

## 5 Datenbanken und Datenbankprogrammierung

Viele Softwaresysteme und praktisch alle betrieblichen Softwaresysteme besitzen eine Schnittstelle zu einer (oder gar mehreren) Datenbanken. Für einen Informatiker ist die Notwendigkeit des Einsatzes von Datenbank-Technologie allgegenwärtig. In diesem Abschnitt werden wir lernen, was Datenbanken sind, in welchen Fällen es sinnvoll ist, sie einzusetzen, wie sie entworfen und gewartet werden können und vor allem, wie man Daten einlesen und aus der Datenbank beziehen kann.

### 5.1 Wozu Datenbanken?

Bevor wir erklären, was eine Datenbank genau ist, welche Arten von Datenbanken es gibt und wie sie verwendet werden können, beschäftigen wir uns zunächst mit den grundlegendsten Fragen: Welche Probleme löst der Einsatz einer Datenbank und welchen Anforderungen genügen Datenbanken?

#### 5.1.1 Daten-Persistenz

In vielen Anwendungen, insbesondere in betrieblicher Software, ist es wichtig, die durch das Programm erzeugten, modifizierten oder analysierten Daten für einen längeren Zeitraum zu speichern. Man spricht in diesem Zusammenhang auch von *persistenten* Daten: Daten, deren Lebensdauer die eines Betriebssystem-Prozesses bzw. einer Programmausführung überschreitet, werden als *persistent* bezeichnet.

#### 5.1.2 Dateisystem als Datenspeicher

Man kann natürlich das Dateisystem dazu verwenden, die Daten persistent zu halten, d. h. die jeweiligen Programmdateien einfach in eine Datei zu schreiben. Es gibt zwei Python-Module, die geeignet sind, die Daten eines Python-Programms in einer Datei zu speichern bzw. Daten aus einer Datei in eine Python-Datenstruktur zu lesen:

- Das Python-Modul `pickle`: Es wandelt beliebige Objekte oder ganze Objektstrukturen in einen Bytestrom (also eine Folge von Bytes) um. Diese Bytefolge kann einfach in eine Datei gespeichert werden.
- Das Python-Modul `shelve`: Es bietet ein persistentes Dictionary-artiges Objekt an, das die gleiche Schnittstelle bereitstellt, wie ein herkömmliches Dictionary-Objekt, jedoch mit der Eigenschaft, dass die darin enthaltenen Daten von Python persistent gehalten werden.

### 5.1.3 Anforderungen an Persistenzmechanismen

Oft genügen diese einfachen Persistenz-Möglichkeiten nicht. Viele Anwendungen stellen hohe Anforderungen an die Zugriffsmechanismen auf persistente Daten, die ein einfaches Dateisystem nicht leisten kann, unter anderem:

1. *Datenredundanz und Datenkonsistenz*: Als *Redundanz* bezeichnet man die Situation, dass bestimmte Informationen sich an unterschiedlichen Stellen verteilt befinden bzw. wiederholt genannt werden. Datenredundanz zieht meist auch Probleme in der Datenkonsistenz nach sich: Wird etwa die sich an unterschiedlichen Stellen befindliche Information nur an einer Stelle geändert und die Anpassung der übrigen Stellen vergessen, so ergibt sich eine Inkonsistenz. In herkömmlichen Dateisystemen kann nicht sichergestellt werden, dass eine bestimmte Information nicht an vielen verschiedenen Stellen (evtl. auch noch in unterschiedlichen Dateiformaten) gespeichert wird.
2. *Schwierigkeiten des Datenzugriffs*: Für jede Art des Datenzugriffs muss möglicherweise eigener Programmtext geschrieben werden.
3. *Datenisolation*: Es gibt potentiell viele verschiedene Dateiformate und es ist schwer über alle Formate hinweg einheitliche Operationen auszuführen.
4. *Datenintegrität*: Integritätsbedingungen sollen die Konsistenz der Daten, d. h. die Korrektheit der Daten untereinander, sicherstellen. Bietet ein Persistenzmechanismus keine Möglichkeiten, die Datenintegrität sicherzustellen, so müssen Integritätsbedingungen im Programmcode fest einprogrammiert werden. Änderungen in Integritätsbedingungen werden somit schwerer, die Programme umfangreicher und schwerer wartbar.
5. *Atomarität von Updates*: Werden Veränderungen an den Daten vorgenommen, dann sollte die Änderung entweder vollständig ausgeführt werden oder gar nicht. Diese Eigenschaft wird im Zusammenhang mit Datenbanken auch als Atomarität bezeichnet. Fehlt diese Eigenschaft, so könnten Fehler oder Programmabstürze die Daten bei halb ausgeführten Änderungen in inkonsistentem Zustand hinterlassen.
6. *Synchronisationsmechanismen*: Dateisysteme können im Allgemeinen nicht effizient sicherstellen, dass mehrere Kopien von sich selbst synchron (d. h. auf gleichem Stand) bleiben.
7. *Zugriffsrechte*: Häufig ist ein Management von Zugriffsrechten erforderlich. Zugriffsrechte sollten abhängig von der jeweiligen Benutzergruppe sein.
8. *Backupmechanismen*: Als „Backup“ bezeichnet man eine Sicherungskopie. Es sollte (zeit- und platz-)effiziente Möglichkeiten geben, Datenbankzustände in regelmäßigen Abständen einzufrieren und Backup-Kopien zu erstellen.

**Aufgabe 5.1**

Ein Dateisystem, das sehr viele Möglichkeiten bietet, ist ZFS von Sun. Einige der eben vorgestellten Anforderungen können durch ZFS erfüllt werden. Erklären Sie, welche Anforderungen mittels Verwendung von ZFS als Datencontainer erfüllt oder teilweise erfüllt werden können und welche Anforderungen nicht erfüllt werden können.

**5.2 Datenbankmanagementsysteme (DBMS)**

Datenbankmanagementsysteme bieten eine Möglichkeit an, Daten persistent zu halten und den oben angedeuteten Anforderungen gerecht zu werden.

**5.2.1 Transaktionskonzept**

Entscheidend ist hierbei das *Transaktionskonzept*: Zusammengehörige Datenoperationen, die als logische Einheit betrachtet werden können, werden hierbei zu einer sogenannten *Transaktion* zusammengefasst. Transaktionen sind *atomar*, d. h., entweder alle in einer Transaktion enthaltenen Datenoperationen werden vollständig ausgeführt, oder – falls etwa eine der Operationen einen Fehler auslöst, oder falls während der Ausführung der Rechner abstürzt – keine der Aktionen wird ausgeführt. Mittels der Operation *Commit* wird im Allgemeinen eine Transaktion abgeschlossen.

Geben wir ein Beispiel für eine Folge von Datenoperationen an, die als Transaktion organisiert sein sollten: Ein Kunde möchte Geld zwischen zwei seiner Konten verschieben. Diese Transaktion besteht aus zwei Aktionen, die zwei Datenoperationen eines DBMS entsprechen: Der Abbuchung des Betrages von Konto<sub>1</sub> und der Buchung genau desselben Betrages auf Konto<sub>2</sub>. Man könnte eine solche Transaktion wie folgt schreiben:

```
BEGIN_TRANSACTION
```

```
    Lösche Betrag  $x$  von Konto1.
```

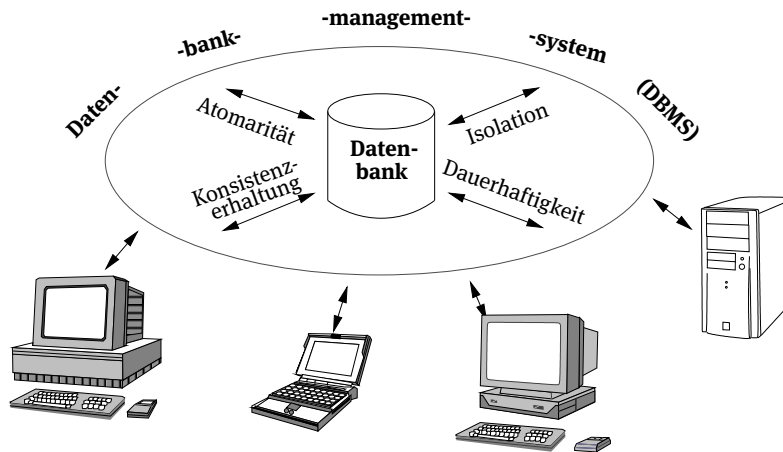
```
    Buche Betrag  $x$  auf Konto2.
```

```
END_TRANSACTION
```

Wären diese beiden Aktionen nicht durch eine Transaktion gruppiert, dann wäre folgender Fehlerfall denkbar: Würde das System etwa abstürzen, nachdem der Betrag von Konto<sub>1</sub> gelöscht wurde und bevor der Betrag auf Konto<sub>2</sub> gebucht wurde, dann würde das Geld einfach verschwinden. Das Transaktionskonzept schließt insbesondere diesen Fehlerfall aus.

### 5.2.2 Funktionsweise eines DBMS

Abbildung 5.1 zeigt den grundsätzlichen Aufbau eines Datenbankmanagementsystems: Dieses bietet Zugriffe auf die eigentlichen Daten durch ein Transaktionskonzept



**Abb. 5.1.** Das Datenbankmanagementsystem stellt die folgenden in vielen Anwendungen erwünschten Eigenschaften – oft als ACID-Eigenschaften bezeichnet – eines Zugriffs auf eine Menge persistenter Daten sicher: Atomarität, Konsistenz-erhaltung (engl: Consistency), Isolation und Dauerhaftigkeit. In der Zeichnung ist zusätzlich angedeutet, dass mehrere (teilweise entfernte) Rechner möglicherweise gleichzeitig auf das Datenbankmanagementsystem zugreifen können.

so an, dass die Zugriffe atomar, konsistent, isoliert und dauerhaft sind. Unter *Atomarität* versteht man (wie schon oben bei der Erläuterung des Transaktionskonzepts dargestellt) die Eigenschaft, dass eine als Transaktion organisierte Sequenz von Datenoperationen entweder vollständig oder überhaupt nicht ausgeführt wird. Unter *Konsistenz* bzw. *Konsistenz-erhaltung* versteht man die Eigenschaft, dass nach jeder Transaktion der Zustand der Datenbank konsistent sein sollte, vorausgesetzt, er war es auch vor Ausführung der Transaktion. Wann die Daten konsistent sind und wann nicht wird über Integritätsbedingungen spezifiziert, die in der Datenbank hinterlegt sind (diese könnten beispielsweise festlegen, dass jeder Professor mindestens einer Vorlesung zugeordnet sein sollte, oder dass eine Matrikelnummer sich in einem bestimmten Zahlenbereich befinden sollte). Unter *Isolation* versteht man die Eigenschaft, dass mehrere gleichzeitig ausgeführte Transaktionen sich während der Ausführung nicht gegenseitig beeinflussen sollten, und dass diese in eine vom DBMS vorgegebene sinnvolle Reihenfolge gebracht werden. Unter *Dauerhaftigkeit* versteht man, dass eine (nach einem Transaktions-Commit) erfolgreiche Änderung eines Datensatzes – aufgrund bestimmter, später eintretender Bedingungen – im Nachhinein nicht mehr wieder automatisiert rückgängig gemacht werden dürfen.

