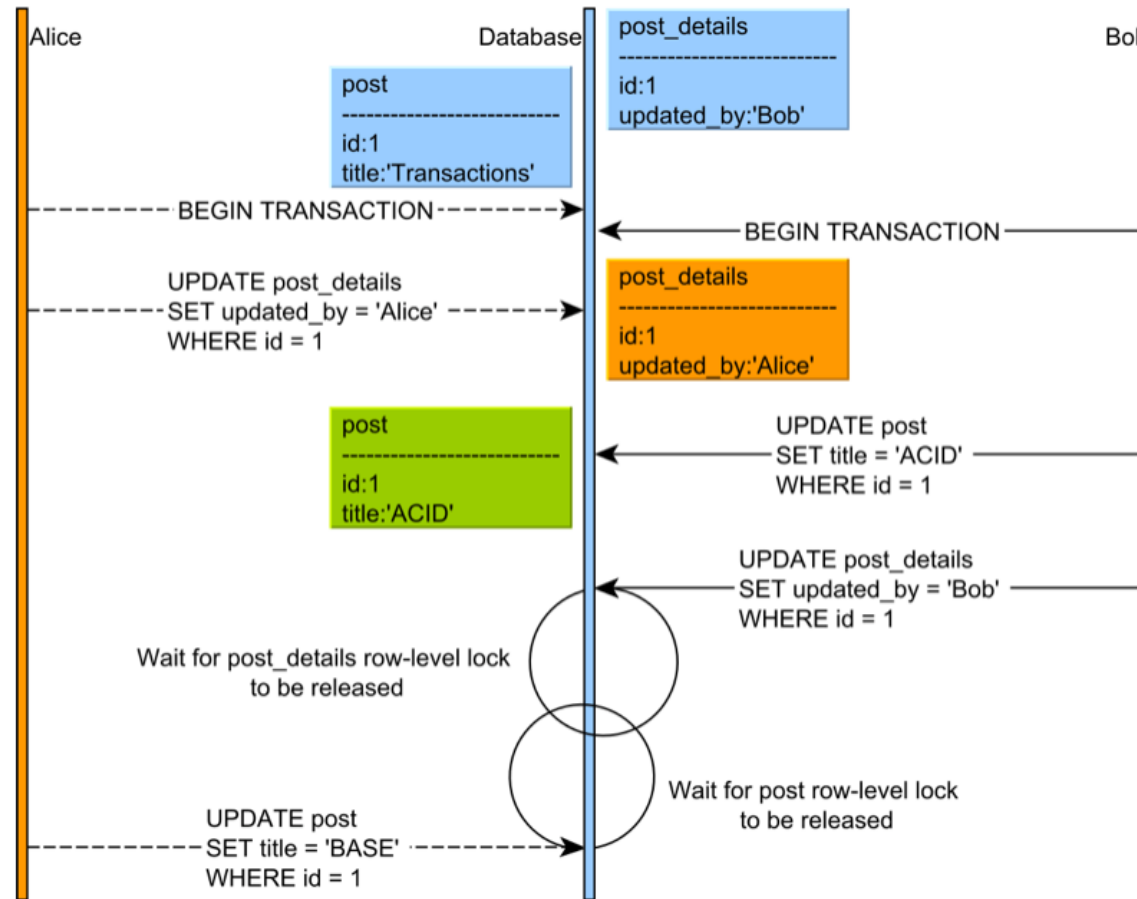
D (Durability): Das Ergebnis einer Transaktion muss dauerhaft, nicht flüchtig gespeichert werden.

Damit Transaktionen im Mehrbenutzerbetrieb störungsfrei durchlaufen können, benötigen wir sogenannte **Locks** (Sperrungen). Eine Sperre kann auf verschiedenen Ebenen gesetzt werden. Beispielsweise kann ein Row-Lock (Zeilensperre) von Benutzer A gesetzt werden, damit Benutzer B während der Transaktionsfolge nicht darauf zugreifen kann und es zu Anomalien kommt.

Neben den Ebenen der Sperre können grundsätzlich zwei Typen unterschieden werden. Read- und Writelocks. Setzt Benutzer A einen Readlock auf einer Ebene, möchte dieser davon lesen. Das hindert zunächst einen weiteren Benutzer B nicht daran, ebenfalls von dieser Ebene lesen zu wollen. Wird hingegen ein Writelock gesetzt, wird sowohl ein Lesezugriff als auch Schreibzugriff darauf verhindert.

