Einführung

in

Datenbank-Systeme

und

SQL

Die Darstellungen stellen zu einem großen Teil eine Bearbeitung eines entsprechenden Skriptes von "Jürgen Burkert et al, Hessisches Institut für Bildungsplanung und Schulentwicklung (HIBS), Wiesbaden" dar.

Bearbeitung: Dieter Lindenberg Version 2019.13

Inhaltsverzeichnis

3-Ebenen-Architektur von Datenbanksystemen 4

Entities und Beziehungen 9

Das Entity-Relationship-Modell 12

- Komplexität 19
- Datenbank Sportverein 32

Das relationale Datenbankmodell 35

- Mathematische Grundlagen des relationalen Datenbankmodells 35
- Anwendung relationaler Operatoren 45

Übertragung des ER-Modells in das relationale Modell 57

- Optimierungen 60
- Aufgaben zur Umsetzung in das relationale Modell 70
- Normalisierung 72
  - Aufgaben zur Normalisierung 81

SQL (Structured Query Language) 89

- Die Entwicklungsumgebung *MySQL* 92
- Umwandlung einer Excel-Datei in eine Datenbank 107
- Abfragen mit *SELECT* 109
- Abfragen mit Vergleichs- und logischen Operatoren 113
- Die Aggregat-Funktionen *COUNT, MAX, MIN, SUM, AVG* 114
- Gruppierung mit *GROUP BY* 116
- Kombination mehrerer Tabellen mit *JOIN* 122
- Unterabfragen 125
- Aktualisierungen 127
  - Erzeugen von Tabellen mit *CREATE* 127
  - Einfügen von Datensätzen mit *INSERT* 130
  - Änderung von Datensätzen mit *UPDATE* 131
  - Löschen von Daten 133
- Die Datenbank „World“ 134
- Die Datenbank „Sportverein“ 138

Rechtliche Aspekte 163

- Datenschutzregelungen 166
- Datenschutz in der Schule 167

In einer Schule werden Daten über Schüler an verschiedenen Stellen benötigt:
Im Sekretariat benötigt man insbesondere die Adressen der Schüler (bzw. der Eltern), die Lehrer müssen die Noten der Schüler eingeben können, und die Schülerbücherei braucht insbesondere die Namen und Klassenzugehörigkeit der Schüler.
Eine (schlechte) Möglichkeit wäre, dass jede Institution (Sekretariat, Lehrer, Bücherei) eine eigene (oder eine Kopie der) Schülerdatei besäße. Die Probleme, die hierbei auftreten, sind ersichtlich:

- Datenredundanz, d.h. dieselben Daten werden mehrfach gespeichert.
- Gefahr der Dateninkonsistenz, d.h. bei Änderungen können wegen der mehrfach gespeicherten Daten Widersprüche auftreten. Verlässt ein Schüler die Schule, so wird er mit dem Programm im Sekretariat gelöscht, aber mit einem anderen immer noch in Kurse eingeteilt (es soll sogar schon vorgekommen sein, dass solche fiktiven Schüler auch Noten erhalten haben).
- Unflexibel gegenüber Änderungen, kurz gesagt: Änderungen der Datenstruktur verlangen Änderungen der Programme; zusätzliche Aufgaben verlangen neue Datenorganisation, da diese an der alten Zielsetzung ausgerichtet waren. Abfragen über mehrere Datenbestände hinweg lassen sich in diesem System kaum realisieren, (z.B. eine Abfrage über die Daten der Bibliothek und der Kurseinteilung: „Lesen Schüler der Deutsch-Leistungskurse mehr klassische Literatur?“.
- Datenschutzprobleme und Datensicherheit, Benutzer bzw. Programme besitzen vollen Zugriff auf die dem Programm zugeordnete Daten; so könnte z.B. das Bibliotheksprogramm aufgrund einer Fehlfunktion sämtliche Ausleiher, d.h. Schülerdaten, löschen.
- Keine Einhaltung von Standards, d.h. Datenbestände werden an unterschiedlichen Stellen in unterschiedlichen Formaten abgespeichert und sind nicht ohne weiteres austauschbar; z.B. können die Schülerdaten im Sekretariat in einem Format vorliegen, das vom Bibliotheksprogramm nicht verstanden wird, so dass diese Daten dort extra eingegeben werden müssen.

All diese Probleme werden von einer modernen Datenbank gelöst.

Ein Datenbank-Management-System (DBMS) verwaltet zentral alle Daten z.B. einer Schule.

Das DBMS stellt Benutzern die Daten in einer für sie brauchbaren Form zur Verfügung. Ändert sich eine Anwendung, dann muss nicht mehr die Datenorganisation geändert werden, sondern lediglich eine neue „externe Sicht" (View) auf die Daten geschaffen werden.

Auch Datensicherheit und Datenschutz können leichter gewährleistet werden. So erhält ein Benutzer durch seine spezifische Sicht nur Zugriff auf Daten, für die er eine Zugriffsberechtigung besitzt. Das DBMS kann durch Kontrollen beim Eingeben, Löschen und Ändern von Daten Fehler vermeiden helfen. So ist es z.B. nicht möglich, einem nicht existierenden Schüler Kurse zuzuweisen oder Kurse zu belegen, deren Kursnummer in der Datenbank nicht vorkommt. Man spricht hier von der Erhaltung der Datenintegrität.

3-Ebenen-Architektur von Datenbanksystemen

Aus ganz unterschiedlichen Betrachtungs- und Problembereichen ist folgende Unterteilung einer Datenbankarchitektur erwachsen. Man unterscheidet sinnvoll drei Ebenen:

Externe Sicht:
So stellt sich die Datenbank dem Benutzer dar.
Interne Strukturen der DB sind für den Anwender uninteressant. Er sollte die DB als Informationsspeicher bzw. als Informationslieferant wahrnehmen, die ihm genau das präsentiert, was er für sein jeweiliges Tätigkeitsfeld benötigt. So kann etwa ein Fachlehrer eines Physikkurses einen ganz anderen Ausschnitt aus der Schüler-Gesamtdatenbank zu sehen bekommen als z.B. das Sekretariat. Dies sind die entsprechenden unterschiedlichen, externen Sichten auf die Datenbank.

Logische/konzeptuelle Sicht:
Hier sind alle logischen Zusammenhänge und Abhängigkeiten aller Daten untereinander beschrieben. Die Abstraktion, die von der realen Welt zu dieser logischen Gesamtsicht führt, bezeichnet man auch als *konzeptuelles Modell*.

Das sog. *relationale Datenmodell* ist das seit vielen Jahren am weitesten verbreitete konzeptuelle Modell. Es basiert rein auf Tabellen (= Relationen) und deren Verknüpfungen. Ein Ausschnitt der Welt (das Problem) wird angemessen modelliert *(Entity-Relationship-Modell)*. Dieser Prozess der Modellbildung,

sprich Datensammlung, Aufteilen in Tabellen, Bestimmung der entsprechenden Datentypen und die Verknüpfung der Tabellen bildet zusammen das sog. *relationale Datenmodell*, die konzeptuelle Sicht der Datenbank.

Interne Sicht:
Hier wird die Realisierung der Daten auf einer Computeranlage beschrieben. Es geht um Fragen wie: Welche Daten werden zu Einheiten (Datensätzen) auf einem Massenspeicher zusammengefasst und wie wird schnell darauf zugegriffen (Suchbaum, Array oder Hashtabelle usw.)?

Im Idealfall sind die drei Ebenen völlig unabhängig voneinander. Das bedeutet z.B., dass eine andere interne Organisation der Daten nichts am konzeptuellen Modell ändern sollte.

Die Verbindungen zwischen den drei Ebenen werden durch Transformationen geschaffen, die natürlich bei Änderungen auf einer Ebene anders gestaltet werden müssten.

Die Benutzer der DB haben nur auf der externen Ebene Zugang zur Datenbank, während es eine Instanz (= eine oder mehrere Personen) gibt, die ein konzeptuelles Modell des Unternehmens erstellt (hat). Eine weitere Instanz (= eine oder mehrere Personen) kümmert sich um die interne Organisation der Daten.

Auf das interne Modell soll nicht weiter eingegangen werden, obwohl natürlich entsprechende Fragen in der Informatik ihre Bedeutung haben. Die interne Sicht wird vom DBMS weitgehend verborgen.

In einem Unternehmen (z.B. in einer Schule) muss eine Datenbank sinnvollerweise in einem Computernetzwerk ablaufen, d.h. das DBMS muss im Regelfall mehrere von außen angeforderte Eingaben oder Abfragen quasi gleichzeitig bearbeiten können.
Dies bedeutet, dass es dem einzelnen Benutzer so erscheint, als würde er allein mit dem System arbeiten, obwohl ganz unterschiedliche Benutzergruppen (oft auch „gleichzeitig“) auf das System zugreifen.

Ein sehr ähnliches Problem läuft auf jedem üblichen Computerprozessor ab: hier greifen auch quasi „gleichzeitig“ mehrere Programme auf den Prozessor zu.

Dabei müssen spezielle Sicherheitsmaßnahmen beachtet werden. So könnte z.B. der Versuch des gleichzeitigen Eintragens der zusätzlichen Fehlstundenanzahl für ein- und denselben Schüler von zwei unterschiedlichen Fachlehrern eine Verletzung der Datenintegrität hervorrufen:

Wenn mehrere Personen das Recht haben, z.B. die gespeicherte Fehlstundenanzahl eines Schülers zu erhöhen, so dürfen diese Personen nicht gleichzeitig die Fehlstundenanzahl ändern.
Beispiel: Lehrer 1 und Lehrer 2 öffnen praktisch gleichzeitig den Datensatz des Schülers Willi Wacker. Beide Lehrer lesen die Fehlstundenanzahl 10. Lehrer 1 erhöht um 3 und speichert den neuen Wert 13, Lehrer 2 erhöht um 7 und speichert -allerdings später als Lehrer 1- den neuen Wert 17. Damit stände in der Datenbank der aktuelle Wert 17. Der richtige Wert wäre allerdings 20 gewesen.

Ein funktionierendes DBMS darf also nur der aktuell ersten Person schreibenden Zugriff erlauben und jeder weiteren Person nur lesenden Zugriff. Der Schreibzugriff für jede weitere Person muss blockiert werden. Das sollte ein gutes DBMS der zweiten Person aber auch mitteilen!

In diesem Zusammenhang gibt es zahlreiche Probleme zu lösen: Soll nur der eine Schüler Willi Wacker blockiert werden oder die ganze Klasse? Sollen bei der Noteneingabe z.B. nur die Physiknote von Willi Wacker blockiert werden oder alle seine Noten? Wenn der Lehrer 1 sich nicht ordentlich vom System abmeldet, wird evtl. der Zugriff für alle anderen Personen auf ewig blockiert.

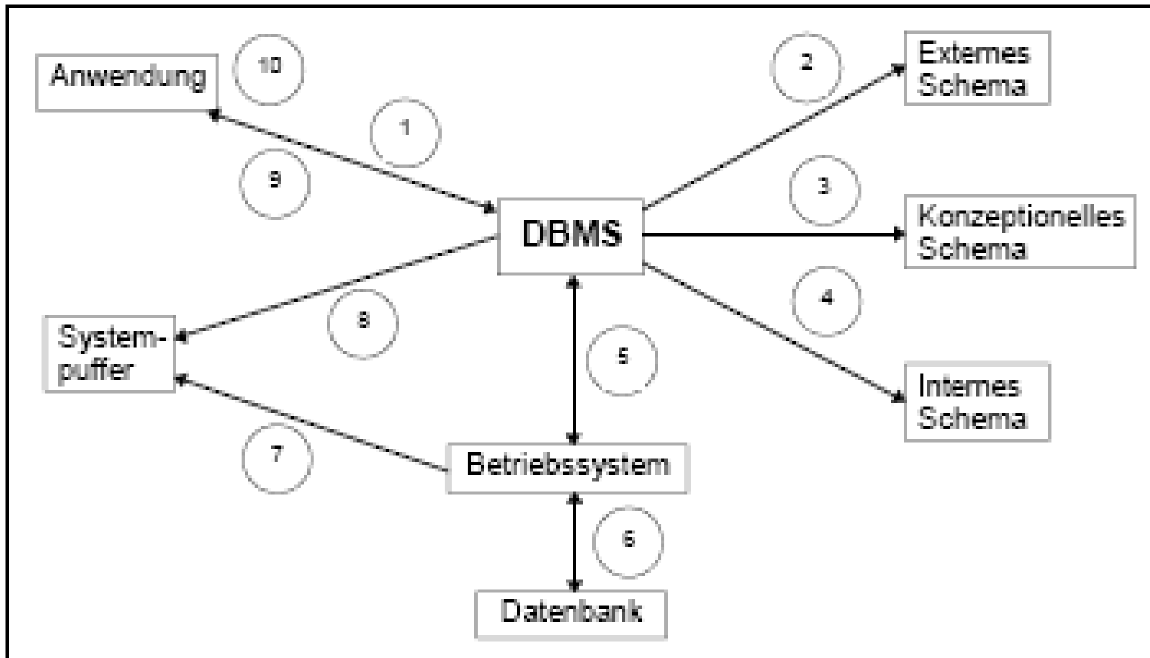
Ein eigenes Programm des DBMS, der so genannte *Transaktionsmanager*, soll derartige Probleme verhindern bzw. lösen.

Wir werden uns mit diesen Problemen im Unterricht nicht weiter beschäftigen. Es sei nur erwähnt, dass es zahlreiche Software gibt, welche derartige Probleme nicht richtig löst.

Völlig analoge Probleme treten z.B. auch auf, wenn mehrere Personen gleichzeitig eine Homepage bearbeiten dürfen (Stichwort: CMS = content management system).

Datenbankverwaltungssysteme (DBMS) stellen also die Verbindung zwischen der Datenbasis und den Datenbankbenutzern bzw. Anwendungsprogrammen her. Dabei erfolgt der Zugriff der Anwendungen auf die Datenbasis nicht direkt, sondern nur über das DBMS.

Das folgende Diagramm veranschaulicht, wie das Datenbankverwaltungssystem eine Anfrage (Query) abarbeitet.



1. Das DBMS empfängt den Befehl des Anwendungsprogrammes, ein bestimmtes Objekt zu lesen.

2. Das DBMS holt sich die benötigten Definitionen des entsprechenden Objekttyps aus dem zugehörigen externen Schema.

3. Das DBMS stellt fest, auf welche konzeptionellen Objekte sich die Anfrage bezieht.

4. Das DBMS stellt fest, welche physischen Objekte zu lesen sind.

5. Das DBMS übergibt dem Betriebssystem die Nummern der zu lesenden Speicherblöcke.

6. Das Betriebssystem greift auf die physischen Speicherblöcke in der Datenbank zu.

7. Das Betriebssystem übergibt die verlangten Blöcke an das DBMS in einen Systempuffer.

8. Das DBMS stellt aus den vorhandenen physischen Sätzen im Systempuffer das verlangte Objekt zusammen.

9. Das DBMS übergibt das Objekt dem Anwendungsprogramm in seinen Arbeitsspeicher.

10. Das Anwendungsprogramm verarbeitet die vom DBMS übergebenen Daten.

Aufgabe 1
Es soll eine Datenbank für eine Bücherei eingerichtet werden. Ein einzelnes Buch sei beschrieben durch die folgenden Merkmale:
Inventarnummer des Buches, ISBN-Nummer, Autor, Titel, Fachgebiet, Verlag, Erscheinungsort und -jahr, Auflage, Preis.
Zu welcher Ebene gehört diese Beschreibung?

Aufgabe 2:
Analysiere bei jeder der folgenden Umstellungen, auf welcher der Ebenen einer bestehenden Datenbank und an welchen Transformationen jeweils etwas geändert werden muss (im Idealfall):

a) Neuer Rechner, abwärts kompatibel, gleiches Betriebssystem, gleiches Datenbanksystem.

c) Neue Festplatten mit anderer Struktur.

f) Eine neue Verbindung zwischen zwei existierenden Typen wird eingeführt, bisher gab es z.B. zwischen Schülern und Lehrern nur die Verbindung, wer Tutor ist. Jetzt gebe es zusätzlich die Verbindung, wer für jeden Schüler frei wählbarer Vertrauenslehrer ist.

Lösungen

Aufgabe 1:
Die Beschreibung gehört zum **konzeptuellen Schema**

Aufgabe 2:
a) Auf keiner Ebene sind Änderungen erforderlich.

c) Da die physischen Datenstrukturen jetzt neu sind, sind Änderungen der internen Sicht notwendig.
Keine Änderungen auf höheren Ebenen. Änderung der Transformation zwischen interner und logischer Sicht.

f) Da eine neue Verknüpfung zwischen den zwei bereits existierenden Typen Schüler und Lehrer eingerichtet werden muss, ist eine Änderung der logischen Gesamtsicht (also konzeptuelle Ebene) notwendig.
Da diese neue Verknüpfung irgendwo gespeichert werden muss, ist auch eine Änderung der internen Sicht erforderlich.
Externe Sichten müssen nur geändert werden, wenn sie die neue Verbindung benutzen.

Entities und Beziehungen

Wir betrachten im Folgenden die Schülerdatenbank einer Schule. Diese enthält außer den eigentlichen Schülerdaten (Name, Adresse, Eltern, Geburtsdatum usw.) auch noch Angaben darüber, welche Kurse in welchen Jahrgangsstufen von welchen Lehrern angeboten werden. Natürlich sind auch die von den Schülern erreichten Noten in den von ihnen gewählten Kursen enthalten.

Informationen wie *Schulname*, *Schülervorname, Kursthema*, stellen **Eigenschaften** der betreffenden **Objekte** Schule, Schüler bzw. Kurs dar.

Die Informationen zur Note oder über Bemerkungen beziehen sich dagegen auf jeweils zwei Objekte gemeinsam, d.h. sie stellen Eigenschaften einer **Beziehung** (engl.: **relationship)** zwischen den Objekten Schüler/Kurs bzw. Schüler/Zeugnis dar. Wir können diesen Beziehungen die Namen „besucht" und „erhält" geben, es entstehen also die Beziehungen

- Schüler *besucht* Kurs
- Schüler *erhält* Zeugnis

zwischen jeweils 2 Objekten.

In der Informatik spricht man im Zusammenhang mit Datenbanksystemen meist nicht von Objekten, sondern benutzt den Begriff *Entity* oder auch *Entität*. Entities können demnach Personen sein, reale Objekte wie Zeugnisse oder Räume, aber auch abstrakte Objekte, wie z.B. Kurse, die nur gedanklich als ein unterscheidbares und identifizierbares Objekt existieren.

Der aus der Objektorientierten Programmierung her bekannte Begriff *Klasse* wird im Zusammenhang mit Datenbanken als *Entity-Typ* bezeichnet. Leider werden in der entsprechenden Literatur zum Thema Datenbanken die beiden unterschiedlichen Begriffe *Entity* und *Entity-Typ* nicht immer deutlich genug voneinander unterschieden.

In der Objektorientierten Programmierung besitzen die Objekte einer Klasse normalerweise nicht nur Attribute (Eigenschaften) sondern auch (gemeinsame) Methoden. In Datenbanksystemen besitzen die Entities nur Eigenschaften (=Attribute).

Ein *Entity-Typ* wird im Wesentlichen durch die gemeinsamen Eigenschaften aller seiner *Entities* beschrieben. Jedes konkrete Entity besitzt für alle seine Eigenschaften jeweils einen Wert. Beispielsweise besitzt der Entity-Typ mit Namen „Kurs“ eine Eigenschaft namens „Kursbezeichnung“. Ein bestimmtes Entity dieses Entity-Typs „Kurs“ besitzt in seiner Eigenschaft „Kursbezeichnung“ den konkreten Wert *„GkPhyQ1“*. Dies entspricht völlig dem Klassen-Objekt Prinzip in der objektorientierten Programmierung.

Die aus der Objektorientierten Programmierung bekannten Klassen werden in Datenbanksystemen ***Entity-Typ*** genannt. Sie entsprechen jeweils **einer ganzen Tabelle**. Jede Eigenschaft des Entity-Typs (= Attribut der Klasse) wird hier durch eine eigene Tabellen**spalte** dargestellt. Die Entities (Objekte) des Entity-Typs (Klasse) sind hier die **Zeilen** der Tabelle.

Schüler

Nr	Name	Adresse	Stufe	Lk 1	Lk2	Gk1	…..	Geschlecht	…
1	Anton Weis	Do, ….	Q1	Ma	Ph	D	….	m	…
2	Ina Klein	Do, ..	Q1	Ph	Ch	E	….	w	…
3	Willi Wang	Schwerte	Q1	D	Pä	Ma	….	m	…
….	…..	….	….	….	….	….	….		…

Wir werden noch genauer untersuchen, wie man Entities und ihre Beziehungen zueinander beschreiben und darstellen kann. Die folgende Darstellung zeigt aber bereits alle wesentlichen Merkmale:

Entity	Beziehung	Eigenschaften
Schule		Schulname Schulort
Kurs		Kursbezeichnung Typ Lehrer
	Schüler(in) *besucht* Kurs	Note
Schüler(in)		Name Vorname Tutor
	Schüler(in) *erhält* Zeugnis	Fehlstunden entschuldigt Fehlstunden unentschuldigt Bemerkung
Zeugnis		Halbjahr Datum

Wichtig: Auch Beziehungen haben Eigenschaften!

Um die jeweilige Modellierung genauer zu beschreiben, ist es für die Angabe der Entities und ihrer Beziehungen zueinander hilfreich, sog. ***Geschäftsregeln*** schriftlich festzuhalten. Diese stellen nichts anderes als Feststellungen über die Objekte der Miniwelt dar, z.B.:

Ein Zeugnis geht genau an einen Schüler und existiert ohne diesen nicht. Ein Schüler kann im Laufe der Oberstufe mehrere Zeugnisse erhalten; es gibt auch Schüler, die noch kein Zeugnis erhalten haben, z.B. in der Jahrgangsstufe 10-1 oder als Neuzugang.
Ein Zeugnis enthält mindestens eine Kursangabe mit Note und Angaben über Fehlstunden und eventuell Bemerkungen; ohne Zeugnis sind diese Angaben sinnlos.

Eine Kursangabe bzw. Fehlstunden und Bemerkungen können in vielen

Zeugnissen auftauchen; es gibt Bemerkungen, die auf keinem Zeugnis stehen.

Wir werden später sehen, dass das Modell unserer Miniwelt umso besser der Realität entspricht, je genauer diese *Geschäftsregeln* festgelegt sind.

Das Entity-Relationship-Modell

Will man eine Datenbank erstellen, so beginnt man logischerweise damit, ein **konzeptuelles Modell** des gegebenen Problems zu erstellen. Dabei wird ganz allgemein ein Abbild der Wirklichkeit geschaffen, bei dem bestimmte Aspekte weggelassen, vereinfacht, zusammengefasst werden. Es lassen sich dabei drei **Abstraktionsmechanismen** unterscheiden:

Klassifikation: Dinge bzw. Objekte (Entities) mit gemeinsamen Eigenschaften werden zu einer Klasse *(Entity-Typ)* zusammengefasst. Dabei werden nicht alle Eigenschaften der Objekte, meist *Attribute* genannt, berücksichtigt, sondern nur die für die Aufgabe relevanten. In unserer Miniwelt Schulverwaltung lassen sich alle Objekte „Schüler“ als Klasse auffassen, in der Miniwelt Bibliothek die Objekte „Buch“.

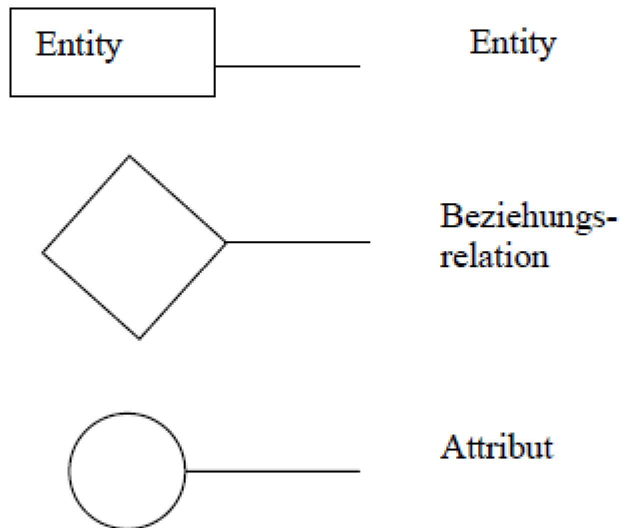
Aggregation: Bereits bestehende Klassen werden zu einer neuen Oberklasse zusammengefasst. So würde sich „Abiturbewertung“ aus „Halbjahresnoten“ und „Prüfungsergebnisse“ zusammensetzen.

Generalisierung oder *Spezialisierung*: Dabei wird eine Teilmengenbeziehung zwischen Elementen verschiedener Klassen definiert. So wäre „Ausleiher in der Schulbibliothek“ eine Generalisierung der Klassen „Schüler“ und „Lehrer“. Umgekehrt wäre „Abiturient“ eine Spezialisierung von „Schüler“.

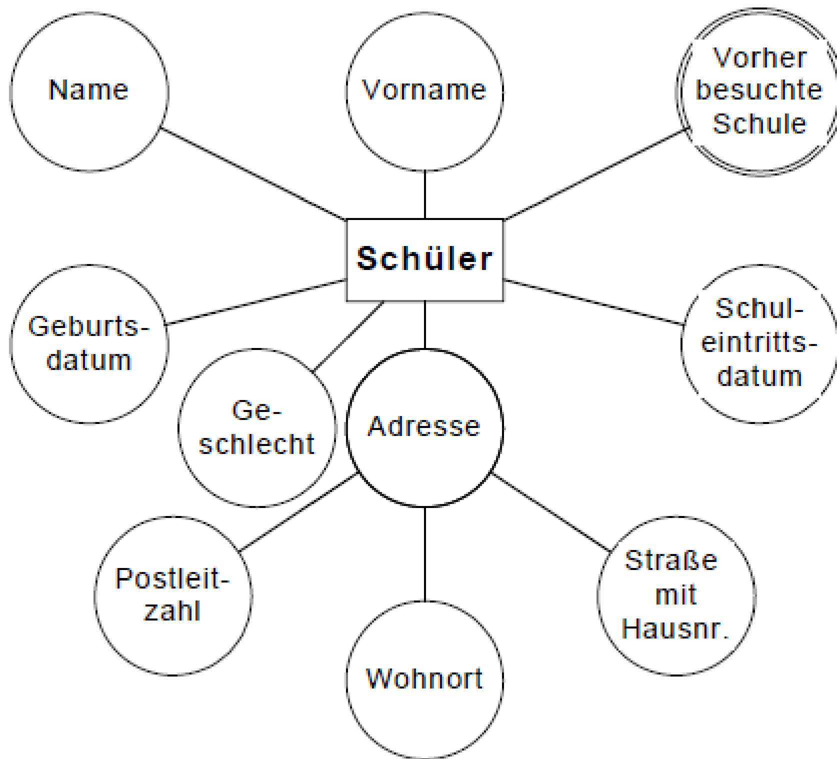
Bei Generalisierung, bzw. Spezialisierung gibt es immer *Vererbung*. So besitzt die untergeordnete Klasse (die Teilmenge) alle Attribute der Oberklasse, kann aber noch zusätzliche Eigenschaften haben. „Ausleiher“ braucht sicher die Attribute „Name, Vorname“, die Teilmenge Schüler besitzt noch „Noten“, „Lehrer“ besitzt zusätzlich noch „Unterrichtsfächer“.

Um die Modellierung der Realität systematischer gestalten zu können, gibt es verschiedene Standardstrategien. Eine der bekanntesten zur Erstellung von Datenbank-Anwendungen ist das so genannte ***Entity-Relationship-Modell*** (kurz ***ER-Modell*** genannt).

Das *ER-Modell* verknüpft Objekte (Entities) und ihre Beziehungen (Relationships) zueinander in einem gemeinsamen Modell.
Zur Modellierung werden im Entity-Relationship-Modell drei Einheiten verwendet:



Die Entities eines Typs besitzen gemeinsame Eigenschaften, *Attribute* genannt, im Beispiel *Name*, *Vorname*, usw. Die Attribute können ***atomar*** sein, d.h. **nur einen** Wert umfassen, mehrere Werte ermöglichen oder selbst wieder aus einzelnen Attributen zusammengesetzt sein. Man kann ein Objekt mit seinen Attributen graphisch sehr gut veranschaulichen.



Dabei wird der Entity-Typ durch ein Rechteck dargestellt, die Attribute durch mit dem Rechteck verbundene Kreise.

Ein zusammengesetztes Attribut wie „Adresse“ wird durch entsprechende Folgekreise gezeichnet.

Ein Attribut wie „Vorher besuchte Schule“, das einen oder mehrere Einträge haben kann, erscheint als Doppelkreis. Solche Attribute heißen *Mehrfachattribute.*

Im *relationalen Datenmodell* werden Attribute durch Spalten(überschriften) von Tabellen dargestellt.

Ein oder mehrere Attribute, die eine Entity eindeutig charakterisieren, werden *Schlüssel* genannt. In unserem Beispiel wäre ein möglicher Schlüssel die Kombination der Attribute „Name, Vorname, Geburtsdatum“, denn es erscheint doch extrem unwahrscheinlich, dass es zwei Schüler mit gleichem Nachnamen, Vornamen und Geburtsdatum gibt.

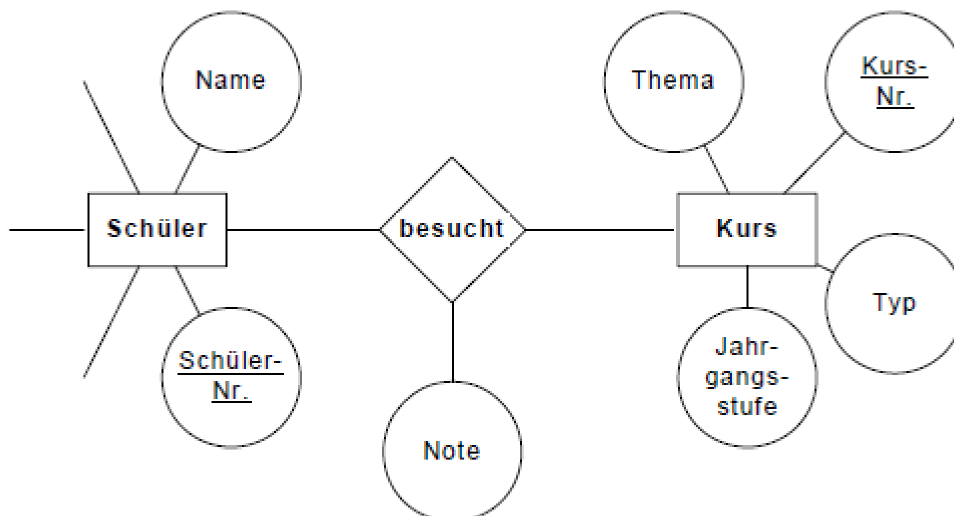
Es ist denkbar, dass noch andere Attributmengen mögliche Schlüssel sind. Normalerweise bevorzugt man einen von evtl. mehreren möglichen Schlüsseln zwecks Identifikation des ***Datensatzes*** (Entity). Diesen ausgewählten Schlüssel nennt man dann *Primärschlüssel (PS)*. Ein Attribut, welches schon alleine (ohne weitere Attribute) den Datensatz eindeutig identifiziert, wird *einfacher Primär-schlüssel* genannt im Gegensatz zu *zusammengesetzten Primärschlüsseln*.

Üblicherweise wird der Primärschlüssel **durch Unterstreichen des Attributnamens kenntlich gemacht**.

Häufig verwendet man auch *künstliche Attribute*, um einen Primärschlüssel zu schaffen. In unserem Beispiel wäre das eine Schüler-Nummer, die jedes Entity eindeutig charakterisiert.

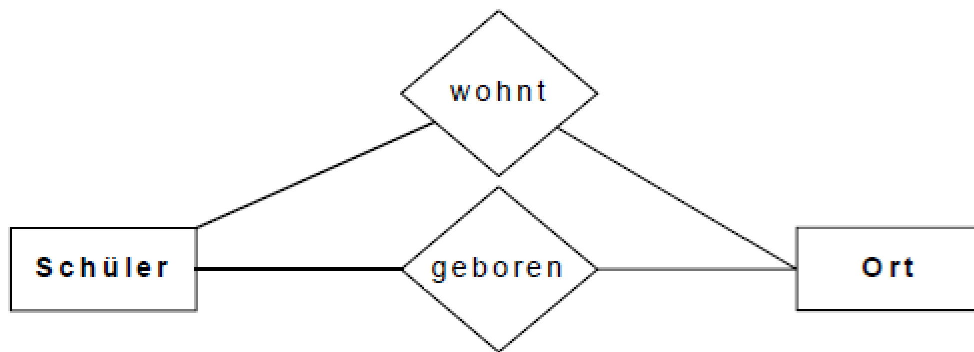
Zwischen den einzelnen Entities können *Beziehungen (Relationships)* bestehen, zwischen Entity-Typen existieren entsprechend *Beziehungs-Typen.* Beziehungs-Typen werden grafisch durch Rauten dargestellt, in die man die Beziehung notiert.

Auch Beziehungs-Typen können Attribute besitzen. Als Beispiel betrachten wir den Beziehungs-Typ „besucht" zwischen den Entity-Typen „Schüler" und „Kurs", die „Note" als Attribut hat.

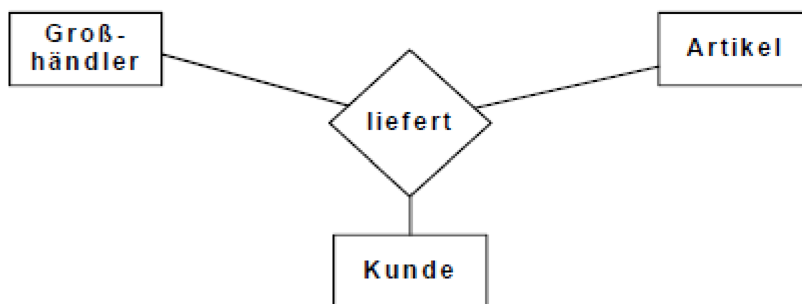


Die Note gehört nicht zum Schüler und auch nicht zum Kurs, sie ist vielmehr ein Produkt der Beziehung zwischen dem Schüler und dem Kurs, den er besucht. Gäbe es die Beziehung nicht, entstünde auch keine Note, daher die Zuordnung zur Beziehungsrelation und nicht zu einer der beiden Entities.

Zwischen Entities kann es mehr als eine Beziehung geben, wie das folgende Beispiel zeigt:



Ein Beziehungs-Typ kann sich auch auf mehr als zwei Entity-Typen beziehen: Verschiedene Großhändler liefern verschiedene Artikel an unterschiedliche Kunden. Die Lieferbeziehung ist also durch ein Tripel (Großhändler, Artikel, Kunde) gekennzeichnet.



Allerdings sind die zweiwertigen Beziehungen die weitaus häufigsten.

Aufgabe zum Entity-Relationship-Modell
Zugrunde gelegt werde das ER-Modell einer Schulverwaltung. Geben Sie bei den nachfolgenden Änderungen an, welche Ebene des DBMS von der Änderung betroffen ist.

1. Der Lehrer Franz Schlauspruch kommt neu an die Schule.

2. Die Lehrer sollen bei der Noteneingabe nicht mehr die Noten des Schülers bei anderen Lehrern abfragen können.

3. Bei den Schülerdaten soll durch eine zusätzliche Indexdatei, in der die Schüler nach Wohnort sortiert sind, ein schnellere Suche nach Schülern eines Ortes ermöglicht werden, um die Busverbindungen besser koordinieren zu können.

4. Zur Erstellung von Altersstatistiken soll auch bei Lehrern das Geburtsdatum gespeichert werden.

5. Der Raum 556 soll als neuer Fachraum für Mathematik verwendet werden.

6. Auf Wunsch des Hausmeisters wird zusätzlich erfasst, welcher Lehrer einen Schlüssel für welchen Fachraum hat.

7. Neben dem Oberstufenleiter sollen auch die Tutoren die komplette Belegung ihrer Schüler einschließlich der bisher vergebenen Noten am Computer einsehen können.

8. Vom Sekretariat sollen die Schüler, die mehr als drei Grundkurse unter 5 Punkten einbringen müssen, per Serienbrief auf die Gefahr der Nichtzulassung zum Abitur hingewiesen werden.