Das Sicherstellen dieser vier Eigenschaften wird oft auch als *ACID-Prinzip* (im Deutschen manchmal auch: „AKID-Prinzip“) bezeichnet.

### 5.2.3 Einsatz von DBMS

Viele Anwendungen verwenden Datenbankmanagementsysteme. Dazu gehören:

- *Content Management Systeme* – wie sie etwa vielen komplexen Web-Seiten oder vielen Informationsportalen von Unternehmen zugrundeliegen. Diese sind eigentlich nichts anderes als Datenbankmanagementsysteme, die einem etwas erweiterten Satz von Anforderungen genügen und eine benutzerfreundliche Oberfläche anbieten.
- *Komplexe Web-Seiten* – die großen Internet-Warenhäuser wie Amazon, oder die großen Internet-Informationssysteme wie Wikipedia halten alle ihre Daten in Datenbankmanagementsystemen.
- *ERP-Systeme* (Enterprise-Ressource-Planning-Systeme) sind Softwaresysteme mit denen Unternehmen ihre Ressourcenplanungen durchführen. Nahezu alle ERP-Systeme halten ihre Daten in Datenbankmanagementsystemen.
- *Bankensoftware* – nahezu alle Banken arbeiten mit kommerziellen Datenbankmanagementsystemen, wie etwa Oracle oder DB/2 von IBM.
- *Statistische Anwendungen und Marktforschungssoftware* – auch sie arbeiten im Allgemeinen mit Datenbankmanagementsystemen.

## 5.3 Relationale DBMS

Es gibt mehrere verschiedene Ansätze, Daten logisch in Datenbanken zu organisieren. Mit Abstand (noch) am weitesten verbreitet ist das *Relationale Datenbankmodell*, das erstmals 1970 von Edgar F. Codd vorgeschlagen wurde [3]. Es basiert auf der mathematischen Relationenalgebra. Die von der Relationenalgebra definierten Operationen sind, wegen der klaren mathematischen Fundierung, auch für Anfänger leicht nachzuvollziehen und anzuwenden. Diese intuitive Handhabbarkeit des relationalen Datenbankmodells ist sicherlich einer der wichtigsten Gründe für seine weite Verbreitung.

### 5.3.1 Tabellen, Schemata, Zeilen, Spalten

Eine relationale Datenbank besteht aus einer Menge von *Relationen*. Eine Relation ist nichts anderes als eine *Tabelle*, und die Begriffe „Relation“ und „Tabelle“ werden im Folgenden synonym verwendet, wobei der Begriff „Tabelle“ vorgezogen wird.

| Student   |           |          |             |            | Professor |           |          |             |
|-----------|-----------|----------|-------------|------------|-----------|-----------|----------|-------------|
| Matr. Nr. | Nach-name | Vor-name | Studiengang | Imm        | Pers. Nr. | Nach-name | Vor-name | Studiengang |
| 20123     | Fischer   | Aaron    | TI          | 2008-10-01 | 1293      | Curry     | Harry    | TI          |
| 19923     | Zeh       | Hilde    | KST         | 2007-03-01 | 8112      | Pech      | Kurt     | WIN         |
| 71232     | Huber     | Gerd     | BKT         | 2006-03-01 | 1822      | Lang      | Eva      | TI          |
| 10012     | Becker    | Ben      | WIN         | 2006-10-01 | 5321      | Rossi     | Guido    | BKT         |
| 62618     | Klein     | Jon      | TI          | 2008-10-01 | 7211      | Hauser    | Gerda    | WIN         |
| 81232     | Lieb      | Juli     | BKT         | 2008-10-01 | 2321      | Gries     | Adam     | BKT         |
| ...       | ...       | ...      | ...         | ...        | ...       | ...       | ...      | ...         |

| Vorlesung |        |     |      | VLStudent |         |
|-----------|--------|-----|------|-----------|---------|
| Nr.       | Titel  | Sem | Prof | VL        | Student |
| 100       | DB     | 4   | 1822 | 405       | 62618   |
| 405       | PI 1   | 1   | 5321 | 950       | 62618   |
| 300       | Mathe1 | 1   | 5321 | 950       | 20123   |
| 950       | PI 2   | 2   | 1822 | 300       | 81232   |
| 500       | Tex    | 3   | 5321 | 405       | 10012   |
| ...       | ...    | ... | ...  | ...       | ...     |

Abb. 5.2. Beispiele von Tabellen einer Relationalen Datenbank.

Abbildung 5.2 zeigt einige Beispiele für logisch miteinander verknüpfte Tabellen. Wie man sieht, besitzt jede Tabelle einen Namen und ein sogenanntes *Schema*, das die Bezeichnung und den Typ der Spalten festlegt. Eine Tabelle besteht aus einer Liste von Zeilen. Jede Zeile enthält ein Werte-Tupel, auch als *Datensatz* bezeichnet. Eine Datenbank sollte keine zwei identischen Datensätze enthalten.

Zwei Tabellen können logisch dadurch verbunden sein, dass Werte bestimmter Spalten miteinander in Verbindung stehen: Einige Werte der Spalte „**Studiengang**“ der Tabelle „**Student**“ entsprechen Werten der Spalte „**Studiengang**“ der Tabelle „**Professor**“. Durch Operationen der Relationenalgebra können so beispielsweise alle einem Studenten durch Studiengangs-Zugehörigkeit zugeordneten Professoren bestimmt werden. Wie dies im Detail erfolgt wird in den nächsten Kapiteln ausführlicher beschrieben.

### 5.3.2 Erstellen einer Tabelle in MySQL mit Python

Wir zeigen, wie man in Python eine Datenbank und die in Abbildung 5.2 gezeigten Tabellen anlegen kann. Wir verwenden hier die MySQL, eine häufig verwendete, leistungsfähige relationale Opensource-Datenbank. Man sollte sich – um alle folgenden Beispiele möglichst einfach ausprobieren zu können – das Datenbankmanagementsystem von MySQL lokal starten; folgendes Kommando startet das DBMS, auch als

den MySQL-Server bezeichnet:

```
$ sudo mysqld
```

Unter Windows kann man sich auch eine der vorgefertigten Umgebung, etwa XAMPP, installieren, die zur Webentwicklung bereitgestellt werden und unter anderem einen MySQL-Server beinhalten. Zusätzlich bieten sie eine graphische Schnittstelle für das Aufsetzen einer ersten Datenbank und das Starten des Datenbankservers.

Nach Ausführung des `mysqld`-Kommandos läuft der MySQL-Datenbankmanagement-Server fortan als Hintergrund-Prozess. Direkt mit der Datenbank interagieren kann man mittels `mysql`. Die Eingabe des Kommandos

```
$ mysql -h 'localhost' -u 'root' -p
```

in der Bash führt in eine interaktive Umgebung, in der die in den folgenden Kapiteln vorgestellten SQL-Kommandos direkt eingegeben und ausgeführt werden können. Die Angabe von `'localhost'` gibt an, dass das DBMS auf dem lokalen Rechner läuft. Man sollte sich jedoch immer bewusst sein, dass sehr häufig das DBMS auf einem eigens dafür vorgesehenen Server-Rechner läuft und das DBMS daher im Allgemeinen von einem entfernten Rechner aus angesprochen wird und diesen mit Informationen versorgt. Mit der Angabe von `'root'` legt man fest, dass man mit Administrator-Rechten mit der MySQL-Datenbank interagiert; die Option `-p` legt fest, dass das MySQL-Passwort angegeben werden muss.

Wir zeigen nun, wie wir Python verwenden können, um mit der MySQL-Datenbank zu interagieren. Die Verwendung von Python hat den Vorteil, dass einem alle Möglichkeiten einer höheren Programmiersprache zur Verfügung stehen, beispielsweise um SQL-Kommandos nach bestimmten Vorgaben zu konstruieren. Folgender Python-Code nimmt eine Verbindung zum eben gestarteten DBMS auf und legt zwei leere Tabellen an:

```
1 import MySQLdb
2 conn = MySQLdb.connect(host='localhost', user='root',
3                        passwd='...', db='test')
4 cursor = conn.cursor()
5 cursor.execute('''
6     CREATE TABLE Professor
7         (Personalnr INTEGER NOT NULL,
8          Nachname CHAR(40),
9          Vorname CHAR(40),
10         Studiengang CHAR(40),
11         PRIMARY KEY (Personalnr))
12 ''')
13 cursor.execute('''
14     CREATE TABLE Vorlesung
15         (Nr INTEGER NOT NULL,
```

```

16     Titel CHAR(40),
17     Sem INTEGER,
18     Prof INTEGER REFERENCES Professor(Personalnr),
19     PRIMARY KEY (Nr))
20 '''

```

**Listing 5.1.** Python-Skript, das das Schema der in Abbildung 5.2 gezeigten Tabelle **Student** erzeugt

Um Python in die Lage zu versetzen, mit MySQL-Datenbanken interagieren zu können, muss das Modul `MySQLdb` geladen werden. Der Aufruf von `connect` in den Zeilen 2 und 3 erstellt ein Verbindungsobjekt zu einer MySQL-Datenbank. Durch den Parameter `host`<sup>1</sup> kann man angeben, auf welchem Rechner (in einem Computernetzwerk) sich die Datenbank befindet; die Angabe `'localhost'` bezieht sich hierbei immer auf den lokalen Rechner, auf dem das Python-Skript selbst abläuft. In Zeile 4 wird ein sogenannter *Cursor* erzeugt. *Cursor* stellen Möglichkeiten zur Verfügung, durch Anfragen entstehende, gegebenenfalls sehr große Ergebnis-Mengen effektiv zu durchlaufen. Eigentlich benötigt nicht jedes Datenbank-Kommando einen *Cursor*. In der Python-Schnittstelle muss man jedoch grundsätzlich einen *Cursor* öffnen, um Datenoperationen ausführen zu können.