Nun kann es zu ungünstigen Konstellationen kommen, in der ein so genannte **Deadlock** entsteht. Schauen wir uns das anhand eines Beispiels genauer an:

Kapitel 13 | Transaktionen



[6]: <https://vladmihalcea.com/database-deadlock/>

Wie ein Deadlock begegnet wird, ist höchst unterschiedlich und hängt vom jeweiligen DBMS ab.

Beispielhaft wird in MySQL die Transaktion terminiert, welcher weniger Zellen betrifft (Victim).

Eine andere Strategie ist es, dass ein Prozess zufallsbasiert einer der beiden Transaktionen A und B auswählt, diesen terminiert und einen entsprechenden Rollback durchführt.

Bisher war unser Ziel stets, ein SQL-Query zu formulieren, welches ein spezifisches Problem löst. Wir haben uns keine Gedanken darüber gemacht, ob unsere Lösung effizient ist. Ineffiziente Lösungen führen in größeren Datenbanken zu Verzögerungen, die wir vermeiden möchten.

Schauen wir uns hierfür folgendes Beispiel an:

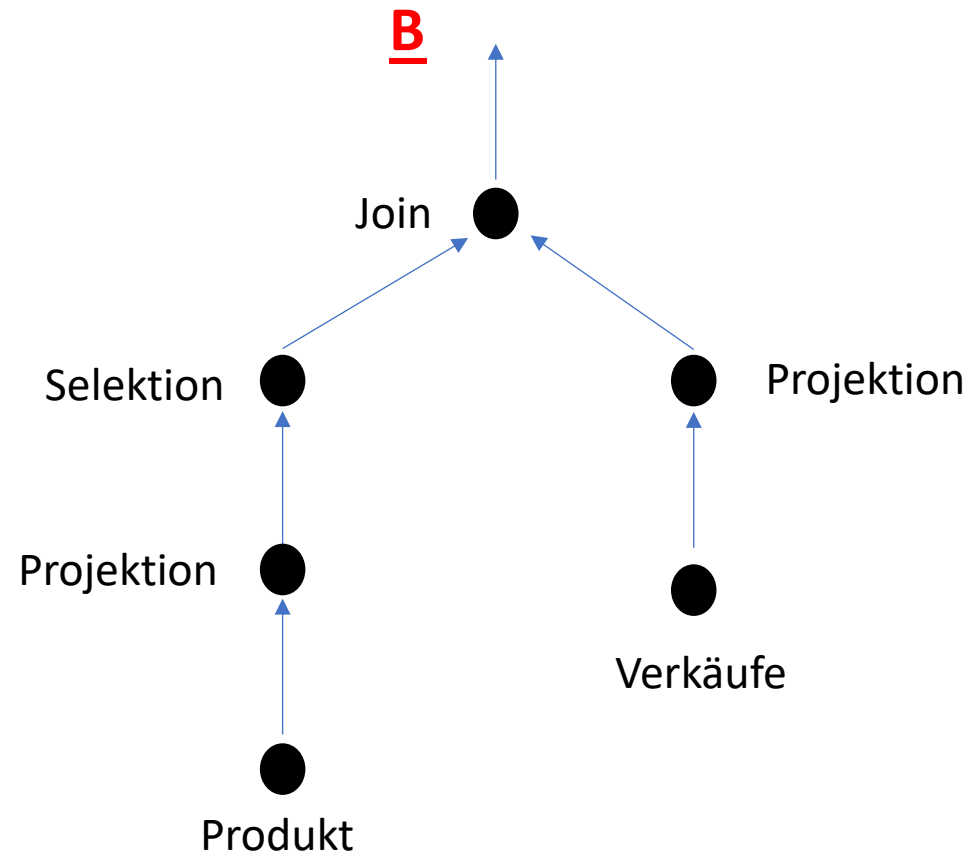
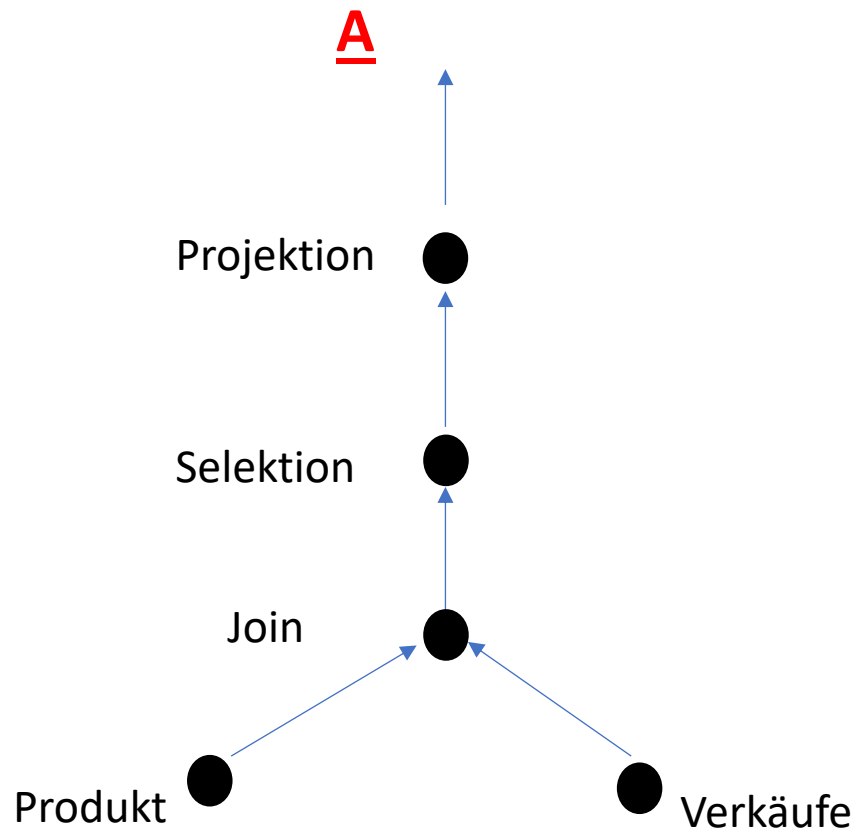
Wir möchten ein SQL-Query schreiben, welches folgende Bedingungen erfüllt.