9. Auf Antrag der SV dürfen Lehrer, die nicht Tutor des entsprechenden Schülers sind, die Fehlstundenzahl nicht mehr einsehen.

10. Daraufhin beschließt die Schulkonferenz, die Fehlstundenspeicherung ganz abzuschaffen.

Lösung

1 Kein Modell ist betroffen. Es werden lediglich zusätzliche Daten eingegeben.
2 Externe Sicht für die Benutzer „Lehrer“ wird geändert.
3 Die interne Organisation der Daten im Rechner wird geändert, also ist das interne Modell betroffen.
4 Das konzeptuelle Modell muss geändert werden. Der Entity-Typ Lehrer erhält ein zusätzliches Attribut.
5 Kein Modell ist betroffen, es werden lediglich Dateneinträge geändert.
6 Das konzeptuelle Modell wird geändert. Zwischen Lehrer und Fachraum wird eine neue Beziehung *hat Schlüssel für* aufgebaut.
7 Die externe Sicht für die Benutzer „Tutoren“ wird geändert.
8 Etwas unklares Beispiel. Es wäre denkbar, dass die externe Sicht für den Benutzer „Textverarbeitungsprogramm im Sekretariat“ geändert werden muss.
9 Externe Sicht für die Benutzer „Lehrer“ wird geändert.
10 Das konzeptuelle Modell ist betroffen. Das Attribut „Fehlstunden“ der Beziehung *besucht* wird gelöscht.

Komplexität

Die bisherige Darstellung sagt aber noch nichts über die Anzahl der jeweils an einer Beziehung beteiligten Entities aus. Im Normalfall unterrichtet z.B. ein Lehrer mehrere Kurse, ein Kurs besitzt aber nur einen Lehrer. Diese Tatsache beschreibt man mittels des Begriffs der *Komplexität* einer Beziehung:

Die Beziehung *Lehrer unterrichtet Kurs* besitzt die Komplexität 1:n, da "1" Lehrer mehrere "n" Kurse unterrichtet.

Insgesamt unterscheidet man meist zwischen drei *Komplexitäten*:
1:1-Beziehungen: Jedes Entity vom Typ E1 steht höchstens mit einem Entity vom Typ E2 in Beziehung und umgekehrt (also mit einem oder keinem).

1:n-Beziehungen: Jedes Entity vom Typ E2 steht höchstens mit einem Entity vom Typ E1 in Beziehung, es können aber mehrere (oder keiner) aus E2 zum selben Entity aus E1 eine Beziehung haben. n:1-Beziehungen sind natürlich ganz analog definiert.

n:m-Beziehungen: Jedes Entity aus E1 kann zu mehreren aus E2 eine Beziehung haben, und jedes Entity aus E2 zu mehreren aus E1.

Beispiele:
Komplexität 1:1: Schüler - erhält - Abiturzeugnis.
Jeder Schüler erhält (höchstens) ein Abiturzeugnis, und jedes Abiturzeugnis gehört eindeutig zu einem Schüler.

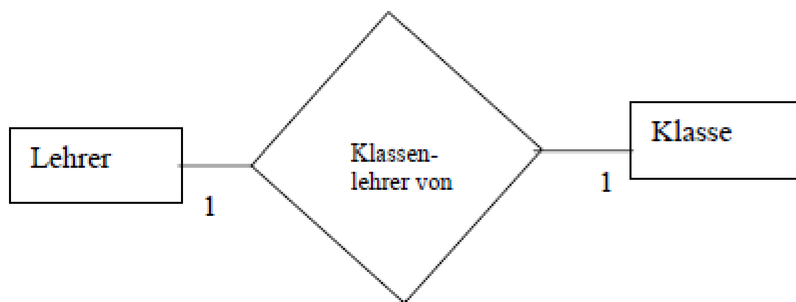
Komplexität 1:n: Lehrer - ist Stufenleiter von - Schüler.
Jeder Schüler hat nur einen Stufenleiter, aber jeder Lehrer, der Stufenleiter ist, hat mehrere Schüler in seiner Jahrgangsstufe.

Komplexität n:m: Schüler - besucht - Kurs.
Jeder Schüler besucht mehrere Kurse, und jeder Kurs wird von mehreren Schülern besucht.

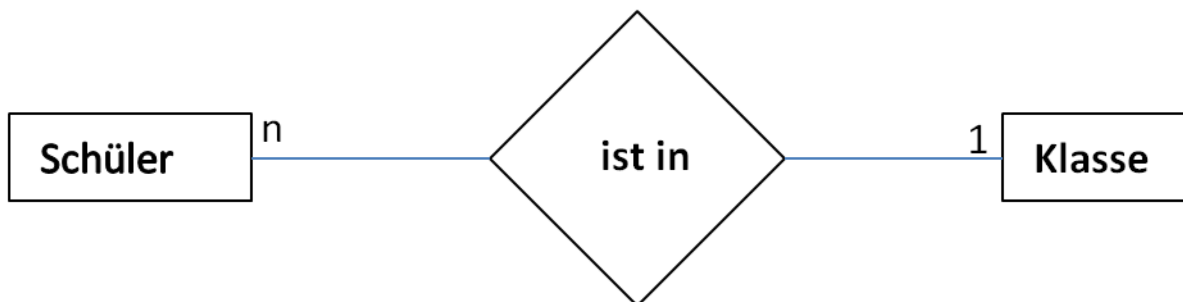
Manchmal findet man noch die Notation "c" für [0,1], also **1:c-Beziehungen**. Sie werden technisch wie 1:n Beziehungen behandelt.

Im ***ER-Diagramm*** werden die Komplexitäten einer Beziehung an dem

entsprechenden Entitätstypen notiert.

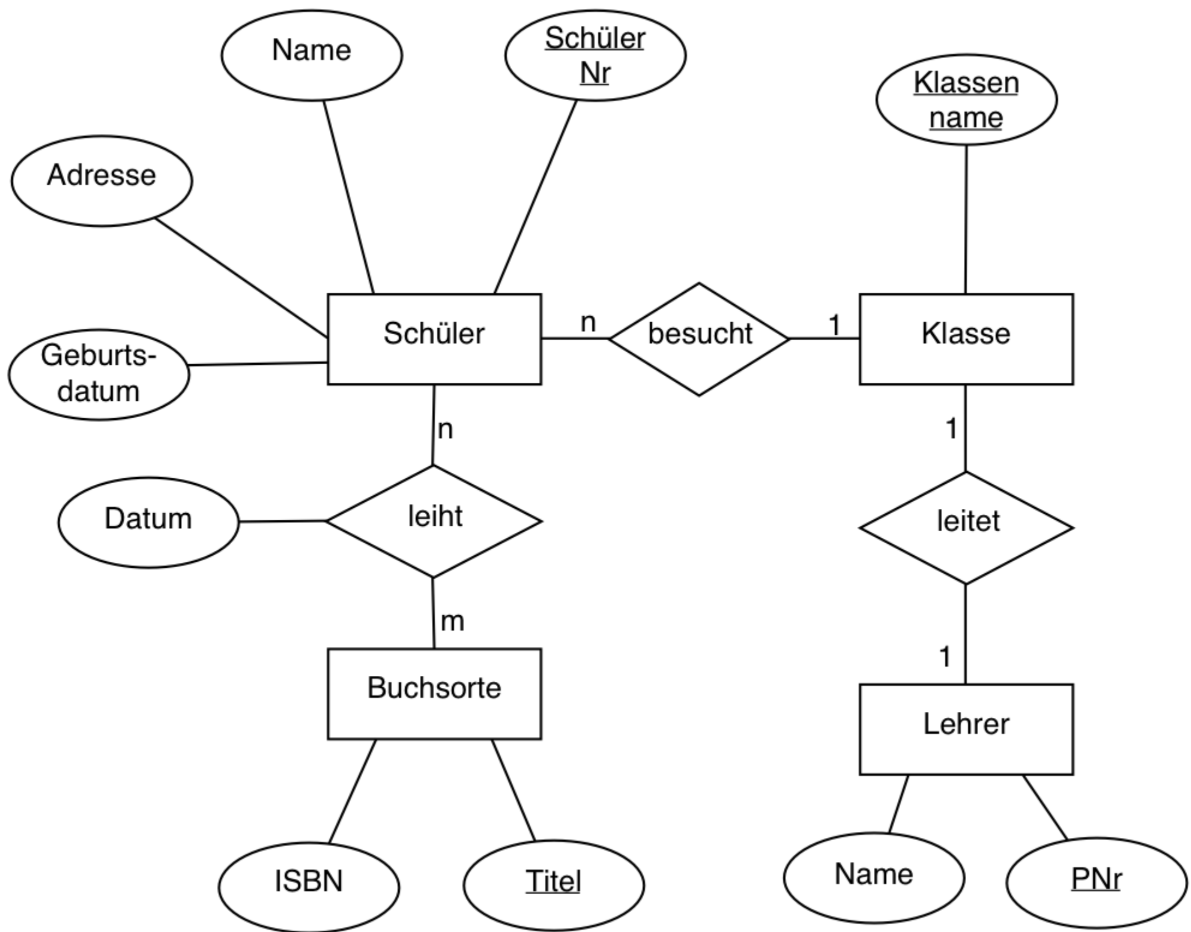


Obiges Diagramm bedeutet: Jeder Lehrer ist Klassenlehrer von höchstens einer Klasse, eventuell ist er aber auch kein Klassenlehrer bzw. Klassenlehrer von 0 Klassen. Jede Klasse besitzt höchstens einen Klassenlehrer, eventuell aber auch gar keinen. Letzteres sollte in den *Geschäftsregeln* ausgeschlossen werden!



Obiges Diagramm bedeutet: Jeder Schüler ist in höchstens einer Klasse. Es gibt eventuell auch Schüler, die in keiner Klasse sind. Letzteres sollte in den *Geschäftsregeln* ausgeschlossen werden! Jede Klasse besitzt mehrere Schüler, eventuell aber auch gar keinen Schüler. Letzteres sollte wieder in den *Geschäftsregeln* ausgeschlossen werden!

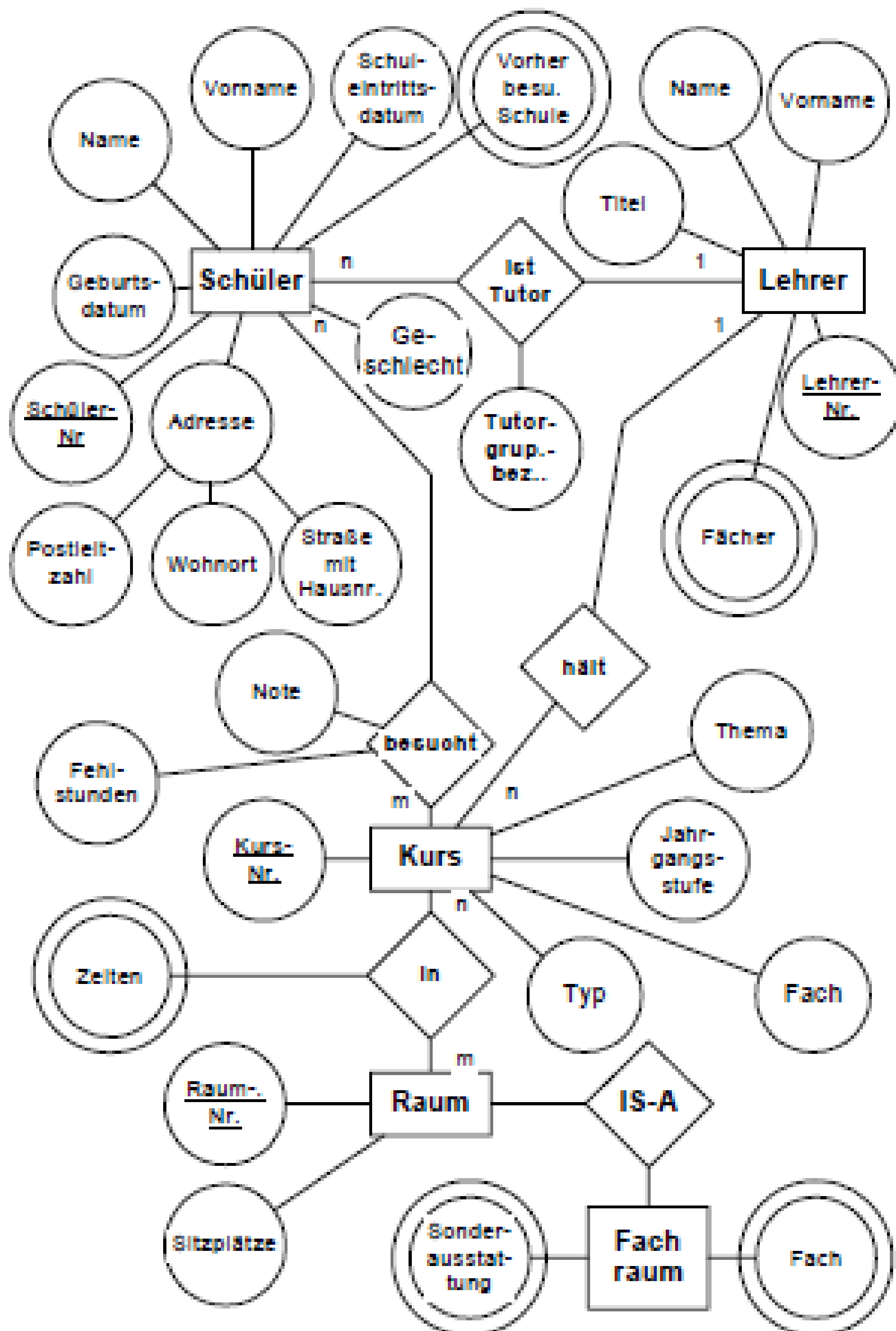
Aufgabe: Beschreibe folgendes konzeptuelles Modell (Entity-Relationship-Modell)!


Name
Schüler Nr
Klassen name
Adresse
Schüler
n
besucht
1
Klasse
Geburts- datum
n
1
Datum
leiht
leitet
m
1
Buchsorte
Lehrer
ISBN
Titel
Name
PNr

Lösung:
Für das obige Beispiel gilt (beachte auch die Geschäftsregeln)

- ein Lehrer ist Klassenlehrer von höchstens einer Klasse.
- eine Klasse wird von mehreren Schülern besucht, allerdings ist jeder Schüler nur in einer Klasse. (Die Geschäftsregeln besagen, dass eine Klasse aus mindestens 1 Schüler bestehen muss, und jeder Schüler muss genau in einer Klasse sein.)
- ein Schüler leiht sich mehrere Bücher verschiedener Sorte und jede Buchsorte kann an verschiedene Schüler verliehen sein. (Geschäftsregeln sind vielleicht: jeder Schüler darf höchstens 7 Bücher gleichzeitig ausleihen).

Aufgabe: Beschreibe folgendes ER-Modell!



Aufgabe zum ER-Modell - „Rechnungschreiben“

Die folgenden Musterrechnungen dokumentieren eine Miniwelt „Rechnungschreiben“ in einer Firma.

a) Stelle in Tabellenform (wie auf Seite 11) die Entity- und Beziehungstypen dar!
b) Erstelle das zugehörige ER-Diagramm mit den Komplexitäten!
c) Formuliere Geschäftsregeln für diese Miniwelt!

Herrn
Horst Staniczek
Birnbaum 3
65510 Hünstetten

Rechnungsnummer: R123 Rechnungsdatum: 11.06.1995
Kundennummer: K002 Rechnungsbetrag: 1397,00

Position	Artikelnummer	Bezeichnung	Anzahl	Einzelpreis	Gesamtpreis
1	A3257	Minitor 17“	2	499,00	998,00
2	A4210	Nadeldrucker	1	399,00	399,00

Firma
Irma Computer GmbH&Co KG
Am Festungswall 45
65189 Wiesbaden

Rechnungsnummer: R457 Rechnungsdatum: 04.03.1996
Kundennummer: K195 Rechnungsbetrag: 359,00

Position	Artikelnummer	Bezeichnung	Anzahl	Einzelpreis	Gesamtpreis
1	A3117	CD-Rom	1	359,00	359,00

Lösung:

Aus den Beispielrechnungen lassen sich offensichtlich die Objekte *Kunde, Rechnung, Artikel* mit folgenden Informationen ableiten:

Kunde: Name, Straße und Hausnummer, PLZ, Ort, Kundennummer

Rechnung: Rechnungsnummer, Rechnungsdatum, Rechnungsbetrag

Artikel: Artikelnummer, Artikelbezeichnung, Artikelpreis

Die Informationen Positionsnummer, Anzahl und Postenpreis lassen sich keinem dieser Objekte direkt zuordnen.

Allerdings bestehen offensichtlich Beziehungen zwischen den Objekten „Kunde" und „Rechnung" bzw. „Rechnung" und „Artikel"!

Insbesondere der Relation **Rechnung_enthält_Artikel** lassen sich die Attribute Positionsnummer, Anzahl und Postenpreis zuordnen.

Der Relation **Kunde_erhält_Rechnung** lassen sich keine weiteren Attribute zuordnen.
Die Objekte und ihren Beziehungen kann man mit ihren Eigenschaften übersichtlich in einer Tabelle darstellen. Dabei sind die Primärschlüssel der Entities unterstrichen.

Objekt (Entity)	**Beziehung (Relation)**	**Eigenschaften (Attribute)**
Kunde		<u>Kundennummer</u> Name Strasse PLZ Ort
	Kunde/Rechnung *erhält*	
Rechnung		<u>Rechnungsnummer</u> Rechnungsdatum Rechnungsbetrag
	Rechnung/Artikel *enthält*	Positionsnummer Anzahl Postenpreis
Artikel		<u>Artikelnummer</u> Artikelbezeichnung Einzelpreis

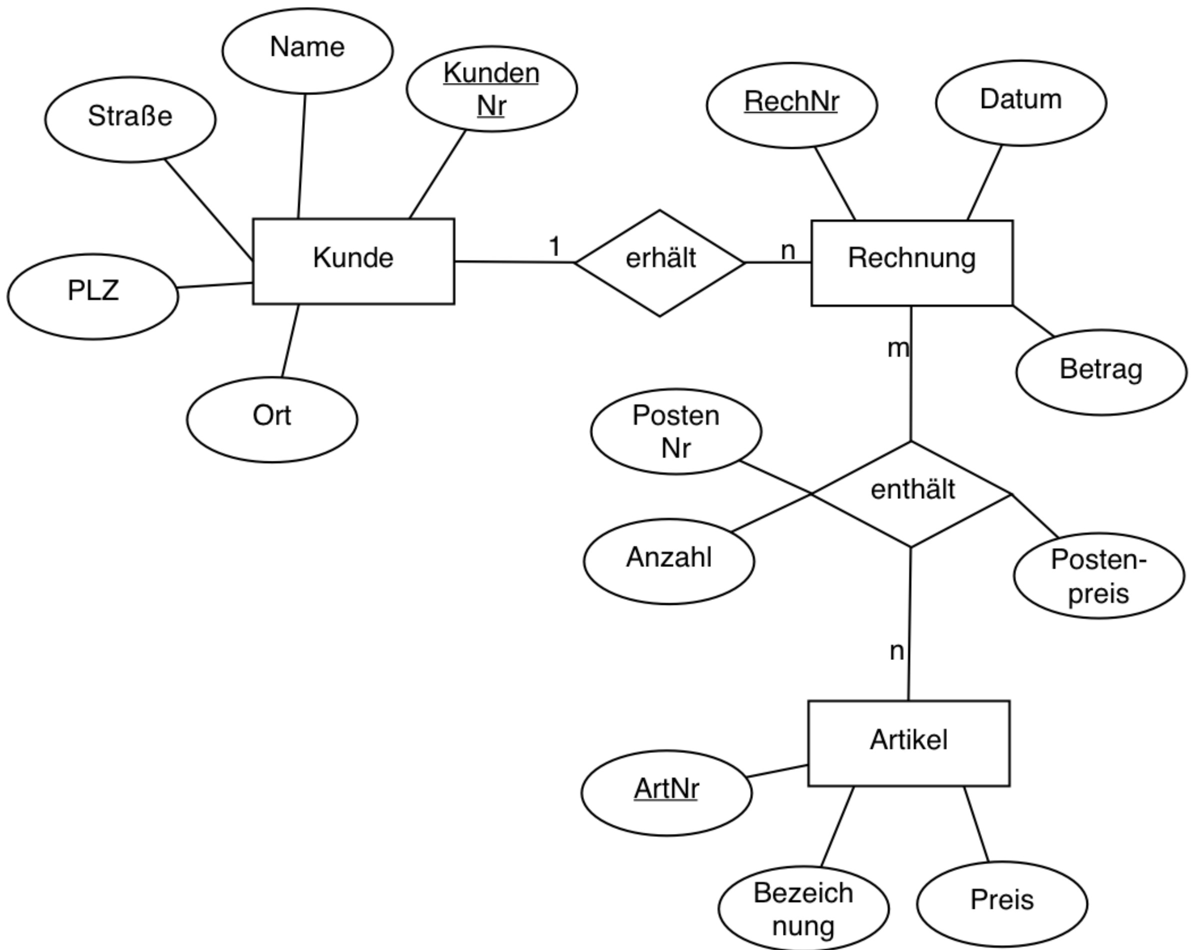
Obige Tabelle beschreibt bereits ein geeignetes Entity-Relationship-Modell.

Darüber hinaus lassen sich bereits detaillierte Beschreibungen über die Zusammenhänge der Elemente der Miniwelt angeben. Solche Aussagen über den Geschäftsgang nennt man Geschäftsregeln.

Es lassen sich folgende Aussagen machen:

- Eine Rechnung geht genau an einen Kunden, ohne Kunde existiert keine Rechnung.
- Ein Kunde kann keine, eine oder mehrere Rechnungen erhalten.
  Eine Rechnung enthält mindestens eine Rechnungsposition. Ohne Rechnung kann eine Rechnungsposition nicht existieren.
- Eine Rechnungsposition betrifft genau einen Artikel und kann ohne diesen nicht existieren.
- Ein Artikel kann in keiner, einer oder mehreren Rechnungspositionen erscheinen.

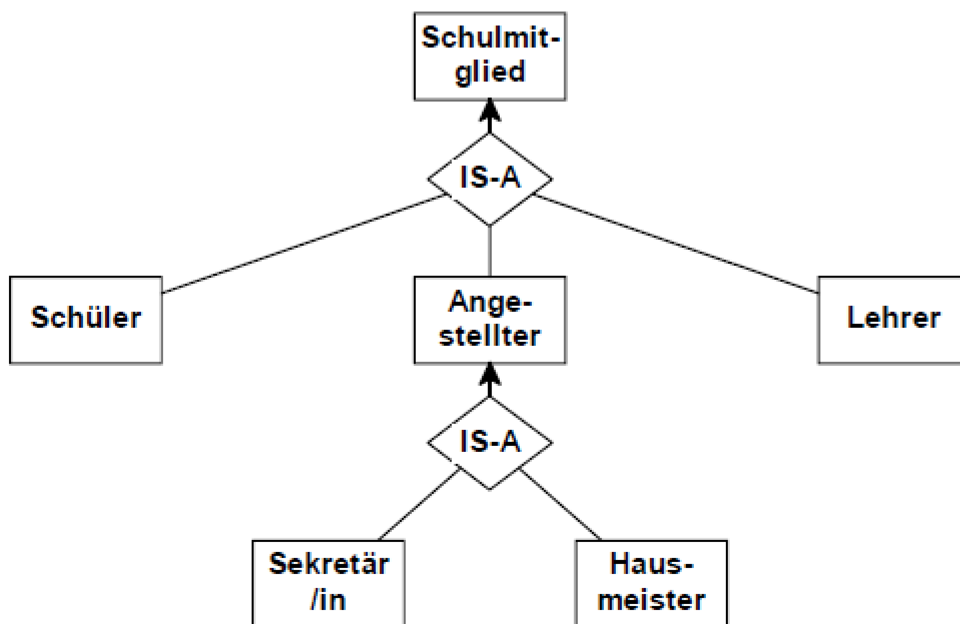
Das Entity-Relationship-Modell lässt sich sofort auch in grafischer Form darstellen:


Name
Kunden Nr
Straße
PLZ
Kunde
Ort
1
erhält
n
RechNr
Datum
Rechnung
Betrag
m
Posten Nr
enthält
Anzahl
Posten-preis
n
Artikel
ArtNr
Bezeich nung
Preis

Eine besondere Art von Beziehungen sind die so genannten “**IS-A**“ Beziehungen. Damit wird es möglich, Teilmengen von Entity-Typen zu erfassen. Diese Beziehungen entsprechen der oben angegebenen speziellen Abstraktionsform *Spezialisieren* bzw. *Generalisieren*.

Bei bestimmten Entities gibt es sinnvolle Teilmengen, die alle Attribute der übergeordneten Menge erben (diese müssen nicht mehr in der grafischen Darstellung angegeben werden), aber noch zusätzliche Attribute besitzen.

Ein Beispiel soll das verdeutlichen.



Auf eine Angabe von Attributen wurde verzichtet. Es lassen sich jedoch ohne Probleme an diesem Beispiel ererbte und neue Attribute erläutern. So wären etwa “Name, Vorname, Adresse“ für alle Schulmitglieder gemeinsam. Bei Lehrer könnten die Zusatzattribute „Titel, Unterrichtsfächer“ auftauchen, bei Angestellter “Besoldungsgruppe“ usw.

Aufgaben zur Komplexität von Beziehungen

Aufgabe 1:
Die folgende Tabelle zeigt jeweils zwei Entity-Typen und den zugehörigen Beziehungstyp. Gib für jede Beziehung die Komplexität an!

E1	Beziehung	E2
Schüler	hat	Tutor
Schüler	bekommt heute	Zeugnis
Schüler	darf arbeiten an	Computer
Schüler	hat ausgeliehen	Buch
Schüler	besucht	Kurs
Schüler	ist befreundet mit	Schüler

Aufgabe 2:
Überlege, welche Komplexität die IS-A-Beziehung besitzt!

Aufgabe 3:
Die folgende Tabelle zeigt jeweils zwei Entity-Typen und den zugehörigen Beziehungstyp. Gib für jede Beziehung die Komplexität an!

E1	Beziehung	E2
Mann	ist Vater von	Kind
Onkel	hat	Neffe
Schüler	hat Unterricht	Lehrer
Person	besitzt	Personalausweis
Bruder	hat	Schwester
Stadt	kürzeste Entfernung	Stadt

Lösungen

Aufgabe 1

E1	Beziehung	E2	Komplexität
Schüler	hat	Tutor	n:1
Schüler	bekommt heute	Zeugnis	1:1
Schüler	darf arbeiten an	Computer	n:m
Schüler	hat ausgeliehen	Buch	1:n
Schüler	besucht	Kurs	n:m
Schüler	ist befreundet mit	Schüler	n:m

Aufgabe 2:
Überlege, welche Komplexität die IS-A-Beziehung besitzt!
Die Antwort ist nicht ganz so einfach, wie man zuerst vermuten könnte. Im Beispiel zur Schulverwaltung ist ein Hausmeister auch genau ein Angestellter und ein Angestellter auch genau ein Mitglied der Schulgemeinde. Es handelt also hier um 1:1 - Beziehungen. Da wir jedoch mit IS-A - Beziehungen ganz allgemein Generalisierung und Spezialisierung erfassen wollen, sind auch 1:n - Beziehungen denkbar. Ein Bsp. wäre die Beziehung zwischen dem Ober-Entity-Typ *Autotyp* und dem Untertyp *Auto*. Zu *Autotyp* würden Attribute wie *Bezeichnung, PS-Zahl, Sitzplätze* usw. gehören, zu Auto z.B. *Kennzeichen, Zulassungsdatum, gefahrene km*. *Auto* erbt alle Attribute von *Autotyp*, aber es gibt natürlich mehrere Autos desselben Typs.

Aufgabe 3:

E1	Beziehung	E2	Komplexität
Mann	ist Vater von	Kind	1:n
Onkel	hat	Neffe	n:m
Schüler	hat Unterricht	Lehrer	n:m
Person	besitzt	Personalausweis	1:1
Bruder	hat	Schwester	n:m
Stadt	kürzeste Entfernung	Stadt	1:1

Aufgaben

4. Geben Sie die Komplexität der folgenden Situationen an. Erstellen Sie jeweils ein ER-Diagramm. Formulieren Sie sinnvolle Geschäftsbedingungen.

a) Eine Ladenkette möchte Informationen über ihre Filialen und über deren Zulieferer speichern. Jeder Zulieferer beliefert mehrere Filialen und jede Filiale kauft von mehreren Zulieferern.

b) Ein Installateur möchte über seine Handwerker und über die Häuser, in denen sie gerade arbeiten, Informationen speichern.

5. Die Computerzubehörfirma Rauchchip verkauft ein Sortiment von Artikeln, die sie von verschiedenen Herstellern bezieht. Außerdem hat sie einen bestimmten Kundenkreis, der bei ihr Bestellungen aufgibt. Eine Bestellung kann natürlich mehrere Artikel umfassen. Derselbe Artikel kann oft von mehreren Herstellern bezogen werden, und ein Hersteller liefert natürlich meist mehr als einen Artikel.

Erstellen Sie im Entity-Relationship-Modell ein sinnvolles Datenmodell für die Firma, das Datenredundanz vermeidet. Wählen Sie geeignete Entities mit notwendigen Attributen und geben Sie die zwischen den Entities bestehenden Beziehungen mit ihrem Komplexitätsgrad an.

6. Hugo Unbedarft besitzt eine große Spedition. Er will seine Auftragsverwaltung auf EDV umstellen und macht sich dazu einen genauen Plan. Seine Aufträge sind immer so, dass sie nur zu einem Ziel führen, es kann allerdings möglich sein, dass mehrere LKWs für einen Auftrag nötig sind. Nicht jeder LKW-Typ ist dazu geeignet, alle Ziele zu erreichen (z.B. zu niedrige Brücken), und nicht jeder Fahrer kann jeden LKW-Typ fahren. Hugo will folgende Daten speichern: AuftragsNr. und LKW-Nr., Ziel, Zielentfernung, Auftragsdatum, LKW-Typ, max. Zuladung eines LKW-Typs, TÜV-Datum, Fahrer-Nr, Fahrer-Name.
Erstellen Sie ein ER-Modell für die Spedition.

Datenbank Sportverein

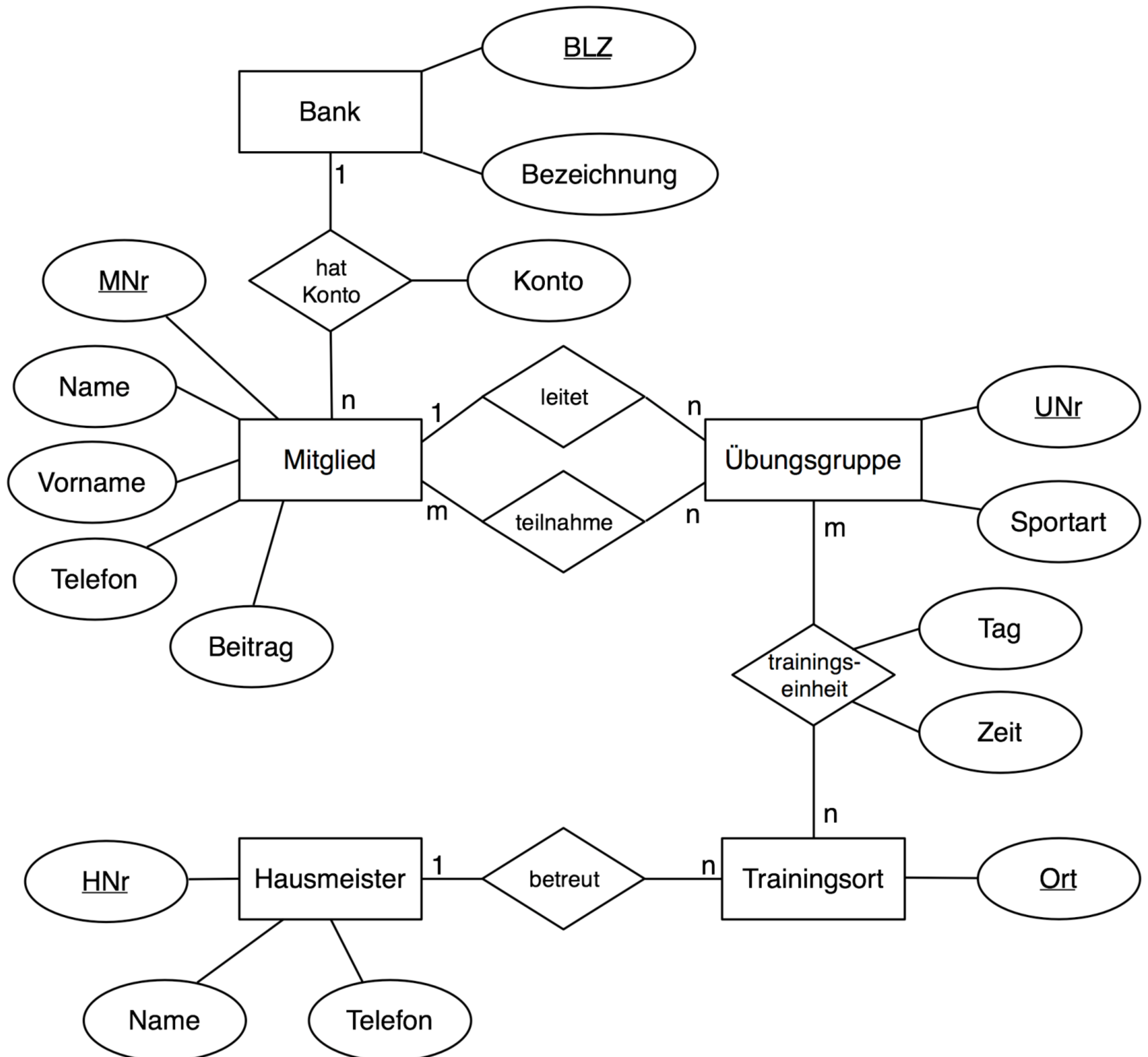
Entwirf eine Datenbank zu folgender Vorgabe:

- Ein Sportverein hat Mitglieder, die eine oder mehrere Sportarten ausüben wollen.
- Dazu sind Übungsgruppen eingerichtet worden, die einen Leiter haben, der ebenfalls ein Mitglied des Sportvereins ist.
- Für jede Übungsgruppe gibt es eine oder mehrere Trainingszeiten pro Woche, zu denen die Gruppe an eventuell verschiedenen Trainingsorten übt. Die Dauer einer Trainingszeit beträgt eine Stunde.
- Hausmeister (Name und Telefon) betreuen einen oder mehrere Trainingsorte.
- Die Mitglieder zahlen einen bestimmten Beitrag und haben deshalb ihre Bankverbindung angegeben. Dabei wird noch das alte Muster Bankname, Bankleitzahl, Kontonummer statt der IBAN verwendet.

Erstelle ein ER-Diagramm mit genauen Komplexitäten!
Entwickele daraus ein Relationenschema für die Datenbank.

Hinweis:
Es ist durchaus möglich, dass zwischen zwei Entitätstypen zwei verschiedene Beziehungen bestehen!

Lösung



Relationenschemata der Datenbank „Sportverein“

mitglied (MNr, Name, Vorname, Telefon, Beitrag)

hat_Konto (MNr, BLZ, Konto)

bank (BLZ, Bezeichnung)

leitet (MNr, UNr)

teilnahme (MNr, UNr)

uebungsgruppe (UNr, Sportart)

trainingseinheit (UNr, Ort, Tag, Zeit)

trainingsort (Ort)

betreut (HNr, Ort)

hausmeister (HNr, Name, Telefon)

Das relationale Datenbankmodell

Bisher liegen die Daten unserer Miniwelt in einem abstrakten Modell vor, das die Realität durch Entitytypen und ihre Beziehungen zueinander widerspiegelt. Das ER-Modell ist – wie wir gesehen haben - in der Phase des konzeptionellen Entwurfs einer Datenbank hervorragend geeignet, ein Datenbanksystem ohne Abhängigkeit von der späteren Implementierung zu gestalten.

Meist ist noch eine Optimierung anhand unterschiedlicher Qualitätskriterien erforderlich.