Die auszuführende Datenoperation wird dem `execute`-Kommando als String mitgegeben. Bei der `'CREATE TABLE'`-Operation handelt es sich um ein SQL-Kommando. *SQL* ist eine standardisierte und weitverbreitete Sprache zur Datendefinition, -abfrage und -manipulation in relationalen Datenbanken. Die in obigem Listing verwendete Datendefinitions-Operation `'CREATE TABLE'` erzeugt eine Tabelle mit Namen „**Professor**“ die die Attribute „Personalnr“, „Nachname“, „Vorname“ und „Studiengang“ besitzt. Außerdem wird mittels der Deklaration `'NOT NULL'` festgelegt, dass der Eintrag „Personalnr“ nie leer sein darf. Die letzte Deklaration in Zeile 11

```
'PRIMARY KEY (Personalnr)'
```

legt fest, dass „Personalnr“ ein Schlüsselattribut sein soll. Dies bedeutet, dass jeder Datensatz der Tabelle „**Professor**“ eindeutig durch das Attribut „Personalnr“ bestimmt ist; es darf also keine zwei unterschiedlichen Datensätze geben, deren Einträge in der Spalte „Personalnr“ identisch sind.

Ab Zeile 14 befindet sich die SQL-Anweisung um die Tabelle „**Vorlesung**“ zu erzeugen. In Zeile 18 legt man mittels der Deklaration

```
'... REFERENCES Professor(Personalnr)'
```

fest, dass das Attribut „Prof“ immer einen Schlüssel der Tabelle „**Professor**“ enthält; solch ein Attribut, dessen Werte immer aus der Menge der Schlüsselwerte einer bestimmten Tabelle stammen, nennt man auch *Fremdschlüssel*. Mittels Fremdschlüssel-Attributen sind Tabellen relationaler Datenbanken logisch miteinander verbunden.

<sup>1</sup> Als *Host* bezeichnet man einen Rechner, der an ein Netzwerk angeschlossen ist und an der Netzwerkkommunikation teilnehmen kann. Für weitere Informationen siehe Abschnitt 6.1.6 auf Seite 175.



Mittels der 'INSERT INTO ... VALUES'-Operation, kann man Wertetupel in die soeben erzeugte Tabelle einfügen; dies ist ein Beispiel für die vorher angesprochene Möglichkeit, SQL-Kommandos dynamisch zu erzeugen:

```
cursor.execute('''
INSERT INTO Professor VALUES
(1293, 'Curry', 'Harry', 'TI'), (8112, 'Pech', 'Kurt', 'WIN'), ...
''')
```

Der Cursor kann mit dem Kommando `cursor.close()` geschlossen werden.

Eine elegantere Möglichkeit besteht darin, die einzufügenden Datensätze in einer Pythonliste zu halten; aus dieser kann mittels Pythons String-Operationen recht einfach die SQL-Einfügeoperation erzeugt werden:

```
21 profs = [(1293, 'Curry', 'Harry', 'TI'), (8112, 'Pech', 'Kurt', 'WIN'),
22          (1822, 'Lang', 'Eva', 'TI'), (5321, 'Rossi', 'Guido', 'BKT'),
23          (7211, 'Hauser', 'Gerda', 'WIN'), (2321, 'Gries', 'Adam', 'BKT')]
24 insertStr = 'INSERT INTO Professor VALUES ' + str(profs)[1:-1]
25 cursor.execute(insertStr)
```

Der Aufruf `str(profs)` wandelt die Liste `profs` in den String

```
'''[(1293,...,
..., "BKT")]' '''
```

um<sup>2</sup>. Anschließend müssen nur noch die einschließenden eckigen Klammern, die sich an der nullten und letzten Position des Strings befinden, entfernt werden – dies geschieht, indem wir den Slice `[1:-1]` von `str(profs)` wählen.

### Aufgabe 5.2

Erstellen Sie eine dritte Tabelle **Student** und eine vierte Tabelle **VLStudent**, in Anlehnung an Abbildung 5.2, und füllen Sie diese mit den Werten aus Abbildung 5.2. Verwenden Sie hierfür Pythons Modul `MySQLdb`.

### Aufgabe 5.3

Schreiben Sie eine Python-Funktion `einfDBTupel`, die einen Cursor einer Datenbank und eine Liste von Werte-Tupeln übergeben bekommt und jedes Wertetupel in die Datenbank einfügt; sollte die Einfüge-Operation mit einem Tupel schief gehen, so soll das Tupel auf dem Bildschirm ausgegeben werden mit der Meldung: `'... konnte nicht eingefuegt werden'`

<sup>2</sup> Pythons `print`-Kommando verwendet übrigens genau diese `str`-Funktion.

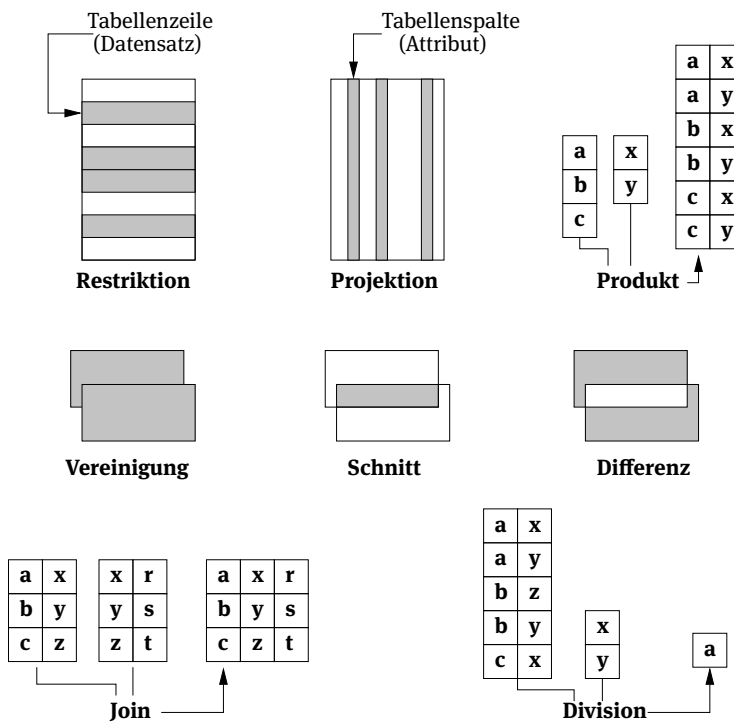
## 5.4 SQL-Abfragen

In diesem Abschnitt lernen wir die Syntax der Standard Query Language und schauen uns einige Beispiele für deren Verwendung an.

### 5.4.1 Relationenalgebra

Alle SQL-Anfragen an eine relationale Datenbank basieren auf der sog. Relationenalgebra. Diese legt die Operationen fest, die es auf Relationen (bzw. Tabellen) gibt: Sie ermöglichen unter anderem eine Verknüpfung von Tabellen, eine Filterung, und eine Suche, auch über mehrere Tabellen hinweg.

Abbildung 5.3 zeigt die wichtigsten Operationen der Relationenalgebra. Fett ge-



**Abb. 5.3.** Die wichtigsten Operationen der Relationenalgebra, ausgeführt entweder auf einer Tabelle (wie Restriktion und Projektion) oder auf zwei Tabellen (wie Vereinigung, Schnitt und Differenz, Produkt, Join und Division).

druckte Buchstaben können hierbei entweder für einen einzelnen Attributwert oder für mehrere Attributwerte (d. h. ein Attributwert-Tupel) stehen. Im Rest dieses Ab-

schnitts gehen wir nun kurz auf die einzelnen Operationen ein und beschreiben in den folgenden Abschnitten deren Umsetzung in SQL.

**Restriktion** Die Restriktion wählt Zeilen, die bestimmten Kriterien genügen, aus einer Tabelle aus.

**Projektion** Die Projektion wählt bestimmte Spalten einer Tabelle aus.

**Produkt** Die Produkt-Operation entspricht dem mathematischen Kreuzprodukt; alle möglichen Kombination von Datensätzen der ersten Tabelle mit Datensätzen der zweiten Tabelle werden erzeugt.

**Vereinigung** Die Datensätze zweier Tabellen werden vereinigt. Dies ist natürlich nur dann möglich, wenn die Schemata der beiden Tabellen übereinstimmen.

**Schnitt** Der Schnitt zweier Tabellen besteht aus denjenigen Datensätzen, die sowohl in der ersten als auch in der zweiten Tabelle vorhanden sind.

**Differenz** Die Differenz zweier Tabellen besteht aus allen Datensätzen der beiden Tabellen, außer denjenigen, die in beiden Tabellen enthalten sind.

**Join** Beim Join zweier Tabellen werden diejenigen Datensätze miteinander kombiniert, bei denen der Inhalt bestimmter Spalten identisch ist. Diese Operation kann mittels eines Kreuzprodukts und anschließender Restriktion umgesetzt werden.

**Division** Die Division von Tabelle<sub>1</sub> durch Tabelle<sub>2</sub> findet alle Daten (bzw. Datentupel) aus den Attributen, die im Schema von Tabelle<sub>1</sub>, nicht aber im Schema von Tabelle<sub>2</sub> enthalten sind, und für die folgende Eigenschaft gilt: Jedes gefundene Datum muss mit jedem Wert aus Tabelle<sub>2</sub> assoziiert sein. Im Beispiel aus Abbildung 5.3 werden alle Werte des ersten Attributs von Tabelle<sub>1</sub> gefunden, die in ihrer Zeile sowohl **x** als auch **y** stehen haben. Im Beispiel aus Abbildung 5.3 ist dies lediglich **a**.