1. Es fasst die Tabellen Produkte und Verkäufe zusammen
2. Sucht Produkte, deren Preis pro Einheit über 100€ liegen
3. Extrahiert die Spalten mit Daten und Produktnamen

Die obige Aufgaben könnten wir wie folge lösen ...

Kapitel 14 | Optimierung

Betrachte beide Lösungen A und B. Welche der beiden Lösungen ist effizienter?



Aus diesem kleinen Beispiel lassen sich bereits 3 Regeln der Optimierung ableiten:

1. Führe i.d.R. erst eine Selektion aus, um die Anzahl der Zeilen zu reduzieren.
2. Führe i.d.R. erst eine Projektion aus, um die Anzahl der Spalten zu reduzieren.
3. Führe erst dann ein Join aus.

Solltest du Probleme mit einer bereits optimierten Abfrage haben, kann man wie folgt vorgehen:

Schritt 1: Erstellung des nicht optimierten SQL-Query

Schritt 2: Query visuell darstellen (siehe vorherige Folie)

Schritt 3: Anhand der Visualisierung den Query entsprechend grafisch umbauen

Schritt 4: Optimierter Query in Code packen.

Der Hintergrund: Der Schützenverein „der glühende Colt“ arbeitet aktuell mit einer Excel 97 Version, in der die Mitglieder der verschiedenen Abteilungen gepflegt werden. Der neu gewählte Vorstand möchte diesen Umstand allerdings ändern, da die Pflege äußerst aufwendig und fehleranfällig ist.

Harry Müller (Vorstandsmitglied) stellt sich eine moderne Lösung vor, in der er die Mitglieder per „Knopfdruck“ anlegen, ändern und löschen kann. Im Gespräch konntet ihr folgende Notizen anfertigen:

”

- Ich muss sehen können, ob ein Mitglied Geburtstag hat, damit ich ein kleines Präsent überreichen kann
- Innerhalb des Vereins existiert eine Gebührenstruktur. Mitglieder in der Bogenabteilung zahlen 8€/Monat. In der Luftdruckabteilung 10€/Monat und in der Feuerwaffenabteilung 15€/Monat. Ist eine Person Mitglied mehrerer Abteilungen kostet es insgesamt immer 20€/Monat.
- Um eine Waffe zu kaufen benötigen die Mitglieder einen Nachweis über ,regelmäßige‘ Teilnahme, die ich ausstellen können muss. Also muss ich eintragen können, wann die Person zum Training erschienen ist. Das System muss anzeigen ob die Voraussetzungen zum Erwerb erfüllt sind. Die Erwerbsvoraussetzungen laut Gesetz lauten wie folgt:
A: Entweder die Person ist einmal im Monat anwesend (12x im Jahr) ODER
B: Die Person ist 18x im Jahr anwesend

“

In dieser „User-Story“ sind einige Vereinfachungen enthalten (aufgrund unserer geringen Kontaktzeit im Unterricht):

Ein Kunde denkt i.d.R. nicht an jeden Sonderfall. In diesem Beispiel würde entweder der Punkt mit dem Geburtstag oder dem Nachweis überhaupt nicht erwähnt werden, da es für den Kunden selbstverständlich ist.