Das relationale Modell beruht auf einfachen mathematischen Grundlagen:

Mathematische Grundlagen des relationalen Datenbankmodells

Das relationale Datenbankmodell basiert auf dem mathematischen Begriff der *Relation.* Eine Relation R ist eine Teilmenge des kartesischen Produkts einer Liste von Wertbereichen W_1, W_2, ...,W_n:

$R \subseteq W_1 \times W_2 \times ... \times W_n$

Elemente einer Relation R sind die n-Tupel der Art $(v_1,v_2,...,v_n)$ mit $v_i \in W_i$. Die Größe n bezeichnet man als Grad der Relation R.

Im Beispiel der Oberstufenverwaltung kommen Kurse als Objekte vor. Kurse werden durch die drei Attribute *Kurs-Nr, Thema* und *Jahrgangsstufe* beschrieben. Die Kursnummer ist eine natürliche Zahl und das Thema eine Zeichenkette (String). Die Wertemengen sind demnach:

$W_1 = N$
$W_2 = \{x \mid \text{ist String}\}$
$W_3 = \{10/I, 10/II, 11/I, 11/II, 12/I, 12/II\}$
Exemplarisch sind hier vier Kurse K_1 bis K_4 in Form von Tripeln (3-Tupel) angegeben:

K_1 = (12, Abbildungsgeometrie, 12/II)
K_2 = (2, Short Stories, 12/I)
K_3 = (38, Datenbanken, 12/II)
K_4 = (19, Antihelden, 11/II)

Sie bilden die vierelementige Relation
KURS = {(13, Analysis 3, 13/II), (2, Short Stories, 12/I),
(38, Datenbanken, 12/II), (19, Antihelden, 11/II}

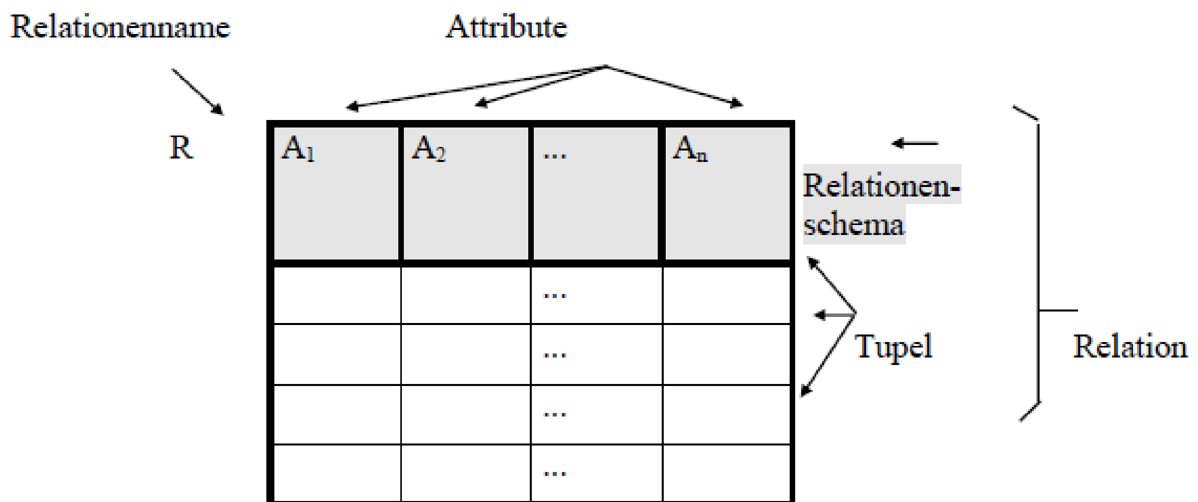
Die Mengenschreibweise ist unübersichtlich. Die dazu äquivalente Tabellen-schreibweise ist besser geeignet und üblich:

KURS

Kurs-Nr	Thema	Jahrgangsstufe
13	Analysis 3	13/II
2	Short Stories	12/I
38	Datenbanken	12/II
19	Antihelden	11/II

In der ersten Zeile der Tabelle werden die jeweiligen Attribute als Spaltenköpfe notiert. Sie gehören somit nicht zur Relation. Vielmehr beschreiben sie das *Relationenschema*, also den hier beschriebenen Entitytyp. Jede weitere Zeile der Tabelle beschreibt ein Tupel der Relation, entspricht also einem Entity.

Der allgemeine Aufbau einer Tabelle im Relationenmodell ist demnach gegeben durch:



Das Relationenschema entspricht dem Entitytyp des ER-Modells. Um über Relationenschemata sprechen zu können, benutzt man die Schreibweise:

Allgemein: $R(A_1, A_2, ..., A_3)$

Beispiel: Kurs(<u>Kurs-Nr</u>, Thema, Jahrgangsstufe)

Schlüsselattribute zeichnet man **durch Unterstreichung** aus.

Bei der Darstellung einer Relation als Tabelle muss man immer beachten, dass die Tabelle lediglich eine übersichtliche Darstellung einer Menge ist.
Dies bedeutet unter anderem, dass in einer Relationentabelle nie zwei gleiche Zeilen vorkommen dürfen und die Reihenfolge der Zeilen keine Bedeutung hat.

Operatoren des Relationenmodells

Mit dem Tabellenkonzept sind wir in der Lage, Entitytypen in das Relationenmodell abzubilden.

Der Abbildung von Beziehungen im ER-Modell werden wir uns später zuwenden.

Um Missverständnissen vorzubeugen, sei betont, dass die *Beziehungen* im ER-Modell nicht dem entsprechen, was wir hier als *Relationen* bezeichnet haben!

Im Folgenden beschäftigen wir uns mit den Operatoren des Relationenmodells, mit denen neben den rein statischen Eigenschaften des ER-Modells auch dynamische Eigenschaften modelliert werden können.

Die Operatoren können auf Relationen (Tabellen) angewendet werden und erzeugen dabei neue Relationen (Tabellen). Wie wir aus der Mathematik wissen, lassen sich Operatoren und Operanden (Relationen) zu komplizierten Ausdrücken verknüpfen, mit denen die Berechnung neuer Relationen beschrieben werden kann. Man spricht daher auch von der *Relationenalgebra*. Sie ist eine präzise Sprache zur Formulierung von Anfragen, die noch unabhängig ist von der Sprache eines bestimmten Datenbankmanagement-systems.

Man kann hier also Dinge noch anfragespracheunabhängig formulieren, egal in welchem DBMS man später implementieren will. Damit *leistet die Relationenalgebra die Verknüpfung unterschiedlicher Tabellen* und damit das

logische wieder zusammen fügen dessen, was aus Modellierungsgründen getrennt wurde.

Operator	Schreibweise	Bedeutung
Durchschnitt	$R \cap S$	Schnittmenge
Vereinigung	$R \cup S$	Vereinigungsmenge
Differenz	$R \setminus S$	Differenz von Mengen
Produkt	$R \times S$	kartesisches Produkt zweier Mengen
Selektion	$\sigma_{Formel}(R)$	Auswahl von Tupel gemäß Formel, Streichung von Zeilen
Projektion	$\pi_{Attribute}(R)$	Auswahl von Attributen, Streichung von Spalten
Join (natural)	$R \bowtie S$	Verknüpfung zweier Relationen zu einer neuen mit den Attributen beider Tabellen über gemeinsames Attribut
Join (Theta)	$R \underset{C}{\bowtie} S$	Verknüpfung zweier Relationen zu einer neuen mit formulierter Bedingung C

Dabei wird zwischen **Sets** und **Bags** unterschieden.

Sind Mehrfachvorkommen eines Tupels (Zeile) erlaubt, so spricht man von Bags (*Eselsbrücke*: Die Ergebnistupel einfach ohne Nachbearbeitung in die „Tasche (bag)" werfen). Werden Duplikate entfernt handelt es sich um Sets. Demnach werden temporär (zeitweilig) erst Bags gebildet, aus denen dann mehrfach vorkommende Elemente entfernt werden, um so als finales Ergebnis ein Set zu erhalten.

Zur Erklärung und Verdeutlichung der Wirkungsweise der Operatoren betrachten wir Beispiele mit folgenden Relationen:

Kurs1

Kurs-Nr	Thema	Jahrgangsstufe
13	Analysis	12/I
25	Short Stories	12/I
3	Datenbanken	11/I

KursLR

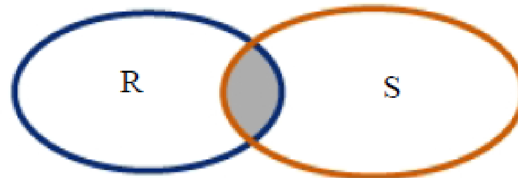
Kurs-Nr	Lehrer	Raum
11	Müller I	123
12	Schulze	124
27	Bauer	14
15	Maier	14
17	Maier	17
3	Zange	211

Kurs2

Kurs-Nr	Thema	Jahrgangsstufe
11	Mechanik I	11/I
12	Mechanik I	11/I
25	Short Stories	12/I
3	Datenbanken	11/I

Durchschnitt

Der Durchschnitt R ∩ S zweier Relationen R und S ist die Menge aller Tupel, die sowohl in R als auch in S enthalten sind.

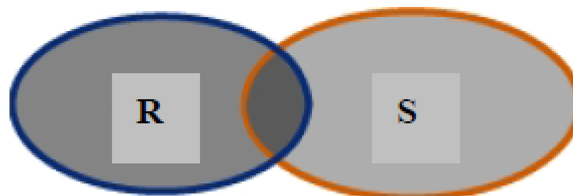


Kurs1 ∩ Kurs2

Kurs-Nr	Thema	Jahrgangsstufe
25	Short Stories	12/I
3	Datenbanken	11/I

Vereinigung

Die Vereinigung R ∪ S zweier Relationen R und S ist die Menge aller Tupel, die in R oder S oder in beiden Relationen enthalten sind.

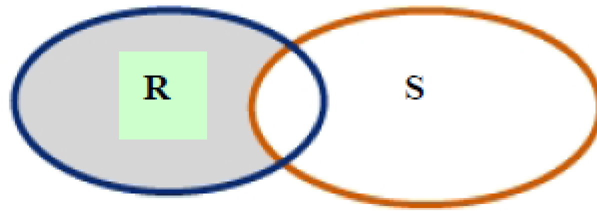


Kurs1 ∪ Kurs2

Kurs-Nr	Thema	Jahrgangsstufe
13	Analysis	12/I
25	Short Stories	12/I
3	Datenbanken	11/I
11	Mechanik I	11/I
12	Mechanik I	11/I

Differenz

Die Differenz R \ S zweier Relationen R und S ist die Menge aller Tupel, die in R aber nicht gleichzeitig in S enthalten sind.



Kurs1 \ Kurs2

Kurs-Nr	Thema	Jahrgangsstufe
13	Analysis	12/I

Kartesisches Produkt - Verbinden zweier Tabellen
Seien R und S Relationen mit Grad n_1 und n_2. Das kartesische Produkt $R \times S$ ist die Menge aller $(n_1 \cdot n_2)$-Tupel, deren ersten n_1 Komponenten ein Tupel in R und deren letzten n_2 Komponenten ein Tupel aus S darstellen. Also wird 'jedes Element mit jedem' gepaart.

Anm.: Wenn Spalten gleichen Namens in den Tabellen auftauchen, dann kann man sie auch über *Tabellenname.Spaltenname* unterscheiden, z.B. Kurs2.Kurs-Nr.

Kurs2 × KursLR

Kurs -Nr	Thema	Jahr-gangsstufe	Kurs-Nr	Lehrer	Raum
11	Mechanik I	11/I	11	Müller 1	123
11	Mechanik I	11/I	12	Schulze	124
11	Mechanik I	11/I	27	Bauer	14
11	Mechanik I	11/I	15	Maier	14
11	Mechanik I	11/I	17	Maier	17
11	Mechanik I	11/I	3	Zange	211
12	Mechanik I	11/I	11	Müller 1	123
12	Mechanik I	11/I	12	Schulze	124
12	Mechanik I	11/I	27	Bauer	14
12	Mechanik I	11/I	15	Maier	14
12	Mechanik I	11/I	17	Maier	17
12	Mechanik I	11/I	3	Zange	211
25	Short Stories	12/I	11	Müller 1	123
25	Short Stories	12/I	12	Schulze	124
25	Short Stories	12/I	27	Bauer	14
25	Short Stories	12/I	15	Maier	14
25	Short Stories	12/I	17	Maier	17
25	Short Stories	12/I	3	Zange	211
3	Datenbanken	11/I	11	Müller 1	123
3	Datenbanken	11/I	12	Schulze	124
3	Datenbanken	11/I	27	Bauer	14

Selektion - Auswahl von Zeilen
Sei F eine Formel. Diese Formel kann Konstanten und Attribute als Operanden, sowie Vergleichsoperatoren ($<, \leq, =, \neq, \geq, >$) und logische Operatoren (and, or, not) enthalten.

Dann ist die Selektion σ_F (R) die Menge aller Tupel in R, die die Formel F erfüllen.

$\sigma_{\text{Kurs-Nr} > 15}$ (KursLR)

Kurs-Nr	Lehrer	Raum
27	Bauer	14
17	Maier	17

Projektion - Auswahl von Spalten
Sei R eine Relation vom Grad n. Dann ist $\pi_{i1}, \pi_{i2}, \ldots, \pi_{im}(R)$ die Abbildung von R auf die Komponenten $i_1, i_2, \ldots, i_m$. Die entstehende Relation ist m-gradig.

$\pi_{\text{Thema,Jahrgangsstufe}}(\text{Kurs1} \cup \text{Kurs2})$

Kurs-Nr	Thema	Jahrgangsstufe
13	Analysis	12/I
25	Short Stories	12/I
3	Datenbanken	11/I
11	Mechanik 1	11/I
12	Mechanik 1	11/I

Duplikate müssen aus der projizierten Tabelle entfernt werden, denn sie muss eine Menge darstellen:

$\pi_{\text{Thema,Jahrgangsstufe}}(\text{Kurs1} \cup \text{Kurs2})$

Thema	Jahrgangsstufe
Analysis	12/I
Short Stories	12/I
Datenbanken	11/I
Mechanik 1	11/I

Join (natürlicher Verbund) - Tabellen verbinden
Der Join R><S der Tabellen R und S wird für Tabellen mit gleichen Attributen

wie folgt berechnet:

Man bildet das kartesische Produkt $R \times S$
Für jedes Attribut, das sowohl in R als auch in S vorkommt, selektiert man die Tupel, für die die Werte der gleichnamigen Attribute übereinstimmen.

Eine der gleichen Spalten wird wegprojiziert.

Als Merkregel formuliert:

Der Join verbindet zwei Tabellen über gleichnamige Spalten bei gleichen Attributwerten.

Der natürliche Verbund ist äußerst wichtig, um Relationen, die aus entwurfstheoretischen Gründen zerlegt wurden, während der Abfrage wieder zu kombinieren. Die Zerlegung findet in der Regel über Schlüsselattribute statt. Dementsprechend findet der Join in aller Regel über ein gemeinsames Schlüsselattribut in den beiden zu verknüpfenden Tabellen statt.

In unserem Beispiel werden durch den Join die Kurs-Informationen mit den Lehrer- und Raum-Informationen verknüpft. So entsteht aus der Kurs-Tabelle und der Lehrer-Raum-Tabelle eine neue Kurs-Lehrer-Raum-Tabelle:

Kurs2 ⊳⊲ KursLR

Kurs-Nr	Thema	Jahrgangsstufe	Lehrer	Raum
11	Mechanik 1	11/I	Müller 1	123
12	Mechanik 1	11/I	Schulze	124
3	Datenbanken	11/I	Zange	211

Zur Berechnung des Join:
Als erstes wird das kartesische Produkt der Ausgangstabellen Kurs2 und KursLR gebildet, die entstehende Tabelle besteht aus 4*6 = 24 Tupeln.

Diejenigen Tupel, die bezüglich des gemeinsamen Attributs *Kurs-Nr* gleiche Werte aufweisen, werden selektiert.

Zuletzt wird die doppelte Spalte *Kurs-Nr* ausprojiziert und das obige Ergebnis hergestellt.

Die folgende Abbildung zeigt diese Vorgehensweise nochmals.

Kurs2 ⋈ KursLR

Kurs-Nr	Thema	Jahrgangsstufe	Kurs-Nr	Lehrer	Raum
11	Mechanik I	11/I	11	Müller 1	123
11	Mechanik I	11/I	12	Schulze	124
11	Mechanik I	11/I	27	Bauer	14
11	Mechanik I	11/I	15	Maier	14
11	Mechanik I	11/I	17	Maier	17
11	Mechanik I	11/I	3	Zange	211
12	Mechanik I	11/I	11	Müller 1	123
12	Mechanik I	11/I	12	Schulze	124
12	Mechanik I	11/I	27	Bauer	14
12	Mechanik I	11/I	15	Maier	14
12	Mechanik I	11/I	17	Maier	17
12	Mechanik I	11/I	3	Zange	211
25	Short Stories	12/I	11	Müller 1	123
25	Short Stories	12/I	12	Schulze	124
25	Short Stories	12/I	27	Bauer	14
25	Short Stories	12/I	15	Maier	14
25	Short Stories	12/I	17	Maier	17
25	Short Stories	12/I	3	Zange	211
3	Datenbanken	11/I	11	Müller 1	123
3	Datenbanken	11/I	12	Schulze	124
3	Datenbanken	11/I	27	Bauer	14
3	Datenbanken	11/I	15	Maier	14
3	Datenbanken	11/I	17	Maier	17
3	Datenbanken	11/I	3	Zange	211

Anwendung relationaler Operatoren

Zur Darstellung eines komplexeren Beispiels zur Relationenalgebra gehen wir von folgenden Tabellen einer Oberstufenverwaltung aus:

Schüler

Schüler-Nr	Name	Vorname	Tutor	Geschlecht
123	Alberti	Hans	Müller	m
034	Glücklich	Gesine	Abel	w
321	Müser	Angelika	Abel	w
111	Weber	Wolfgang	Zange	m

Kurs

Kurs-Nr	Typ	Fach	Thema	Jahrgangsstufe
13	GK	Mathematik	Analysis 2	12/I
11	GK	Physik	Mechanik 1	11/I
03	GK	Informatik	Datenbanken	12/II
25	LK	Englisch	Short Stories	12/I
89	GK	Informatik	Compilerbau	13/II

Besucht

Schüler-Nr	Kurs-Nr	Fehlstunden	Punkte
123	03	00	12
123	25	03	07
321	89	00	14
111	03	21	03

Es soll die Frage beantwortet werden, welche Mädchen Informatikkurse besuchen und welche Punktzahlen sie dabei erreicht haben.

1. Bestimmung der Informatikkurse

Informatikkurse = $\pi_{\text{Kurs-Nr}}(\sigma_{\text{Fach = Informatik}}$ (Kurs))

Kurs-Nr
03
89

2. Join mit der *Besucht*-Tabelle über das gemeinsame Schlüsselattribut *Kurs-Nr* liefert die Informatikschüler

   Informatikschüler = Informatikkurse >< Besucht

Kurs-Nr	Fach	Schüler-Nr	Fehlstunden	Punkte
03	Informatik	123	00	12
03	Informatik	111	21	03
89	Informatik	321	00	14

3. Projektion auf die benötigten Attribute *Schüler-Nr* und *Punkte*:

   InformatikschülerPunkte = $\pi_{\text{Schüler-Nr, Punkte}}$ (Informatikschüler)

Schüler-Nr	Punkte
123	12
111	03
321	14

4. Join mit *Schüler*-Tabelle über das gemeinsame Schlüsselattribut *Schüler-Nr*.

   InformatikschülerPunkteName = InformatikschülerPunkt >< Schüler

Schüler-Nr	Punkte	Name	Vorname	Tutor	Geschlecht
123	12	Alberti	Hans	Müller	m
111	03	Weber	Wolfgang	Zange	m
321	14	Müser	Angelika	Abel	w

Selektion der Mädchen und Projektion auf die geforderten Angaben.

Ergebnis = $\pi_{\text{Punkte,Name,Vorname}}(\sigma_{\text{Geschlecht=w}}(\text{InformatikschülerPunkteName}))$

Punkte	Name	Vorname
14	Müser	Angelika

Aufgabe 1

Gegeben seien drei Relationen mit den folgenden Tupeln:

Besucht

Gast	Bistro
Hans	Uno
Ede	Uno
Ede	Dos
Ede	Chico
Karl	Dos
Karl	Chico
Heini	Uno

Serviert

Bistro	Getränk
Uno	Wasser
Uno	Kaffee
Dos	Kaffee

Mag

Gast	Getränk
Hans	Wasser
Ede	Wasser
Ede	Kaffee
Karl	Kaffee

a) Bilden Sie *Serviert* × *Mag*.

b) Bilden Sie *Serviert* >< *Mag*. Welche Informationen beinhaltet diese Relation?

c) Geben Sie alle Bistros aus, die Getränk servieren, die Karl mag. Überprüfen Sie Ihre Operation in der Relationenalgebra anhand des Beispiels.

d) Geben Sie alle Gäste aus, die mindestens ein Bistro besuchen, die auch das Getränk serviert, das sie mögen. Formulieren Sie die Anfrage mit Operationen der Relationenalgebra.

Lösung

Aufgabe 1

a) Bilden Sie Serviert × Mag.

Bistro	Getränk	Gast	Getränk
Uno	Wasser	Hans	Wasser
Uno	Wasser	Ede	Wasser
Uno	Wasser	Ede	Kaffee
Uno	Wasser	Karl	Kaffee
Uno	Kaffee	Hans	Wasser
Uno	Kaffee	Ede	Wasser
Uno	Kaffee	Ede	Kaffee
Uno	Kaffee	Karl	Kaffee
Dos	Kaffee	Hans	Wasser
Dos	Kaffee	Ede	Wasser
Dos	Kaffee	Ede	Kaffee
Dos	Kaffee	Karl	Kaffee

b) weiß unterlegte Tupel

c) $\pi_{Bistro}\ (\sigma_{Gast = Karl}\ (Serviert \bowtie Mag)$ oder effizienter

$\pi_{Bistro}\ (Serviert \bowtie \pi_{Getränk}\ (\sigma_{Gast = Karl}\ (Mag)))$

1. Tabellen verbinden $\bowtie$
2. auf Karl reduzieren σ
3. Frage beantworten π

d) $\pi_{Gast}\ (Besucht \bowtie (\pi_{Bistro,\ Gast}\ (Serviert \bowtie Mag)$

Aufgabe 2
Gegeben seien folgende Relationen (# ist das Zeichen für *Nummer*):

Lieferanten (L#, LName, Status, Stadt)
Teile (T#, TName, Farbe, Gewicht, Stadt)
Projekte (P#, PName, Stadt)
Lieferungen (L#, T#, P#, Anzahl)

Hierbei bedeutet Stadt einmal die Stadt, in der ein Lieferant sitzt, die Stadt, in der das entsprechende Teil hergestellt wird, bzw. die Stadt, in der ein Projekt stattfindet.

Lösen Sie die folgenden Aufgaben durch Operationen aus der Relationenalgebra:

a) Finde Sie alle Lieferungen mit Anzahlen zwischen 300 und 750 und geben Sie alle dazu in der Relation Lieferungen verzeichneten Informationen aus.

b) Geben Sie alle Städte aus, in denen Lieferanten sitzen.

c) Geben Sie alle vorkommenden Paarungen TName, Stadt aus.

d) Finden Sie alle schwarzen Teile. Geben Sie ihre Nummer und ihren Namen aus.

e) Finden Sie alle Lieferanten, die in einer Einzellieferung mehr als 150 Teile geliefert haben. Geben Sie ihren Namen aus.

LÖSUNG

Aufgabe 2

In vielen Fällen wurde versucht, die Größe der Zwischentabellen durch frühzeitige Projektion oder Selektion zu minimieren.

a) $\sigma_{\text{Anzahl} \geq 300 \text{ und Anzahl} \leq 750}$ (Lieferungen)

b) π_{Stadt} (Lieferanten)

c) $\pi_{\text{TName, Stadt}}$ (Teile)

d) $\pi_{\text{T\#, TName}}$ ($\sigma_{\text{Farbe = schwarz}}$ (Teile))

e) π_{Lname} ($\pi_{\text{L\#}}$ ($\sigma_{\text{Anzahl} > 150}$ (Lieferungen)) $\bowtie$ $\pi_{\text{L\#, Lname}}$ (Lieferanten))

f) $\pi_{\text{T\#}}$ ($\pi_{\text{L\#}}$ ($\sigma_{\text{Stadt = London}}$ (Lieferanten)) $\bowtie$ $\pi_{\text{L\#, T\#}}$ (Lieferungen))

π_{TName} ($\pi_{\text{L\#}}$ ($\sigma_{\text{Stadt = London}}$ (Lieferanten)) $\bowtie$ $\pi_{\text{L\#, T\#}}$ (Lieferungen) $\bowtie$ Teile)

g) π_{Stadt} (Lieferanten $\bowtie$ Projekte)

h) $\pi_{\text{P\#}}$ ($\pi_{\text{P\#, L\#}}$ (Lieferungen) $\bowtie$ $\pi_{\text{P\#, Stadt}}$ (Projekte) $\bowtie$ $\pi_{\text{L\#, Stadt}}$ (Lieferanten))

i) $\pi_{\text{T\#}}$ ($\pi_{\text{L\#}}$ ($\sigma_{\text{Lname = Lux}}$ (Lieferanten) $\bowtie$ $\pi_{\text{L\#, P\#}}$ (Lieferungen)) $\bowtie$ Teile

Aufgabe 3
Gegeben sind die folgenden Tabellen:

Tabelle1

A	B	C
4	2	8
2	2	1
6	7	3

Tabelle2

C	D
8	2
3	6

Tabelle3

B	E
5	3
4	4
5	4

Tabelle4

B	C	D
1	2	1
4	2	1
1	2	9

Führen Sie folgende relationalen Operationen durch und stellen Sie die Ergebnistabelle auf!
Beschreiben Sie die Aufgabenstellung mittels der behandelten Symbolik!

a) Selektion von Tabelle1 mit der Bedingung B=2
b) Projektion von Tabelle3 auf E
c) Join Tabelle1 und Tabelle2 nach dem gemeinsamen Attribut C
d) (Selektion von Tabelle 1 mit B>C) vereinigt mit (Selektion von Tabelle1 mit A<5)

Lösung:

Aufgabe 3

a)

A	B	C
4	2	8
2	2	1

b)

E
3
4

c)

A	B	C	D
4	2	8	2
6	7	3	6

d)

A	B	C
4	2	8
2	2	1
6	7	3

Aufgabe 4
Gegeben seien folgende Tabellen

GK-Fach 1

Raum	Fach	Lehrer
137	Mathematik	Müller
221	Deutsch	Schmidt
104	Englisch	Lehmann

GK-Fach 2

Raum	Fach	Lehrer
127	Informatik	Müller
104	Englisch	Lehmann
123	Physik	Paulsen
018	Musik	Schmidt

Themen

Nr.	Thema	Klasse
001	Analysis	12/I
002	Klassik	13/1

Bilden Sie
a) GK-Fach 1 GK-Fach 2
b) GK-Fach 1 GK-Fach 2
c) GK-Fach 1 \ GK-Fach 2
d) Themen x GK-Fach 2
e) $\sigma_{\text{Name = 'Müller'}}$(Schüler)
f) π Name(Schüler)
g) π Vorname(σ Name
h) Join(Schüler, Kurs)

Lösung:

Aufgabe 4

a) GK-Fach 1 ∩ GK-Fach 2

Der Durchschnitt A ∩ B zweier Tabellen A und B ist die Menge aller Tupel, die sowohl in A als auch in B enthalten sind.

Raum	Fach	Lehrer
104	Englisch	Lehmann

b) GK-Fach 1 ∪ GK-Fach 2

Die Vereinigung A ∪ B zweier Tabellen A und B ist die Menge aller Tupel, die in A oder in B oder in beiden Relationen enthalten sind.

GK-Fach 1

Raum	Fach	Lehrer
137	Mathematik	Müller
221	Deutsch	Schmidt
104	Englisch	Lehmann

GK-Fach 2

Raum	Fach	Lehrer
127	Informatik	Müller
104	Englisch	Lehmann
123	Physik	Paulsen
018	Musik	Schmidt

→ GK-Fach 1 ∪ GK-Fach 2

Raum	Fach	Lehrer
137	Mathematik	Müller
221	Deutsch	Schmidt
104	Englisch	Lehmann
127	Informatik	Müller
123	Physik	Paulsen
018	Musik	Schmidt

c) GK-Fach 1 \ GK-Fach 2

Die Differenz A \ B zweier Tabellen A und B ist die Menge aller Tupel, die in A aber nicht gleichzeitig in B enthalten sind.

GK-Fach 1

Raum	Fach	Lehrer
137	Mathematik	Müller
221	Deutsch	Schmidt
104	Englisch	Lehmann

GK-Fach 2

Raum	Fach	Lehrer
127	Informatik	Müller
104	Englisch	Lehmann
123	Physik	Paulsen
018	Musik	Schmidt

→ GK-Fach 1 \ GK-Fach 2

Raum	Fach	Lehrer
137	Mathematik	Müller
221	Deutsch	Schmidt

d) Themen x GK-Fach 2

Das Produkt A x B zweier Tabellen A vom Grad n und B vom Grad m ist die

Menge aller Kombinationstupel (n x m-Tupel).

Themen

Nr.	Thema	Klasse
001	Analysis	12/I
002	Klassik	13/1

GK-Fach 2

Raum	Fach	Lehrer
127	Informatik	Müller
104	Englisch	Lehmann
123	Physik	Paulsen
018	Musik	Schmidt

Nr.	Thema	Klasse	Raum	Fach	Lehrer
001	Analysis	12/I	127	Informatik	Müller
001	Analysis	12/I	104	Englisch	Lehmann
001	Analysis	12/I	123	Physik	Paulsen
001	Analysis	12/I	018	Musik	Schmidt
002	Klassik	13/1	127	Informatik	Müller
002	Klassik	13/1	104	Englisch	Lehmann
002	Klassik	13/1	123	Physik	Paulsen
002	Klassik	13/1	018	Musik	Schmidt

e) $\sigma_{\text{Name = 'Müller'}}$(Schüler)

f) π_{Name}(Schüler)
Bei der Projektion werden Spalten aus einer Tabelle ausgewählt, die bestimmten Eigenschaften genügen. Ist ein Eintrag mehrfach vorhanden, so wird er nur einmal angezeigt.

Schüler

SNr	Vorname	Name
4711	Paul	Müller
0815	Erich	Schmidt
7472	Sven	Lehmann
1234	Olaf	Müller
2313	Jürgen	Paulsen

π Name(Schüler)

Name
Müller
Schmidt
Lehmann
Paulsen

g) $\pi_{\text{Vorname}}(\sigma_{\text{Name = 'Müller'}}(\text{Schüler}))$
Aus der Tabelle Schüler sollen die Vornamen aller Schüler angezeigt werden, deren Nachname Müller ist. Die Abfrage hat also die Form:
$\pi_{\text{Vorname}}(\sigma_{\text{Name = 'Müller'}}(\text{Schüler}))$

Schüler

SNr	Vorname	Name
4711	Paul	Müller
0815	Erich	Schmidt
7472	Sven	Lehmann
1234	Olaf	Müller
2313	Jürgen	Paulsen

π Vorname(σ Name 'Müller'(Schüler))

Vorname
Paul
Olaf

h) Join(Schüler, Kurs)
Ein Join ist das Verbinden von zwei Relationen zu einer neuen Tabelle.
Die Literatur unterscheidet eine größere Anzahl Joins, für die Schule sind nur die mit

Schüler

SNr	Vorname	Name
4711	Paul	Müller
0815	Erich	Schmidt
7472	Sven	Lehmann
1234	Olaf	Müller
2313	Jürgen	Paulsen

Kurse

SNr	KNr	Fehlstunden	Punkte
0815	03	0	12
4711	03	12	03
1234	23	3	14
0987	09	9	09

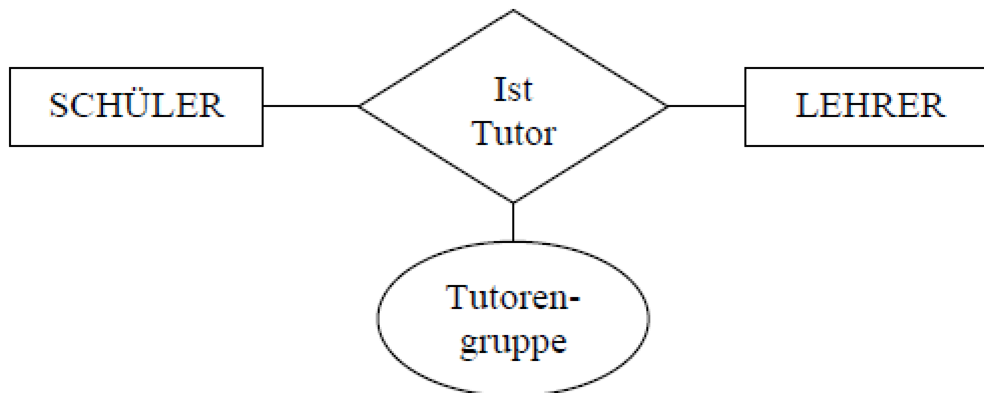
Join_{SNr}(Schüler, Kurs)

SNr	Vorname	Name	KNr	Fehlstunden	Punkte
0815	Erich	Schmidt	03	0	12
4711	Paul	Müller	03	12	03
1234	Olaf	Müller	23	3	14

Da zu den SNr.7472, 2314 und 0987 der Tabelle Schüler bzw. Kurse keine "Gegenstücke" in der anderen Tabelle existieren, werden diese in der neuen Relation nicht angezeigt.

Übertragung des ER-Modells in das relationale Modell

Bei der Einführung des relationalen Modells haben wir schon gesehen, wie Entitytypen auf Relationenschemata abgebildet werden. Wir wenden uns nun dem Problem zu, Beziehungstypen in das relationale Modell abzubilden. Dazu betrachten wir als Beispiel die *Ist-Tutor*-Beziehung zwischen Schüler und Lehrer:



Schüler

<u>SNr</u>	**Name**	**Vorname**	**Geschlecht**	**….**
…	…	…	…	…
214	Meier	Willi	m	
215	Mors	Tanja	w	
216	Müller	Sabine	w	
…	…	…	…	…

Lehrer

<u>LNr</u>	**Name**	**Vorname**	**Fach 1**	**….**
…	…	…	…	…
41	Kapphaus	Hubert	Physik	
42	Kortmann	Franz	Sport	
43	Pleuder	Gabi	Pädagogik	
…	…	…	…	…

<u>Grundregel</u>:

Jeder <u>Beziehungstyp</u> wird in eine eigene Tabelle abgebildet. Die Attribute der Tabelle sind die Primärschlüssel der beiden beteiligten Entitytypen zusätzlich der beziehungseigenen Attribute.

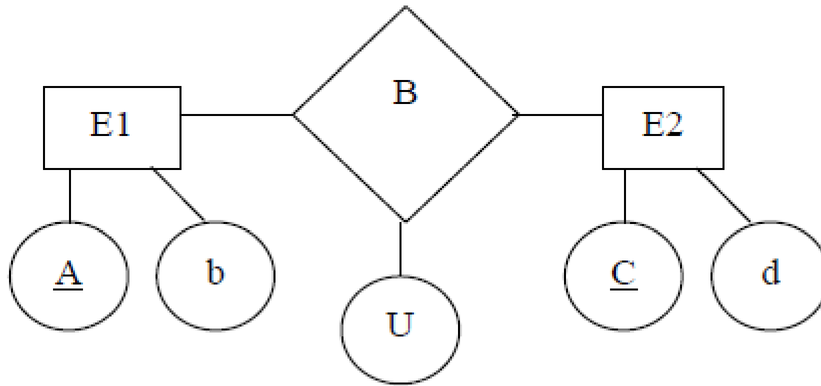
IstTutor

LNr	<u>SNr</u>	Tutorengruppe
…	…	…
41	267	Unterstufe
41	315	Unterstufe
43	105	Oberstufe
…	…	…

Die *Ist-Tutor*-Beziehung wird demnach übersetzt in die Tabelle:
IstTutor(Lehrer-Nr, <u>Schüler-Nr</u>, Tutorengruppe)
Die Lehrernummer kann kein Schlüssel dieser neuen Relation sein, denn zu jeder Lehrernummer gibt es verschiedene Schülernummern. Da jeder Schüler **genau einen Tutor** hat, ist hingegen die Schülernummer ein Schlüssel der neuen Relation.