Diese Operationen der Relationenalgebra haben ihre Entsprechung in der Standard Query Language.

#### 5.4.2 Das SELECT-Kommando

Mittels des SQL-SELECT-Kommandos kann man Zeilen einer Tabelle auswählen. Will man alle Zeilen und alle Spalten einer Tabelle auswählen, so kann dies folgendermaßen geschehen:

```
>>> cursor.execute('SELECT * FROM Student')
6L
>>> cursor.fetchall()
((20123L, 'Fischer', 'Aaron', 'TI'), (19923L, 'Zeh', 'Hilde', 'KST'),
 (71232L, 'Huber', 'Gerd', 'BKT'), (10012L, 'Becker', 'Ben', 'WIN'),
 (62618L, 'Klein', 'Jon', 'TI'), (81232L, 'Lieb', 'Juli', 'BKT'))
```

Nach Ausführung des SQL-Kommandos mittels `execute` liefert `fetchall()` die gesamte Ergebnismenge aus der Datenbank zurück. Besonders dann, wenn die Ergebnismenge sehr groß ist (was bei entsprechend umfangreichen Datenbanken oft vorkommen kann) ist es ratsam, die Funktion `fetchone()` zu verwenden, die immer nur eine Zeile der Ergebnismenge zurückliefert; falls sich keine weiteren Zeilen mehr in der Ergebnismenge befinden, wird `None` zurückgeliefert.

#### Aufgabe 5.4

- Verwenden Sie eine Schleife, um mittels `fetchone()` alle Zeilen der Ergebnismenge in einer Liste zu speichern.
- Schreiben Sie eine Funktion `myexecute`, die einen Cursor und ein SQL-Statement übergeben bekommt und falls sich mehr als 1000 Zeilen in der Ergebnismenge befinden, das `fetchall`-Kommando verwendet, um die Ergebnisse auf dem Bildschirm auszugeben und – falls sich mehr als 1000 Zeilen in der Ergebnismenge befinden – mittels `fetchone` die Stringrepräsentationen der Zeilen der Ergebnismenge in der Datei `myexecute.txt` speichert.

Eine Projektion, d. h. eine Auswahl bestimmter Spalten, erhält man einfach dadurch, dass man das `'*'` im `SELECT`-Kommando durch die Attribut-Namen (d. h. die Überschriften der jeweiligen Spalten) ersetzt:

```
>>> cursor.execute('SELECT Matrikelnr,Nachname FROM Student')
6L
>>> cursor.fetchall()
((20123L, 'Fischer'), (19923L, 'Zeh'), (71232L, 'Huber'),
 (10012L, 'Becker'), (62618L, 'Klein'), (81232L, 'Lieb'))
```

Eine Selektion, d. h. die Auswahl bestimmter Zeilen, erhält man, indem man dem `SELECT`-Statement eine `WHERE`-Klausel folgen lässt.

Wir geben einige Beispiele; im Folgenden schreiben wir statt des einbettenden Python-Kommandos `cursor.execute('⟨SQL-Kommando⟩')` einfach das entsprechende SQL-Kommando:

- |  |  |
|--|--|
| <p>Alle Studenten deren Matrikelnummer größer als 70000 ist.</p> <p>a)</p> | <pre>SELECT Vorname,Nachname FROM Student WHERE Matrikelnr &gt; 70000;</pre> |
|--|--|

- Die Matrikelnummer(n)  
des (bzw. der) Studenten  
mit Nachnamen  
„Fischer“.

```
SELECT Matrikelnr FROM Student
WHERE Nachname = 'Fischer';
```

- Alle Studenten die sich  
seit 2008 immatrikuliert  
haben und im Studiengang  
MAB studieren.

```
SELECT Vorname, Nachname FROM Student
WHERE Imm>='2008-1-1'
AND Studiengang='MAB';
```

- Alle Studenten deren  
Nachname Meyer oder  
Maier ist.

```
SELECT Vorname, Nachname FROM Student
WHERE Nachname='Meyer'
OR Nachname='Maier';
```

Mit SQL kann man einfache Berechnungen ausführen. Die folgenden beiden Beispiele zeigen Berechnungen über Datumwerte. Beispiel (a) verwendet das Schlüsselwort **AS** um der Spalte der Ergebnistabelle, die die berechneten Werte enthält, einen Namen zu geben. Die SQL-Funktion `CURDATE()` liefert das aktuelle Datum, die Funktionen `YEAR(...)`, `MONTH(...)` und `DAY(...)` liefern das Jahr, den Monat bzw. den Tag einer Datumsangabe als Zahl zurück. Während Beispiel (a) eine Berechnung für die Erzeugung von Spalteninformationen durchführt, verwendet Beispiel (b) eine Berechnung für eine Restriktion in einer **WHERE**-Klausel.

- Die Namen aller Studenten  
zusammen mit der  
Spalte „Jahre“, die an-  
(a) gibt, wie viele Jahre der  
jeweilige Student bereits  
immatrikuliert ist.

```
SELECT (YEAR(CURDATE())-YEAR(Imm))
AS Jahre, Nachname
FROM Student;
```

- Alle Studenten, die sich  
(b) im Dezember immatriku-  
liert haben.

```
SELECT Vorname, Nachname FROM Student
WHERE MONTH(Imm)=12;
```

Man kann auch reguläre Ausdrücke innerhalb von **SELECT**-Anweisungen verwenden:

- Vor- und Nachname aller  
Studenten, deren Nach-  
(a) name mit einem A be-  
ginnt.

```
SELECT Vorname, Nachname FROM Student
WHERE Nachname REGEXP '^A';
```

- Die Matrikelnummern  
aller Studenten, deren  
(b) Nachname mindestens 8  
Buchstaben lang ist.

```
SELECT Matrikelnr, Nachname FROM Student
WHERE Nachname REGEXP '.....';
```

- Die Matrikelnummern,  
 (c) deren Vorname mit Ar beginnt und mit ra endet.
- |  |  |
|--|--|
| <pre>SELECT Matrikelnr FROM Student WHERE Nachname REGEXP '^Ar.*ra\$';</pre> |  |
|--|--|

### Aufgabe 5.5

Geben Sie ein `SELECT`-Statement an, das ...

- (a) die Vor- und Nachnamen aller Studenten liefert, die seit mehr als 3 Jahren immatrikuliert sind.
- (b) die Matrikelnummern aller Studenten liefert, deren Nachname mit 'Z' beginnt.
- (c) die Namen aller Studenten des Studiengangs KST liefert, deren Nachname mit 'M' beginnt und mit 'r' endet und ein 'e' oder 'a' enthalten, und die seit mindestens 2 Jahren immatrikuliert sind.
- (d) alle Vorlesungstitel des Professors mit der Personalnummer 5321 liefert, die mit 'P' beginnen.

### 5.4.3 Zählen und Statistiken

SQL-`SELECT`-Anweisungen können für einfache Statistiken verwendet werden. Mittels der `COUNT`-Funktion ist es möglich, Trefferlisten zu zählen, wie folgende Beispiele zeigen:

- Anzahl der Einträge in der Tabelle
- a) „**Student**“.
- |  |  |
|--|--|
| <pre>SELECT COUNT(*) FROM Student;</pre> |  |
|--|--|
- Anzahl der Studenten, die mindestens seit dem 1.1.2004 immatrikuliert sind.
- b)
- |   |  |
|---|--|
| <pre>SELECT COUNT(*) FROM Student WHERE Imm&lt;='2004-1-1';</pre> |  |
|---|--|

Mittels der Anweisung `GROUP BY` ist es möglich, Datensätze nach einer bestimmten Spalte (siehe Beispiel a) oder nach mehreren Spalten (siehe Beispiel b) zu gruppieren. In Verbindung mit `COUNT` können so Datensätze nach bestimmten Kriterien geordnet und gezählt werden.

- Anzahl der Studenten pro Studiengang; fetchall könnte z. B. folgen:
- a) de Datensätze zurückliefern:
- |  |  |
|--|--|
| <pre>(( 'BKT',141), ( 'KST',136), ( 'MAB',166), ( 'TI',138), ( 'WIN',140))</pre> | <pre>SELECT Studiengang, COUNT(*) FROM Student GROUP BY Studiengang;</pre> |
|--|--|

Die Anzahl der Immatrikulationen  
b) zu einem bestimmten Datum, pro  
Studiengang.

```
SELECT Studiengang, Imm, COUNT(*)
FROM Student
GROUP BY Studiengang, Imm;
```

### Aufgabe 5.6

Verwenden Sie SQL und Python, um ...

- (a) zu bestimmen, welcher Studiengang die meisten Studenten hat.
- (b) zu bestimmen, welcher Student sich als erstes immatrikuliert hat.
- (c) zu bestimmen, welcher Student des Studiengangs TI sich als erstes immatrikuliert hat.
- (d) zu bestimmen, welcher Student des Studiengangs BKT den kürzesten Nachnamen hat.

#### 5.4.4 Joins: Verknüpfung von Tabellen

Viele sinnvolle Abfragen müssen zwei oder mehrere Tabellen miteinander verknüpfen, um die gewünschten Informationen zu suchen. Will man beispielsweise wissen, wie die Vorlesungen heißen, die ein bestimmter Professor liest, so müssen die Tabellen „**Professor**“ und „**Vorlesung**“ verknüpft werden: Da das Feld „Prof“ der Tabelle „**Vorlesung**“ eine Referenz auf den entsprechenden Datensatz der Tabelle „**Professor**“ ist, kann man mittels einer Join-Operation (siehe Tabelle 5.3 auf Seite 144) die gewünschten Informationen sammeln. Folgende SQL-Anweisung findet die Titel der Vorlesungen, die der Professor „Guido Rossi“ hält:

```
SELECT Vorlesung.Titel
FROM Vorlesung, Professor
WHERE Professor.Nachname = 'Rossi' AND
      Professor.Vorname='Guido' AND
      Vorlesung.Prof = Professor.Personalnr;
```

Es folgen zwei weitere Beispiele für Joins. In Beispiel (a) wird gezeigt, wie sogar drei Tabellen verknüpft werden müssen, um die gewünschten Informationen zu suchen. Beispiel (b) zeigt das Erstellen einer Statistik über zwei Tabellen hinweg.