Zu welchen Problemen wird das aus Sicht des Entwicklers führen?

Wie kann man solche Probleme vermeiden bzw. zumindest verringern?

Welcher Punkt fällt in diesem Projekt besonders auf?

Im Lernfeld 8 können wir nicht in die Tiefen des Projektmanagements einsteigen oder diese behandeln!

Einige Gedanken/Schritte können wir allerdings auch ohne dieses Vorwissen ausführen:

1. Aus dem Gespräch die **Funktionalitäten** herausfiltern und notieren!
2. Die beiden Endbenutzer scheinen älter und unerfahren zu sein. Die Rahmenbedingungen haben Auswahl auf den Ablauf meines Prozesses! (näheres in Projektmanagement)
3. Welche **Technologien** setze ich ein, um das Problem zu lösen? Nur weil ich Java in der Schule gelernt habe, heißt das nicht zwingend, dass man Java für jedes Projekt einsetzen sollte. Nicht meine erlernte Programmiersprache spielt eine Rolle bei der Lösung, sondern die Erfordernisse des Projektes! In diesem Projekt beispielsweise macht **prototyping** Sinn. Einen Prototyp mit JavaFX zu erstellen ist möglich, aber sicher nicht zielführend (unabhängig davon, wie gut ihr euch mit Java**FX** auskennt).

Vielleicht wäre die Verwendung von HTML+Javascript/PHP+SQLite/MySQL sinnvoller?
Vielleicht könnte man ein Framework wie electron verwenden, um die Software „ausführbar“ zu machen. Vielleicht sollten wir Python verwenden, um schnelle Ergebnisse erzielen zu können.
Vielleicht doch C# und WPF? Vielleicht möchte der Schützenverein eine fancy IOS-App?

Die Wahl der Technologien sollte wohlüberlegt sein. Es vereinfacht bzw. verkompliziert die Arbeit später enorm.

Beliebter Fehler Nummer 1: Dem Kunden ist es (i.d.R) vollkommen egal, was ihr verwendet. Dieser möchte sein Problem gelöst bekommen. Diskussionen wie „... aber Programmiersprache XY ist besser“ ist hier völlig fehl am Platz!

Beliebter Fehler Nummer 2: Over-Engineering!

Unser Beispielperson ist älter und mit der Verwendung von Excel 97 vertraut. Vermutlich trägt er keine Apple-Watch und AirPods und vermutlich benötigt er keine Möglichkeit, über seine Apple-Watch auf die Datenbank des Schützenvereins zugreifen zu können!

Dieses Extrembeispiel verdeutlicht eine Problematik, die häufig auftritt. Entwickler möchten das beste Ergebnis abliefern. Grundsätzlich ist das eine lobliche Einstellung. Allerdings spielen in der realen Welt Zeit und Geld eine große Rolle. Wenn der Kunde mit einer Lösung glücklich ist, die in einer Woche entwickelt werden kann, macht es keinen Sinn dafür 4 zu verschwenden. Es macht den Kunden nicht glücklicher. Ganz im Gegenteil. Der Umsatzausfall ist enorm! Außerdem gilt die Regel: Umso komplexer die Software, umso höher der Wartungsaufwand!

Perfektionismus hat hier keinen Platz! Versteht das nicht falsch. Eine Software muss gut durchdacht und geplant werden. Unnötige Sonderfeatures, die vllt. „cool und fancy“ sind aber der Kunde nicht benötigt, sollten auch nicht implementiert werden! **Keep-it-simple!**

Kapitel X1 | Kundenauftrag

Aufgaben

Folgende Aufgaben sind bis zur gegebenen Deadline auszuführen:

1. Entwicklung einer lauffähigen Lösung anhand der genannten Punkte!
2. Erstellung einer Dokumentation, die die folgende Punkte beinhaltet:
 - 2.1 Ein Kapitel zur Vorgehensweise, Teamgröße und anderer Kenndaten
 - 2.2. ER-Modell
 - 2.3. Datenbankstruktur
 - 2.4. Verwendete weitere Software/Technologien (evtl. Quellcode/Kommentare)
 - 2.5. Aufgewendete Zeit in Stunden, gruppiert nach einzelnen Aufgabenfelder (z. B. „2h – Erstellung ER-Modell“. Endsumme mit 45€/Std multiplizieren. Damit erhalten wir interessante Daten für eine Abschlussbesprechung. Daher bitte gewissenhaft und ehrlich die Zeit **tracken**!
3. Auf Basis der oben genannten Daten eine kleine Rechnung erstellen. Auftragnehmer ist die jeweilige Stammgruppe.
4. Die Dokumentationen + Beispielrechnung zusätzlich in englischer Sprache
6. Erstellung einer kurzen Präsentation eurer Lösung für unser Lernfeld

Kapitel X1 | Kundenauftrag

Aufgaben

Im Skript wird sehr deutlich vor Over-Engineering gewarnt. Allerdings dient unser Lernfeld der Ausbildung. Sollte euch der ein oder andere Punkt einfallen, der einen Mehrwert für das Projekt darstellt, könnt ihr diesen gerne einbauen. Auch hier gilt: Zeitaufwand notieren!

Ihr könnt auch diese – nicht explizit genannten – Funktionen/Features in der Beispielrechnung separat ausweisen. Das würde eindrucksvoll aufzeigen, wie hoch der Extraaufwand (zeitlich und monetär) wäre.

Viel Erfolg und Spaß mit dem Projekt 😊

Der Markt der Computerspiele wächst immer rasanter. Plattformen wie Steam, Epic Games Store usw. bieten die Möglichkeit, Spiele zu bewerten, zu favorisieren und vorzumerken. Leider sind diese Plattformen aufgebläht mit Funktionen, die wir nicht benötigen. Daher bauen wir in diesem Projekt eine eigene Datenbankanwendung zur Verwaltung von Computerspielen, die auf die wesentlichen Funktionen reduziert werden.

Das Projekt besteht aus einigen Pflichtfunktionalitäten und optionalen Funktionalitäten. Pflichtfunktionen müssen zwingend im Programm vorhanden sein. Die Art und Weise der Implementierung bleibt der jeweiligen Gruppe überlassen!

Folgende Rahmenbedingungen gelten:

- Die Wahl der Programmiersprach(en)/Stacks ist frei wählbar
- Ein **Ablaufplan** mit **Meilensteinen** muss erstellt werden
- Eine Kundendokumentation & Rechnung ist **nicht** Bestandteil des Projekts

Kapitel X2 | Privatprojekt

Aufgaben

Unser Programm besitzt eine GUI, welche folgende Funktionalitäten beinhaltet:

- Das Anlegen, Ändern & Löschen von einzelnen Spielen
- Das Anlegen, Ändern & Löschen von Rubriken (Adventure, MMO's usw.)
- Jedes Spiel besitzt eine kurze Beschreibung und Bild. Daher müssen Bilder hinzugefügt werden können (optional: automatisches Skalieren!)
- Spiele können als Favorit markiert und zudem bewertet werden
- Spiele können nach Rubrik, Name, Favorit(en) und Bewertungen sortiert werden
- (optional: Eine eigene Rubrik „gelöschte Spiele“, die wiederhergestellt werden können)
- (optional: Das automatisierte Einfügen der Titelbilder)
- (optional: Der Import kompletter Inhalte via .csv)
- (optional: Durchschnittliche Bewertungen von Metascore oder anderen, evtl. automatisiert)
- (optional: Ein Kaufpreis wird angezeigt (z. B. Amazon, Steam usw.)
- (optional: Jedes Spiel kann per Druckfunktion ausgedruckt werden. Es beinhaltet ebenfalls ein Bild, Preis, Beschreibung & Bewertung. Idee: Als potenzieller Wunschzettel)

Kapitel X2 | Privatprojekt

Aufgaben

Die Grundfunktionalitäten sollten nach dem ersten Projekt (X₁) einfach und schnell zu implementieren sein. Schwieriger bzw. neu ist das Arbeiten mit verschiedenen API's.

Es sollte daher unbedingt vorher geklärt werden, welche Technologien (Sprachen, DBMs usw.) hier Sinn machen bzw. die Implementierung vereinfachen.