Wir sehen, dass die Schlüsselwahl letztlich von der Komplexität der Beziehung abhängt. **Im Beispiel ist die Komplexität n:1, also wird der Primärschlüssel der n-Seite zum Primärschlüssel der Beziehungsrelation.**

Bei einer n:m-Beziehung, beispielsweise *Lehrer unterrichtet Schüler*, bilden die Primärschlüssel der beiden Entityrelationen zusammen den Primärschlüssel der Beziehungsrelation. **Bei einer 1:1-Beziehung wählt man einen der beiden Primärschlüssel zum Primärschlüssel der Beziehungsrelation.**



ER-Diagramm Tranformationsregeln

Hier erkennen wir noch mal allgemein, wie ein ER-Diagramm in das relationale Modell umgewandelt wird. Es sind die beiden Entities E_1 und E_2 aufgeführt, mit ihren Attributen A, b und C, d, wobei A und C den Primärschlüssel der jeweiligen Entity darstellen.

Die beiden Entities sind über die Beziehungsrelation B verknüpft, die ihrerseits das Attribut U besitzt. Entsprechend ergibt sich nach dem Transformations-schema folgende Tabellenstruktur im relationalen Modell:

E_1($\underline{A}$, b)
E_1BE_2(A, C, U)
E_2($\underline{C}$, d)

Präzisierung der Grundregel:
Eine ER-Beziehung zwischen den Entitytypen $E_1(\underline{a_1}, a_2, ..., a_n)$ und $E_2(\underline{b_1}, b_2, ..., b_m)$ mit den Primärschlüsseln a_1 und b_1 sowie den Beziehungs-attributen $B(c_1, c_2, ..., c_k)$ wird auf die Beziehungsrelation $BE_{12}(a_1, b_1, c_1, ..., c_k)$ abgebildet. Der Schlüssel der Beziehungsrelation wird nach folgender Tabelle gebildet:

Komplexität der Beziehung	Schlüssel der Beziehungsrelation
1 : 1	a_1 oder b_1
1 : n	b_1
n : m	a_1 und b_1

Optimierungen

Die Grundregel kann immer angewendet werden. Sie liefert unabhängig von der Komplexität der Beziehung stets drei Tabellen für eine binäre Beziehung.

Es gibt Sonderfälle, in denen eine ER-Beziehung zwischen zwei Entitytypen auf lediglich zwei Relationen oder gar nur eine Relation abgebildet werden können.

Wir werden gleich zeigen, dass diese Sonderfälle bei 1:n-Bezehungen (und damit auch bei 1:1-Beziehungen) auftreten **können**, nicht aber bei n:m-Beziehungen. Letztere müssen immer auf zwei 1:n-Beziehungen aufgebrochen werden, da konkrete DBMS keine n:m-Beziehungen direkt darstellen können.

Das bisher benutzte Komplexitätsmaß ist zu grob, um eine Entscheidung darüber treffen zu können, ob man mit weniger Tabellen auskommt. Der Beziehungstyp muss genauer untersucht werden. Dazu betrachten wir die folgenden Relationenschemata etwas genauer:

Schüler(Schüler-Nr, Name,Vorname, Geburtsdatum, Adresse, Schuleintrittsdatum)

IstTutor(Schüler-Nr, Lehrer-Nr, Tutorgruppe)

Lehrer(Lehrer-Nr, Name, Vorname, Titel, Fächer)

In der *Ist-Tutor*-Beziehung können die zwei Tabellen für *Schüler* und *IstTutor* problemlos **über ihren gemeinsamen Schlüssel** zu einer einzigen Tabelle namens *Schüler-Tutor* zusammengefasst werden:

Schüler

Schüler-Nr	Name	Vorname	….
123	Alberti	Hans	…
034	Glücklich	Gesine	
321	Müser	Angelika	
111	Weber	Wolfgang	

IstTutor

Schüler-Nr	Lehrer-Nr	Tutorengruppe
123	42	LK/Bio
034	05	LK/Ph
321	37	LK/D
111	42	LK/Bio

Schüler-Tutor

<u>Schüler-Nr</u>	Name	Vorname	….	Lehrer-Nr	Tutorengruppe
123	Alberti	Hans	…	42	LK/Bio
034	Glücklich	Gesine		05	LK/Ph
321	Müser	Angelika		37	LK/D
111	Weber	Wolfgang		42	LK/Bio

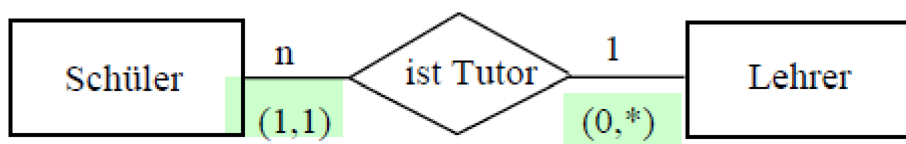
Die Zusammenfassung funktioniert, weil ein Schüler <u>**genau einen**</u> Lehrer als Tutor hat. In der Beziehung *IstTutor* kommt jede Schülernummer **genau einmal** vor. Deshalb haben die Tabellen *Schüler* und *IstTutor* auch denselben Schlüssel.

Bei den Lehrernummern ist das anders. Lehrernummern können in der Tabelle *IstTutor* keinmal, einmal oder mehrmals vorkommen. Keinmal kommt sie für Lehrer ohne Tutorengruppe vor, mehrmals für Lehrer mit Tutorengruppe.

Ausgehend von dieser Beobachtung wird die Komplexität einer Beziehung in der Theorie auch durch Angabe von **minimaler und maximaler Anzahl**, mit der jeweils ein Objekt eines Entitytyps in einer Beziehung vorkommt, angegeben. Ein Schüler kommt in der Beziehung *IstTutor* minimal und maximal einmal vor - Schreibweise (1, 1).

Ein Lehrer muss in der Beziehung nicht, kann aber mehr als einmal vorkommen - Schreibweise (0, *). Man sagt auch, dass Schüler ***obligatorisch*** (zwingend) in der Relation vorkommen, Lehrer hingegen ***optional***.

Über die ***obligatorische*** Mitgliedschaft an einer Relation müssen in den Geschäftsregeln während des konzeptionellen Entwurfs Aussagen gemacht werden.



Anmerkung:
Es gibt noch andere Schreibweisen. Falls man auf diese stoßen sollte, dazu hier eine Übersicht:

Unsere Notation	Bedeutung	alternative Notation	Bedeutung
[1, 1]	genau 1	1	1
[0, 1]	0 oder 1	c	<u>c</u>an
[1, *]	mindestens 1	m	<u>m</u>ultiple
[0, *]	keiner oder beliebig viele	mc	<u>m</u>ultiple <u>c</u>an

Als Beispiel einer **nicht obligatorischen** Beziehung betrachten wir aus dem ER-Diagramm des Bibliotheksystems die Beziehung:

Ein Ausleiher kommt in dieser Beziehung entweder gar nicht oder einmal oder mehrmals vor (d.h. er kann minimal 0, maximal mehr als ein Buch ausleihen) - Schreibweise (0, *). Ein Ausleihbuch kommt entweder gar nicht oder nur einmal vor (d.h. es kann minimal 0, maximal 1-mal ausgeliehen sein - Schreibweise (0,1).

Ausleihbuch

Inventar-Nr	Buchtyp-Nr	Ausleihzeit
5201000593	3400	180
5201000693	3400	270
5201000793	3500	90
5201000893	3500	180

Ausleiher

Ausleiher-Nr	Name	Vorname
101	Lächa	Roland
204	Möller	Melchior
236	Schulz	Sieglinde
103	Meier	Baltasar

leiht aus

Inventar-Nr	Ausleiher-Nr	Ausleihdatum
5201000593	101	12.07.96
5201000793	204	13.08.96
5201000893	101	29.08.96

Wenn man jetzt versucht, die *leiht-aus*-Tabelle in die *Ausleihbuch*-Tabelle zu integrieren, so entstehen bei nicht ausgeliehenen Büchern Nullwerte in der verbundenen Tabelle:

Ausleihbuch-leiht aus

Inventar-Nr	Buchtyp-Nr	Ausleihzeit	Ausleiher-Nr	Ausleihdatum
5201000593	3400	180	101	12.07.96
5201000693	3400	270	NULL	NULL
5201000793	3500	90	204	13.08.96
5201000893	3500	180	101	29.08.96

Nullwerte sollten bei einem Datenbankentwurf möglichst vermieden werden. Sie sind Ursache unnötiger Komplikationen. Beispielsweise müsste geklärt werden, was das Ergebnis der beiden Selektionen

$\sigma_{\text{Ausleiher-Nr}<200}$ (Ausleihbuch-Ausleiher) und
$\sigma_{\text{Ausleiher-Nr}\geq 200}$ (Ausleihbuch-Ausleiher)
und insbesondere das Ergebnis der Mengenvereinigung
$\sigma_{\text{Ausleiher-Nr}<200}$ (Ausleihbuch-Ausleiher) $\cup$ $\sigma_{\text{Ausleiher-Nr}\geq 200}$ (Ausleihbuch-Ausleiher)
ist. Ergibt die Mengenvereinigung die ursprüngliche Tabelle oder nicht?

Die Ursache dieser Komplikationen liegt in den Nullwerten, welche dann in der verbundenen Tabelle entstehen, wenn die n-Seite optional, also nicht obligatorisch an der Beziehung beteiligt ist.

Regel 1 für 1:n-Beziehungen

Ist bei einer ER-Beziehung B der Komplexität 1:n zwischen den Entitytypen E1 und E2 der Entitytyp E2 **obligatorisch** in der Beziehungsrelation B, so können die Relationenschemata von E2 und B zu einem Schema verbunden werden. Zur Abbildung der Beziehung benötigt man dann nur zwei Tabellen.

$E_1(\underline{a_1}, a_2, \ldots, a_n)$ $\quad$ $E_1(\underline{a_1}, a_2, \ldots, a_n)$

$B(\underline{a_1}, \underline{b_1}, c_1, c_2, \ldots, c_k) \Rightarrow$

$E_2(\underline{b_1}, b_2, \ldots, b_m)$ $\quad$ $E_2B(\underline{b_1}, b_2, \ldots, b_m, a_1, c_1, c_2, \ldots, c_k)$

Im Sonderfall, dass die Beziehungsrelation keine eigenen Attribute aufweist, reduziert sich die Abbildung einer 1:n-Beziehung darauf, dass man das Schlüsselattribut a_1 von E_1 in das Relationsschema von E_2 aufnimmt. Ein Join

über das in beiden Tabellen enthaltene Schlüsselattribut a_1 kann die Tabellen E_1 und E_2B miteinander verbinden.

Betrachten wir nun Beziehungen der Komplexität 1:1 am Beispiel von Spinden in der Schule.

Nach der Grundregel erhalten wir drei Relationenschemata:
Schüler(Schüler-Nr, Name, Vorname)
Besitzt(Schüler-Nr, Spind-Nr)
Spind(Spind-Nr, Standort)

Haben alle Schüler einen Spind, sind also Schüler **obligatorisch** an der *Besitzt*-Beziehung beteiligt, so kann wie oben die *Besitzt*-Relation mit der *Schüler*-Relation verbunden werden.

Schüler(Schüler-Nr, Name, Vorname, Spind-Nr)
Spind(Spind-Nr, Standort)

Hätte auch nur ein Schüler keinen Spind (***optionale*** Beziehung), so würde bei diesem Schüler ein Nullwert als Spindnummer auftreten. Die beiden Tabellen dürften nicht verbunden werden.

Nehmen wir an, es gibt wenige Spinde und viele Schüler, die Spinde wären also obligatorisch an der Beziehung beteiligt. In diesem Fall könnte man die *Besitzt*-Relation mit der *Spind*-Relation verbinden, ohne dass Nullwerte entstehen:

Schüler(Schüler-Nr, Name, Vorname)
Spind(Spind-Nr, Standort, Schüler-Nr))

Wenn bei einer 1:1-Beziehung beide Entitytypen obligatorisch an der Beziehung teilnehmen, zum Beispiel in der Beziehung *Klasse hat Klassenlehrer* (unter der unüblichen Voraussetzung, dass jeder Lehrer auch Klassenlehrer ist), **so können die drei Relationen zu einer einzigen Relation verbunden werden:**

```
Klasse(KName,KRaum)
                                                    KlasseLehrer(Lehrer-Nr,    Name,
hat(KName,Lehrer-Nr)                  ⇒             Vorname, KName, KRaum)

Klassenlehrer(Lehrer-Nr, Name, Vorname)
```

Der Entschluss, sie dennoch separat aufzunehmen, könnte z.B. daher rühren, dass man einen Teil der Daten evtl. für sensibel hält – z.B. das Gehaltsfeld -, während der andere Teil – z.B. die Adresse – für eine breitere Klientel nutzbar sein soll.

Regel für 1:1-Beziehungen

Ist bei einer ER-Beziehung der Komplexität 1:1 zwischen den Entitytypen E_1 und E_2 **einer der beiden** Entitytypen E' obligatorisch in der Beziehungsrelation B, so können die Relationenschemata von E' und B zu einem Schema verbunden werden. Zur Abbildung der Beziehung benötigt man nur zwei Tabellen.

Sind **beide** Entitytypen obligatorisch in der Beziehungsrelation, so reicht **ein einziges gemeinsames** Schema für die ER-Beziehung aus:

$E_1(\underline{a_1}, a_2, ..., a_n)$

$B\ (a_1, b_1, c_1, c_2, ..., c_k) \Rightarrow E_1BE_2(\underline{a_1}, a_2, ..., a_n, b_1, b_2, ..., b_m, c_1, c_2, ..., c_k)$

$E_2(\underline{b_1}, b_2, ..., b_m)$

Beziehungen der Komplexität n:m können immer nur auf drei Relationenschemata abgebildet werden.

Die beiden hier beteiligten Entitytypen werden auf je ein Relationenschema abgebildet, die Grundregel gibt an, wie die Beziehung abgebildet wird. Es ist nicht möglich, zwei der drei entstehenden Tabellen zu verbinden, ohne dass Nullwerte entstehen würden.

Es lohnt sich, die Beziehung zwischen den Entitytyp- und Beziehungsschemata genauer zu betrachten. Die ER-Beziehung *Lehrer unterrichtet Schüler*

ergibt die Relationen

Lehrer(Lehrer-Nr, Name, ...)
unterrichtet(Lehrer-Nr, Schüler-Nr)
Schüler(Schüler-Nr, Name, ...)

mit Tabellen der Art

Lehrer

Lehrer-Nr	Name	...
17	Maier	
68	Schulze	
25	Bauer	

unterrichtet

Lehrer-Nr	Schüler-Nr
17	123
17	034
68	123
68	111
25	111

Schüler

Schüler-Nr	Name	
123	Alberti	
034	Glücklich	
321	Müser	
111	Weber	

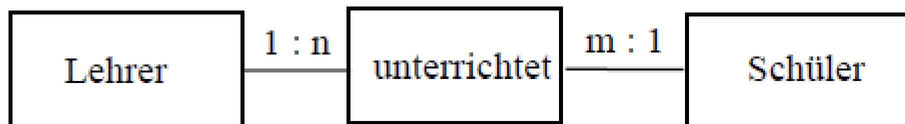
1:n n:1

Zwischen der Tabelle *Lehrer* und der Tabelle *Schüler* gibt es eine Beziehung. Diese Beziehung wird ja gerade in der Tabelle *unterrichtet* dargestellt. Interessanterweise gibt es demzufolge auch eine (hier 1:n-) Beziehung zwischen der Tabelle *Lehrer* und der Tabelle *unterrichtet*, denn jeder Lehrer unterrichtet viele Schüler. Diese Beziehung ist auf der n-Seite obligatorisch, weil ein Lehrer, der einen Schüler unterrichtet, per Konstruktion in der *Lehrer*-Tabelle enthalten

ist.

Analoges gilt für die Beziehung zwischen Paaren der *unterrichtet*-Tabelle und Schülern. Hier besteht eine obligatorische n:1-Beziehung. Man kann also sagen:

Regel für n:m-Beziehungen
Eine n:m-Beziehung kann auf je eine obligatorische 1:n- und m:1-Beziehung aufgeteilt werden.



Diese Regel ist bei der Implementierung eines Datenbanksystems mit einem realen DBMS hilfreich, weil *kommerzielle Systeme lediglich 1:1- und 1:n-Beziehungen unterstützen*!

Sonderfälle

Man muss sich noch Gedanken über die Umsetzung von strukturierten Attributen (Schüleradresse), Mehrfachattributen (Autor beim Buchtyp, Fach beim Lehrer) und *is-a*-Beziehungen machen.

Strukturierte Attribute wie zum Beispiel *Adresse* können einfach durch Übernahme der Teilattribute *Postleitzahl*, *Wohnort*, *Straße* und *Hausnummer* in Tabellen abgebildet werden.

Der Versuch, **Mehrfachattribute** in das Entity-Relationenschema aufzunehmen, scheitert. Entweder man erhält nicht atomare Attributwerte, welche ähnlich wie Nullwerte bei der Selektion Probleme bereiten, oder redundante Wiederholungsgruppen:

nicht atomare Attributwerte

Lehrer -Nr	Name	Fach
17	Maier	{M, Ph, Inf}
68	Schulze	{D, E}
25	Bauer	{Spo}

…

Wiederholungsgruppen

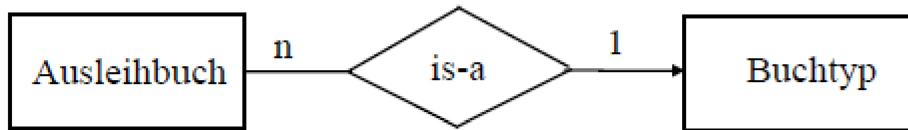
Lehrer-Nr	Name	Fach
17	Maier	M
17	Maier	Ph
17	Maier	Inf
68	Schulze	D
68	Schulze	E
25	Bauer	Spo

Zur Abbildung von **Mehrfachattributen** auf das relationale Modell benötigt man eine zweite Tabelle, welche aus dem Primärschüsselattribut des Entitytyps und dem Mehrfachattribut besteht:

Lehrerfach

Lehrer-Nr	Fach
17	M
17	Ph
17	Inf
68	D
68	E
25	Spo

Is-a-Beziehungen sind stets obligatorische 1:1-Beziehungen (Schüler is-a Ausleiher) oder n:1-Beziehungen (Ausleihbuch is-a Buchtyp). Nach den Regeln 1 und 2 reicht es daher, **in das Relationenschema des spezielleren Entitytyps zusätzlich den Primärschlüssel des allgemeineren Entitytyps aufzunehmen**.



spezieller: Ausleibuch *Allgemeiner*: Buchtyp

Also: Primärschlüssel von 'Buchtyp' (die *Buchtyp-Nr*) zusätzlich in das Relationsschema von 'Ausleihbuch' aufnehmen.

Buchtyp

Buchtyp-Nr	Titel	E-Jahr	Verlag	...
1000	MS Access Benutzerhandbuch	1992	Microsoft	
1001	MS Access Sprachverzeichnis	1992	Microsoft	
1004	MS Access Graph	1993	Microsoft	
1005	MS-Excel 4.0 Schnellübersicht	1992	Markt+Technik	
1006	Multiplan	1986	Hanser	
1007	Quattro Pro für Windows	1993	Borland	
1202	Microsoft Word für Windows	1992	Microsoft	

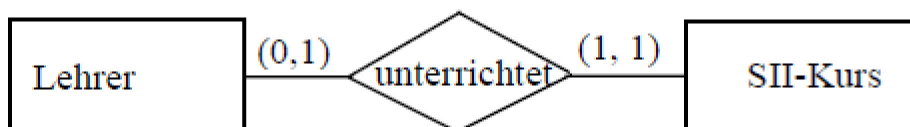
Ausleihbuch

Inventar-Nr	**Buchtyp-Nr**	Ausleihzeit
5201000593	1004	180
5201000693	1006	270
5201000793	1202	0
5201000893	1007	180

Aufgaben zur Umsetzung in das relationale Modell

Aufgabe 1

Was bedeuten die folgenden vier Diagramme (Prosa-Erklärung)? Die Attribute wurden zur Vereinfachung weggelassen. Geben Sie jeweils die Komplexität in der Form 1:1, 1:n bzw. n:m an.



Lösung

Aufgabe 1

a) Jeder Lehrer hat mindestens einen Kurs, und jeder Kurs hat genau einen Lehrer. 1:n

b) Es gibt auch Kurse ohne Lehrer und Lehrer ohne Kurs. Ein Kurs hat höchstens einen Lehrer. 1:n

c) Jeder Lehrer hat mindestens einen Kurs, und ein Kurs kann mehrere Lehrer haben. n:m

d) Jeder Lehrer hat höchstens einen Kurs, und ein Kurs hat genau einen Lehrer. 1:1

Normalisierung

Es gibt noch eine andere Möglichkeit, um auf mehr formalem Weg die benötigten Tabellen mit ihren Beziehungen zu konstruieren. Er führt über die so genannte *Normalisierung*.
Codd hat 1972 drei Normalisierungsregeln angegeben die zur 1. Normalform, zur 2. und 3. Normalform einer Relation (Tabelle) führen. Verschiedene Autoren haben später weitere Regeln und Normalformen hinzugefügt, die aber in der Praxis keine große Rolle spielen.

Unter *Normalisierung* versteht man das Aufteilen aller relevanten Daten in Relationen in der Art und Weise, dass sie am Ende den Normalisierungsregeln entsprechen. Hauptgründe für die Normalisierung sind:

- die Vermeidung unerwünschter Anomalien beim Einfügen, Löschen und Verändern, die zu einer Inkonsistenz der Daten führen.

- die Vermeidung von überflüssiger Information (Redundanz).

- der Zwang zum systematischen Entwurf der Datenbank.

- die bessere Übersichtlichkeit für Benutzer und Programmierer.

Wir wollen uns am Beispiel eines Personalinformationssystems, das die relevanten Daten der Mitarbeiter einer Firma verwaltet, die möglichen Anomalien verdeutlichen.

Die drei Tabellen bilden die Entity-Typen *Mitarbeiter* und *Projekt* sowie den Beziehungs-Typ *arbeitet an* ab:

Mitarbeiter

PersNr	Name	Vorname	Beruf	Abteilung	AbtNr
123	Abt	Josef	Buchhalter	Zentrale	4
234	Müller	Lieselotte	EDV-Dozentin	EDV	6
287	Schulz	Eva	Operator	EDV	2

Projekt

ProjNr	Projekt	Leiter
5	DV2000	Müller
5	DV2000	Müller
1	Kosten	Abt

Arbeitet an

ProjNr	PersNr	Telefon	Zeit
5	234	5432	100%
5	287	5433	50%
1	123	65421	100%

Aufgabe: Was ist an folgenden Szenarien im Zusammenhang mit obiger Datenbank problematisch?

a) Eine neue Mitarbeiterin wird eingestellt. Sie erhält natürlich einen eigenen Schreibtisch mit eigenem Telefonanschluss. Da sie sich erst einarbeiten muss, wird sie noch keinem Projekt zugeordnet.

b) Ein Projekt wird abgeschlossen und kann aus der Datenbank entfernt werden.

c) Eine Mitarbeiterin, die gleichzeitig Projektleiterin ist, ändert durch Heirat ihren Namen.

Lösung:

a) **Einfüge-Anomalie:**
Die Telefonnummer der neuen Mitarbeiterin kann nicht gespeichert werden.

b) **Lösch-Anomalie:**
Die Informationen über die Telefonnummern gehen verloren.

c) **Änderungs-Anomalie**:
Der Name wird in der Tabelle *Mitarbeiter* geändert, verbleibt aber in der Tabelle *Projekt* unverändert.

Um solche Anomalien und Redundanzen zu vermeiden, werden die Normalisierungsregeln angewandt. Man geht dabei von allen relevanten Daten aus und stellt sie in einer Tabelle zusammen. In einer nicht normalisierten Tabelle gibt es meist Wiederholungsgruppen (Mehrfachattribute) und Redundanzen.

PersNr	**Name**	**AbtNr**	**ProjNr**
123	Müller	5	3, 6

Wenn eine Zelle nichtatomare Informationen enthält, könnte man einfach die entsprechende Zeile vervielfältigen:

PersNr	**Name**	**AbtNr**	**ProjNr**
123	Müller	5	3
123	Müller	5	6

Allerdings ist jetzt der alte Schlüssel kein Schlüssel mehr. Er muss durch ein weiteres Attribut zu einem Schlüssel ergänzt werden.

Deshalb werden Wiederholungsgruppen üblicherweise in eine neue Relation ausgegliedert:

PersNr	**Name**	**AbtNr**
123	Müller	5

PersNr	**ProjNr**
123	3
123	6

Die so bereinigte Relation befindet sich dann in der *1. Normalform.*

1. Normalform
Eine Relation befindet sich in der ersten Normalform, wenn an allen Kreuzungspunkten von Zeilen und Spalten nur atomare Werte auftreten.

Es gibt also in der 1.Normalform *keine Wiederholungsgruppen* mehr. Diese können formal entweder durch mehrere Zeilen aufgelöst werden oder gleich – was geschickter ist – durch Ausgliederung in eine eigene Tabelle, verbunden über den Primärschlüssel der ursprünglichen Tabelle.

Beispiel:

Rgnr	Name	Vorname	Straße	Nr	Plz	Ort	**ArtNr1**	**ArtBez1**	***ArtNr2***	***ArtBez2***
1	Peters	Klaus	Xyz Gasse	1	33333	Testingen	12345	Wasser	45678	Limo
2	Schulze	Fried	Abc Gasse	4	22222	Mohingen	99999	Bier	44444	Sekt

wird aufgelöst, indem mehrere Zeilen für die Wiederholungsgruppe spendiert werden:

Rgnr	Name	Vorname	Straße	Nr	Plz	Ort	**ArtNr**	**ArtBez**
1	Peters	Klaus	Xyz Gasse	1	33333	Testingen	12345	Wasser
3	Peters	Klaus	Xyz Gasse	1	33333	Testingen	45678	Limo
2	Schulze	Fried	Abc Gasse	4	22222	Mohingen	99999	Bier
4	Schulze	Fried	Abc Gasse	4	22222	Mohingen	44444	Sekt

oder besser durch eine eigene Tabelle, in der die Wiederholungsgruppe sowie der Primärschlüssel der Ursprungstabelle mit aufgenommen werden.

Rgnr	Name	Vorname	Straße	Nr	Plz	Ort
1	Peters	Klaus	Xyz Gasse	1	33333	Testingen
2	Schulze	Fried	Abc Gasse	4	22222	Mohingen

Rgnr	ArtNr	ArtBez
1	12345	Wasser
3	45678	Limo
2	99999	Bier
4	44444	Sekt

Dabei ist die neue Tabelle über den Primärschlüssel *Rgnr* mit ihrer Ursprungstabelle verbunden. Zu beachten ist, dass allerdings nun der alte Primärschlüssel, sprich *Rgnr*, nicht mehr eindeutig ist, denn es können ja mehrere Artikel in ein und derselben Rechnung erscheinen.

Entsprechend muss hier zu einem zusammengesetzten Primärschlüssel übergegangen werden!

Die folgende Tabelle eines Personalinformationssystems befindet sich in der 1. Normalform (es sind der Übersichtlichkeit wegen nicht alle Attribute dargestellt):

Firma

PersNr	Name	AbtNr	Abteilung	ProjNr	Projekt	Zeit
123	Müller	5	EDV	6	Novell	50%
123	Müller	5	EDV	3	DV2000	50%
876	Schulze	3	Personal	3	DV2000	100%

Bemerkung: In obiger Tabelle wäre nur die Kombination aus *PersNr* und *ProjNr* ein Schlüssel.

Um die weiteren Normalisierungsschritte vornehmen zu können, sind die Begriffe *funktional abhängig* und *transitiv abhängig* zu klären. Sie beziehen sich auf die Abhängigkeit zwischen den Schlüssel- und den Nicht-Schlüssel-Attributen.

In einer Relation ist das Attribut B vom Attribut A *funktional abhängig*, wenn zu jedem Wert von A genau ein Wert von B gehört.

In unserer Tabelle sind die Attribute *Name, AbtNr, Abteilung* funktional abhängig vom Attribut *PersNr.* Dieses Attribut eignet sich daher als Schlüssel für eine Tabelle, die nur diese Daten enthält.

Das Attribut *Projekt* hängt funktional vom Attribut *ProjNr* ab, das deshalb Schlüssel einer weiteren Tabelle werden kann.

Das Attribut *Zeit* hängt dagegen nur von der Kombination der beiden Schlüsselattribute *PersNr* und *ProjNr* ab, nicht dagegen von einem Teil dieses zusammengesetzten Schlüssels.

2. Normalform

Eine Relation befindet sich in der 2. Normalform, wenn sie sich in der ersten befindet und jedes Nicht-Schlüssel-Attribut funktional abhängig ist nur vom Gesamtschlüssel, nicht dagegen von Schlüsselteilen.

Das Einhalten der 2. Normalform muss also nur dann überprüft werden, wenn der Gesamtschlüssel aus mehr als einem Attribut besteht!

Ferner sollte jede Tabelle nur Daten aus einem Objektbereich beinhalten. Objektbereiche wie z.B. Personaldaten und Abteilungsdaten werden damit entzerrt.

Nicht-Schlüsselattribute, die nur von einem Teil des Primärschlüssels abhängen, werden mit diesem Teil zusammen in eine neue Tabelle ausgelagert, so dass das Kriterium für die 2.Normalform damit erfüllt ist.

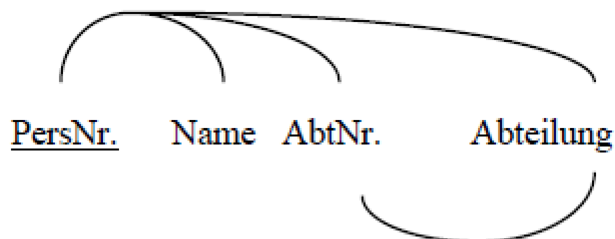
Die Aufspaltung der obigen Tabelle in 3 Tabellen überführt diese in die 2. Normalform. Die verwendete Tupel-Schreibweise ist uns schon bekannt und erleichtert den Überblick:

Mitarbeiter(PersNr, Name, AbtNr, Abteilung)
Projekt(ProjNr, Projekt)
Arbeitet an(PersNr, ProjNr, Zeit)

Ohne die Tabellen nochmals mit Werten hinzuschreiben, macht man sich leicht klar, dass es trotzdem noch Datenredundanz gibt. So müsste bei jedem Mitarbeiter, der eingestellt wird, neben der *AbtNr* auch *Abteilung* gespeichert werden. Das kann natürlich auch zu Einfüge- und Änderungsanomalien führen.

Wie man leicht einsieht, liegt der Grund dafür in der funktionalen Abhängigkeit des **Nichtschlüsselattributes** *Abteilung* vom ebenfalls **Nichtschlüsselattribut** *AbtNr.*

Diesen Sachverhalt kann man wieder mathematisch so ausdrücken: Das Nicht-Schlüssel-Atttribut *Abteilung* ist nur *transitiv abhängig* vom Schlüssel-Attribut *PersNr,* weil es über den Umweg *AbtNr* von *PersNr* abhängig ist.



Man kann das auch so formulieren: *Abteilung* hängt funktional von *AbtNr* ab und *AbtNr* wieder von *PersNr.* Wenn B von A abhängt und C von B, dann hängt C über B indirekt von A ab, eben transitiv.

Im Gegensatz zur transitiven Abhängigkeit wollen wir von *direkter Abhängigkeit* sprechen, wenn es keinerlei mögliches „Zwischenglied" für die Abhängigkeit gibt.

Wenn wir die transitive Abhängigkeit beseitigen wollen, müssen wir die Tabelle *Mitarbeiter* zerlegen, es entstehen zwei Tabellen. Diese befinden sich dann in der 3. Normalform nach Codd:

Mitarbeiter(PersNr, Name, AbtNr)
Abteilung(AbtNr, Abteilung)

3. Normalform
Eine Relation befindet sich in der 3. Normalform, wenn sie sich in der 1. und 2. Normalform befindet und keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Attributen existieren. Anders ausgedrückt: Die Relation darf keine transitiven Abhängigkeiten aufweisen.

Nicht-Schlüsselattribute, die von anderen Nicht-Schlüsselattributen abhängig sind, werden also mit diesen zusammen als Attribut in eine neue Tabelle ausgelagert, so dass das Kriterium für die 3.Normalform damit erfüllt ist.

Damit haben wir zumindest in unserem Beispiel redundante Daten entfernt und die erwähnten Anomalien vermieden. Es lassen sich jetzt komplexere Beispiele konstruieren, in denen doch wieder Probleme auftreten, obwohl sich die Tabellen in 3NF befinden. Dazu gibt es weitere Normalisierungsmethoden, unter anderem die so genannte Boyce-Codd-Normalform, die noch über die 3NF hinausgehen.

Reflexion Datenbankdesign

Mit unseren jetzigen Kenntnissen können wir noch mal der Frage nach dem allgemeinen prinzipiellen Vorgehen beim Datenbankentwurf nachgehen und gelangen zu etwa folgender Vorgehensweise:

1. Bestimmung der Aufgaben der Datenbank: WAS soll sie überhaupt leisten?

2. Tabellen / Entitäten / Objekte bestimmen

3. Felder (Attribute) der Tabellen bestimmen

4. Beziehungen definieren

5. Entwurfsverfeinerung mittels Normalformen

6. Umsetzung mit einem RDBMS

Zu 1: Es hilft oft, sich typische Fragen zu notieren, die die Datenbank einmal beantworten können muss, z.B.
- Wer ist der Lieferant des meistverkauften Artikel?
- Wie groß war der Umsatz im Mai vorigen Jahres und diesen Jahres?
- Welcher Sachbearbeiter ist für den Kunden X zuständig?

Dann sollte man vorliegende Dokumente beschaffen und analysieren, die den abzubildenden Gegenstandsbereich der Datenbank betreffen, z.B. Berichte, Rechnungen, Auftragsbestätigungen usw. So kann man fundiert Objekte und ihre Attribute identifizieren und gewinnt ferner ein schärferes Bild vom Gegenstandsbereich, der dem Datenbankentwickler ja häufig fremd ist!

Zu 2: Vermeiden von Redundanzen durch Komponentenbildung. Ergibt sich oft auf natürliche Art und Weise, in dem man das, was logisch zusammen gehört, auch als ein Objekt betrachtet und auf eine Tabelle abbildet, z.B. Personendaten, Bestelldaten, Produktdaten…

Aufgaben zur Normalisierung

Aufgabe 4

Eine Tabelle mit Lehrerdaten sei wie folgt formuliert:

Nr	Nachname	Vorname	Amtsbez.	Besoldungsgruppe	Klasse
1	Bavaria	Eusebia	StRin	A13	8a, 10b
2	Bachmann	Hanna	StRin	A13	8b
3	Kalkulus	Carl-Johann	StD	A15	7c
4	Spike	Moses	OStR	A14	7d

Analysieren Sie die Tabelle! Welche Schwächen fallen auf? Normalisieren Sie sie anschließend.

Lösung Aufgabe 4:

Redundanzen: Die Besoldungsgruppe ist abhängig von der Amtsbezeichnung. Zusammenhang *Amtsbez.* und *Besoldungsgruppen* einmal in eigener Tabelle erfassen

Einfügeanomalie: Es kann keine neue Besoldungsgruppe Ax ohne einen Lehrer (Nr) geben, der sie hat (Situation z.B. bei Datenbank-Neuanlage).

Löschanomalie: Ohne Mitarbeiter (alle gelöscht) sind auch die Amtsbezeichnungen und die Besoldungsgruppen weg.

Änderungsanomalie: Wird eine Besoldungsgruppe Ax geändert, so muss jeder einzelne Datensatz kontrolliert werden statt es einmal zentral zu formulieren durch eine Zuordnung Amtsbez. ⇔ Besoldungsruppe

Transitive Abhängigkeit: Besoldungsgruppe ist nur von Amtsbez. abhängig, nicht vom Primärschlüssel.