Alle Studenten, die  
a) an der Vorlesung  
„DB“ teilnehmen.

```
SELECT Student.Vorname, Student.Nachname
FROM Student, Vorlesung, VLStudent,
WHERE Vorlesung.Titel='DB'
      AND Student.Matrnr = VLStudent.Student
      AND Vorlesung.Nr = VLStudent.VL;
```

Alle Professoren  
zusammen mit der  
b) Anzahl der von  
Ihnen gehaltenen  
Vorlesungen.

```
SELECT COUNT(Vorlesung.Prof) AS VLs,
       Professor.Nachname
FROM Professor, Vorlesung
WHERE Professor.Personalnr
       = Vorlesung.Prof
GROUP BY Professor.Nachname,
         Professor.Personalnr;
```

### Aufgabe 5.7

Verwenden Sie ein SQL-SELECT-Statement, um folgende Fragen zu beantworten:

- (a) Wie viele Studenten hat der Studiengang TI?
- (b) Welcher Studiengang hat die meisten Studenten?
- (c) Wie viele Professoren haben die einzelnen Studiengänge?
- (d) Wie viele Studenten nehmen an den einzelnen Vorlesungen teil?
- (e) Wie ist das Betreuungsverhältnis im Studiengang 'BKT'?
- (f) Welche Vorlesungen hört Student Noam Chomsky?
- (g) Welche Studenten haben sich vor mehr als 5 Jahren eingeschrieben?
- (h) Mit welchen Studenten hat der Prof. Eva Lang über Ihre Vorlesungen zu tun?
- (i) Welche Studenten hören Vorlesungen von studiengangsfremden Professoren?
- (j) Die fleißigen Studenten: Alle Studenten die Vorlesungen höherer Semester hören.
- (k) Die Nachzügler: Alle Studenten die Vorlesungen niederer Semester hören.

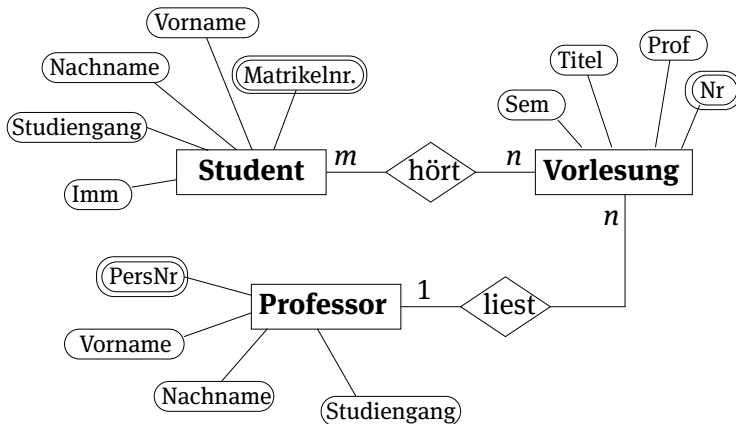
## 5.5 Entwurf relationaler DBMS

Der Entwurf einer relationalen Datenbank sollte, insbesondere auch aufgrund der Schwierigkeiten, ein relationales Datenmodell im Nachhinein zu ändern, sehr sorgfältig erfolgen. Eine häufig angewandte Modellierungstechnik verwendet Entity-Relationship-Diagramme, um relationale Datenbank-Schemata zu beschreiben.

### 5.5.1 Entity-Relationship-Diagramme

Abbildung 5.4 zeigt das Entity-Relationship-Diagramm einer einfachen Datenbank, die die Beziehungen zwischen Professoren, Vorlesungen und Studenten modelliert. Ein Entity-Relationship-Diagramm besteht aus den folgenden Komponenten:





**Abb. 5.4.** Ein Entity-Relationship-Diagramm, das in seiner Umsetzung als relationale Datenbank genau den in Abbildung 5.2 gezeigten Tabellen entspricht.

### Entitäten

Die Rechtecke modellieren die sogenannte *Entitäten*, das sind die realen Objekte, die in der relationalen Datenbankschema modelliert werden sollen. In Abbildung 5.4 werden die drei Entitäten „**Student**“, „**Professor**“ und „**Vorlesung**“ modelliert.

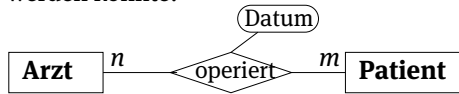
### Attribute von Entitäten

Attribute bezeichnen Eigenschaften einer Entität. So hat die Entität „**Professor**“ die Eigenschaften „PersNr“, „Vorname“, „Nachname“ und „Studiengang“. Ein Attribut (bzw. Attributkombinationen) dessen Wert ein bestimmtes Objekt des Entitätstyps eindeutig bestimmt, nennt man auch *identifizierendes Attribut*, manchmal auch *Schlüsselattribut*. Beispielsweise ist das Attribut „Matrikelnr“ der Entität „**Student**“ das Schlüsselattribut: die Matrikelnummer bestimmt eindeutig ein bestimmtes Objekt der Entität „Student“.

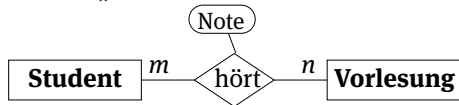
### Beziehungen

Explizit modelliert werden außerdem die Beziehungen zwischen den Entitäten, indem die entsprechenden Entitäten durch einfache Linien miteinander verbunden sind. Diese Linien werden zusätzlich mit einem Verb benannt, das die Art der Beziehung beschreiben soll. Das Entity-Relationship-Diagramm aus Abbildung 5.4 beschreibt beispielsweise die Beziehung zwischen **Student** und **Vorlesung** durch das Verb „hört“, d. h. ein Student „hört“ eine Vorlesung. Diese Beziehungsbezeichnungen werden üblicherweise durch eine Raute dargestellt.

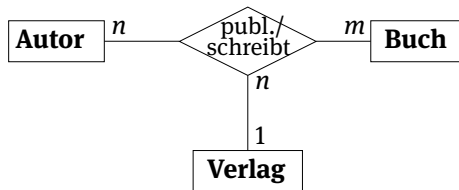
Auch Beziehungen können Attribute besitzen. Ein einfaches Beispiel wäre die Beziehung „operiert“ zwischen **Arzt** und **Patient**, der das Attribut „Datum“ zugewiesen werden könnte:



Ebenso wäre vorstellbar, der Beziehung „hört“ zwischen **Student** und **Vorlesung** ein Attribut „Note“ zuzuweisen.

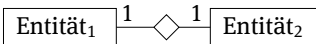


Auch sogenannte  $n$ -äre Beziehungen können modelliert werden. So könnte man etwa die 3-äre Beziehung zwischen „**Autor**“, „**Buch**“ und „**Verlag**“ folgendermaßen modellieren:

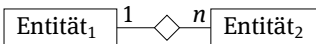


### Kardinalitäten von Beziehungen

Zu jeder Beziehung gehören *Kardinalitäten*, die festlegen, wie viele der zugehörigen Entitäten an der Beziehung beteiligt sein können. Vier unterschiedliche Kardinalitätstypen sind für eine Beziehung zwischen zwei Entitäten, Entität<sub>1</sub> und Entität<sub>2</sub>, denkbar:

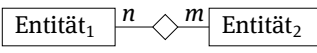
- Eine (1 : 1)-Beziehung: 

Diese legt fest, dass ein bestimmtes Objekt des Typs Entität<sub>1</sub> immer *genau* einem Objekt des Typs Entität<sub>2</sub> zugeordnet ist. Würde man beispielsweise „**Professor**“ und „**Lehrstuhl**“ als zwei getrennte Entitäten modellieren, so wäre eine (1 : 1)-Beziehung zwischen diesen denkbar: Ein Professor ist genau einem Lehrstuhl zugeordnet, und ein bestimmter Lehrstuhl ist genau einem Professor zugeordnet.

- Eine (1 :  $n$ )-Beziehung: 

Diese legt fest, dass ein bestimmtes Objekt des Typs Entität<sub>1</sub> immer mehreren Objekten des Typs Entität<sub>2</sub> zugeordnet werden kann. Ein Objekt des Typs Entität<sub>2</sub> jedoch muss immer genau einem Objekt des Typs Entität<sub>1</sub> zugeordnet sein. Dies ist bei der Beziehung zwischen „**Professor**“ und „**Vorlesung**“ der Fall: Ein Profes-

sor kann mehrere Vorlesungen halten; eine bestimmte Vorlesung jedoch ist genau einem Professor zugeordnet.

- Eine  $(n:m)$ -Beziehung: 

Diese legt fest, dass sowohl jedes Objekt des Typs Entität<sub>1</sub> mehreren Objekten des Typs Entität<sub>2</sub> zugeordnet werden kann als auch jedes Objekt des Typs Entität<sub>2</sub> mehreren Objekten des Typs Entität<sub>1</sub>. Dies ist bei der Beziehung zwischen „Student“ und „Vorlesung“ der Fall: Ein Student hört im Allgemeinen mehrere Vorlesungen, und jede Vorlesung wird von mehreren Studenten besucht.

### 5.5.2 Umsetzung in ein relationales Modell

Es gibt einige einfache Regeln, die festlegen, wie ein Entity-Relationship-Diagramm in ein relationales Modell – sprich: in ein relationales Datenbankschema – umgesetzt werden kann.