In diesem Projekt entfällt sowohl die Dokumentation in DE/En, als auch die Kundendokumentation und die Rechnung. Die gesamte Zeit wird demnach für die Entwicklung der Software verwendet.

Viel Erfolg und Spaß mit dem Projekt 😊

Abschluss

Herzlichen Glückwunsch! Du bist am Ende des Skripts angelangt. Solltest du bis hierher alle Aufgaben gelöst und die Inhalte gelernt haben, bist du bestens für die IHK-Prüfung in diesem Bereich vorbereitet.

Das Thema Datenbanken bietet allerdings noch viel mehr als in diesem Skript gezeigt. Beispielsweise wurde im Skript das Thema verteilte Datenbanken nicht angesprochen.

Daher möchte ich dich ermuntern, dich noch tiefer in diesem Gebiet zu informieren, damit du für alle Eventualitäten gewappnet bist.

Kritik/Lob zum Skript kannst du mir gerne via Mail (joshua.schumacher@bbs1-mainz.de) mitteilen. 😊

Changelog:

- Version 0.11: Kapitel 4 hinzugefügt
- Version 0.12: Kleinere Fehler korrigiert
- Version 0.13: Kapitel 5 hinzugefügt; Changelog hinzugefügt; Kap 6 Thema geändert; Kap 4 Übungsaufgaben erweitert; Verlinkungen hinzugefügt; zusätzliche Literatur hinzugefügt; Lizenzbestimmungen hinzugefügt.
- Version 0.14: Kapitel 6 hinzugefügt
- Version 0.15: Kapitel 7 hinzugefügt, Kapitel 8 hinzugefügt. Kapitel 8 in 8 & 9 aufgeteilt, Inhaltsangabe erweitert
- Version 0.16: Kapitel 8/9 erstellt
- Version 0.17: Kapitel 9 ausgebaut
- Version 0.18: Kapitel 10 Views hinzugefügt; Inhaltsangabe erweitert.
- Version 0.19: Kapitel 11/12 vertauscht und hinzugefügt.
- Version 0.20: Projekt X2 hinzugefügt, Inhaltsangabe angepasst.
- Version 0.21: Kapitel Sicherheit hinzugefügt
- Version 1.00: Github Repo hinzugefügt. Inhaltangabe angepasst. Fehler korrigiert.
- Version 1.01: Diverse Fehler korrigiert und Hinweise in Kap. Normalformen hinzugefügt.
- Version 2.0: Kapitel 13 & 14 hinzugefügt. Projekt X1 leicht vereinfacht. Kleinere Anpassungen verschiedener Kapitel
- Version 2.1: Kundenprojekt vereinfacht und zusammengefasst.

Quellen:



Icon made by ultimatearm

<https://www.flaticon.com/authors/ultimatearm>

[2] One does not ...

<https://spontan-wild-und-kuchen.de/archive/2485>

[3] Mengenlehre

[https://de.wikipedia.org/wiki/Menge_\(Mathematik\)#Vereinigung_\(Vereinigungsmenge\)](https://de.wikipedia.org/wiki/Menge_(Mathematik)#Vereinigung_(Vereinigungsmenge))



Icon made by pixelperfect

<https://www.flaticon.com/authors/pixelperfect>

[5] JOIN-Typen

https://www.w3schools.com/sql/sql_join.asp

[6] Deadlock

<https://vladmihalcea.com/database-deadlock/>

Nützliche Literatur:

*Ich übernehme keine Verantwortung bezüglich der Inhalte der folgenden Links!
Sie dienen lediglich der Einarbeitung in das Thema. Sollten Links nicht mehr funktionieren
oder auf den folgenden Seiten „problematische“ Inhalte publiziert werden, bitte ich um eine Rückmeldung.
Der betreffende Link wird dann entfernt!*

Datenbankkurs von Tino Hempel

<https://www.tinohempel.de/info/info/datenbank/index.htm>

Datenbanken verstehen

<https://datenbanken-verstehen.de/>

Informatik Manga zum Thema Datenbanken (sehr gut!)

https://www.amazon.de/Informatik-Manga-Datenbanken-German-Mana-Takahashi/dp/3834809837/ref=sr_1_1?mk_de_DE=%C3%85M%C3%85%C5%BD%C3%95%C3%91&dchild=1&keywords=Datenbank+manga&qid=1609164358&sr=8-1

Nützliche Literatur:

Interaktive Apps für SQL & Er-Modelle:

<https://sql-island.informatik.uni-kl.de>

<https://www.monst-er.de>