Zunächst Überführung in die 1.NF (es darf nur atomare Werte geben):

Nichtatomare Werte gibt es in der ersten Zeile:

Nr	Nachname	Vorname	Amtsbez.	Besoldungsgruppe	Klasse
1	*Bavaria*	*Eusebia*	*StRin*	*A13*	*8a*
1	*Bavaria*	*Eusebia*	*StRin*	*A13*	*10b*
2	Bachmann	Hanna	StRin	A13	8b
3	Kalkulus	Carl-Johann	StD	A15	7c
4	Spike	Moses	OStR	A14	7d

!!! Die **Nr** identifiziert jetzt NICHT mehr EINDEUTIG einen Datensatz (Zeile, Tupel). Daher ist der neue PS ein zusammengesetzter PS aus den Attributen *Nr* und *Klasse*. !!!

Sollte ein Lehrer in einer Klasse mehr als ein Fach unterrichten, so wäre der PS um das Attribut *Fach* zu ergänzen.

Nun Überführung in die 2. NF
(Jedes Nicht-Schlüssel-Attribut ist funktional abhängig nur vom Gesamtschlüssel, nicht dagegen von Schlüsselteilen):

Der Primärschlüssel ist zusammengesetzt, daher ist die 2.NF überhaupt noch zu

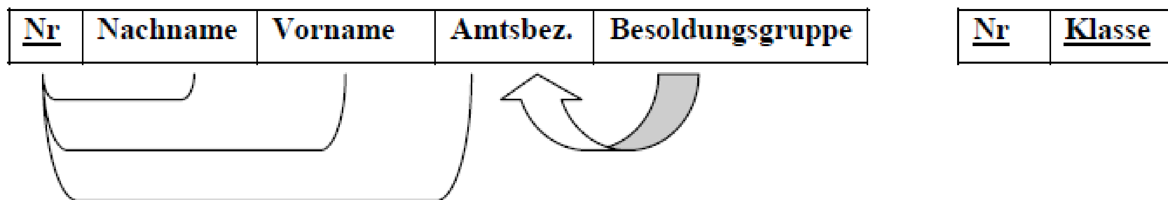
prüfen.

Nachname, Vorname, Amtsbez. und Besoldungsgruppe haben aber nichts mit den Klassen zu tun, die hängen nur von der Nr. ab. Ausgliedern in eine eigene Tabelle mit PS als Fremdschlüssel.

Nr	Nachname	Vorname	Amtsbez.	Besoldungsgruppe
1	Bavaria	Eusebia	StRin	A13
2	Bachmann	Hanna	StRin	A13
3	Kalkulus	Carl-Johann	StD	A15
4	Spike	Moses	OStR	A14

Nr	Klasse
1	8a
1	10b
2	8b
3	7c
4	7d

Überführung in die 3. NF
(Es darf keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Attributen geben, es darf also keine transitiven Abhängigkeiten geben)



Es gibt hier also transitive Abhängigkeiten. Auflösen durch eigene Tabelle.

Lehrer

Nr	Nachname	Vorname	Amtsbez.
1	Bavaria	Eusebia	StRin
2	Bachmann	Hanna	StRin
3	Kalkulus	Carl-Johann	StD
4	Spike	Moses	OStR

Lehrer_unterrichtet_in

Nr	Klasse
1	8a
1	10b
2	8b
3	7c
4	7d

Gehalt

Amtsbez.	Besoldungsgruppe
StRin	A13
StD	A15
OStR	A14

Aufgabe 6
Gegeben sei eine Tabelle mit folgender Struktur. Normalisiere sie bis zur 3. Normalform. Dokumentiere dabei ausführlich deine Schritte mit der entsprechenden Begründung (Definition der Normalformen und ihre Folgerungen).

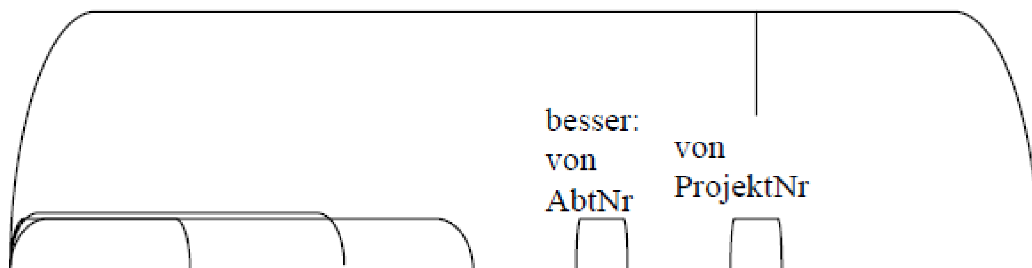
PersNr	Nachname	Vorname	Abteilung	Projekt	Stunden
1	Lorenz	Christian	Einkauf	Verkaufsanalyse	198, 201
2	Baumann	Peter	Verkauf	Optimierung	120, 189, 43
3	Petersen	Anna	Personal	Weiterbildung	120

Lösung der Aufgabe 6:

Überführung in die1. Normalform (nur atomare Werte sind erlaubt)
Der Primärschlüssel muss einen Datensatz eindeutig identifizieren. In der Regel sind dazu nun zusammengesetzte PS nötig.

PersNr	Nachname	Vorname	Abteilung	Projekt	Stunden
1	Lorenz	Christian	Einkauf	Verkaufsanalyse	198
1	Lorenz	Christian	Einkauf	Verkaufsanalyse	201
2	Baumann	Peter	Verkauf	Optimierung	120,
2	Baumann	Peter	Verkauf	Optimierung	189
2	Baumann	Peter	Verkauf	Optimierung	43
3	Petersen	Anna	Personal	Weiterbildung	120

Wir analysieren nun die Tabelle, um aus ihr die 2. und 3. NF ablesen zu können.



PersNr	Nachname	Vorname	Abteilung	Projekt	Stunden
1	Lorenz	Christian	Einkauf	Verkaufsanalyse	198
1	Lorenz	Christian	Einkauf	Verkaufsanalyse	201
2	Baumann	Peter	Verkauf	Optimierung	120,
2	Baumann	Peter	Verkauf	Optimierung	189
2	Baumann	Peter	Verkauf	Optimierung	43
3	Petersen	Anna	Personal	Weiterbildung	120

Überführung in die 2. Normalform (Jedes Nicht-Schlüssel-Attribut darf nur funktional abhängig vom Gesamtschlüssel sein, nicht dagegen von Schlüsselteilen)

Wir haben hier einen zusammengesetzten PS vorliegen, deshalb müssen wir uns überhaupt erst um die 2.NF kümmern, sonst wären die Bedingungen ja schon

erfüllt.
Nachname, Vorname und Abteilung hängen nur von der PersNr ab, nicht von der Stundenzahl. Daher auslagern in eine eigene Tabelle:

Personal

PersNr	Nachname	Vorname	Abteilung
1	Lorenz	Christian	Einkauf
2	Baumann	Peter	Verkauf
3	Petersen	Anna	Personal

Arbeiten_an

PersNr	Projekt	Stunden
1	Verkaufsanalyse	198
1	Verkaufsanalyse	201
2	Optimierung	120,
2	Optimierung	189
2	Optimierung	43
3	Weiterbildung	120

Um Inkonsistenzen und Redundanzen zu vermeiden, sollte man noch Abteilungsnummern und Projektnummern einführen und die entsprechenden Tabellen ausgliedern:

Abteilung

AbtNr	Abteilung
I	Einkauf
II	Verkauf
III	Personal

Projekt

ProjNr	Projekt
A	Verkaufsanalyse
B	Optimierung
C	Weiterbildung

Personal

PersNr	Nachname	Vorname	AbtNr
1	Lorenz	Christian	I
2	Baumann	Peter	II
3	Petersen	Anna	III

Arbeiten_an

PersNr	ProjNr	Stunden
1	A	198
1	A	201
2	B	120,
2	B	189
2	B	43
3	C	120

3. Normalform
(Es darf keine funktionalen Abhängigkeiten zwischen Nicht-Schlüssel-Attributen geben, es darf also keine transitiven Abhängigkeiten geben)

Die Datenbank befindet sich bereits in 3.NF
Damit enthält die Datenbank folgende Tabellen:

Personal(PersNr, Nachname, Vorname, AbtNr)
Abteilung(AbtNr, Abteilung)
Projekte(ProjNr, Projekt)
Arbeitet_an(ProjNr, PersNr, Stunden)

Aufgabe 10
Eine Tabelle sei in Kurzschreibweise wie folgt gegeben. Bringe sie in die 3.NF !

Rechnung

RechNr	KundenNr	Datum	Artikel
1	2	03.03.2020	2 Apfelbäume
2	5	04.01.2008	2 Birnbäume, 3 Apfelbäume
3	12	01.01.1999	2 Pflaumenbäume, 3 Kirschbäume

Lösung der Aufgabe 10:

Es sind nicht-atomare Attributwerte enthalten, also auflösen:
Überführung in die 1. Normalform:

<u>RechNr</u>	KundenNr	<u>Position</u>	Datum	Anzahl	Artikel
1	2	1	03.03.2020	2	Apfelbaum
2	5	1	04.01.2008	2	Birnbaum
2	5	2	04.01.2008	3	Apfelbaum
3	12	1	01.01.1999	2	Pflaumenbaum
3	12	2	01.01.1999	3	Kirschbaum

Nun ist ein zusammengesetzter PS nötig, um einen Datensatz eindeutig zu identifizieren. Dazu wurde das Feld *Position* hinzugefügt.
Es gibt einen zusammengesetzten PS, daher ist die 2. NF überhaupt noch zu diskutieren.

Datum und *KundenNr* sind nur abhängig von dem Teilschlüssel *Rechnungsnummer*, nicht von der *Position*, entsprechend muss ausgelagert werden.

Überführung in die 2. Normalform:
Rechnung

<u>RechNr</u>	**KundenNr**	**Datum**
1	2	03.03.2020
2	5	04.01.2008
3	12	01.01.1999

Rechnungdetails

<u>RechNr</u>	**<u>Position</u>**	**Anzahl**	**Artikel**
1	1	2	Apfelbaum
2	1	2	Birnbaum
2	2	3	Apfelbaum
3	1	2	Pflaumenbaum
3	2	3	Kirschbaum

Die Datenbank befindet sich jetzt auch schon in 3. NF

SQL (Structured Query Language)

Entscheidend für die Qualität einer Datenbank ist, wie die in den Tabellen „steckenden“ Informationen wieder zurück gewonnen und mit anderen Informationen verknüpft werden können.

Das relationale Modell stellt hierfür den theoretischen Hintergrund bereit. Die Umsetzung der relationalen Operationen in eine *Abfragesprache* ist allerdings je nach Einsatz spezieller Datenbanksoftware unterschiedlich realisiert. Insofern ist es sinnvoll, sich eines Quasi-Standards zu bedienen, der in den letzten Jahren noch an Bedeutung hinzugewonnen hat: *SQL (Structured Query Language).*

SQL gibt es seit den 80er Jahren. *SQL* lässt sich innerhalb der Programmiersprachen als Sprache der *4. Generation* bezeichnen (*4GL)*, denn sie ist im Gegensatz zu den prozeduralen Sprachen der 3. Generation, die durch Datenstrukturen und Steuerstrukturen gekennzeichnet und bei Dateien satzorientiert sind, nicht-prozedural und mengenorientiert.

Ein kleines Beispiel zur Ausgabe aller Titel und Themen aus einer Datei namens *Buchtyp*, die vor 1990 erschienen sind, möge dies verdeutlichen:

<table>
<tr><th>problemorientierte Sprache (3GL)</th><th>SQL (4GL)</th></tr>
<tr><td><pre>open(buchtyp);
while not eof(buchtyp) {
  read(buch);
  if (buch.jahr < 1999)
    print(buch.titel, buch.autor);
}</pre></td><td><pre>SELECT titel, autor
FROM buchtyp
WHERE jahr < 1999</pre></td></tr>
</table>

Die *SQL*-Anweisungen beinhalten also nicht, **wie** etwas gemacht werden soll, sondern nur, **was** gemacht werden soll. Das **„wie“** wird dem Betriebssystem bzw. dem Interpreter der Sprache *SQL* überlassen.

SQL ist allerdings keine universelle Programmiersprache wie z.B. *C*, *Delphi* oder *Java*, sondern beinhaltet im Wesentlichen nur sehr wenige Befehle zur Verwaltung einer Datenbank. Deshalb ist es heute auch in vielen höheren Programmiersprachen wie z.B. *Java* oder *Delphi* möglich, *SQL*-Statements „einzubetten“ und ausführen zu lassen.

Die Gesamtheit der *SQL*-Befehle wird gewöhnlich in drei Klassen eingeteilt:
- **DDL**-Befehle (DataBase Definition Language) zur Definition bzw. Erzeugung von Datenstrukturen (Datenbanken, Tabellen, Typen von Attributen usw.)
- **DCL**-Befehle zur Kontrolle der Zugriffsberechtigungen.
- **DML**-Befehle zur Manipulation der Datenbank und Formulierung von Abfragen.

Wir wollen uns hauptsächlich nur mit den DML-Befehlen beschäftigen, weil sie den wesentlichen Kern von *SQL* ausmachen.

Die *DDL*-Befehle sind etwas unhandlich, weswegen die meisten *SQL*-Umgebungen komfortablere Methoden zur Verfügung stellen.

Zur Groß-/Kleinschreibung:
Normalerweise ist *SQL* relativ unempfindlich bei der Schreibweise. Allerdings sollte man den Namen der Datenbank exakt wiedergeben. Das liegt daran, dass diese unter ihrem Namen auf der Festplatte gespeichert ist und *SQL*- reagiert beim Öffnen und Schreiben von Dateien ähnlich empfindlich wie Linux-Systeme.

Üblicherweise schreibt man *SQL*-Anweisungen komplett in Großbuchstaben.

Wichtig: Man unterscheidet zwischen der eigentlichen Programmiersprache *SQL* und den sog. Entwicklungs- oder Programmierumgebungen für *SQL.*
Analog unterscheidet man auch zwischen der Programmiersprache *JAVA* und den *Java*-Umgebungen Net-Beans, Eclipse oder BlueJ.
Es gibt viele *SQL*-Umgebungen. Beispiele sind etwa **MySQL, Oracle, Microsoft-SQL-Server, SQLite** oder auch **MS-Access**.
Leider gibt es einige wenige Unterschiede zwischen den Entwicklungsumgebungen. Manche Befehle funktionieren zwar in der einen, aber nicht in der anderen Umgebung und umgekehrt. Aber Ähnliches kennt man ja auch von den verschiedenen Browsern her (Internet-Explorer, Firefox, usw.). Warum sollte auch alles einfach sein?

Textstrings werden in MySQL mit einfachen Hochkommas eingeschlossen.

Leider ist *MySQL* nicht sonderlich stringend in der Benutzung von Anführungszeichen. Manchmal ist es egal, ob man einfache, doppelte oder gar keine Anführungszeichen benutzt. Als einfaches Anführungszeichen wird meistens das Hochkomma links neben der *Enter*-Taste benutzt; manchmal besteht das

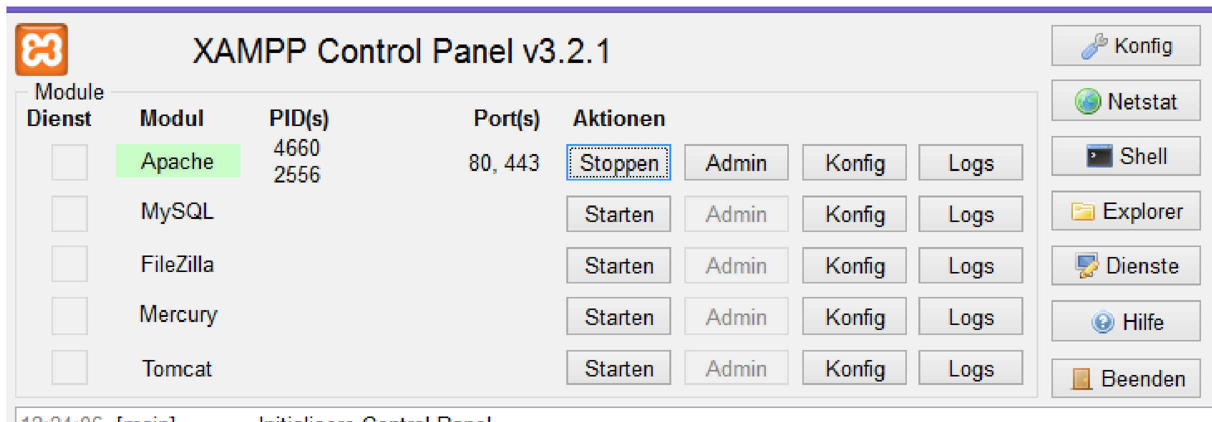
System aber auch auf die Verwendung des Zeichens rechts neben der ***ß***-Taste (mit Shift). Und leider unterscheidet sich das sogar bei zwei praktisch identischen Rechnern mit (offensichtlich) demselben Betriebssystem und (anscheinend) derselben *SQL*-Entwicklungsumgebung.

Zwei aufeinanderfolgende SQL-Anweisungen müssen durch ein Semikolon getrennt werden.

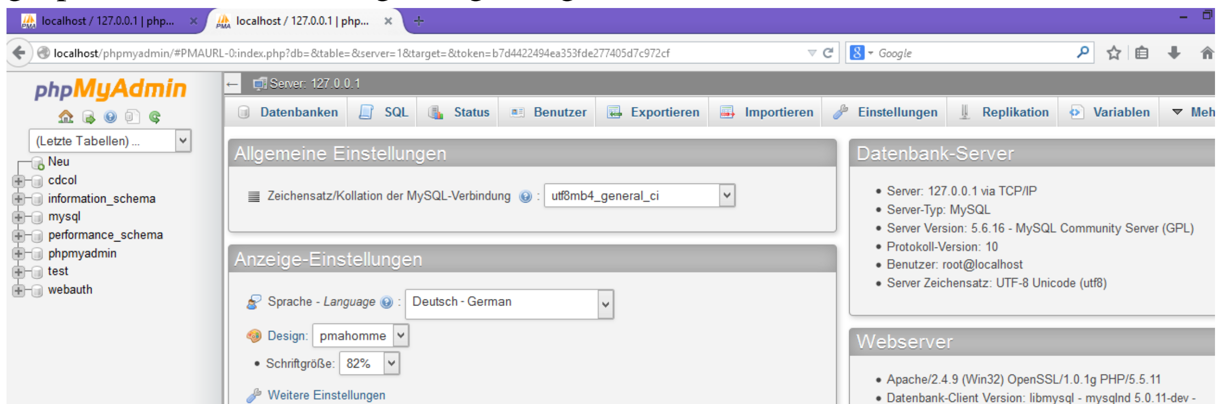
Die Entwicklungsumgebung *MySQL*

Installiere das im Internet kostenlos erhältliche Programm *XAMPP*. Dieses enthält unter anderem eine Programmierumgebung für das Datenbanksystem *SQL*.

Starte anschließend das Programm xampp-control:



Starte das Modul *Apache*, warte einige Sekunden und starte anschließend das Modul *MySQL* (Hinweis: Nach Beendigung der Programmierung sollte man diese beiden Module auch wieder stoppen). Klicke danach auf den Admin-Button in der *MySQL*-Zeile. Danach öffnet der Standardbrowser folgende, lokal gespeicherte Entwicklungs-umgebung:



Wähle oben das Menü *Datenbanken* und erzeuge anschließend eine neue Datenbank mit dem Namen *TestDB*. Die Auswahlmöglichkeiten im Menü *Kollation* werden ignoriert.
Es erfolgt die Mitteilung „Die Datenbank wurde erzeugt“, und links im *phpMyAdmin*-Menü ist der Name der neuen, noch leeren Datenbank zu sehen.

Bemerkung: trotz der Großbuchstaben im eingegebenen Namen wird dieser nur mit Kleinbuchstaben weiterverarbeitet.

Wähle nun links im *phpMyAdmin*-Menü die gerade angelegte Datenbank *testdb* aus! Wir wollen unsere erste Tabelle in unserer Datenbank *kunden* nennen. Gib also als Tabellennamen *kunden* ein und bei "Anzahl der Spalten" den Wert **2** (jede Tabelle kann beliebig viele Spalten – Attribute – haben). Betätige den Button „OK"!
Die 2 Felder müssen nun im folgenden Formular korrekt gefüllt werden, erst danach wird unsere Tabelle erzeugt.

Für die beiden Spalten unserer Kunden-Tabelle wählen wir die Namen **name** und **adresse**. Befülle das Formular wie im Screenshot gezeigt:

Server: 127.0.0.1 » Datenbank: testdb » Tabelle: kunden

Anzeigen | Struktur | SQL | Suche | Einfügen | Exportieren | Import

Tabellenname: kunden | 1 Spalte(n) einfügen | OK

Name	Typ	Länge/Werte	Standard	Kollation
Name	VARCHAR	60	Kein(e)	
Adresse	VARCHAR	100	Kein(e)	

Tabellenkommentar: | Tabellenformat: InnoDB | Kollation:

Beide Spalten sind vom Typ ***VARCHAR***, das heißt, die Spalten erwarten Strings als Input. Die Eingabe ***Länge/Werte*** ist abhängig vom Typ. Bei *VARCHAR* muss dort ein Wert eingetragen werden und zwar **eine Zahl zwischen 1 und 255**. Die Zahl gibt an, wie lang der jeweilige String jeweils sein darf. Der Name unserer Kunden soll nicht länger als 60 Zeichen sein, die Adresse nicht länger als 100 Zeichen. Die restlichen Werte werden zunächst ignoriert.

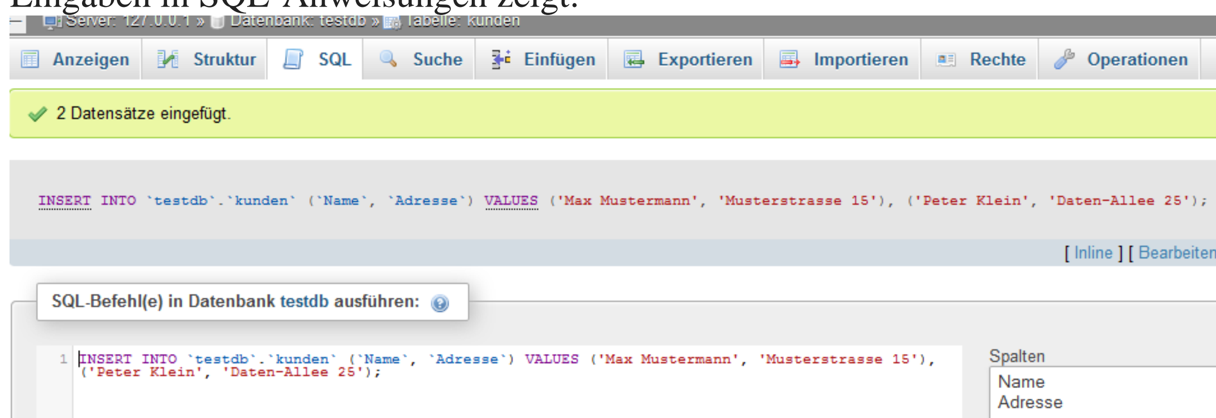
Der ganze Vorgang wird natürlich mit dem Button *Speichern* abgeschlossen.

Jetzt existiert schon eine Datenbank mit einer einzigen Tabelle. Diese muss jetzt noch mit Werten gefüllt werden.

Werte eintragen
Wähle nun links im *phpMyAdmin*-Menü die Tabelle *kunden* in der Datenbank *testdb* aus! Hinweis: In diesem Untermenü erscheint üblicherweise auch die Option *Neu*. Damit lassen sich einfach weitere Tabellen bzw. innerhalb einer Tabelle auch weitere Spalten erzeugen.
Klicke oben auf den Reiter *Einfügen*. Je nach Tabelle werden nun unterschiedlich viele Eingabefelder angezeigt, in unserem Falle 2. Standardmäßig gibt die *PHPMyAdmin*-Umgebung immer 2 Formulare aus, für den Fall, dass man nicht nur einen sondern gleich zwei Datensätze eintragen möchte.

Trage im ersten Feld, den (String-) ***Wert*** *Max Mustermann* ein. Im Feld *Adresse* nehmen wir analog die Straße *Musterstraße 15*. Im zweiten Formular tragen wir den Namen *Peter Klein* ein und bei *Adresse* die Angabe *Daten-Allee 27*. Mit einem Klick auf den untersten Button "OK" werden alle Daten in die Datenbank übernommen.

Es erscheint anschließend folgendes Fenster, welches die Übersetzung unserer Eingaben in SQL-Anweisungen zeigt:



Um für die weitere Bearbeitung mehr Daten zu haben, kopiere jetzt in das *SQL*-Anweisungsfeld folgende Befehle:

```
INSERT INTO kunden (name, adresse) VALUES ('Michael Meier', 'Saagengasse 13');
INSERT INTO kunden (name, adresse) VALUES ('Berta Brecht', 'Kalossenweg 8');
INSERT INTO kunden (name, adresse) VALUES ('Lilly Lagerfeld', 'Blumenweg 19');
INSERT INTO kunden (name, adresse) VALUES ('Peter Paulus', 'Kirchenstrasse 22');
INSERT INTO kunden (name, adresse) VALUES ('Sarah Seil', 'Roteneck 12');
INSERT INTO kunden (name, adresse) VALUES ('Adam Aldenau', 'Grüner Weg 5');
INSERT INTO kunden (name, adresse) VALUES ('Emil Entenich', 'Hinterhausen 12');
```

Teste jetzt z.B. folgende Anweisung:

```
SELECT * FROM kunden
WHERE name < 'Peter Paulus'
```

Wie oben beschrieben, wurde mithilfe der SQL-Entwicklungsumgebung eine Datenbank einschließlich ihrer Tabellen und einiger Einträge erzeugt. Dasselbe Resultat kann man auch erhalten, indem man alles durch SQL-Anweisungen (CREATE, DELETE, INSERT usw.) erzeugen lässt (siehe obige Abbildungen!).

Eine Datenbank wird als eine Textdatei gespeichert; sie erhält im Dateinamen das Anhängsel ***.sql*** . Wenn man eine derartige Datei mit einem Text-Editor öffnet, sieht man die Auflistung aller SQL-Anweisungen, die zur Erzeugung dieser Datenbank (mit allen Tabellen und Eintragungen, einschließlich Kommentaren zum besseren Verständnis) notwendig sind.

Leider sind diese ***.sql***-Dateien etwas unterschiedlich für Windows- bzw. Linuxsysteme. Die Unterschiede kann man jedoch mit etwas Aufwand manuell im Text-Editor anpassen.

Bestimmte Datensätze auswählen

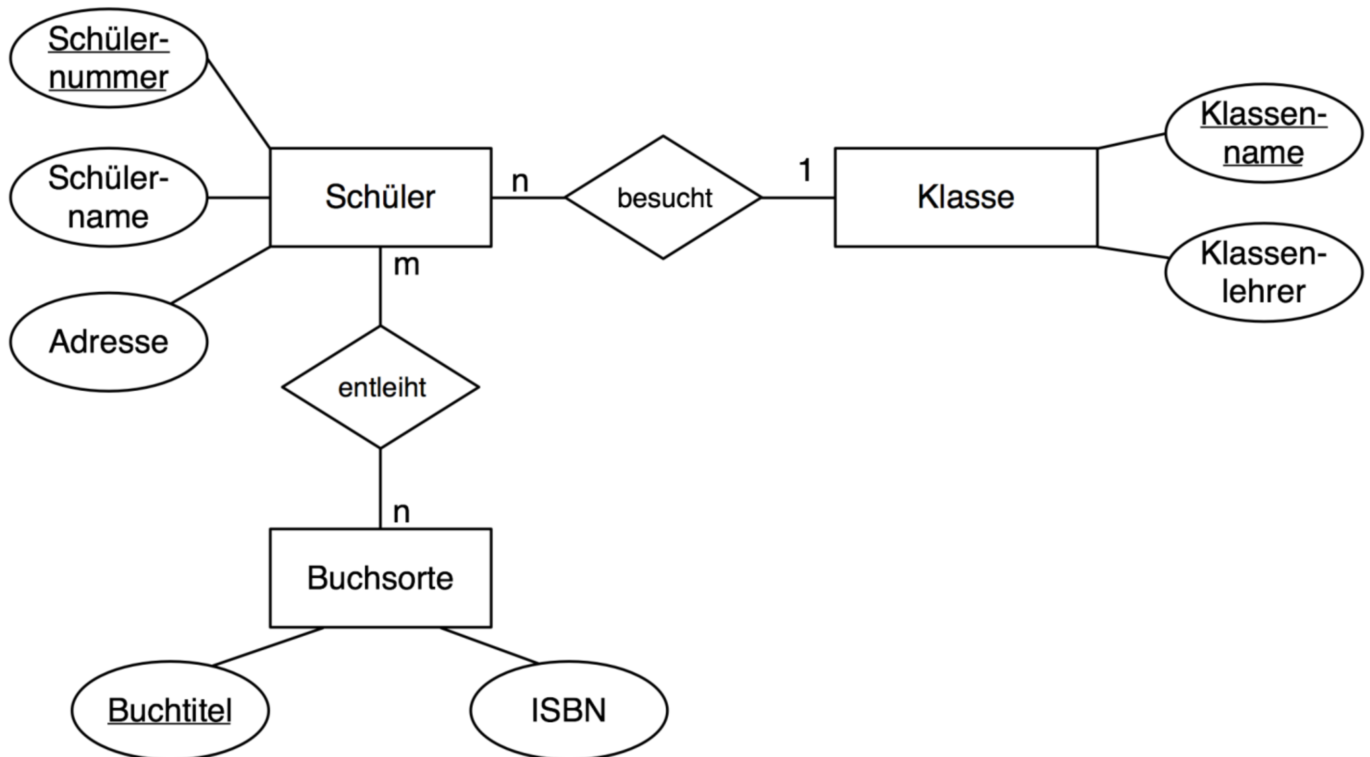
Im *PHPMyAdmin* –Hauptmenü findet man (nach Auswahl der gewünschten Tabelle in der gewünschten Datenbank) einen Reiter "*SQL*". Klickt man darauf, so sieht man ein großes Textfeld, in dem schon etwas *SQL*-Code zu sehen ist. Dieser voreingestellte Code variiert je nachdem, ob man eine ganze Datenbank angewählt hat oder nur eine Tabelle. Er hängt auch davon ab, ob die Datenbank nur eine oder mehrere Tabellen besitzt.

```
SELECT * FROM kunden WHERE 1
```

Klicke unten auf den Button "OK" und es sollten alle bereits eingetragenen Datensätze ausgegeben werden. Hinweis: Die Bedingung *WHERE 1* kann auch weggelassen werden.

Am Beispiel der Datenbank *Schulbuchverwaltung* wollen wir nun einen ersten Einblick in die Sprache *SQL* gewinnen.

ER-Diagramm der Datenbank „Schulbuchverwaltung"



Die Relationenschemata der Datenbank „Schulbuchverwaltung"

schueler (<u>Schuelernummer</u>, Schuelername, Adresse, ↑Klassenname)
klasse (<u>Klassenname</u>, Klassenlehrer)
buchsorte (<u>Buchtitel</u>, ISBN)
entleihvorgang (<u>Leihnummer</u>, ↑Schuelernummer, ↑Buchtitel)

Beachte, dass in dem Datenbanksystem nur 4 Tabellen existieren; Die Beziehung *besucht* kann mit der Entity *Schüler* zu einer einzigen Tabelle zusammengefasst werden.

Erzeuge zunächst eine neue, leere Datenbank mit dem Namen „Schulbuchverwaltung“. Öffne diese leere Datenbank und importiere anschließend die vom Lehrer zur Verfügung gestellte Datei namens `schulbuchverwaltung.sql` !
Interessanterweise werden hierdurch nur einzelne Tabellen importiert. Der Name der Datenbank muss vorher schon existieren und die leere Datenbank selbst muss schon geöffnet sein.

Falls diese Datei nicht zur Verfügung stehen sollte (erzeuge zunächst auf jeden Fall die neue, leere Datenbank „Schulbuchverwaltung“!), so kann man die nötigen Tabellen auch erzeugen, indem man nachfolgende *SQL*-Anweisungen angibt.
Hinweis: Die folgenden Anweisungen bzgl. des anzuwendenden Zeichensatzes bewirken leider, dass deutsche Umlaute nicht richtig dargestellt werden. Eine nachträgliche Änderung des Zeichensatzes für einzelne Tabellen oder gar für die gesamte Datenbank erweist sich je nach Versionsnummer von *MySQL* als äußerst schwierig.

```
CREATE TABLE 'buchsorte' (
  'Buchtitel' varchar(30) CHARACTER SET latin1
COLLATE latin1_german1_ci NOT NULL,
  'ISBN' varchar(15) NOT NULL DEFAULT '',
  PRIMARY KEY ('Buchtitel')
) ENGINE=MyISAM DEFAULT CHARSET=latin1

INSERT INTO 'buchsorte' ('Buchtitel', 'ISBN') VALUES
('Grammatik', '3-412-02312-3'),
('Mein Deutschbuch', '3-513-22312-4'),
('Learning Englisch 7', '3-221-41123-2'),
('Learning Englisch 6', '3-221-41122-2'),
('Learning Englisch 5', '3-221-41121-2'),
('Mathematik heute 7', '7-231-24554-8'),
('Mathematik heute 6', '7-231-24553-3'),
('Mathematik heute 5', '7-231-24552-3'),
('Übungsheft Mathemax 7', '5-789-31521-2');

CREATE TABLE 'entleihvorgang' (
  'Leihnummer' int(6) NOT NULL AUTO_INCREMENT,
  'Schuelernummer' int(4) NOT NULL DEFAULT '0',
  'Buchtitel' varchar(30) NOT NULL DEFAULT '',
  PRIMARY KEY (`Leihnummer`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1
AUTO_INCREMENT=66 ;
```

```
INSERT INTO 'entleihvorgang' ('Leihnummer',
'Schuelernummer', 'Buchtitel') VALUES
(1, 1, 'Grammatik'),
(2, 1, 'Mein Deutschbuch'),
(3, 1, 'Learning Englisch 7'),
(4, 1, 'Mathematik heute 7'),
(5, 2, 'Mein Deutschbuch'),
(6, 2, 'Learning Englisch 7'),
(7, 2, 'Mathematik heute 7'),
(8, 3, 'Grammatik'),
(9, 3, 'Mein Deutschbuch'),
(10, 3, 'Learning Englisch 7'),
(11, 3, 'Mathematik heute 7'),
(12, 3, 'Übungsheft Mathemax 7'),
(13, 4, 'Mein Deutschbuch'),
(14, 4, 'Learning Englisch 7'),
(15, 4, 'Mathematik heute 7'),
(16, 5, 'Grammatik'),
(17, 5, 'Mein Deutschbuch'),
(18, 5, 'Learning Englisch 7'),
(19, 5, 'Mathematik heute 7'),
(20, 6, 'Mein Deutschbuch'),
(21, 6, 'Learning Englisch 7'),
(22, 6, 'Mathematik heute 7'),
(23, 7, 'Grammatik'),
(24, 7, 'Mathematik heute 6'),
(25, 7, 'Mein Deutschbuch'),
(26, 7, 'Learning Englisch 6'),
(27, 8, 'Mathematik heute 6'),
(28, 8, 'Mein Deutschbuch'),
(29, 8, 'Learning Englisch 6'),
(30, 9, 'Grammatik'),
(31, 9, 'Mathematik heute 6'),
(32, 9, 'Mein Deutschbuch'),
(33, 9, 'Learning Englisch 6'),
(34, 10, 'Mathematik heute 6'),
(35, 10, 'Mein Deutschbuch'),
(36, 10, 'Learning Englisch 6'),
(37, 11, 'Grammatik'),
(38, 11, 'Mathematik heute 6'),
(39, 11, 'Mein Deutschbuch'),
(40, 11, 'Learning Englisch 6'),
(41, 12, 'Mathematik heute 6'),
(42, 12, 'Mein Deutschbuch'),
```

```
(43, 12, 'Learning Englisch 6'),
(44, 13, 'Grammatik'),
(45, 13, 'Learning Englisch 5'),
(46, 13, 'Mathematik heute 5'),
(47, 13, 'Mein Deutschbuch'),
(48, 14, 'Learning Englisch 5'),
(49, 14, 'Mathematik heute 5'),
(50, 14, 'Mein Deutschbuch'),
(51, 15, 'Grammatik'),
(52, 15, 'Learning Englisch 5'),
(53, 15, 'Mathematik heute 5'),
(54, 15, 'Mein Deutschbuch'),
(55, 16, 'Learning Englisch 5'),
(56, 16, 'Mathematik heute 5'),
(57, 16, 'Mein Deutschbuch'),
(58, 17, 'Grammatik'),
(59, 17, 'Learning Englisch 5'),
(60, 17, 'Mathematik heute 5'),
(61, 17, 'Mein Deutschbuch'),
(62, 18, 'Learning Englisch 5'),
(63, 18, 'Mathematik heute 5'),
(64, 18, 'Mein Deutschbuch');


CREATE TABLE 'klasse' (
  'Klassenname' char(2) NOT NULL DEFAULT '',
  'Klassenlehrer' varchar(30) NOT NULL DEFAULT '',
  PRIMARY KEY ('Klassenname')
) ENGINE=MyISAM DEFAULT CHARSET=latin1
COMMENT='Enthält die Relation Name der Klasse und
deren Klassenlehrer';


INSERT INTO 'klasse' ('Klassenname', 'Klassenlehrer')
VALUES
('5a', 'Albrecht'),
('5b', 'Brescher'),
('6a', 'Hingsen'),
('6b', 'Ingold'),
('7a', 'Laurach'),
('7b', 'Schmidt'),
('7c', 'Lempel');
```

```
CREATE TABLE 'schueler' (
  'Schuelernummer' int(4) NOT NULL AUTO_INCREMENT,
  'Schuelername' varchar(30) NOT NULL DEFAULT '',
  'Adresse' varchar(50) NOT NULL DEFAULT '',
  'Klassenname' char(2) NOT NULL DEFAULT '',
  PRIMARY KEY ('Schuelernummer')
) ENGINE=MyISAM  DEFAULT CHARSET=latin1
AUTO_INCREMENT=19 ;
```

INSERT INTO 'schueler' ('Schuelernummer', 'Schuelername', 'Adresse', 'Klassenname') VALUES
(1, 'Hilger, Simone', 'Seifenweg 5, 45300 Bankhofen', '7a'),
(2, 'Kruschwitz, Tolga', 'Ingwerstrasse 53, 45300 Bankhofen', '7b'),
(3, 'Kaplan, Heike', 'Kirchgasse 1, 45300 Bankhofen', '7c'),
(4, 'Prinz, Jochen', 'Am Berg 12a, 45300 Bankhofen', '7b'),
(5, 'Kaplan, Torsten', 'Kirchgasse 1, 45300 Bankhofen', '7c'),
(6, 'Morman, Moriz', 'Ullinger Weg 2, 45300 Bankhofen', '7b'),
(7,'Abrasi, Tino','Falkenweg 9,45300 Bankhofen','6a'),
(8, 'Zurren, Dina','Am Dorn 4,45300 Bankhofen','6b'),
(9, 'Brentano, Luigi', 'Beim Italiener 3, 45300 Bankhofen', '6a'),
(10, 'Engemann, Jasmin', 'Küppersweg 8, 45300 Bankhofen', '6b'),
(11, 'Chiong, Na Hu', 'Weiler 3, 45300 Bankhofen', '6a'),
(12, 'Matras, Ingo', 'Trassenweg 7, 45300 Bankhofen', '6b'),
(13, 'Thomas, Laura', 'Rügener Strasse 6, 45300 Bankhofen', '5a'),
(14, 'Meier, Konstantin', 'Am Weiher 2, 45300 Bankhofen', '5b'),
(15, 'Weihershäuser, Gerlinde', 'Hauptstrasse 4, 45300 Bankhofen', '5a'),
(16, 'Onsberg, Karl-Heinz', 'Jupiterweg 7, 45300 Bankhofen', '5a'),
(17, 'Jagoda, Else', 'Schulstrasse 3, 45300 Bankhofen', '5b'),
(18, 'Liborius, Konrad', 'Steinweg 3, 45300 Bankhofen', '5b');

Teste die SQL-Anweisungen mit der Datenbank *Schulbuchverwaltung* und notiere jeweils in der rechten Spalte, was sie bewirken!
Hinweis: Zwischen der Anweisung **COUNT** und der nachfolgenden Klammer darf kein Leerzeichen stehen! Analoges gilt auch für andere Anweisungen.