- Eine Entität wird durch eine Tabelle einer relationalen Datenbank realisiert.
- Eine  $(1:1)$ -Beziehung der Entitäten „Entität<sub>1</sub>“ und „Entität<sub>2</sub>“ wird dadurch realisiert, dass eine der beiden Tabellen um den Fremdschlüssel der anderen erweitert wird.
- Eine  $(1:n)$ -Beziehung der Entitäten „Entität<sub>1</sub>“ und „Entität<sub>2</sub>“ wird folgendermaßen realisiert: Die Tabelle, die der mit „n“ markierten Entität entspricht, wird um den Fremdschlüssel der Tabelle erweitert, die der mit „1“ markierten Entität entspricht.
- Eine  $(n:m)$ -Beziehung der Entitäten „Entität<sub>1</sub>“ und „Entität<sub>2</sub>“ wird folgendermaßen realisiert: Es wird eine neue (Hilfs-)Tabelle angelegt, die Fremdschlüssel der „Entität<sub>1</sub>“-Tabelle und Fremdschlüssel der „Entität<sub>2</sub>“-Tabelle enthält. Enthält diese Beziehung selbst Attribute (wie etwa in obigem Beispiel „Student hört(e) Vorlesung mit Note ...“), so werden diese zusätzlich in diese Hilfstabelle mit aufgenommen. Betrachten wir die auf Seite 153 als Beispiel präsentierte Arzt-Patienten-Beziehung „operiert“ mit dem Beziehungsattribut „Datum“. Diese Beziehung würde man wie in Abbildung 5.5 gezeigt als relationale Datenbank realisieren:

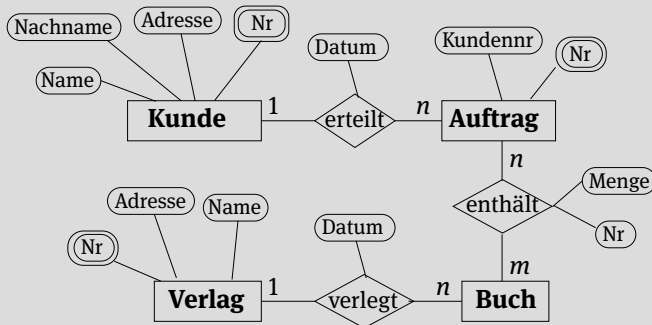
| Arzt   |       |     | Patient |           |     | ArztPatient |      |          |
|--------|-------|-----|---------|-----------|-----|-------------|------|----------|
| Persnr | Name  | ... | Nr.     | Name      | ... | Patient     | Arzt | Datum    |
| 100    | Freud | ... | 231     | Nietzsche | ... | 231         | 100  | 1-1-1882 |
| ...    | ...   | ... | ...     | ...       | ... | ...         | ...  | ...      |

Abb. 5.5. Realisierung einer  $(n:m)$  – Beziehung

- Eine  $n$ -äre Beziehung (= Beziehung mit  $n$  beteiligten Entitäten) der Entitäten „Entität<sub>1</sub>“, „Entität<sub>2</sub>“, ..., „Entität <sub>$n$</sub> “ kann folgendermaßen realisiert werden: Es wird – ähnlich wie bei der Realisierung einer  $(n:m)$ -Beziehung – eine Hilfetabelle eingefügt, die als Attribute die Fremdschlüssel aller Beteiligten Entitäten enthält, inklusive der Attribute, die die  $n$ -äre Beziehung selbst hat.

### Aufgabe 5.8

Gegeben sei folgendes Entity-Relationship-Modell:



- Erstellen Sie mit Pythons MySQLdb-Modul eine MySQL-Datenbank, die dieses Entity-Relationship-Modell realisiert. Füllen Sie die Tabellen mit einigen Werten.
- Geben Sie SQL-Abfragen an, die folgende Informationen liefern.
  - Welche Aufträge hat der Kunde „Michael Maier“ seit dem 1.1.2007 erteilt?
  - Welche Bücher hat der Kunde „Michael Maier“ seit dem 1.1.2007 bestellt?
  - Wie oft wurde das Buch „Einführung in die Informatik“ bestellt?
  - Welche Kunden haben Bücher des Verlages „Oldenbourg“ bestellt?

### Aufgabe 5.9

Ein Patient mit einer eindeutigen „Patienten-ID“, einem „Namen“ und einem „Geburtsdatum“ wird zu einem bestimmten Datum von genau einem Arzt (mit

einer eindeutigen „Personal-ID“ und einem „Namen“) operiert; an dieser Operation beteiligt können mehrere Krankenschwestern (bzw. Krankenpfleger) sein (mit einer eindeutigen „Personal-ID“ und einem „Namen“).

- (a) Erstellen Sie ein passendes Entity-Relationship-Modell.
- (b) Realisieren Sie ein passendes relationales Datenbankmodell mit Hilfe von Pythons `MySQLdb`-Modul. Fügen sie einige konkrete Patienten, Ärzte und Krankenschwestern ein, die Operationen an den Patienten durchführen.
- (c) Erstellen Sie SQL-Abfragen, die folgende Informationen liefern:
  1. Welche Patienten wurden im Zeitraum vom 1.1.2008 bis 1.6.2008 vom Arzt „Hagen Schmidt“ operiert?
  2. Zu welchen Zeitpunkten war der Krankenpfleger „Harald Rether“ an Operationen beteiligt?
  3. Wie viele Operationen gab es 2009?

### 5.5.3 Normalisierung

Es kann zu inkonsistenten Daten alleine dadurch kommen, dass ein relationales Datenbankschema ungeschickt entworfen wurde. Solche Inkonsistenzen können vermieden werden, indem man ein relationales Datenbankschema sukzessive in die sogenannte 1. Normalform, 2. Normalform, 3. Normalform, 4. Normalform und 5. Normalform überführt. Diese Normalformen legen, grob gesprochen, Regeln fest, welche Abhängigkeiten zwischen Schlüssel- und Nicht-Schlüsselwerten bestehen dürfen. Wir skizzieren im Folgenden die ersten 3 Normalformen.

#### 1. Normalform

Die 1. Normalform fordert, dass jedes Attribut einen *atomaren*, also keinen zusammengesetzten, Wert beinhalten soll. Beispielsweise würde ein Attribut „Adresse“ diese Forderung verletzen, da es zusammengesetzt ist (etwa aus PLZ, Ort, Straße, Hausnummer). Ein solches zusammengesetztes Attribut kann leicht Inkonsistenzen erzeugen, wie etwa in folgendem Beispiel zwei unterschiedliche Städtenamen zur selben Postleitzahl; noch dazu sind Ort und Straßename einmal durch ein Komma und einmal durch einen Strichpunkt getrennt:

| Kunde |         |          |                               |
|-------|---------|----------|-------------------------------|
| Nr    | Vorname | Nachname | Adresse                       |
| 100   | Hugo    | Müller   | 89134 Blaustein, Rehstrasse 1 |
| 101   | Egon    | Möller   | 89134 Berlin; Uhuweg 2        |
| ...   | ...     | ...      | ...                           |

Um diese Tabelle in die 1. Normalform zu übertragen sollte man das Attribut „Adresse“ in mindestens die drei Attribute „Plz“, „Ort“ und „Straße“ aufteilen.

## 2. Normalform

Eine Tabelle befindet sich in der 2. Normalform, wenn sie sich in der 1. Normalform befindet und wenn kein Nichtschlüsselattribut von einer (echten) Teilmenge der Schlüsselattribute abhängt. Nehmen wir als Beispiel folgende Tabelle, die die 2. Normalform verletzt:

| Rezensionen |              |                       |                  |
|-------------|--------------|-----------------------|------------------|
| ISBN        | Rezensent-ID | Zusammenfassung       | Rezensent-Email  |
| 3834813729  | 21           | Sehr gut              | twon@t-online.de |
| 3834813729  | 11           | Ganz gut, nur zu lang | elev@web.de      |
| 3486586270  | 21           | Super                 | twoo@t-online.de |
| ...         | ...          | ...                   | ...              |

Die beiden Attribute „ISBN“ und „Rezensent-ID“ bilden zusammen die Schlüsselattribute der Tabelle. Das Attribut „Rezensent-Email“ hängt aber von einem echten Teil des Schlüssels ab, nämlich (ausschließlich) von dem Attribut „Rezensent-ID“. Darum befindet sich die Tabelle „Rezensionen“ nicht in der 2. Normalform. Tatsächlich kann diese Abhängigkeit von einem Teil des Schlüssels sehr leicht zu Inkonsistenzen führen, wie dies in obiger Tabelle auch tatsächlich der Fall ist: Die Email des Rezensenten 21 ist nicht konsistent, nämlich in Zeile 1 ist sie `twon@t-online.de` und in Zeile 3 dagegen `twoo@t-online.de`.

Um diese Tabelle in die 2. Normalform zu überführen, müsste man eine zweite Tabelle „Autor“ erstellen und die Email-Adresse des Autors in dieser zweiten Tabelle speichern.

### Aufgabe 5.10

- Teilen Sie, wie im Text beschrieben, die Tabelle „Rezensionen“ in zwei Tabellen auf, so dass diese sich in der 2. Normalform befinden.
- Verwenden Sie Pythons `MySQLdb`-Modul, um eine kleine relationale Datenbank mit diesen Tabellen zu erstellen, und füllen Sie die Tabellen mit einigen Beispielwerten.
- Geben Sie SQL-Kommandos an, die folgende Fragen beantworten:
  - Wie viele Bücher hat der Autor mit der EMail-Adresse `elev@web.de` rezensiert?
  - Es sollen alle Zusammenfassungen von Rezensionen des Buches mit der ISBN 3834813729 ausgegeben werden.

3. Es soll gezählt werden, wie viele Zusammenfassungen, die der Rezensent mit der „ID“ 21 geschrieben hat, das Wort „schlecht“ enthalten.

### Aufgabe 5.11

Gegeben sei eine Tabelle einer relationalen Datenbank. Der Schlüssel dieser Tabelle besteht aus nur einem Attribut. Wir nehmen an, diese Tabelle befinde sich in der 1. Normalform.

Befindet sich diese Tabelle damit auch automatisch in der 2. Normalform? Oder anders gefragt: Ist es möglich, dass diese Tabelle sich nicht in der 2. Normalform befindet?

### 3. Normalform

Eine Tabelle befindet sich in der 3. Normalform, wenn sie sich in der 2. Normalform befindet und wenn kein Attribut transitiv von einem Schlüsselattribut abhängt. Eine transitive Abhängigkeit eines Attributs  $A$  von einem Schlüsselattribut  $S$  liegt dann vor, wenn es ein Attribut  $A'$  gibt, so dass  $A'$  von  $S$  abhängt (formal:  $S \rightarrow A'$ ) und  $A$  von  $A'$  abhängt (formal:  $A' \rightarrow A$ ).