Nr	SQL-Befehl	Wirkung
1	**SHOW TABLES**	
2	**SHOW COLUMNS FROM schueler**	
3	**SELECT * FROM schueler**	
4	**SELECT * FROM schueler WHERE Klassenname = '7a'**	
5	**SELECT Schuelername, Klassenname FROM schueler ORDER BY Klassenname, Schuelername**	
6	**SELECT COUNT(Klassenname) FROM schueler**	
7	**SELECT COUNT(DISTINCT Klassenname) FROM schueler**	
8	**SELECT Klassenname, COUNT(Klassenname) FROM schueler GROUP BY Klassenname**	
9	**SELECT * FROM schueler, klasse**	
10	**SELECT * FROM schueler, klasse WHERE schueler.Klassenname = klasse.Klassenname**	
11	**SELECT * FROM schueler JOIN klasse ON schueler.Klassenname = klasse.Klassenname**	
12	**SELECT * FROM schueler JOIN klasse USING(Klassenname)**	

13	**SELECT Schuelername, Klassenlehrer** **FROM schueler** **JOIN klasse USING(Klassenname)**	
14	**SELECT *** **FROM klasse** **JOIN schueler USING(Klassenname)** **JOIN entleihvorgang** **USING(Schuelernummer)**	
15	**SELECT Schuelername, schueler.Klassenname, Klassenlehrer, Buchtitel** **FROM klasse** **JOIN schueler USING(Klassenname)** **JOIN entleihvorgang** **USING(Schuelernummer)** **WHERE LEFT(schueler.Klassenname,1) = '7'** **ORDER BY** **schueler.Klassenname, Schuelername**	
16	**SELECT Schuelernummer,** **COUNT(Leihnummer) AS Anzahl** **FROM entleihvorgang** **GROUP BY Schuelernummer** **ORDER BY Anzahl DESC, Schuelernummer ASC**	
17	**SELECT MAX(Anzahl)** **FROM (SELECT COUNT(Leihnummer) AS** **Anzahl FROM entleihvorgang** **GROUP BY Schuelernummer) AS t**	
18	**UPDATE schueler** **SET Klassenname = '7c'** **WHERE Klassenname = ' 7a ' ;** **SELECT * FROM schueler**	

Lösung

Nr	SQL-Befehl	Wirkung
1	**SHOW TABLES**	**Nur die Namen der Tabellen werden aufgelistet.**
2	**SHOW COLUMNS FROM schueler**	**Alle Spalten mit ihren Eigenschaften (String, Int usw.) werden aufgelistet.**
3	**SELECT * FROM schueler**	**Zeigt alle Datensätze der Tabelle *schueler*.**
4	**SELECT * FROM schueler WHERE Klassenname = '7a'**	**Zeigt nur die Datensätze der Schüler aus der Klasse 7a.**
5	**SELECT Schuelername, Klassenname FROM schueler ORDER BY Klassenname, Schuelername**	**Nur die Schülernamen und ihre Klassennamen werden angezeigt alphabetisch geordnet zuerst nach Klasse, dann nach Name.**
6	**SELECT COUNT(Klassenname) FROM schueler**	**Die Anzahl der Einträge in der Spalte *Klassenname* wird angegeben. Weil jeder Schüler in einer Klasse ist, entspricht dies auch der Anzahl der Schüler.**
7	**SELECT COUNT(DISTINCT Klassenname) FROM schueler**	**Es wird die Anzahl der unterschiedlichen Klassennamen ausgegeben.**
8	**SELECT Klassenname, COUNT(Klassenname) FROM schueler GROUP BY Klassenname**	**Für jede Klasse wird die Anzahl ihrer Schüler ausgegeben.**
9	**SELECT * FROM schueler, klasse**	**Jeder Datensatz (Zeile) aus der Tabelle *schueler* wird (unsinnigerweise) mit jedem Datensatz aus der Tabelle *klasse* zu einem neuen Datensatz kombiniert. Die in beiden Tabellen enthaltene Spalte *Klassenname* erscheint dann doppelt.**
10	**SELECT * FROM schueler, klasse WHERE schueler.Klassenname = klasse.Klassenname**	**Nur sinnvolle Kombinationen beider Tabellen werden gezeigt. Für jeden Schüler wird aus der Tabelle *klasse* der richtige Klassenlehrer mit ausgegeben. Die Spalte Klassenname erscheint zweimal.**
11	**SELECT * FROM schueler JOIN klasse ON schueler.Klassenname = klasse.Klassenname**	**Dieselbe Wirkung wie in Nr. 10**

12	**SELECT * FROM schueler JOIN klasse USING(Klassenname)**	**Dieselbe Wirkung wie in Nr. 10, aber die Spalte *Klassenname* wird nur einmal ausgegeben (und zwar als erste Spalte).**
13	**SELECT Schuelername, Klassenlehrer FROM schueler JOIN klasse USING(Klassenname)**	**Aus der Kombination der beiden Tabellen werden nur die Spalten *Schuelername* und (zugehöriger) *Klassenlehrer* ausgegeben.**
14	**SELECT * FROM klasse JOIN schueler USING(Klassenname) JOIN *entleihvorgang* *USING(Schuelernummer)***	**Es werden nur diejenigen Schüler (zusammen mit ihrem Klassenlehrer) aufgelistet, die ein Buch ausgeliehen haben. Für jedes Buch gibt es (zusammen mit dem Schüler) eine neue Zeile.**
15	**SELECT Schuelername, schueler.Klassenname, Klassenlehrer, Buchtitel FROM klasse JOIN schueler USING(Klassenname) JOIN entleihvorgang USING(Schuelernummer) WHERE LEFT(schueler.Klassenname,1) = '7' ORDER BY schueler.Klassenname, Schuelername**	**Wie Nr. 14, allerdings beschränkt auf die Jahrgangsstufe 7; außerdem geordnet zuerst nach Klassen, und dann innerhalb jeder Klasse nach Schülernamen.**
16	**SELECT Schuelernummer, COUNT(Leihnummer) AS Anzahl FROM entleihvorgang GROUP BY Schuelernummer ORDER BY Anzahl DESC, Schuelernummer ASC**	**Ausgegeben werden die Nummern derjenigen Schüler, die Bücher ausgeliehen haben, zusammen mit der jeweiligen Anzahl der von einem Schüler ausgeliehenen Bücher; geordnet zuerst absteigend nach der Bücheranzahl, bei gleicher Bücheranzahl aufsteigend nach Schülernummer. Die Spalte mit der Bücheranzahl erhält die Überschrift *Anzahl*.**
17	**SELECT MAX(Anzahl) FROM (SELECT COUNT(Leihnummer) AS Anzahl FROM entleihvorgang GROUP BY Schuelernummer) AS t** ***Wenn die Unterabfrage nicht funktioniert, dann mit ..FROM schueler JOIN (SELECT…) ON ….***	**Die maximale Anzahl von Büchern, die an einen einzelnen Schüler ausgegeben worden ist, wird angezeigt. Aus SQL-syntaktischen Gründen muss die temporär erzeugte Untertabelle einen eigenen Namen haben (hier: t).**
18	**UPDATE schueler SET Klassenname = '7c' WHERE Klassenname = '7a'; SELECT * FROM schueler**	**Für alle Schüler der 7a wird der Klassenname in '7c' umbenannt Zwei aufeinander folgende *SQL*-Anweisungen müssen durch Semikolon getrennt werden.**

Aufgaben zur Schulbuchverwaltung

Aufgrund der letzten SQL-Beispiele sollte man nun schon die folgenden Aufgaben lösen können. Schreibe für jede Aufgabe die entsprechende SQL-Anweisung auf! Erst danach werden wir uns genauer mit den einzelnen SQL-Anweisungen beschäftigen.

1. Erstelle eine Liste, die zu jedem Buchtitel (alphabetisch sortiert) angibt, an welche Schüler (alphabetisch sortiert) dieses Buch gerade ausgeliehen wird. Sortierung zuerst nach Buchtiteln, dann nach Namen.

2. Erstelle eine Liste, die zu jedem Buchtitel angibt, an wie viele Schüler dieses Buch ausgeliehen wird. Sortierung nach Anzahl der Ausleihen (das am häufigsten ausgeliehene Buch zuerst).

3. Gib nur denjenigen Buchtitel aus, der am häufigsten ausgeliehen wird!

4. Sorge mit dem SQL-Editor dafür, dass Herr Albrecht zusätzlich auch Klassenlehrer der (neu einzufügenden) Klasse 8a wird und Herr Brescher soll zusätzlich Klassenlehrer der (neu einzufügenden) 8b werden. Füge ebenfalls mit dem SQL-Editor in die beiden neuen Klassen jeweils mehrere Schüler ein!

a) Erstelle nun eine Liste mit allen Klassenlehrern, ihren Klassen und ihren Schülern!

b) Erstelle eine Liste mit allen Klassenlehrern, ihren Klassen und (nur) der Anzahl ihrer Schüler in den einzelnen Klassen!

c) Erstelle eine Liste mit allen Klassenlehrern und der Gesamtanzahl ihrer Schüler (egal in welcher Klasse diese sind)!

d) Ermittle denjenigen Klassenlehrer, der die meisten Schüler hat!

Lösung

Aufgabe 1

```
SELECT Buchtitel, Schuelername
FROM buchsorte JOIN entleihvorgang USING(Buchtitel)
     JOIN schueler USING(Schuelernummer)
ORDER BY Buchtitel, Schuelername
```

Aufgabe 2

```
SELECT COUNT(Schuelernummer) AS Anzahl, Buchtitel
FROM entleihvorgang
GROUP BY Buchtitel
ORDER BY Anzahl DESC
```

Aufgabe 3

```
SELECT Buchtitel, MAX(Anzahl)
FROM (SELECT Buchtitel, COUNT(Schuelernummer) AS Anzahl
      FROM entleihvorgang
      GROUP BY Buchtitel) AS T
```

Aufgabe 4a

```
SELECT  Klassenlehrer, Klassenname, Schuelername
FROM klasse Join schueler USING(Klassenname)
ORDER BY Klassenlehrer,  Klassenname, Schuelername
```

Aufgabe 4b

```
SELECT  Klassenlehrer, Klassenname, COUNT(Schuelername)
FROM klasse Join schueler USING(Klassenname)
GROUP BY Klassenname
ORDER BY Klassenlehrer, Klassenname
```

Aufgabe 4c

```
SELECT  Klassenlehrer, COUNT(Schuelername)
FROM klasse Join schueler USING(Klassenname)
GROUP BY Klassenlehrer
ORDER BY Klassenlehrer
```

Aufgabe 4d

```
SELECT MAX(T.Anzahl)
FROM klasse JOIN   (SELECT  Klassenlehrer, COUNT(Schuelername) AS Anzahl
                    FROM klasse Join schueler USING(Klassenname)
                   GROUP BY Klassenlehrer) AS T
             USING(Klassenlehrer)
```

Umwandlung einer Excel-Datei in eine Datenbank

Zunächst speichert man die vorhandene Excel-Datei als ***.csv - Datei*** (Trennzeichen-getrennt)!
Hinweis: Probleme ergeben sich allerdings, wenn in einer Excel-Zelle zwei Zeilen vorhanden sind. Das sollte man vermeiden!

Danach erzeugt man in *MySQL* eine neue Datenbank mit beliebigem Namen. Dabei sollte man unter der Option *Kollation* zum Beispiel den Schriftsatz ***utf8_german2_ci*** wählen. Dadurch werden auch die deutschen Umlaute richtig dargestellt.
Anschließend importiert man in dieser neuen Datenbank die obige **.csv-Datei*.
Bei dem Import dieser Datei hat man einige Auswahlmöglichkeiten:

Formatspezifische Optionen:

☐ Daten aktualisieren, wenn doppelte Schlüssel beim Importieren erkannt werden (ON DUPLICATE KEY UPDATE hinzufügen)

Spalten getrennt mit: ;

Spalten eingeschlossen von:

Spalten escaped mit:

Zeilen enden auf: auto

☑ Die erste Zeile der Datei enthält die Spaltennamen *(wenn diese Option nicht aktiv ist, wird die erste Zeile als Datenzeile interpretiert)*

☐ Bei INSERT Fehler nicht abbrechen

OK

Bei der Option *„Spalten getrennt mit“* sollte man das Semikolon wählen (das hängt natürlich davon ab, wie die *.*csv-Datei* strukturiert ist (MSExcel strukturiert anders als z.B. Libre-Office). Aber das erkennt man leicht, wenn man diese mit einem Texteditor, z.B. *Notepad++*, öffnet).
Wenn die Excel-Datei in der ersten Zeile die Spaltenüberschriften enthält, sollte man hier oben das entsprechende Häkchen setzen.
Die anderen Wahlmöglichkeiten stellt man so, wie oben dargestellt, ein.

Als Ergebnis erhält man eine Datenbank, welche eine einzige Tabelle enthält.

Aufgabe:
Erstelle eine Excel-Tabelle mit folgenden Spalten: Nachname, Vorname, Gebdat, Klasse, Wohnort, Geschlecht!
Die Tabelle sollte etwa 20 fiktive Datensätze enthalten. Sorge dafür, dass mindestens zwei Datensätze den Nachnamen Meier haben und mindestens drei Datensätze den Nachnamen Schmidt!
Konvertiere diese Tabelle in eine Datenbank!

Sortiere anschließend diese Datenbank nach verschiedenen Kriterien!

Erstelle eine Liste aller Mädchen, die älter als 15 Jahre sind!

Versuche, (nur) alle Datensätze mit demselben Nachnamen aufzulisten!

Abfragen mit *SELECT*

Eine **Abfrage** mit *SQL* dient dazu, die relationalen Operationen ***Projektion*** und ***Selektion* zu realisieren**. (Eine Projektion liefert eine bzw. mehrere Tabellenspalten, eine Selektion liefert eine bzw. mehrere Tabellenzeilen, also Datensätze). Wenn die Abfrage sich auf *mehrere Tabellen* bezieht, wird vorher ein *Join* (entspricht der Vereinigung von Tabellen) durchgeführt.
Das Ergebnis einer SELECT-Abfrage ist nach dem relationalen Modell wieder eine Tabelle, die evtl. auch aus nur einer Zeile und Spalte bestehen oder sogar ganz leer sein kann (daraus folgt, dass SELECT-Abfragen sich schachteln lassen). Diese Tabelle wird bei einer Auswahl-Abfrage nicht etwa neu angelegt, sondern stellt nur eine Sicht auf die Datenbank dar.

Die Syntax des Standard-SELECT-Befehls, mit dem solche Abfragen in *SQL* formuliert werden können, ist recht kompliziert, vor allem wegen der theoretisch beliebigen Schachtelungstiefe durch Unterabfragen.

Eine einfache *SQL*-Abfrage an unsere Datenbank wie

```
SELECT  *  FROM buchtyp;
```

liefert uns alle Zeilen und Spalten der Tabelle *buchtyp*, dabei steht das Symbol * für "alle Spalten der Tabelle".

Werden anstatt des Symbols * Spaltennamen (Attribute) angegeben, liefert der Befehl nur diese Spalten (Projektion):

```
SELECT  Titel, Autor  FROM buchtyp;
```

Beachte, dass damit auch die Reihenfolge der Spalten festgelegt wird. In der Ausgabe steht die Spalte *Titel* vor der Spalte *Autor*.

Mit der Anweisung DISTINCT werden doppelte oder gar mehrfache Ausgaben vermieden:

```
SELECT  DISTINCT Autor  FROM buchtyp;
```

Die *WHERE*-Klausel erzeugt eine **Einschränkungsbedingung**:

```
SELECT  Autor, Erscheinungsjahr  FROM buchtyp
WHERE Erscheinungsjahr > 1995;
```

```
SELECT Name
FROM Student
WHERE Name LIKE 'F%';
```

listet die Namen aller Studenten auf, deren Name mit F beginnt. (im Beispiel etwa: Fichte und Fauler).

LIKE kann mit verschiedenen Platzhaltern belegt werden: **_** steht für genau ein fehlendes Zeichen, **__** für zwei fehlende Zeichen und `%` steht für eine beliebige Zeichenfolge (auch für kein Zeichen). Bei der ***SQL-Version* von Microsoft** steht manchmal (leider nicht immer eindeutig) auch `*` für eine beliebige Zeichenfolge und **?** für ein einzelnes Zeichen. So können mit der Abfrage auch Felder nach Inhalt durchsucht werden.

Man kann den ***LIKE*-Operator** auch zusammen mit **NOT** benutzen: **NOT LIKE**
Hinweis: die Anweisung *LIKE* macht Probleme, wenn z.B. deutsche Umlaute benutzt werden. Beispiel ….. `WHERE StadtName LIKE 'Lü%';` liefert nicht nur Lünen und Lüneburg, sondern auch z.B. Lyon.

```
SELECT * FROM meineTabelle
WHERE feldName1 LIKE '%abc%';
```

Es werden alle Zeilen der Tabelle gelesen, deren Element in der Spalte *feldName1* den Teilstring 'abc' enthält).

Suchbegriff	gefundene Einträge
Beck%	Beck, Beckmann
%mann	Vormann, Hoffmann, Beckmann
L%t	Lippert, Lucht
P%l%	Paul, Paulaner, Pollmann
_p%	Spix, Opp, Apelt
____	Namen mit 4 Zeichen (Küll, Abba)
% %	Findet Namen aus zwei Wörtern (Leerzeichen bei Müller Lüdenscheid)

```
SELECT * FROM meineTabelle
WHERE feldName1 IN (11, 13, 17);
SELECT * FROM meineTabelle
WHERE feldName1 IN ('Meier', 'Müller' );
```

Es werden diejenigen Zeilen selektiert, in denen der Wert von *feldName1* in der angegebenen Menge enthalten ist.

```
SELECT * FROM Telefon
WHERE Vorwahl IN ('030', '03342', '040');
```

```
SELECT * FROM meineTabelle1
WHERE feldName1
IN (SELECT feldName2 FROM meineTabelle2);
```

Wie vorher, aber die angegebene Menge ist das Resultat einer weiteren Abfrage (mit einspaltigem Ergebnis).

```
SELECT attribut1, attribut2
FROM meineTabelle1 AS A, meineTabelle2 AS B
WHERE A.Nummer > 100;
```

Hier werden sog. **Alias**namen angelegt, die man im Weiteren auch benutzen kann.
Hinweis: In manchen *SQL*-Entwicklungsumgebungen kann das Schlüsselwort *AS* auch weggelassen werden. Man sollte es jedoch benutzen, weil es das Verständnis erleichtert.

Es ist entscheidend, an welcher Stelle ein Aliasname definiert wird, weil die *SQL*-Anweisungen hierarchisch operieren. In obigem Beispiel wird (logischerweise) zuerst die *FROM*-Anweisung ausgeführt. Man muss ja zuerst wissen, mit welchen Tabellen man überhaupt arbeiten soll. Wenn man hier einen Alias-namen festlegt, wird der im nächsten Schritt bekannt sein.
Danach wird die *WHERE*-Anweisung ausgeführt, damit die Anzahl der weiter zu verarbeitenden Datensätze schon mal möglichst gering sein wird.
Zum Schluss werden die Datensätze projiziert auf die gewünschten Spalten bzw. Attribute.

Folgendes Beispiel wäre also fehlerhaft:

```
SELECT attribut1 AS C, attribut2
FROM meineTabelle
WHERE  C > 100
```

Begründung: Bei der Ausführung der *WHERE*-Anweisung ist der Alias C noch gar nicht bekannt.

SELECT SUBSTR(Name, 1, 5) FROM meineTabelle;
Mit der Abfrage ***SUBSTR(Name, Startposition, Länge)*** kann man Teilstrings extrahieren.

```
SELECT * FROM meineTabelle
WHERE feldName1 IS NULL AND feldName2 IS NOT NULL;
```

SQL returniert ***NULL***, wenn ein Feld leer ist. Es gibt normalerweise keine Leerstrings. *NULL* kann nicht mit Vergleichsoperatoren geprüft werden, sondern mit **IS NULL** bzw. **IS NOT NULL**.

Das Ergebnis einer Abfrage ist wieder eine Tabelle, die aber keinen eigenen Namen hat. Wenn man mit dieser Ergebnistabelle weiterarbeiten will (insbesondere in geschachtelten *SQL*-Anweisungen), benötigt man zwingend einen Aliasnamen für diese Ergebnistabelle:

```
(SELECT titel, verfasser, auflage
 FROM Band
 WHERE auflage = '2006') AS Baende2006
```

```
SELECT Spaltenname1 AS "neuerName"
FROM meineTabelle
WHERE Spaltenname2 = 'xxx';
```

Ausgabe aller Spaltenname1-Inhalte (Überschrift der neu erzeugten Spalte ist "neuerName", bei denen in zweitangesprochener Spalte 'xxx' steht.
Beachte die unterschiedlichen Hochkommata!

Abfragen mit Vergleichs- und logischen Operatoren

Es sollte keine Probleme bereiten, die folgenden Abfragen nachzuvollziehen.
Es sei angemerkt, dass "!=“ oder auch "<>“ ungleich bedeutet.

```
SELECT * FROM Kunde WHERE KundenID = 3411;
SELECT * FROM Kunde WHERE KundenID > 3411;
SELECT * FROM Kunde WHERE KundenID >= 3411;
SELECT * FROM Kunde WHERE KundenID != 3411;
SELECT * FROM Kunde WHERE KundenID <> 3411;
SELECT * FROM Kunde WHERE KundenID IS NOT 3411;
SELECT * FROM Kunde WHERE KundenID BETWEEN 4000 AND 5000;
SELECT * FROM Kunde WHERE Name BETWEEN 'C' AND 'L ';
SELECT * FROM Kunde WHERE KundenID IN (3411, 3412);
SELECT * FROM Kunde WHERE KundenID NOT IN (3411, 3412);
SELECT * FROM Kunde WHERE KundenID = 3411 AND
Name = 'Bugs Bunny';
SELECT * FROM Kunde WHERE KundenID > 3000 OR
Name = 'Bugs Bunny';
```

Natürlich ist es auch jetzt möglich, nur einzelne Spalten auszugeben.

```
SELECT Name, Anschrift
FROM Kunde
WHERE KundenID > 3000 AND Name > 'Bugs Bunny' ;
```

Die Aggregat-Funktionen *COUNT, MAX, MIN, SUM, AVG*

Mit diesen Aggregatfunktionen (es gibt noch weitere) kann man statistische Auswertungen numerischer Daten in einer Datenbank vornehmen. Jede Aggregatfunktion wird auf ein Attribut einer Tabelle angewendet und liefert als Ergebnis nur einen einzigen Zahlenwert zurück.

Die Funktion ***COUNT*** kann in zwei verschiedenen Versionen genutzt werden: **COUNT(Spaltenname)** liefert die Anzahl der nichtleeren Zellen in der angegebenen Spalte.

Will man nur die Anzahl der unterschiedlichen Werte einer Spalte erhalten, so benutzt man **COUNT(DISTINCT** Spaltenname).

COUNT(*) hingegen gibt die Anzahl aller Zeilen (*NULL*-Werte werden mitgezählt) gemäß dem GROUP-BY-Abschnitt zurück. Dabei wäre es dann auch egal, welches Attribut man in der Klammer dieser Anweisung schreiben würde: Deshalb das Sternchen * .

```
SELECT COUNT(Klassenname)
FROM schueler;
```

liefert:

Count(Klassenname)
18

Leider wurden hier mehrfach vorkommende Klassennamen auch mehrfachgezählt. In dieser Tabelle gibt es genau achtzehn Datensätze.

```
SELECT COUNT(DISTINCT Klassenname)
FROM schueler;
```

liefert:

Count(DISTINCT Klassenname)
7

Es gibt in dieser Tabelle nur 7 unterschiedliche Klassen.

```
SELECT COUNT(*)
FROM schueler
WHERE Klassenname = '7b';
```
liefert:

Count(*)
3

Es gibt in der Klasse 7b nur 3 Schüler.

MAX, MIN, SUM, AVG

Auch diese Abfragen dürften nicht schwer zu verstehen sein; deshalb nur ein paar Beispiele:

```
SELECT MAX(schuelernummer) FROM schueler;
SELECT MIN(adresse) FROM schueler;    // alphabetisches Minimum wird gesucht.
SELECT MIN(adresse) FROM schueler WHERE klassenname = '7b';
SELECT SUM(schuelernummer) FROM schueler WHERE klassenname = '7b'; // sinnlos, aber funktioniert.
SELECT AVG(schuelernummer) FROM schueler; // auch sinnlos
SELECT MIN(schuelernummer), MAX(schuelernummer), SUM(schuelernummer), AVG(schuelernummer) fROM schueler; // auch sinnfrei
```

Hinweis: In einer ***WHERE*-Klausel** ist keine Aggregatfunktion möglich!

Gruppierung mit *GROUP BY*

In den obigen Beispielen gab es immer nur eine Zeile in der Ausgabe als Ergebnis des Funktionsaufrufs. Mit einer ***GROUP BY***-Klausel können wir auch die Ausgabe mehrerer Zeilen erreichen:
Hat man die Datensätze mittels ***GROUP BY*** in Gruppen eingeteilt, so kann man nur noch Angaben über die Gruppen machen, nicht mehr über einzelne Datensätze.

```
SELECT zahlungsEmpfaenger, SUM(Betrag)
FROM rechnungen
GROUP BY zahlungsEmpfaenger;
```

GROUP BY reduziert die Zeilen pro Group-Wert auf eine Zeile.
GROUP BY wird normalerweise zusammen mit Aggregatfunktionen (*COUNT, MIN, MAX, SUM, AVG* ...) genutzt.

Die Anweisung GROUP BY spaltet (als Zwischenschritt) die eigentliche Tabelle in entsprechend viele Teiltabellen auf, berechnet dann mit einer Aggregatfunktion den gewünschten Wert und gibt anschließend diese Teilergebnisse für jede Gruppe als Einzeiler aus.

```
SELECT klassenname, COUNT(*)
FROM schueler
GROUP BY klassenname;
```

Obige Abfrage liefert für unsere Tabelle folgendes Ergebnis:

Klassenname	**COUNT(*)**
5a	3
5b	3
6a	3
6b	3
7b	3
7c	3
7f	1

wichtig: **Jeder *SELECT*-Befehl darf einschließlich aller Unterabfragen höchstens eine *GROUP BY*-Klausel enthalten.**

Typischer Fehler:
Der typische Fehler im Zusammenhang mit der Gruppierung besteht darin, dass nach der Gruppenbildung noch versucht wird, Informationen von einzelnen

Datensätzen auszugeben. Zum Beispiel:
SELECT Region, *Name*, SUM(Einwohner)
FROM cia
GROUP BY Region

Man kann zwar für jede Gruppe die Region und die Summe der Einwohner angeben, aber nicht den Namen, denn der ist für jedes Land einer Region anders.

Die meisten *SQL*-Entwicklungsumgebungen weisen obige *SELECT*-Anweisung mit einer Fehlermeldung zurück. Sinngemäß heißt es meist, dass das Attribut *Name* nicht Bestandteil der *GROUP BY*-Klausel ist. *MySQL* bildet eine Ausnahme, akzeptiert die *SELECT*-Anweisung und liefert dann aber ein nicht sinnvolles Ergebnis.

Merke also:
Falls gruppiert wird, müssen alle mit *SELECT* ausgewählten Attribute (außer dem, nach dem gruppiert wird) mit einer Aggregatfunktion versehen werden.

Gruppenauswahl - die HAVING-Klausel

Die *HAVING*-Klausel ist wie die *WHERE*-Klausel, nur dass sich die angegebene Bedingung auf das Ergebnis einer Aggregationsfunktion bezieht, zum Beispiel **`HAVING SUM(betrag) > 0.`**

Wenn man nicht an allen Gruppen interessiert ist, so kann man mit der **HAVING-Klausel** die in Frage kommenden Gruppen auswählen. Bildlich bedeutet das, dass man einige der entstandenen Gruppen von der weiteren Betrachtung ausschließt.

Aber nach welchen Kriterien kann man Gruppen auswählen? Das geht nur mit einer Gruppeneigenschaft, also entweder danach, wie die Gruppen gebildet wurden oder mit einer Aggregatfunktion, die eine Gruppeneigenschaft bestimmt. Deshalb wird in einer ***HAVING*-Klausel** in aller Regel eine Aggregatfunktion benutzt, während in einer ***WHERE*-Klausel** keine Aggregatfunktion möglich ist.

Beispiel:
Es sollen die Regionen angezeigt werden, die mehr als 100 Millionen Einwohner haben.

```
SELECT region, SUM(einwohner)
FROM cia
GROUP BY region HAVING SUM(einwohner) > 1E08
```

Neues Beispiel:
Wir betrachten nun nur die Länder mit mehr als 100 Millionen Einwohner, gruppieren diese nach ihrer Region (so dass die einzelnen Länder nicht mehr erkennbar sind), zeigen von all diesen Regionen aber nur diejenigen Regionen mit (aus diesen *>100Mill-Ländern* gebildeten) mehr als 250 Millionen Einwohnern an:

```
SELECT region, SUM(einwohner)
FROM cia
WHERE einwohner > 1E08
GROUP BY region
HAVING SUM(einwohner) > 250000000
```

Falls es im obigen Beispiel eine Region geben sollte, welche nur 3 Länder enthält und in jedem dieser 3 Länder 90 Millionen Einwohner leben sollten, so wird diese Region bei obiger Abfrage nicht ausgegeben.

Kommen in einer Abfrage ***WHERE***, ***GROUP BY*** und ***HAVING*** vor, so wird in dieser Reihenfolge das Ergebnis bestimmt. Zuerst werden die zu betrachtenden Datensätze mittels ***WHERE*** selektiert, dann werden diese mit ***GROUP BY*** gruppiert und zum Schluss werden mit ***HAVING*** die gewünschten Gruppen ausgewählt.

Wie ein *SQL*-Interpreter einen speziellen Befehl abarbeitet, ist für den Anwender uninteressant. Zum besseren Verständnis (und auch zur Erkennung bzw. Vermeidung von Fehlern) ist es aber sinnvoll, die einzelnen Klauseln in einer bestimmten Reihenfolge zu betrachten, um sich selbst die Arbeitsweise zu veranschaulichen. Hierbei ist folgendes Modell hilfreich:

Klausel	Betrachtungsschritt	Erläuterungen
SELECT	6	streicht alle nicht genannten Spalten
FROM	1	bildet das Kreuzprodukt aller genannten Tabellen
WHERE	2	streicht alle Zeilen, welche die Bedingung nicht erfüllen
GROUP BY	3	bildet Gruppen
HAVING	4	streicht alle Gruppen, welche die Bedingung nicht erfüllen. Nur mit Gruppierung verwendbar. Bezieht sich fast immer auf Aggregatfunktionen.
ORDER BY	5	sortiert auf- oder absteigend

Nach diesem Modell ist es also sinnvoll, sich zuerst die beteiligten Tabellen anzusehen, bei mehreren beteiligten Tabellen den *Join* durchzuführen und anschließend die Selektionen und Anordnungen zu betrachten. Ganz zuletzt ist dann die Projektion auf die benannten Spalten durchzuführen.

Einfache Auswahl-Abfragen

Betrachte folgende Beispielsabfrage:

```
SELECT *
FROM buchsorte
WHERE buchtitel like 'Learning%'
ORDER BY buchtitel
```

Mittels obiger Schrittmethode zum Verständnis des *SQL*-Befehls lässt dieser sich leicht verstehen:

1. Die Tabelle *Buchsorte* wird bearbeitet (*FROM*)

2. Es werden diejenigen Zeilen (Datensätze) gesucht, welche die Bedingung hinter *WHERE* erfüllen.

3. Die gefundenen Datensätze werden nach dem Buchtitel alphabetisch geordnet.

4. Ausgegeben werden alle Spalten (Attribute).

Wir wollen nun eine Abfrage direkt in *SQL* formulieren. Uns interessiert, wie viele Bücher die verschiedenen Verlage in unserer Bibliothek stellen. Haben wir im *SQL*-Editor bereits die Tabelle *Buchtyp* ausgewählt, so ist das Grundgerüst unserer Abfrage bereits im Editor-Fenster zu sehen:

Die Aggregat-Funktion ***COUNT*** lässt sich zur Gruppierung verwenden, so dass wir ergänzen können:

```
SELECT COUNT(titel) AS Anzahl, Verlag
FROM Buchtyp
GROUP BY Verlag
```

Nach der Eingabe obiger Anweisungen wird das Ergebnis der Abfrage angezeigt.

Anzahl	Verlag
24	
6	Addison-Wesley
4	Autodesk
2	BI
1	Birkhäuser
41	Borland
2	Central Point
2	Corel
1	Digitalk
2	Dümmler
2	Hanser
1	Heise
1	IBM
1	IWT
2	Klett

Sie lässt erkennen, dass wir mit **`AS Anzahl`** eine Spaltenüberschrift definiert haben, die zweite Spalte zeigt die Verlagsnamen an. Die *NULL*-Einträge bei 24 Büchern lassen sich durch einen Blick in die Tabelle *Buchtyp* leicht erklären: Das sind Bücher, für die wir bisher in der Ausgangstabelle noch keinen Verlagsnamen eingetragen haben - unsere Datenbank besitzt demnach noch einige „Löcher", die noch zu schließen sind.

Wollen wir die Ausgabe der Buchanzahlen ohne Verlagsangabe unterdrücken, führt die Hinzunahme der **`HAVING`**-Klausel zum Ziel:

```
SELECT COUNT(Titel) AS Anzahl, Verlag
FROM Buchtyp
GROUP BY Verlag
HAVING Verlag IS NOT NULL
```

Weitere Beispiele:

Die folgende *SQL*-Anweisung liefert anhand der Tabelle *Kurs* die Kurse der Jahrgangsstufe 12 geordnet nach Lehrernummer:

```
SELECT KursNr, Thema, LehrerNr
FROM Kurs
WHERE (Jahrgangsstufe='12/I') OR (Jahrgangsstufe ='12/II')
ORDER BY LehrerNr;
```

Die nicht ausgeliehenen Spinde aus der Tabelle *Spind* erhält man so:

```
SELECT SpindNr, Standort
FROM Spind
WHERE SchuelerNr IS NULL;
```

Kombination mehrerer Tabellen mit *JOIN*

Bisher haben wir (fast immer) nur Daten aus jeweils einer einzigen Tabelle ausgewertet. Dazu braucht man eigentlich keine Datenbank. Die folgende *SQL*-Abfrage liefert hingegen Informationen, die man nur durch Kombination von zwei Tabellen erhalten kann.

```
SELECT *
FROM schueler, klasse
WHERE schueler.Klassenname = klasse.Klassenname
ORDER BY Klassenlehrer
```

Beachte, dass die Spalte Klassenname im erzeugten Listen-Ausdruck zweimal vorkommt! Begründung: In der *SELECT*-Anweisung steht, dass alle Spalten ausgegeben werden sollen. Und in der Bedingungsabfrage müssen die beiden Spalten nicht den gleichen Namen haben; es ginge auch:

WHERE schueler.spalte1 = klasse.spaltenname5

Das Ergebnis sieht so aus:

Schuelernummer	Schuelername	Adresse	Klassenname	Klassenname	Klassenlehrer
15	Weihershäuser, Gerlinde	Hauptstrasse 4, 45300 Bankhofen	5a	5a	Albrecht
13	Thomas, Laura	Rügener Strasse 6, 45300 Bankhofen	5a	5a	Albrecht
16	Onsberg, Karl-Heinz	Jupiterweg 7, 45300 Bankhofen	5a	5a	Albrecht
17	Jagoda, Else	Schulstrasse 3, 45300 Bankhofen	5b	5b	Brescher
18	Liborius, Konrad	Steinweg 3, 45300 Bankhofen	5b	5b	Brescher
14	Meier, Konstantin	Am Weiher 2, 45300 Bankhofen	5b	5b	Brescher
11	Chiong, Na Hu	Weiler 3, 45300 Bankhofen	6a	6a	Hingsen
9	Brentano, Luigi	Beim Italiener 3, 45300 Bankhofen	6a	6a	Hingsen
7	Abrasi, Tino	Falkenweg 9, 45300 Bankhofen	6a	6a	Hingsen
12	Matras, Ingo	Trassenweg 7, 45300 Bankhofen	6b	6b	Ingold
10	Engemann, Jasmin	Küppersweg 8, 45300 Bankhofen	6b	6b	Ingold
8	Zurren, Dina	Am Dorn 4, 45300 Bankhofen	6b	6b	Ingold
5	Kaplan, Torsten	Kirchgasse 1, 45300 Bankhofen	7c	7c	Lempel
3	Kaplan, Heike	Kirchgasse 1, 45300 Bankhofen	7c	7c	Lempel
2	Kruschwitz, Tolga	Ingwerstrasse 53, 45300 Bankhofen	7b	7b	Schmidt
6	Morman, Moriz	Ullinger Weg 2, 45300 Bankhofen	7b	7b	Schmidt
4	Prinz, Jochen	Am Berg 12a, 45300 Bankhofen	7b	7b	Schmidt

Dasselbe Ergebnis kann man auch mit der sog. ***Join***-Anweisung erhalten. Mit einem ***Join*** kann man zwei Tabellen zu einer Tabelle zusammenführen. Dabei wird immer ein Datensatz aus der ersten Tabelle mit einem Datensatz aus der zweiten Tabelle zu einem neuen Datensatz der neuen Tabelle zusammengesetzt. Das macht man natürlich nur für die Datensätze, die auch zusammen gehören.

Üblicherweise sollte man nur bestimmte Datensätze miteinander verknüpfen. Wenn man keine weiteren Verknüpfungsbedingungen angibt, dann würde jeder

Datensatz aus der ersten Tabelle mit jedem Datensatz aus der zweiten Tabelle kombiniert (was üblicherweise keinen Sinn ergibt).

```
SELECT *
FROM schueler JOIN klasse
ON schueler.Klassenname = klasse.Klassenname
ORDER BY Klassenlehrer
```

Mit der folgenden Anweisung wird die Spalte Klassenname nur einmal ausgedruckt, allerdings an erster Stelle:

```
SELECT *
FROM schueler JOIN klasse USING(Klassenname)
ORDER BY Klassenlehrer
```

Der Gebrauch von ***USING*** setzt also voraus, dass in beiden beteiligten Tabellen eine Spalte mit identischem Namen vorhanden ist.
Leider kann man bei *USING* keine zusätzlichen Bedingungen mehr angeben.
Also das Folgende würde nicht funktionieren:

```
SELECT *
FROM schueler JOIN klasse
USING(Klassenname) AND Klassenname = ‘5a‘
ORDER BY Klassenlehrer
```

Mit ON wäre es allerdings möglich:

```
SELECT *
FROM schueler JOIN klasse
ON schueler.Klassenname = klasse.Klassenname AND klasse.Klassenname = ‘5a‘
ORDER BY Klassenlehrer
```

Die Anweisung *Join* wird benutzt, um zwei Tabellen (üblicherweise unter einer Bedingung) miteinander zu verbinden. Die Anweisung *WHERE* soll nur für das Herausfiltern von einzelnen Datenzeilen innerhalb einer einzigen (evtl. gerade erst mit *JOIN* zusammengefügten) Tabelle genutzt werden.

Die oben benutzte Version

```
SELECT *
FROM schueler, klasse
WHERE schueler.Klassenname = klasse.Klassenname
```

gilt also als schlecht!

Will man nicht alle Spalten, sondern nur einzelne ausgeben, so kann es passieren, dass ein Spaltenname in beiden Tabellen identisch wäre. In dem Falle müsste man dann präziser angeben **SELECT tabelle1.spaltenname ….**

Abfragen lassen sich auch auf mehr als zwei Tabellen ausdehnen. Betrachten wir noch einmal die einzelnen Tabellen unserer Datenbank ***schulbuchverwaltung*:**

buchsorte(Buchtitel, ISBN),
entleihvorgang(Buchtitel, Leihnummer, Schuelernummer),
klasse(Klassenlehrer, Klassenname),
schueler(Adresse, Klassenname, Schuelername, Schuelernummer)

Gesucht sei im Folgenden eine Namensliste aller Schüler, die sich ein oder mehrere Bücher ausgeliehen haben, zusammen mit den zugehörigen ISBN-Nummern der ausgeliehenen Bücher. Dafür muss man drei Tabellen kombinieren:

SELECT schuelername, ISBN FROM
schueler JOIN (buchsorte JOIN entleihvorgang USING(Buchtitel)) USING(Schuelernummer)
ORDER BY schuelername

Unterabfragen

SQL-Anweisungen lassen sich auch schachteln. Eine Unterabfrage kann in der Attributliste der *SELECT*-Anweisung und als Vergleichsoperand in der *WHERE*- oder *HAVING*-Klausel verwendet werden.

Die Unterabfrage wird syntaktisch in runde Klammern gesetzt und vom Interpreter zuerst ausgewertet.

Da in der Unterabfrage in der *FROM*-Klausel auch eine andere Tabelle als in der Hauptabfrage benannt werden kann, ist damit eine – allerdings unübersichtlichere – Alternative zum *Join* gegeben.

In der ***WHERE*-Klausel** der Hauptabfrage einer geschachtelten *SELECT*-Anweisung darf man einen Vergleichsoperator oder den ***IN***-Operator verwenden. Dann darf allerdings die Unterabfrage nur eine einzige Spalte liefern.
Hat man in der ***WHERE*-Klausel** der Hauptabfrage einen Vergleichsoperator benutzt, so darf die Unterabfrage nur einen einzigen Wert als Ergebnis liefern.

Würde unsere Datenbank in der Tabelle *Buchtyp* noch das Attribut *Ausleihzahl* enthalten, um eine Ausleihstatistik aufzubauen, so könnte der folgende *SQL*-Befehl dazu dienen, die überdurchschnittlich ausgeliehenen Bücher herauszufinden (die Funktion ***AVG*** liefert den Mittelwert):

```
SELECT Titel, Ausleihzahl
FROM Buchtyp
WHERE Ausleihzahl > (SELECT AVG(Ausleihzahl)
                     FROM Buchtyp);
```

Um die ausgeliehenen Bücher mit den Daten zu finden, müssen die Tabellen *Ausleihbuch* und *Ausleihe* miteinander verknüpft werden. Die *SQL*-Abfrage mit Unterabfrage zur Ausgabe der ausgeliehenen Bücher verwendet die ***IN***-Klausel, damit das Ergebnis der Unterabfrage eindeutig ist:

```
SELECT DISTINCT *
FROM Ausleihbuch
WHERE InventarNr IN
    (SELECT InventarNr FROM Ausleihe
      WHERE Ausleihbuch.InventarNr = Ausleihe.InventarNr);
```

Ein analoges Ergebnis liefert der Join der beiden Tabellen mit dem zusätzlichen Vorteil, dass auf die Attribute beider Tabellen zurückgegriffen werden kann:

```
SELECT DISTINCT *
FROM Ausleihbuch JOIN Ausleihe
ON Ausleihbuch.InventarNr = Ausleihe.InventarNr;
```

Aktualisierungen

Erzeugen von Tabellen mit *CREATE*

Bislang dienten die *SQL*-Befehle dazu, Abfragen zu formulieren und damit Anfragen an die Datenbank zu richten. Die Ergebnistabellen stellen eine virtuelle Sicht des Benutzers auf die Datenbank dar, weil die Ergebnistabellen nicht als externe Dateien gespeichert werden.

Dies ist nach dem relationalen Datenbankmodell auch nicht sinnvoll, da sonst redundante Daten erzeugt würden. In der betrieblichen Realität sind allerdings bestimmte Abfragen das tägliche Brot der Datenbank-Anwendung und hier kann es ohne weiteres sinnvoll sein, nicht nur die Abfrage zu speichern, sondern aus Zeitgründen auch die fertige Ergebnistabelle.

Mit *SQL* lässt sich eine **Ergebnistabelle** direkt speichern mit

```
CREATE TABLE AS
  SELECT *
  FROM Buchtyp
  WHERE Erscheinungsjahr > 2000
  AS Neubuch;
```

Der Teil von *SQL*, mit dem man Tabellen und sogar ganze Datenbanken erstellt, wird ***DDL*** - *Data Definition Language* genannt.

Man kann natürlich auch komplett neue Tabellen innerhalb einer Datenbank erstellen, welche also nicht als Ergebnisabfrage zustande gekommen sind:

Mit *create table Tabellenname* erhält die Tabelle einen Namen. Dann gibt man alle Attribute der Tabelle samt ihren Datentypen an. Die Primärschlüssel kennzeichnet man mit *primary key*.

• varchar(n) n-Zeichen lange Zeichenkette
• int(n) Integer-Datentyp mit der Anzeigebreite n
• date Datumtyp vom Format 'JJJJ-MM-TT'
• enum(...) ein Aufzähltyp

create table schueler (
SNr **int**(5) **primary key**,
Nachname **varchar**(40),
Vorname **varchar**(30),
Geburtsdatum **date**,
StrasseNr **varchar**(40),
PLZ **varchar**(7),
Ort **varchar**(40)
);

create table Kurs (
KursNr **varchar**(10) **primary key**,
Fach **varchar**(5),
Thema **varchar**(50),
Art **enum**('GK', 'LK'),
Halbjahr **varchar**(10),
Stunden **int**(1)
);

create table Belegt (
SNr **int**(5),
KursNr **varchar**(10),
Punkte **int**(2),
primary key (SNr, KursNr)
);

Wesentlich wichtiger ist die Möglichkeit, die Datenbank zu bearbeiten bzw. zu aktualisieren. Dazu gibt es in *SQL* die Befehle: ***INSERT***, ***UPDATE*** und ***DELETE***.

Diese Befehlsgruppe wird auch als ***DML*** - *Data Manipulation Language* bezeichnet.

Hier geht es nicht darum, ein oder zwei Datensätze zu aktualisieren. Das könnte man mit dem *SQL*-Editor wesentlich einfacher und schneller erledigen. Die folgenden Aktualisierungsanweisungen werden vielmehr nur eingesetzt, wenn es um die Aktualisierung einer Vielzahl von Datensätzen geht.

Einfügen von Datensätzen mit *INSERT*

Wie fügt man Daten in eine Tabelle ein? Mit dem *INSERT*-Befehl kann man Daten in eine Tabelle einfügen:

INSERT INTO schueler **VALUES** (1024, 'Mueller', 'Heinz', '1988-08-14', 'Hüttenweg 6', '64536', 'Oberdorf');
INSERT INTO kurs **VALUES** ('12|34', 'Inf', 'Datenbanken', 'GK', '2019/01', 2);
INSERT INTO belegt **VALUES** (1024, '12I34', NULL);

Mit diesen drei *INSERT*-Befehlen fügt man in jeweils eine der drei Tabellen *schueler*, *kurs* oder *belegt* einen Datensatz ein. Ist ein Attributwert nicht bekannt, wie z.B. die Punktezahl für den belegten Kurs, so kann man dafür einen *NULL*-Wert benutzen.

Der normale *INSERT* - Befehl hat den Aufbau:
INSERT INTO Tabellenname **VALUES** (Wert_1, Wert_2,... , Wert_n);

Beim ***INSERT***-Befehl in der bisherigen Form muss man immer Werte für alle Spalten angeben. Bei der zweiten Form des ***INSERT***-Befehls ist das nicht nötig, denn hier gibt man hinter dem Tabellennamen die einzelnen Spalten an, die einen Wert erhalten sollen:

INSERT INTO Tabellenname(Spalte_1, Spalte_2, Spalte_3,...) **VALUES** (Wert_1, Wert_2, Wert_3, ...);

Beispiele:
INSERT INTO schueler(SNr, Nachname, Vorname) **VALUES** (1024, 'Mueller', 'Heinz');
INSERT INTO kurs(KursNr, Art, Thema) **VALUES** ('12I34', 'GK', 'SQL');

Änderung von Datensätzen mit *UPDATE*

Und wie ändert man Daten? Dafür gibt es den ***UPDATE***-Befehl. Auch dafür drei Beispiele:

UPDATE schueler **SET** Nachname = 'Müller' **WHERE** SNr = 1024;
UPDATE kurs **SET** Thema = 'Datenbanken mit SQL', Stunden = 3 **WHERE** KursNr = '12I34';
UPDATE belegt **SET** Punkte = 14 **WHERE** SNr = 1024 **AND** KursNr = '12I34';

Mit *UPDATE Tabellenname* gibt man an, in welcher Tabelle Daten geändert werden sollen.
Mit *SET Spalte = Wert* gibt man den neuen Wert für die Spalte an.
Mit *WHERE Bedingung* wählt man die Datensätze aus, die geändert werden sollen.

Fehlt die *WHERE*-Bedingung, so werden alle Datensätze geändert.
Gibt man in der *WHERE* -Bedingung einen Schlüssel an, so wird nur der zugehörige Datensatz geändert.

Der *UPDATE*-Befehl hat den Aufbau: **UPDATE** Tabelle **set** Spalte_1 = Wert_1, Spalte_2 = Wert_2,... **WHERE** Bedingung;

UPDATE kundentabelle
SET name = 'Gisela Müller'
WHERE name = 'Gisela Meier'

Zunächst wird festgelegt, welche Tabelle aktualisiert werden soll. Danach folgt der Spaltenname und der neue Wert. Das nun folgende WHERE ist wichtig, um das UPDATE auf eine Zeile zu beschränken, denn ohne WHERE würden alle Zeilen der Tabelle *kundentabelle* aktualisiert werden.

Man ist aber nicht auf eine Spalte pro Update beschränkt:

UPDATE kundentabelle
SET name = 'Donald Duck', adresse = 'Entenhausen'
WHERE name = 'Emil Enterich'

```
UPDATE Versicherungsnehmer
SET Ort = 'Leipzig', PLZ = '04178'
WHERE PLZ = '04430';
```

Einzelne Datensätze kann man üblicherweise viel eleganter mit Hilfe des *SQL*-Editors ändern. Wirklich Sinn (und Freude) macht die Anweisung *UPDATE* eigentlich erst, wenn man viele Datensätze ändern will, wie im folgenden Beispiel:

```
UPDATE beamtentabelle
SET Gehalt = Gehalt * 1.5
WHERE Schulname = 'Goethe-Gymnasium-Dortmund'
```

Löschen von Daten

Man löscht Datensätze mit dem ***DELETE*** -Befehl. Das geht so:

DELETE FROM schueler **WHERE** SNr = 1024;
DELETE FROM kurs **WHERE** Fach = 'D';
DELETE FROM belegt;

Beim ersten *DELETE*-Befehl wird als *WHERE*-Bedingung ein Schlüsselwert angegeben. Dadurch wird genau ein Datensatz gelöscht.
Beim zweiten *DELETE* - Befehl werden alle Deutsch-Kurse gelöscht.
Beim dritten *DELETE* -Befehl fehlt die *WHERE* -Bedingung, dadurch werden alle Datensätze gelöscht.

Der *DELETE* -Befehl hat den Aufbau:
DELETE FROM Tabellenname **WHERE** Bedingung;

DELETE FROM kunden
WHERE name = 'Donald Duck'

Im Grunde sagt man nur, aus welcher Tabelle man etwas löschen möchte und dann, welchen Datensatz bzw. welche Datensätze genau. Auch hier ist es wieder wichtig, die *WHERE*-Bedingung einzusetzen, sonst werden alle Datensätze einer Tabelle gelöscht.

In einer Bücherei könnte die Aktualisierung dazu verwendet werden, Ausleiher-Nummern aus datenschutzrechtlichen Gründen zu löschen.

Tabellen leeren und löschen

Mit *DELETE FROM Tabellenname* löscht man (nur) **alle Datensätze** einer **Tabelle**.
Aber die Tabelle namens *Tabellenname* existiert immer noch. Nur ihr gesamter Inhalt wurde gelöscht.

Zum Löschen einer **Tabelle** nimmt man den *DROP Tabellenname* - Befehl,
z. B. **DROP** kurs

Danach existiert die Tabelle namens *kurs* nicht mehr!

Die Datenbank „World“

Erzeuge zunächst eine neue leere Datenbank mit dem Namen *„world“*. Öffne diese leere Datenbank und importiere anschließend die vom Lehrer zur Verfügung gestellte Datei namens `world.sql` (Die Aktualität der Daten entspricht etwa dem Stand aus dem Jahre 2010). Leider kann man nicht ohne weiteres beliebig große Dateien importieren. Die maximal zu ladende Größe wird angezeigt. Sie beträgt im Jahre 2019 etwa 2 MB. Größere Dateien zu laden ist zwar auch möglich, erfordert aber zum Beispiel Änderungen in den Einstellungen von *MySQL*.
Interessanterweise werden hierdurch nur einzelne Tabellen in die (vorher leere) Datenbank importiert. Diese muss natürlich schon geöffnet sein.

Die Datenbank *World* enthält folgende Tabellen:

city(ID, Name, CountryCode, District, Population)
country(Code, Name, Continent, Region, SurfaceArea, IndepYear, Population, LifeExpectancy, GNP, GNPOld, LocalName, GovernmentForm, HeadOfState, Capital, Code2)
countrylanguage(CountryCode, Language, ISOfficial, Percentage)

Bemerkung: In der Tabelle *countrylanguage* bilden die beiden Attribute *CountryCode* und *language* zusammen einen Primärschlüssel.

Schreibe in die rechte Spalte der nachfolgenden Tabelle die zugehörigen *SQL*-Anweisungen!

Aufgabe	**SQL-Anweisung**
Alle Städte, die mit „Do“ beginnen	
Alle Städte, nach Einwohnerzahl geordnet	
Alle Städte, nach Einwohnerzahl geordnet, mit Information über Distrikt und Land	
Alle deutschen Städte, nach Einwohnerzahl geordnet, mit Information über Distrikt	
Alle Städte, die mit „Do“ beginnen, nach Einwohnerzahl geordnet, mit Information über Distrikt und Land	
Die größte Einwohnerzahl (ohne Angabe des zugehörigen Stadtnamens)	
Die Stadt mit der größten Einwohnerzahl	
Alle Länder, nach Anzahl der Städte geordnet	
Alle Paare von Ländern, die eine offizielle Sprache gemeinsam haben, nach Summe der Prozentzahlen dieser Sprache geordnet	

Lösung:

Aufgabe	SQL-Anweisung
Alle Städte, die mit „Do“ beginnen	SELECT * FROM City WHERE Name LIKE ‘Do%’ *Bemerkung: evtl. LIKE ‘Do*’ eingeben. Achte auf richtige Hochkommata!*
Alle Städte, nach Einwohnerzahl geordnet	SELECT * FROM City ORDER BY Population DESC, Name ASC
Alle Städte, nach Einwohner-zahl geordnet, mit Information über Distrikt und Land	SELECT City.Name AS Stadt, District AS Bezirk, Country.Name AS Land, City.Population AS Einwohnerzahl FROM City JOIN Country ON CountryCode = Code ORDER BY City.Population DESC *Bemerkung: die neu gewählten Spaltenüberschriften kann man auch mit einfachen oder doppelten Anführungszeichen angeben, also z.B.* *City.Name AS ‚Stadt‘ oder auch City.Name AS „Stadt“*
Alle deutschen Städte, nach Einwohnerzahl geordnet, mit Information über Distrikt	SELECT City.Name AS Stadt, District AS Bezirk, City.Population AS Einwohnerzahl FROM City JOIN Country ON CountryCode = Code WHERE Country.Name = ‘Germany’ ORDER BY City.Population DESC *Interessant: Möglich wäre auch: ORDER BY Einwohnerzahl DESC*
Alle Städte, die mit „Do“ beginnen, nach Einwohner-zahl geordnet, mit Information über Distrikt und Land	SELECT City.Name AS Stadt, District AS Bezirk, Country.Name AS Land, City.Population AS Einwohnerzahl FROM City JOIN Country ON CountryCode = Code WHERE City.Name LIKE ‘Do%’ ORDER BY City.Population DESC
Die größte Einwohnerzahl (ohne Angabe des zugehörigen Stadtnamens)	SELECT MAX(Population) AS Maximum FROM City
Die Stadt mit der größten Einwohnerzahl	SELECT * From City WHERE Population = (SELECT MAX(Population) FROM City)
Alle Länder, nach Anzahl der Städte geordnet	*Nach Länderkürzeln:* SELECT CountryCode, COUNT(CountryCode) AS Anzahl FROM City GROUP BY CountryCode ORDER BY Anzahl DESC *Nach Ländernamen:* SELECT Country.Name, COUNT(Country.Name) AS Anzahl FROM City JOIN Country ON CountryCode = Code GROUP BY Code ORDER BY Anzahl DESC
Alle Paare von Ländern, die eine offizielle Sprache gemeinsam haben, nach Summe der Prozentzahlen dieser Sprache geordnet	SELECT (Sprache1.Percentage + Sprache2.Percentage) AS Prozent, Sprache1.Language, Land1.Name, Land2.Name FROM CountryLanguage AS Sprache1 JOIN Country AS Land1 ON Land1.Code = Sprache1.CountryCode JOIN CountryLanguage AS Sprache2 ON Sprache1.Language = Sprache2.Language JOIN Country AS Land2 ON Land2.Code = Sprache2.CountryCode WHERE Sprache1.IsOfficial = 'T' AND Sprache2.IsOfficial = 'T' AND Land1.Name < Land2.Name ORDER BY Prozent DESC, Sprache1.Language ASC, Land1.Name ASC

Aufgabe:
Erstelle eine Liste aller Sprachen dieser Welt zusammen mit der Anzahl der Menschen, welche die jeweilige Sprache sprechen, egal ob es die offizielle Landessprache ist oder nicht. Die Liste soll absteigend nach der Anzahl der Menschen geordnet werden.

Lösung

```
SELECT Language, SUM(Percentage * Population / 100) AS Anzahl
FROM country JOIN countrylanguage ON CountryCode = Code
GROUP BY language
ORDER BY Anzahl DESC
```

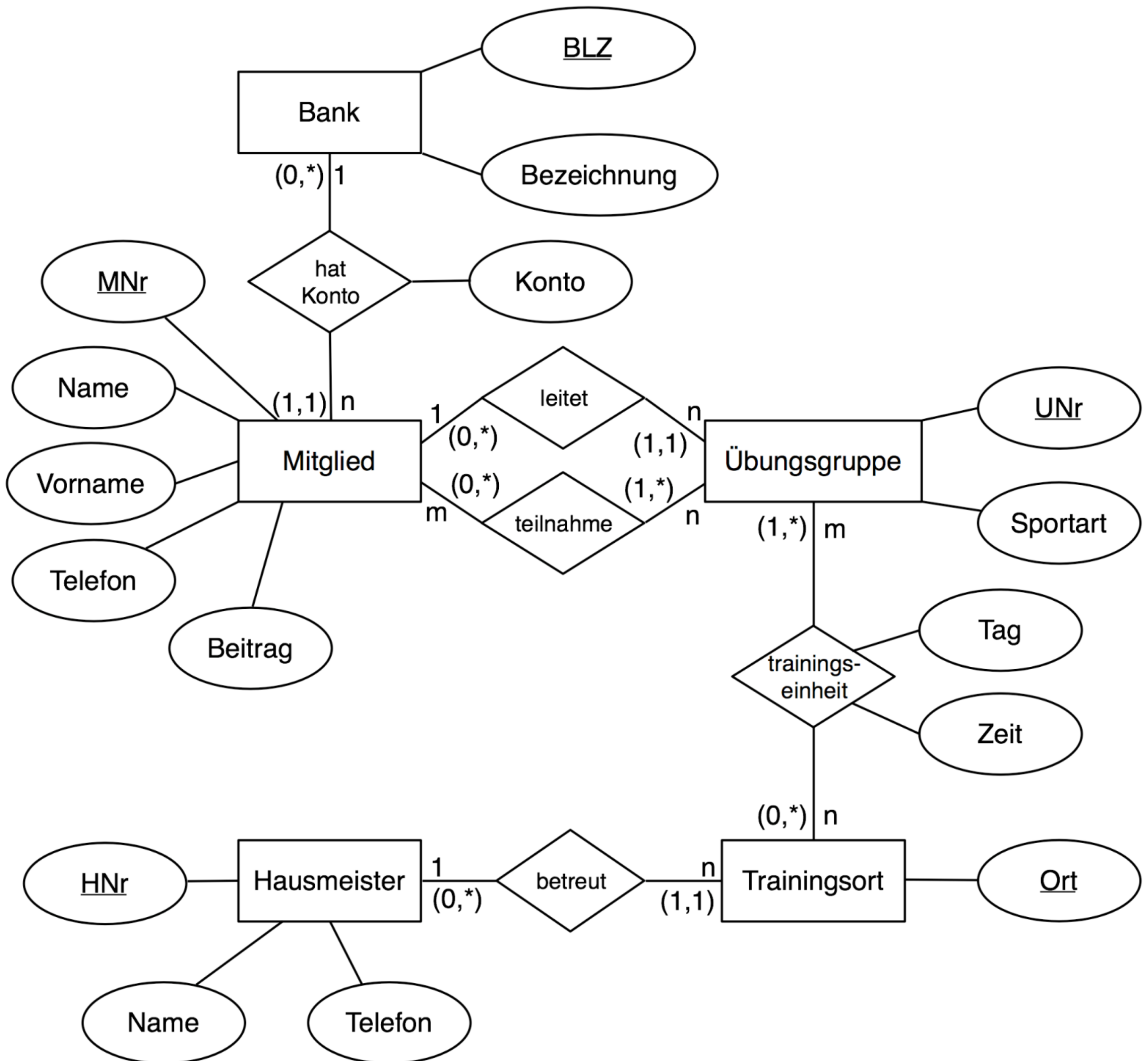
Aufgabe:
Liste alle Ländernamen auf, in denen Deutsch gesprochen wird. Die Liste soll nach der Anzahl der Menschen, die Deutsch sprechen, geordnet sein.

Lösung

```
SELECT Name, (Percentage * Population / 100) AS Anzahl
FROM country JOIN countrylanguage ON CountryCode = Code
WHERE language = 'German'
ORDER BY Anzahl DESC
```

Die Datenbank „Sportverein“

ER-Modell „Sportverein“



Relationenschemata der Datenbank „Sportverein"

mitglied (MNr, Name, Vorname, Telefon, ↑BLZ, Konto, Beitrag)
bank (BLZ, Bezeichnung)
teilnahme (↑MNr, ↑UNr)
uebungsgruppe (UNr, Sportart, ↑MNr) // MNr ist die Mitgliedsnummer des Leiters
trainingseinheit (↑UNr, ↑Ort, Tag, Zeit)
trainingsort (Ort, ↑HNr)
hausmeister (HNr, Name, Telefon)

Für die nachfolgende Aufgabe Nr. 18 ist die Kenntnis einiger SQL-Funktionen hilfreich:

Die Anweisung **`SELECT TIME_TO_SEC('22:23:00');`**
liefert das Ergebnis 80580
Das entspricht der Anzahl der Sekunden bei einer Zeit von 22 Stunden und 23 Minuten.

Analog liefert **`SELECT TIME_TO_SEC('00:39:38');`** die Zahl 2378

Die Anweisung
```
SELECT TIMEDIFF('2014-12-31 23:59:59','2014-12-30 01:01:01');
```
liefert die Zeitangabe '46:58:58'

Die Anweisung **`SELECT ABS(-24);`** liefert natürlich den Betrag: 24

Aufgaben zur Datenbank „Sportverein“

Nr	Aufgabe	Hinweise
1	Welche Trainings-orte gibt es?	
2	Welche Sportarten werden angeboten? (ohne Dopplungen)	
3	Wie viele Mitglieder hat der Verein?	
4	Welchen Beitrag nimmt der Verein insgesamt ein?	
5	Der Beitrag wird für alle Mitglieder um 20% angehoben.	
6	Für Übungs-gruppenleiter wird der Mitglieds-beitrag halbiert.	Man kann in die WHERE-Bedingung auch wieder eine komplette SELECT-Abfrage integrieren (Unterabfrage). Der IN-Operator gibt an, ob ein Wert in der Ergebnistabelle vorhanden ist.
7	Gib die Vornamen der Mitglieder nach Häufigkeit absteigend geordnet aus.	
8	Gib Namen und Nummer des Mitglieds mit der höchsten Mitgliedsnummer an.	Schachtelung zweier SELECT-Anweisungen
9	Ist Mitglied Zwick ein Übungsgruppen-leiter?	

10	Ordne die Sportarten nach Teilnehmerzahl!	
11	Bestimme die maximale Teilnehmerzahl einer Sportart.	Schachtelung zweier SELECT-Anweisungen (Unterabfrage). Das Ergebnis der inneren SELECT-Abfrage muss mit AS einen Aliasnamen erhalten.
12	Wie viele Mitglieder üben pro Woche in Bad1?	
13	Welche Sportarten übt Mitglied Zwick aus?	
14	Führe die Namen der Fußballer mit Konto bei der Sparkasse Rhynern nach Mitglieds-nummer geordnet auf.	
15	Übungsgruppenleit er Ellerbusch ist krank. Deshalb fallen seine Übungsstunden aus. Welche Haus-meister müssen informiert werden?	

16	Wie viele Mitglieder betreuen die verschiedenen Hausmeister?	
17	Gibt es Übungsgruppenleiter, die in der gleichen Sportart Teilnehmer sind?	Self-Join erforderlich
18	Gibt es bei den Trainingsorten Doppel-belegungen? Es sollen auch die betroffenen Übungsgruppen-leiter angezeigt werden.	Self-Join erforderlich. Hilfreiche Funktionen u.a.: **`TIMEDIFF, TIME_TO_SEC, ABS`**
19	Gibt es für Mitglieder Überschneidungen bei den Trainings-einheiten?	Sehr komplex, vermutlich mehrfacher Self-Join nötig. (Lösung passt hier nicht in diese Tabellenzelle)

20	Wie viele Mitglieder üben genau eine, zwei, drei, vier Sportarten aus?	Schachtelung zweier SELECT-Anweisungen (Unterabfrage). Das Ergebnis der inneren SELECT-Abfrage muss mit AS einen Aliasnamen erhalten.
21	Welchen Beitrag zahlt das Mitglied Beck?	
22	Liste alle Übungsgruppenleiter auf!	
23	Ordne die Sportarten nach Beliebtheit! Gib dabei die Teilnehmerzahl an!	
24	Welche Sportarten übt das Mitglied Auer aus?	