Anders ausgedrückt: Eine Tabelle entspricht der 3. Normalform, wenn alle Attribute direkt vom Schlüssel abhängen. Jedes Attribut, das verändert werden sollte, wenn sich ein anderes Nichtschlüsselattribut verändert, verletzt also die 3. Normalform. Beispielsweise verletzt die folgende Tabelle die 3. Normalform:

| Autor   |         |          |       |           |
|---------|---------|----------|-------|-----------|
| AutorID | Vorname | Nachname | PLZ   | Stadt     |
| 125     | Udo     | Beck     | 89134 | Blaustein |
| ...     | ...     | ...      | ...   | ...       |

Das Attribut „Stadt“ hängt transitiv vom Schlüssel „AutorID“ ab, denn „AutorID“  $\rightarrow$  „PLZ“ und „PLZ“  $\rightarrow$  „Stadt“. Das Attribut „Stadt“ hängt also nicht direkt vom Schlüssel ab und folglich befindet sich diese Tabelle nicht in der 3. Normalform. Man muss diese Tabelle in zwei Tabellen aufspalten, um die 3. Normalform herzustellen: Eine Tabelle „Autor“, mit den Attributen „Vorname“ und „Nachname“ und einer Adress-ID; und einer Tabelle „Adresse“ mit Schlüssel „PLZ“ und weiteren Attributen, unter anderem eben den Städtenamen.

**Aufgabe 5.12**

Wie im Text erläutert, wird durch obige Tabelle „Autor“ die 3. Normalform verletzt. Die Normalformen sollen ja immer auch sicherstellen, dass keine Dateninkonsistenzen auftreten können.

Beschreiben Sie, welche Inkonsistenzen durch das Schema der Tabelle „Autor“ entstehen können.

**5.6 Nicht-Relationale DBMS**

Der Ansatz der relationalen Datenbanken, alle Daten in vordefinierten Tabellen zu halten, hat sich lange bewährt und ist immer noch der de-facto-Standard, große Datenmengen zu verwalten. Für viele Anwendungen ist der relationale Ansatz jedoch nicht passend, insbesondere dann nicht, wenn viele große Dokumente (inklusive deren Volltext-Inhalten) verwaltet werden sollen, oder wenn eine vordefinierte Struktur der Daten nicht sinnvoll ist.

Ferner sind relationale Datenbanksysteme grundsätzlich darauf ausgelegt, auf einem Zentralrechner (in diesem Zusammenhang oft als „Server“ bezeichnet) zu laufen. Dagegen lassen sich viele nicht-relationale Datenbanken, wie beispielsweise CouchDB, einfacher auf viele einzelne Rechner an vielen unterschiedlichen Standorten verteilen. Genau diese Anforderung, Daten verteilt zu halten und Anfragen verteilt zu bearbeiten, ist für viele moderne Web-Anwendungen ausschlaggebend – man denke nur an Google, Amazon usw.

**5.6.1 CouchDB: Datenverfügbarkeit vs. Datenkonsistenz**

Ein bekannter aufstrebender Vertreter dieser nicht-relationalen Datenbanken ist die CouchDB[1], die ein schemafreies Dokumentenmodell unterstützt.

Relationale Datenbankmanagementsysteme legen im Allgemeinen großen Wert auf die Datenkonsistenz; jedoch kann es insbesondere bei sehr großen Datenbeständen sehr aufwändig sein, Konsistenzprüfungen über den gesamten Datenbestand durchzuführen. Hinzu kommt, dass das Überführen eines relationalen Datenbankentwurfs in die 3. Normalform oder gar 4. Normalform – ein notwendiger Schritt, wenn man auf Datenkonsistenz großen Wert legt – immer einhergeht mit der Aufspaltung von Tabellen. Die Folge davon ist, dass für Abfragen teilweise viele Joins notwendig sind, die wiederum rechenauf-



wändig sind. Es besteht ein *Trade-Off*<sup>3</sup> zwischen einer Datenverfügbarkeit und *Skalierbarkeit* und zum anderen Datenkonsistenz. Skalierbarkeit meint in diesem Zusammenhang die Möglichkeit, das DBMS auf mehrere Rechner, mehrere Festplatten oder mehrere Server<sup>4</sup> zu verteilen und während des Betriebs zusätzliche Festplatten, Rechner oder Server anzuschließen bzw. abzustellen.

CouchDB strebt, wie auch andere NoSQL<sup>5</sup>-Datenbanken, einen Kompromiss zugunsten hoher Verfügbarkeit und Skalierbarkeit an: Anstatt eine über die ganze Zeit hinweg vollständige Datenkonsistenz anzustreben, verfolgt CouchDB eine sogenannte „*Eventual Consistency*“-Strategie, d. h. die Daten sind letztendlich (irgendwann einmal) konsistent; temporär inkonsistente Zustände werden zugunsten einer hohen Verfügbarkeit bewusst in Kauf genommen. Dies mag wohl bei Banktransaktionen nicht hinnehmbar sein, tatsächlich ist dieser Kompromiss so aber für sehr viele Anwendungen sinnvoll.

### 5.6.2 Funktionsprinzipien

Die im Folgenden beschriebenen Funktionsprinzipien mögen nicht für alle nicht-relationalen Datenbanken gleichermaßen gelten. Zwar mag für jede sog. NoSQL-Datenbank das Prinzip der Schemafreiheit gelten; genau die Tatsache, dass relationale Datenbankmanagementsysteme aufgrund ihrer Bindung an feste Tabellenschemata zu unflexibel sind, ist ja ein wichtiger Grund dafür, auf nicht-relationale Datenbankmanagementsysteme auszuweichen. Wie eine nicht-relationale Datenbank jedoch Datenkonsistenz sicherstellt (bei CouchDB mittels MVCC), und ob oder wie Daten repliziert werden (bei CouchDB mittels inkrementeller Replikation), kann von Datenbankmanagementsystem zu Datenbankmanagementsystem sehr verschieden sein.

#### Schema-Freiheit

Daten werden in CouchDB nicht in Tabellen eines festen Schemas gespeichert; ein bestimmter Datensatz, in CouchDB auch oft als *Dokument* bezeichnet, wird stattdessen schemafrei im sogenannten JSON-Format (siehe Abschnitt 5.6.3) gespeichert. Der Name „Dokument“ mag hier tatsächlich etwas irreführend sein, denn ein CouchDB-

<sup>3</sup> Der englische Begriff *Trade-Off* stammt eigentlich aus der Volkswirtschaftslehre, wird aber in der Informatik recht häufig verwendet; man könnte ihn mit Zielkonflikt oder Kosten-Nutzen-Abwägung übersetzen. Ein Trade-Off liegt dann vor, wenn man eine Verbesserung eines Aspektes nur unter Inkaufnahme der Verschlechterung eines anderen Aspektes erreichen kann.

<sup>4</sup> Unter *Server* versteht man einen Rechner, der sich (vereinfacht gesprochen) in ständiger Bereitschaft befindet, um evtl. über das Netzwerk ankommende Anfragen von „Außen“ zu bearbeiten und die anfragenden Rechner entsprechend mit Daten zu „bedienen“. Mehr zu den Begriffen „Server“ und „Client“ in Abschnitt 6.1.6.

<sup>5</sup> NoSQL ist ein Akronym für: „not only SQL“. Damit werden gemeinhin Datenbanken bezeichnet, die kein oder kein ausschließliches relationales Konzept verfolgen.

Dokument (d. h. ein JSON-Datensatz) besteht aus einer Menge von Schlüssel-Wert-Paaren, eingeschlossen in die geschweiften Klammern '{' und '}' – also einer Struktur ganz ähnlich einem Python-Dictionary-Objekt. Im Unterschied zu Python-Dictionary-Objekten sind in JSON-Strukturen aber lediglich Strings als Schlüssel erlaubt. Be beispielsweise könnte folgende JSON-Struktur ein CouchDB-Dokument sein:

```
{ "name": "Tobias Haerberlein",
  "plz": "89134",
  "ort": "Blaustein",
  "email": ["haeberlein@hs-albsig.de", "tobias.haerberlein@gmx.de"] }
```

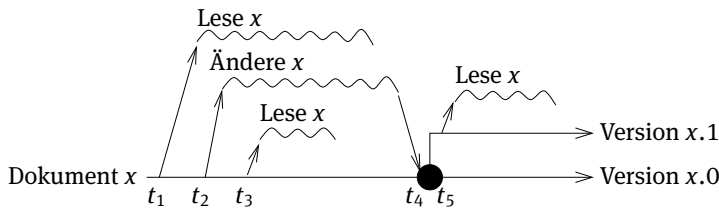
Aufgrund der Tatsache, dass man bei Verwendung von CouchDB nicht an ein bestimmtes Schema gebunden ist, ist es problemlos möglich einen weiteren Personendatensatz einzufügen, der andere Schlüssel enthält, etwa durch folgendes Dokument:

```
{ "vorname": "Gregory",
  "nachname": "Bateson",
  "ort": "London",
  "buecher": ["steps to an ecology of mind", "mind and nature"] }
```

### MVCC statt Locks

Hier gibt es keine Locks, d. h. Leseoperationen müssen nicht wie bei relationalen Datenbanksystemen auf andere Schreiboperationen warten. Dadurch, dass beliebig viele Clients ohne diese Verzögerungen von der Datenbank lesen können, ist eine hohe Verfügbarkeit sichergestellt.

Multi-Version Concurrency Control (MVCC), stellt sicher, dass eine Transaktion grundsätzlich nie auf Datenbankzugriffe warten muss. MVCC funktioniert folgendermaßen: Sobald ein Dokument geändert wird, wird eine neue Version des Dokuments erzeugt und *zusätzlich* zur alten Version gespeichert – ganz ähnlich, wie dies auch in einem Versions-Kontrollsystem geschieht. Abbildung 5.6 veranschaulicht die Funktionsweise von MVCC anhand eines einfachen Beispiels.