Lösungen:

Nr	Aufgabe	SQL-Anweisung
1	Welche Trainings-orte gibt es?	SELECT Ort FROM trainingsort
2	Welche Sportarten werden angeboten? (ohne Dopplungen!)	SELECT Sportart FROM uebungsgruppe GROUP BY Sportart oder: SELECT DISTINCT Sportart FROM uebungsgruppe
3	Wie viele Mitglieder hat der Sportverein?	SELECT COUNT(*) AS Anzahl FROM mitglied
4	Welchen Beitrag nimmt der Verein insgesamt ein?	SELECT SUM(Beitrag) AS Summe FROM mitglied
5	Der Beitrag wird für alle Mitglieder um 20% angehoben.	UPDATE mitglied SET Beitrag = Beitrag * 1.2
6	Für Übungsgruppen-leiter wird der Mitgliedsbeitrag halbiert.	UPDATE mitglied SET Beitrag = Beitrag / 2 WHERE mitglied.MNr IN (SELECT DISTINCT MNr FROM uebungsgruppe)
7	Gib die Vornamen der Mitglieder nach Häufigkeit abstei-gend geordnet aus.	SELECT Vorname, COUNT(Vorname) AS Anzahl FROM mitglied GROUP BY Vorname ORDER BY Anzahl DESC, Vorname ASC
8	Gib den Namen und die Nummer des Mitglieds mit der höchsten Mitglieds-nummer an.	SELECT Name, Vorname, MNr FROM Mitglied WHERE MNr = (SELECT MAX(MNr) FROM Mitglied)
9	Ist Mitglied Zwick ein Übungsgruppen-leiter?	SELECT uebungsgruppe.* FROM uebungsgruppe JOIN mitglied USING (MNr) WHERE Name = 'Zwick'
10	Ordne die Sportarten nach Beliebtheit. Gib dabei die Teilnehmerzahl an.	SELECT Sportart, COUNT(DISTINCT teilnahme.MNr) AS Anzahl FROM teilnahme JOIN uebungsgruppe USING(UNr) GROUP BY Sportart ORDER BY Anzahl DESC
11	Bestimme die maximale Teilnehmerzahl einer Sportart.	SELECT MAX(Anzahl) AS Maximum FROM (SELECT Sportart, COUNT(DISTINCT teilnahme.MNr) AS Anzahl FROM teilnahme JOIN uebungsgruppe USING(UNr) GROUP BY Sportart) AS t
12	Wie viele Mitglieder üben pro Woche in Bad1?	SELECT COUNT(DISTINCT teilnahme.MNr) AS Anzahl FROM teilnahme JOIN uebungsgruppe USING(UNr)

		JOIN trainingseinheit USING(UNr) WHERE Ort = 'Bad1'
13	Welche Sportarten übt Mitglied Zwick aus?	SELECT Sportart FROM uebungsgruppe JOIN teilnahme USING(UNr) JOIN mitglied ON mitglied.MNr = teilnahme.MNr WHERE Name = 'Zwick' SELECT Sportart FROM mitglied JOIN teilnahme USING (MNr) JOIN uebungsgruppe USING (UNr) WHERE Name = 'Zwick' SELECT Sportart FROM uebungsgruppe, teilnahme, mitglied WHERE Name = 'Zwick' AND mitglied.MNr = teilnahme.MNr AND uebungsgruppe.UNr = teilnahme.UNr
14	Führe die Namen der Fußballer mit Konto bei der Sparkasse Rhynern nach Mitgliedsnummer geordnet auf.	SELECT DISTINCT name, mitglied.MNr FROM uebungsgruppe JOIN teilnahme USING (UNr) JOIN mitglied ON mitglied.MNr = teilnahme.MNr JOIN bank USING (BLZ) WHERE Sportart = 'Fußball' AND Bezeichnung = 'Sparkasse Rhynern' ORDER BY mitglied.MNr
15	Übungsgruppenleiter Ellerbusch ist krank. Deshalb fallen seine Übungsstunden aus. Welche Haus-meister müssen informiert werden?	SELECT hausmeister.Name, Ort, Tag, Zeit FROM hausmeister JOIN trainingsort USING (HNr) JOIN trainingseinheit USING (Ort) JOIN uebungsgruppe USING (UNr) JOIN mitglied ON mitglied.MNr = uebungsgruppe.MNr WHERE mitglied.Name = 'Ellerbusch'
16	Wie viele Mitglieder betreuen die verschiedenen Hausmeister?	SELECT COUNT(DISTINCT teilnahme.MNr) AS Anzahl, Name FROM hausmeister JOIN trainingsort USING (HNr) JOIN trainingseinheit USING (Ort) JOIN uebungsgruppe USING (UNr) JOIN teilnahme USING (UNr) GROUP BY Name

17	Gibt es Übungsgruppen-leiter, die in der gleichen Sportart Teilnehmer sind?	**SELECT DISTINCT mitglied.Name, uebungsgruppeMod.Sportart FROM uebungsgruppe AS uebungsgruppeMod JOIN mitglied USING (MNr) JOIN uebungsgruppe AS uebungsgruppeTN ON uebungsgruppeTN.SportArt = uebungsgruppeMod.SportArt JOIN teilnahme ON uebungsgruppeMod.MNr = teilnahme.MNr AND uebungsgruppeTN.UNr = teilnahme.UNr WHERE uebungsgruppeTN.MNr != uebungsgruppeMod.MNr**
18	Gibt es bei den Trainingsorten Doppelbelegungen? Es sollen auch die betroffenen Übungs-gruppenleiter angezeigt werden.	**SELECT Einheit1.Ort, Einheit1.Tag, Einheit1.Zeit, Sportart, Einheit1.UNr, Name, Einheit2.UNr FROM uebungsgruppe JOIN mitglied USING (MNr) JOIN trainingseinheit AS Einheit1 USING (UNr) JOIN trainingseinheit AS Einheit2 ON Einheit1.Tag = Einheit2.Tag AND Einheit1.Ort = Einheit2.Ort WHERE ABS(TIME_TO_SEC(TIMEDIFF(Einheit2.Zeit,Einheit1.Zeit))) < 3600 AND Einheit1.UNr != Einheit2.UNr ORDER BY Einheit1.Ort, Einheit1.Tag, Einheit1.Zeit**
19	Gibt es für Mitglieder Überschneidungen bei den Trainingseinheiten?	**SELECT Name, Einheit1.Ort, Einheit1.Tag, Einheit1.Zeit, Übungsgruppe1.Sportart, Einheit2.Ort, Einheit2.Tag, Einheit2.Zeit, Übungsgruppe2.Sportart FROM mitglied JOIN teilnahme AS Teilnahme1 USING (MNr) JOIN uebungsgruppe AS Übungsgruppe1 USING (UNr) JOIN trainingseinheit AS Einheit1 USING (UNr) JOIN teilnahme AS Teilnahme2 ON Teilnahme2.MNr = mitglied.MNr JOIN uebungsgruppe AS Übungsgruppe2 ON Teilnahme2.UNr = Übungsgruppe2.UNr JOIN trainingseinheit AS Einheit2 ON Einheit1.Tag = Einheit2.Tag AND Übungsgruppe2.UNr = Einheit2.UNr WHERE Übungsgruppe1.UNr < Übungsgruppe2.UNr AND ABS(TIME_TO_SEC(TIMEDIFF(Einheit2.Zeit,Einheit1.Zeit))) < 3600 ORDER BY Name, Einheit1.Ort, Einheit1.Tag, Einheit1.Zeit** Gibt es eine elegantere Lösung?

20	Wie viele Mitglieder üben genau eine, zwei, drei, vier Sportarten aus?	**SELECT AnzahlSportarten, COUNT(AnzahlSportarten) FROM (SELECT mitglied.MNr, COUNT(DISTINCT Sportart) AS AnzahlSportarten FROM mitglied JOIN teilnahme USING (MNr) JOIN uebungsgruppe USING (UNr) GROUP BY mitglied.MNr) AS SportartenProMitglied GROUP BY AnzahlSportarten**

Aufgaben zu SQL-Abfragen

Aufgabe 1
Gegeben ist eine Datenbank mit folgenden Relationen:
LIEFERER(LiefererNr, LieferName, Strasse, Plz, Ort, Rabatt)
ARTIKEL(ArtikelNr, Artikel, Preis, GruppenNr, Meldebestand, Bestand)
ARTIKELLIEFERER(ArtikelNr, LiefererNr, BestellNr, Angebotspreis)
WARENGRUPPE(GruppenNr, Gruppe)

a) Was bewirkt folgende Anweisung?

```
SELECT DISTINCT ArtikelLieferer.LieferNr,
     ArtikelLieferer.ArtikelNr, Lieferer.LieferName
FROM Lieferer JOIN ArtikelLieferer
ON Lieferer.LieferNr = ArtikelLieferer.LieferNr
WHERE ArtikelLieferer.ArtikelNr = 1616;
```

b) Eine Lieferliste ist zu erstellen, die nach Lieferer-Ort geordnet ist!

c) Was bewirkt die folgende Aktionsabfrage?

```
UPDATE DISTINCT ArtikelLieferer
SET ArtikelLieferer.Angebotspreis
    = Angebotspreis*0,97
WHERE ArtikelLieferer.LiefererNr = 12345;
```

d) Welche Lieferanten gewähren einen Rabatt, der niedriger ist als der durchschnittliche Rabattsatz?

Lösung:

a) Es werden die beiden Tabellen *LIEFERER* und *ARTIKELLIEFERER* durch einen *Join* verbunden, wobei das Auswahlkriterium dieselbe Artikelnummer ist. Aus der entstandenen Tabelle werden die Artikeldaten derjenigen Zeilen (Datensätze) ausgegeben, welche die Artikelnummer 1616 besitzen.

b)
```
SELECT DISTINCT Lieferer.LieferName AS Lieferant,
Lieferer.Strasse, Lieferer.Ort
FROM LIEFERER
ORDER BY LIEFERER.ORT, Lieferer.LieferName
```

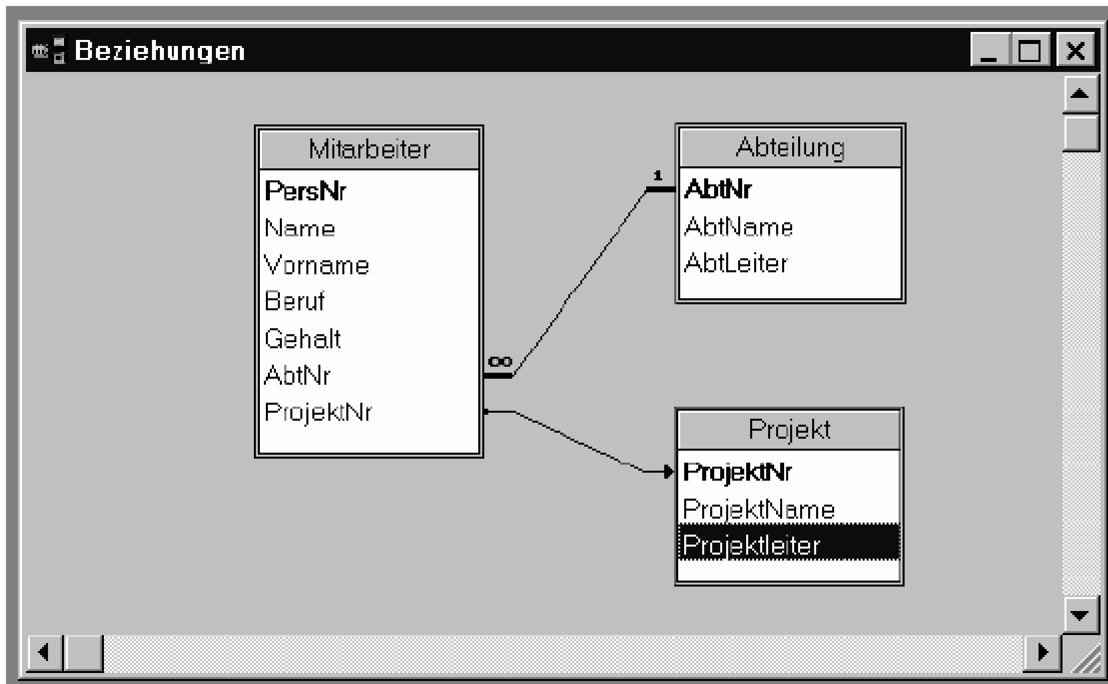
bewirkt eine Ausgabe in der Form:

Lieferant	**Ort**	**Strasse**
Müller&Co	Wiesbaden	Mainzer Str. 768
Testiege P.	Mainz	Im Rad 3
…	…	…

c) In der Tabelle *ARTIKELLIEFERER* wird ein Update an denjenigen Zeilen (Datensätzen) vorgenommen, welche die Lieferernummer 12345 besitzen. Der Angebotspreis wird auf 97% reduziert.

d) Die Abfrage lässt sich mit einer Unterabfrage bewältigen:
```
SELECT Lieferer.LieferName,Lieferer.Rabatt
FROM Lieferer
WHERE Lieferer.Rabatt < (SELECT AVG(Rabatt) FROM
                          Lieferer);
```

Aufgabe 2
Gegeben ist die folgende Datenbank:



In einer Firma gibt es Mitarbeiter. Jeder Mitarbeiter ist eindeutig einer Abteilung zugeordnet. Manche Mitarbeiter sind an einem Projekt beteiligt. Ein Mitarbeiter darf nur an einem Projekt arbeiten. Jede Abteilung hat einen Abteilungsleiter und jedes Projekt hat einen Projektleiter. Projektleiter und Abteilungsleiter sind natürlich auch Mitarbeiter.

Teste die folgenden *SQL*-Befehle nach Erstellen der Datenbank und beschreibe ihre Wirkung:

a) zu Projektion und Selektion

```
SELECT * FROM Mitarbeiter   WHERE PersNr = 28;
```

```
SELECT PersNr, Name, Vorname  FROM Mitarbeiter
WHERE AbtNr = 20
ORDER BY Name, Vorname;
```

b) zu logischen Verknüpfungen und Wildcards

```
SELECT PersNr, Name, Vorname  FROM Mitarbeiter
WHERE Name IN ("Fritsch", "Schuster", "Roller");
```

```
SELECT PersNr, Name, Vorname, Gehalt
```

```
FROM Mitarbeiter
WHERE Gehalt BETWEEN 2000 AND 3000;
```

```
SELECT PersNr, Name, Vorname  FROM Mitarbeiter
WHERE Name LIKE "Me*"
```

```
SELECT PersNr, Name, Vorname
FROM Mitarbeiter
WHERE ProjektNr is Null
```

c) zur Ausgabe und zur Gruppierung

```
SELECT Mitarbeiter.PersNr, Mitarbeiter.Name AS [Name
des Mitarbeiters], Mitarbeiter.Gehalt AS
Monatsgehalt, 1996 AS Jahr
FROM Mitarbeiter
ORDER BY Mitarbeiter.Name;
```

```
SELECT COUNT(*) FROM Mitarbeiter
```

```
SELECT AbtNr, COUNT(PersNr) AS Mitarbeiter
FROM Mitarbeiter
GROUP BY AbtNr
```

```
SELECT AbtNr, COUNT(PersNr) AS Mitarbeiter,
MAX(Gehalt) AS maxGehalt
FROM Mitarbeiter
GROUP BY AbtNr
```

```
SELECT AbtNr, COUNT(PersNr) AS Mitarbeiter,
SUM(Gehalt) AS maxGehalt
```

```
  FROM Mitarbeiter
  WHERE AbtNr >= 20
  GROUP BY AbtNr
  HAVING SUM(Gehalt) >= 50000
```

d) zu geschachtelten Abfragen

```
SELECT Name, Beruf, AbtNr FROM Mitarbeiter
WHERE AbtNr = 21 AND Beruf IN
(SELECT DISTINCT Beruf FROM Mitarbeiter
   WHERE AbtNr = (SELECT AbtNr FROM Abteilung
                      WHERE AbtName = "Produktion"))
```

Lösungen:

a) - Alle Daten der Tabelle *MITARBEITER* mit Personalnummer 28, d.h. des betreffenden Mitarbeiters
- Personalnummer, Name und Vorname der Mitarbeiter in Abteilung 20 sortiert nach Name

b) - Mitarbeiterdaten Personalnummer, Name und Vorname derjenigen Mitarbeiter, die Fritsch, Schuster oder Roller heißen.

- Analog die Daten derjenigen Mitarbeiter, deren Gehalt zwischen 2000 und 3000 DM liegt.

- Analog die Daten derjenigen Mitarbeiter, deren Name mit „Me“ beginnt.

- Analog die Daten derjenigen Mitarbeiter, die in keinem Projekt mitarbeiten.

c) - Es werden Spaltenüberschriften erzeugt für die Spalten „Name des Mitarbeiters“ und „Monatsgehalt“; in der Spalte „Jahr“ steht die Konstante 1996, obwohl ein Attribut JAHR fehlt.

- Es wird die Zahl aller Mitarbeiter ausgegeben.

- Es wird, gruppiert nach Abteilungen, die Zahl der Mitarbeiter und das maximale Gehalt ermittelt und mit Spaltenüberschrift ausgegeben.

- Es erfolgt zuerst eine Selektion anhand der WHERE-Klausel, d.h. für Abteilungsnummer >=20 und anschließend wird gruppiert nach Abteilungen entsprechend der HAVING-Klausel, d.h. mit einer Gehaltssumme >=50000.

d) - Alle Mitarbeiter der Abteilung 21 werden ausgegeben, die einen Beruf besitzen, der auch in der Abteilung „Produktion“ vorkommt.

Aufgaben zum Join

Aufgabe 1
Was bewirken die folgenden *SQL*-Befehle bei Verwendung der in der Aufgabe 2 (Aufgaben zu *SQL*-Abfragen) verwendeten Datenbank?

```
SELECT Name, AbtName  FROM Abteilung, Mitarbeiter
WHERE Abteilung.AbtNr = Mitarbeiter.AbtNr
```

```
SELECT AbtName, Name
FROM Abteilung JOIN Mitarbeiter
ON Abteilung.AbtNr = Mitarbeiter.AbtNr
```

```
SELECT AbtName, Name
FROM Abteilung JOIN Mitarbeiter
ON Abteilung.AbtNr = Mitarbeiter.AbtNr
WHERE AbtName = "Entwicklung"
```

Lösung:
Die beiden ersten *SQL*-Befehle liefern dasselbe Ergebnis: Es werden Name und Abteilungsname aller Mitarbeiter ausgegeben; die WHERE-Klausel unterdrückt sinnlose Kombinationen bei der Bildung des Join.

Der dritte *SQL*-Befehl schränkt diese Ausgabe auf die Abteilung „Entwicklung“ ein.

Aufgaben zu Aktualisierungen

Aufgabe 1
Was bewirken die folgenden *SQL*-Befehle bei Verwendung der in Aufgabe 2 (Aufgaben zu *SQL*-Abfragen) dargestellten Datenbank? Teste die Wirkung aus!

```
INSERT INTO Mitarbeiter
VALUES (500, "Walkes", "Otto", "Techniker", 4100, 20,
NULL)


INSERT INTO Mitarbeiter (PersNr, Name, AbtNr, Beruf)
VALUES (501, "Didi", 20, "Techniker")


INSERT INTO Mitarbeiter (PersNr, Name, AbtNr, ProjektNr)


SELECT 502, "ALF", AbtNr, NULL
FROM Abteilung
WHERE AbtName = "Produktion"
UPDATE Mitarbeiter SET Gehalt = Gehalt + 150 WHERE
AbtNr IN
  (SELECT AbtNr FROM Abteilung WHERE AbtLeiter IN
     (SELECT PersNr FROM Mitarbeiter WHERE Name =
"Schubert"))


DELETE FROM Mitarbeiter WHERE PersNr = 501
```

Lösung:

Es handelt sich um Aktionsabfragen, welche die Datenbank in folgender Weise verändern:

- Es wird ein neuer Datensatz in die Tabelle *MITARBEITER* mit den angegebenen Werten eingefügt.

- Analog wird ein neuer Datensatz eingefügt, wobei die Reihenfolge der Daten nicht der Struktur der Tabelle entspricht.

Die Spezifizierung des neuen Datensatzes erfolgt diesmal nicht mit VALUES, sondern mittels SELECT anhand der Tabelle ABTEILUNG.

- Es wird das Gehalt aller Mitarbeiter um 150 erhöht, die in einer Abteilung tätig sind, deren Leiter „Schubert" heißt.

- Es wird der Mitarbeiter mit der Personalnummer 501 aus der Tabelle entfernt.

Aufgabe 2
Die folgenden Fragen beziehen sich wieder auf die in Aufgabe 2 dargestellte Datenbank. Schreibe die entsprechenden *SQL*-Abfragen!

1. Welche Mitarbeiter arbeiten in der Abteilung 21? Gefragt sind Abteilungsnummer, Personalnummer und Name.

2. Welche Mitarbeiter der Abteilung 21 haben den Beruf Techniker? Gefragt sind Abteilungsnummer, Personalnummer, Name und Beruf.

3. In welchen Abteilungen arbeiten die Mitarbeiter mit den Namen Sturm, Frey, Winter und Fischer? Gefragt sind Abteilungsnummer, Personalnummer und Name.

4. Welche Mitarbeiter in der Abteilung 20 sind Laborant oder Techniker? Gefragt sind Abteilungsnummer, Personalnummer, Name und Beruf.

5. Welche Mitarbeiter der Abteilung 20 verdienen zwischen 3000 und 4000 Euro? Gefragt sind Abteilungsnummer, Personalnummer, Name und Gehalt.

6. Welche Mitarbeiter arbeiten für irgendein Projekt? Gefragt sind Personalnummer, Name und Projektnummer.

7. Welche Mitarbeiter haben einen Namen, der mit H beginnt? Gefragt sind Personalnummer und Name.

8. Welche verschiedenen Berufe werden in der Abteilung 22 ausgeübt?

9. Geben Sie eine Liste der Mitarbeiter aus, die in Abteilung 20 arbeiten. Die Liste soll nach Namen und Vornamen sortiert sein. Gefragt sind Abteilungsnummer, Personalnummer, Name und Vorname.

10. Geben Sie eine nach Projektnummer und Personalnummer sortierte Liste der Mitarbeiter aus, die für irgendein Projekt arbeiten. Gefragt sind Projektnummer, Personalnummer und Name.

11. Wie hoch ist das höchste Gehalt in Abteilung 21?

12. Wie hoch ist jeweils das höchste Gehalt in den einzelnen Abteilungen?

13. Wie hoch ist jeweils das niedrigste und das höchste Gehalt der Abteilungen, die mehr als 10 Mitarbeiter haben?

14. Welche Mitarbeiter haben ein Gehalt, das mindestens 3500 Euro höher ist als das niedrigste Gehalt aller Mitarbeiter? Gefragt sind Personalnummer, Name und Gehalt.

15. Welche Mitarbeiter sind in der Abteilung Informatik beschäftigt? Gefragt sind bei Join Personalnummer, Name und Abteilungsname; bei Unterabfrage Personalnummer, Name und Abteilungsnummer.

16. Welches Gehalt hat die Leitung der Abteilung Recht?

17. Welche Mitarbeiter der Produktionsabteilung arbeiten mit am Projekt Wirkstoff ABC? Gefragt sind bei Join Abteilungsname, Projektname, Name und Personalnummer; bei Unterabfrage Abteilungsnummer, Projektnummer, Name und Personalnummer.

18. Welche Mitarbeiter der Abteilung 20 haben einen Vornamen, der auch in der Abteilung 22 vorkommt? Gefragt sind Personalnummer, Name und Vorname.

19. In welchen Projekten arbeiten Mitarbeiter, die einer Abteilung angehören, die vom Mitarbeiter mit der Personalnummer 28 geleitet wird? Gefragt sind

Projektnummer und Projektname. Die Aufgabe soll durch eine mehrfach geschachtelte Abfrage gelöst werden.

Lösungen:

1.
```
SELECT AbtNr, PersNr, Name FROM Mitarbeiter
WHERE AbtNr = 21
```

2.
```
SELECT AbtNr, PersNr, Name, Beruf FROM Mitarbeiter
WHERE AbtNr = 21 AND Beruf = "Techniker"
```

3.
```
SELECT AbtNr, PersNr, Name FROM Mitarbeiter
WHERE Name IN ("Sturm", "Frey", "Winter",
"Fischer")
```

4.
```
SELECT AbtNr, PersNr, Name, Beruf FROM Mitarbeiter
WHERE AbtNr = 20 AND Beruf IN ("Laborant",
"Techniker")
```

5.
```
SELECT AbtNr, PersNr, Name, Gehalt
FROM Mitarbeiter
WHERE AbtNr = 20 AND Gehalt BETWEEN 3000 AND 4000
```

6.
```
SELECT PersNr, Name, ProjektNr FROM Mitarbeiter
WHERE ProjektNr IS NOT NULL
```

7.
```
SELECT PersNr, Name FROM Mitarbeiter
WHERE Name LIKE "H*"
```

8.
```
SELECT DISTINCT Beruf FROM Mitarbeiter
WHERE AbtNr = 22
```

9.
```
SELECT AbtNr, PersNr, Name, Vorname
FROM Mitarbeiter
WHERE AbtNr = 20
ORDER BY Name, Vorname
```

10.
```
SELECT ProjektNr, PersNr, Name FROM Mitarbeiter
WHERE ProjektNr IS NOT NULL
```

```
    ORDER BY ProjektNr, PersNr
```

```
11. SELECT Max (Gehalt) FROM Mitarbeiter
    WHERE AbtNr = 21
```

```
12. SELECT AbtNr, Max (Gehalt) FROM Mitarbeiter
    GROUP BY AbtNr
```

```
13. SELECT AbtNr, Min(Gehalt), Max (Gehalt)
    FROM Mitarbeiter
    GROUP BY AbtNr
    HAVING COUNT(PersNr) > 10
```

nicht möglich ist folgende Konstruktion:

```
    SELECT AbtNr, Min(Gehalt), Max (Gehalt)
    FROM Mitarbeiter
    WHERE COUNT(PersNr) > 10
    GROUP BY AbtNr
```

```
14. SELECT PersNr, Name, Gehalt FROM Mitarbeiter
    WHERE Gehalt >= (SELECT MIN(Gehalt) + 3500
    FROM Mitarbeiter)
```

```
15. SELECT PersNr, Name, AbtName
    FROM Mitarbeiter, Abteilung
    WHERE Abteilung.AbtNr = Mitarbeiter.AbtNr.
    AND AbtName = "Informatik"
```

eine weitere Lösung mit geschachteltem SELECT wäre:

```
    SELECT PersNr, Name, AbtNr FROM Mitarbeiter
    WHERE AbtNr = (SELECT AbtNr FROM Abteilung WHERE
    AbtName = "Informatik")
```

```
16. SELECT Gehalt FROM Mitarbeiter, Abteilung
    WHERE AbtName = "Recht" AND PersNr = AbtLeiter
```

eine zweite Lösung ist:

```
    SELECT Gehalt FROM Mitarbeiter
    WHERE PersNr = (SELECT AbtLeiter FROM Abteilung
```

```
WHERE AbtName = "Recht")
```

17.
```
SELECT PersNr, Name, AbtName, ProjektName
FROM Mitarbeiter, Abteilung, Projekt
WHERE Abteilung.AbtNr = Mitarbeiter.AbtNr
AND AbtName = "Produktion"
AND Mitarbeiter.ProjektNr = Projekt.ProjektNr
AND Projektname = "Wirkstoff ABC"
```

eine zweite Lösung ist:

```
SELECT AbtNr, PersNr, Name, ProjektNr
FROM Mitarbeiter
WHERE AbtNr = (SELECT AbtNr FROM Abteilung
               WHERE AbtName = "PRODUKTION")
AND ProjektNr = (SELECT ProjektNr FROM Projekt
WHERE ProjektName = "Wirkstoff ABC")
```

18. Die Aufgabe ist ein Beispiel für Schnittmengen:

```
SELECT PersNr, Name, Vorname FROM Mitarbeiter
WHERE AbtNr = 20 AND
Vorname IN (SELECT Vorname FROM Mitarbeiter WHERE
            AbtNr =22)
```

19.
```
SELECT ProjektNr, ProjektName FROM Projekt
WHERE EXISTS
(SELECT PersNr FROM Mitarbeiter
 WHERE Mitarbeiter.ProjektNr = Projekt.ProjektNr
 AND AbtNr IN SELECT AbtNr FROM Abteilung
             WHERE AbtLeiter =28))
ORDER BY ProjektNr
```

Rechtliche Aspekte

Die Arbeit mit Datenbanken berührt an vielen Stellen rechtliche Probleme, da die verarbeiteten Daten und die hierzu eingesetzten Werkzeuge nicht von jedem Nutzer nach eigenem Belieben verwendet werden können.

Beispiel 1:
Zwei ehemalige Schülerinnen, die vor fünf Jahren Abitur gemacht haben, kommen ins Sekretariat der Schule und bitten um eine Adressenliste ihres Jahrgangs, um eine Wiedersehens-Party zu feiern. Die Sekretärin weigert sich mit Hinweis auf das Datenschutzgesetz, die Adressen herauszugeben. Die Ehemaligen staunen nicht schlecht, dass gleichzeitig ein Polizeibeamter die Liste einer Klasse der Jahrgangsstufe 11 erhält, die am letzten Samstag auf dem Schulgelände eine Party feierte, wobei erheblicher Sachschaden am Schulgebäude entstand.

Beispiel 1:
Die Weitergabe von personenbezogenen Daten an Dritte außerhalb des öffentlichen Bereichs, so auch an Ehemalige, ist im Schulbereich in Hessen unzulässig, sofern der Betroffene nicht zustimmt. Dies ist im Falle der Wiedersehensparty nicht gegeben, die Sekretärin handelt richtig.

Die Weitergabe von Daten an die Polizei liegt innerhalb des öffentlichen Bereichs und ist zulässig, wenn dies zur Erfüllung ihrer Aufgaben erforderlich ist und die Datenübermittlung im Rahmen der ursprünglichen Zweckbestimmung erfolgt. Dies ist im Beispiel nicht gegeben. Das HDSG liefert aber in §12 die Ausnahmefälle, z.B. bei Abwehr erheblicher Nachteile für das Allgemeinwohl oder wenn Anhaltspunkte für eine Straftat vorliegen. Die Schule muss aber die Zulässigkeit sehr genau prüfen, Auskünfte darf nur der Schulleiter oder ein von ihm Beauftragter erteilen. Inwieweit die Sekretärin also zulässig handelt, hängt vom konkreten Einzelfall ab.

Beispiel 2:
Auf dem Computer des Schulsekretariats werden die Personaldaten der Schüler wie auch Würdigungsberichte des Schulleiters über Lehrerinnen und Lehrer gespeichert. Der Personalrat der Lehrer fordert, dass diese Daten nicht ungeschützt allen Sekretariatsmitarbeiterinnen zugänglich sind. Der Schulträger installiert darauf ein Schutzsystem, das sich in das Betriebssystem einklinkt und jeden Zugang über Passwörter schützt. Die Daten werden verschlüsselt gespeichert. Nach einem Diebstahl des Rechners sind allerdings die Sicherungskopien wertlos, da sie nicht ohne Originalrechner gelesen werden können.

Beispiel 3:
Die Schule beschafft für den Unterricht eine Lizenz für ein Textverarbeitungssystem, das auf einem frei zugänglichen Rechner installiert wird. Kundige Schüler und Lehrer kopieren sich die Software illegal vom Rechner. Nachdem ein bekannter Software-Konzern die Schulen schriftlich auf die Strafbarkeit von Raubkopien aufmerksam gemacht hat, wird der Rechner in das Schulnetz integriert. Damit sind Kopien vom Server des Netzes, wo jetzt die Software installiert ist, für Anwender nicht mehr möglich.

Beispiel 4:
Die Schülerbibliothek der Schule verwendet ein Datenbanksystem zur Speicherung der Bücherdaten und der Ausleihvorgänge. Da die Bibliothekarin vom Förderverein der Schule bezahlt wird, fordert dieser von ihr einen Nachweis der erbrachten Arbeitsleistung und der Ausleihvorgänge. Die Schulleitung außerdem möchte von der Bibliothekarin wissen, inwieweit die verwendeten Mittel für neue Bücher auch sinnvoll eingesetzt und die gekauften Bücher auch ausgeliehen wurden. Die Datenschutzbeauftragte der Schule weist die Bibliothekarin darauf hin, dass Ausleihvorgänge nur während der Leihzeit ausleiherbezogen gespeichert werden dürfen. Anschließend seien sie zu löschen. Die Bibliothekarin weigert sich, den Bibliotheksrechner zur Arbeitskontrolle einsetzen zu lassen. Da die Ausleihvorgänge bisher nur personenbezogen gespeichert sind, muss das Programm umgeschrieben werden.

Diese vier realistischen Beispiele zeigen, dass die automatisierte Datenverarbeitung und hier vor allem die Verwendung von Dateiverwaltungssystemen und Datenbanken eine Fülle von Rechtsgütern und Rechtsvorschriften berührt, was Konsequenzen im Alltag nach sich zieht. Personen, in der Schule vor allem Schülerinnen und Schüler, Lehrerinnen und Lehrer, Sekretariatsangestellte und Mitarbeiter verschiedener Behörden sind gleichzeitig Informationsgegenstände wie Informationskonsumenten. Ihr Recht auf Informationsfreiheit kollidiert dabei oft mit den Rechten anderer, z.B. dem Recht auf Schutz der Persönlichkeit, wobei beide Rechte im Grundgesetz der Bundesrepublik verankert sind.
Solche Normenkonflikte werden in einem Rechtsstaat in der Regel durch Gesetze und Verordnungen gelöst. In den oben genannten Beispielen werden Belange des Datenschutzes, der Datensicherheit und des Urheberrechts berührt, die jeweils durch entsprechende Vorschriften des Bundes und der Länder geregelt sind.
In einigen Fällen ist auch das Betriebsverfassungsgesetz heranzuziehen, wenn Computer zur Steuerung und Kontrolle von Arbeitsvorgängen dienen.

Unter *Datensicherheit* versteht man die Gesamtheit aller - meist technischer und organisatorischer - Maßnahmen, die zum Schutz von Daten vor unberechtigtem Zugriff, Verfälschung oder Verlust ergriffen werden.

Unter *Datenschutz* versteht man die Gesamtheit aller - meist juristischer - Maßnahmen zum Schutze *personenbezogener Daten* vor unberechtigtem Zugriff, vor Manipulation oder Zerstörung.

Unter *Urheberrecht* sind die Vorschriften zum Schutze eines Urhebers an seinem Werk gemeint, sofern dieses schutzwürdigen Belangen entspricht, z.B. fällt auch umfangreichere Software unter das Urheberrecht.

Aufgrund dieser Begriffsbestimmung fällt es leicht, die in den Beispielen angesprochenen Tatbestände zu klassifizieren. In Beispiel 1 geht es um personenbezogene Daten, im speziellen Fall um Adresslisten. Inwieweit eine Schule solche Daten weitergeben darf, unterliegt der Datenschutz-Gesetzgebung.

Ähnliches gilt in den Beispielen 2 und 4. Hier zeigt sich allerdings, dass Datenschutz und Datensicherheit nicht getrennt betrachtet werden dürfen: Entsprechende Maßnahmen zur Datensicherheit gewähren auch Datenschutz, sofern z.B. den schutzwürdigen Belangen dadurch Rechnung getragen wird, dass verschiedene Benutzergruppen nur ihre spezielle Sicht auf die Daten haben oder eine spezielle Sicht für unberechtigte Benutzer nicht möglich ist.

In Beispiel 3 zeigt sich ebenfalls, dass urheberrechtlichen Belange mit geeigneten Maßnahmen zur Datensicherheit Rechnung getragen werden kann: Moderne Netzwerkumgebungen verhindern nicht nur Manipulationen an den Datenbeständen, sondern sichern auch Software gegen Raubkopieren ab.

Datenschutzregelungen

Es kommt nicht von ungefähr, dass Fragen des Datenschutzes immer an erster Stelle genannt werden, wenn es um den Einsatz von Computern bei der Speicherung und Verarbeitung von Daten geht. Dies rührt sicherlich im Wesentlichen daher, dass jeder Mensch Informationen über sich selbst als sein Eigentum und eine Weitergabe oder gar Verfälschung als Verletzung seiner Intimsphäre betrachtet.

Insofern ist das Anliegen des Datenschutzes nicht neues: Arzt- und Beicht-geheimnis, Steuergeheimnis usw. sind keine Errungenschaften unserer Zeit. Aber durch den Einsatz automatisierter Datenverarbeitungen hat das Anliegen des Schutzes personenbezogener Daten erst an Brisanz gewonnen:

Nach Schätzungen ist jeder Bundesbürger durchschnittlich in etwa 200 Datenbanken gespeichert, hierzu gehören vor allem die Datenbanken der Melde- und Finanzbehörden, der Renten-, Arbeitslosen- und Krankenversicherung, der Banken und allgemeinen Versicherungen, des Kraftfahrbundesamtes in Flensburg und der Polizeibehörden, der Telekom usw.

Diese Daten lassen sich schnell kopieren, über weltweite Netze übermitteln, manipulieren und löschen, ohne dass der Betroffene es überhaupt bemerken muss.

Im Jahre 1978 wurde das erste Bundesdatenschutzgesetz verabschiedet, inzwischen sind die Datenschutzgesetzte des Bundes und der Länder mehrfach der Entwicklung angepasst worden.

In den Datenschutzgesetzen werden meist in einem allgemeinen Teil Begriffsbestimmungen vorgenommen, dann die Rechtsgrundlagen definiert und die Rechte der Betroffenen beschrieben, die Einrichtung des Datenschutzbeauftragten geregelt und Strafvorschriften bei Verstößen gegen das Gesetz behandelt.

Im Folgenden sind einige wesentliche Paragraphen des hessischen Datenschutz-gesetzes als Beispiel für eine solche gesetzliche Regelung aufgeführt.

Datenschutz in der Schule

In der Schule werden seit jeher die Daten von Schülerinnen und Schülern, Erziehungsberechtigten und Unterrichtenden verarbeitet, ohne dass die Betroffenen ausdrücklich zustimmen.

Aufgabe 6:
Ein häufiger Diskussionspunkt ist die Frage, ob die Eltern volljähriger Schülerinnen und Schüler über die Leistungen oder das Fernbleiben vom Unterricht von der Schule informiert werden dürfen.

Kläre diese Frage mit Hilfe der Bestimmungen über die Datenübermittlung von Schuldaten!

Auch hier ist die Rechtslage eindeutig: Die Datenübermittlung ist nur mit Einwilligung des Betroffenen, also des Schülers, möglich.