**Abb. 5.6.** Multi-Version-Concurrency-Control in Aktion: Zum Zeitpunkt  $t_1$  liest eine Transaktion das Dokument  $x$ . Ein Concurrency-Control-Mechanismus, der Locks verwendet, um Datenkonsistenz sicherzustellen, würde daraufhin das Dokument  $x$  so lange sperren, bis die Transaktion die Leseaktion beendet hat (also etwa bis Zeitpunkt  $t_4$ ). MVCC arbeitet jedoch nicht mit Locks und erlaubt einer zweiten Transaktion kurz darauf (zum Zeitpunkt  $t_2$ ) das Dokument zu ändern, ohne dass Verzögerungen entstehen. Auch während diese Änderungen gerade durchgeführt werden, kann zum Zeitpunkt  $t_3$  eine weitere Transaktion ohne Verzögerungen eine Leseoperation durchführen. Sobald die vorige Transaktion die Änderung in die Datenbank schreibt, wird automatisch eine neue Version des Dokumentes erzeugt.

### Aufgabe 5.13

Zeichnen Sie ein ähnliches Bild wie das in Abbildung 5.6 und stellen Sie dieses Mal den zeitlichen Verlauf dar, der in einer Lock-basierten Datenbank entstehen würde.

### Aufgabe 5.14

Überlegen Sie, was passieren könnte, wenn zum Zeitpunkt  $t_3$  statt einer Leseoperation eine Schreiboperation ausgeführt werden würden. Sie können hierbei die folgenden beiden Fälle unterscheiden: (a) die zum Zeitpunkt  $t_3$  stattfindende Schreiboperation ist *vor* Zeitpunkt  $t_4$  beendet; (b) die zum Zeitpunkt  $t_3$  stattfindende Schreiboperation ist *nach* Zeitpunkt  $t_4$  beendet.

### Inkrementelle Replikation

Als *Replikation* bezeichnet man die Situation, dass dieselben Daten über mehrere Standorte (bzw. Rechner oder Festplatten) verteilt mehrmals gespeichert sind. Dies kann aus einem bzw. aus einer Kombination der folgenden beiden Gründe erforderlich sein. **1.** Man will sicherstellen, dass die Daten sicher sind, d.h. dass die Daten auch dann, wenn ein Standort bzw. Rechner ausfällt, noch erhalten bleiben. **2.** Man will eine hohe Verfügbarkeit der Daten bzw. kurze Antwortzeiten bei vielen gleichzeitigen Zugriffen auf die Datenbank sicherstellen.

Replikation erfordert immer einen Mechanismus, die Daten zu synchronisieren, d. h. abzugleichen. Je nach dem, welche Synchronisationstechnik verwendet wird, können auch Konflikte entstehen, beispielsweise dann, wenn ein an zwei unterschiedlichen Stellen gespeichertes Datum zwischen zwei Synchronisationen in beiden Datenquellen geändert wurde. Es gibt unterschiedliche Strategien mit solchen Konflikten umzugehen, wir gehen hier jedoch nicht auf Details ein.

CouchDB verwendet eine sog. *Inkrementelle Replikation*: Die Replikation erfolgt nicht bei jeder Dokumentenänderung, sondern zu bestimmten (eventuell durch den Benutzer) vordefinierten Zeitpunkten.

### 5.6.3 JSON

JSON (= JavaScript Object Notation) ist neben XML ein häufig, insbesondere für viele Web-Anwendungen verwendetes textbasiertes<sup>6</sup> Datenaustauschformat. Es ist hilfreich, um etwa die durch eine Anwendung erzeugten Daten temporär zu speichern und dadurch einer anderen Anwendung verfügbar zu machen, oder einfach, um die Daten einer Anwendung persistent zu speichern.

Sehr oft wird JSON in Web-Anwendungen verwendet, um Daten von einem Web-Service an eine im Webbrowser ausführbare JavaScript-Anwendung zu transportieren. Und das geht deshalb besonders einfach, da JSON-Datensätze einer syntaktischen Teilmenge von JavaScript-Ausdrücken entsprechen – daher eben auch der Name „JavaScript Object Notation“.

Pythons Standardbibliothek `json` bietet alle notwendigen Funktionalitäten, um Pythondaten in JSON umzuwandeln und um JSON-Daten in Python einzulesen:

```
>>> import json
>>> adict = {'eins' :1, 'zwei' :{'ersteins' :1, 'nocheins' :1}, 3 :
'drei' }
>>> json.dumps(adict)
'{"3": "drei", "eins": 1, "zwei": {"nocheins": 1, "ersteins": 1}}'
```

Die Funktion `dumps` liefert die JSON-Repräsentation eines Pythonobjektes. Wie man sieht, entspricht ein Python-Dictionary-Objekt recht genau einem JSON-Objekt: Beide bestehen aus einer Menge von Schlüssel-Wert-Paaren, eingeschlossen in geschweifte Klammern und beide können potentiell geschachtelt werden. Ein wichtiger Unterschied besteht darin, dass ein JSON-Objekt lediglich Strings als Schlüssel zulässt; ein

---

<sup>6</sup> Mit „textbasiert“ ist gemeint, dass die Daten so repräsentiert sind, dass sie durch einen Texteditor erstellt, betrachtet und geändert werden können; im Allgemeinen entspricht jedes einzelne Byte eines textbasierten Datenformats – zumindest dann, wenn die einzelnen Zeichen im ASCII-Format vorliegen – einem (etwa in einem Texteditor) darstellbaren Zeichen. Ein binäres Datenaustauschformat dagegen kann jedes einzelne Bit ausnutzen, um Daten zu repräsentieren; ein einzelnes Byte entspricht daher im Allgemeinen keinem darstellbaren Zeichen.

Dictionary-Objekt unterstützt dagegen Werte eines beliebigen (nicht-veränderbaren) Typs – Integer, Floats, Tupel, usw. – als Schlüssel. Wie man sieht, wurde bei der Umwandlung des obigen Dictionary-Objekts in ein JSON-Objekt der Integer-Schlüssel 3 in einen String "3" umgewandelt.

### Aufgabe 5.15

Betrachten wir folgendes in der Variablen `adict` gespeichertes Python-Dictionary-Objekt:

```
>>> adict = {'name' : 'knuth' , 'job' : 'computer scientist' }
```

Gilt `str(adict) == json.dumps(adict)` ?

Neben Dictionary-artigen Mengen von Schlüssel-Wert-Paaren, kennt JSON auch noch Listen (durch Kommata getrennte, in eckige Klammern eingeschlossene Werte), Integer, Strings, die Booleschen Werte `true` und `false` und den Wert `null`.

### Aufgabe 5.16

Pythons Modul `json` stellt neben `dumps` die Funktion `loads` zur Verfügung, die einen JSON-String in ein entsprechendes Python-Objekt umwandelt. Beantworten Sie zunächst ohne die Hilfe eines Rechners und danach durch Ausprobieren in Python die folgenden beiden Fragen.

- (a) Sei `s` ein JSON-String, der ein JSON-Objekt, d. h. eine Liste von Schlüssel-Wert-Paaren enthält. Unter welchen Umständen gilt, dass `dumps(loads(s))` den String `s` unverändert zurückliefert?
- (b) Sei `d` ein Python-Dictionary. Unter welchen Umständen gilt, dass `loads(dumps(s))` das Dictionary `d` unverändert zurückliefert?

## 5.6.4 Erzeugen einer CouchDB-Datenbank mit Python

Unter Verwendung des Python-Moduls `couchdb.client` kann man mit einem laufenden CouchDB-Datenbankserver interagieren. Dieses Modul ist nicht im Standardsatz der Module, die mit Python installiert werden, und muss mit

```
$ python -m pip install couchdb
```

nachinstalliert werden. Listing 5.2 zeigt ein Python-Skript, das eine neue CouchDB-Datenbank erzeugt und Datensätze in diese einfügt.

```

from couchdb.client import *
from json import dumps, loads

# Verbindung zum Server aufbauen
srv = Server('http://localhost:5984')

# Erzeuge Datenbank mit Namen test
mydb = srv.create('test')

person = {
    'name' : 'Tobias Haeberlein',
    'age' : 38,
    'ort' : 'Blaustein',
    'strasse' : 'Blauweg 11',
    'land' : 'Deutschland',
    'email' : ['haeberlein@hs-albsig.de', 'tobias.haeberlein@gmx.de'] }
mydb.create(dumps(person)) # Erstellen eines neuen Dokuments

person = {
    'name' : 'Gregory Bateson',
    'age' : '107',
    'adresse' : 'London',
    'land' : 'England',
    'email' : ['bateson@web.de', 'greg.bateson@gmx.de'] }
mydb.create(dumps(person))

```

**Listing 5.2.** Python-Skript, das eine neue CouchDB-Datenbank 'test' erzeugt und zwei Personendatensätze einfügt

In Zeile 5 wird die Verbindung zum CouchDB-Server aufgebaut; hierfür ist die Angabe einer URL (= Unified Resource Locator; siehe Abschnitt 6.4.2 auf Seite 182) notwendig. In diesem Beispiel verwenden wir `localhost` als URI, d. h. wir gehen davon aus, dass der CouchDB-Server sich auf dem selben Rechner befindet, auf dem auch das Python-Skript ausgeführt wird. CouchDB kommuniziert standardmäßig über den Port 5984. In Zeile 8 wird mittels `create` eine neue Datenbank erzeugt. Die Variable `mydb` enthält den entsprechenden Verweis darauf.

In den Zeilen 10 bis 16 und 19 bis 24 werden zwei Personendatensätze als Python-`dict`-Objekte erzeugt. Damit diese in die CouchDB eingefügt werden können, müssen sie im JSON-Format vorliegen. Die Umwandlung des `dict`-Objektes in das JSON-Format erledigt die Funktion `dumps` des in der Standardbibliothek enthaltenen Moduls `json`. In den Zeilen 17 und 25 werden die beiden ins JSON-Format umgewandelten Datensätze mittels der `create`-Anweisung in die Datenbank `test` eingefügt.

**Aufgabe 5.17**

Sollte die Datenbank `test` schon existieren, dann bricht das in Listing 5.2 gezeigte Skript ab. Fügen Sie mittels `try` und `except` ein einfaches Exception-Handling ein, das diesen möglichen Fehler folgendermaßen abfängt: Sollte die CouchDB schon eine Datenbank mit Namen `test` haben, so soll eine kurze Fehlermeldung ausgegeben werden, anschließend soll diese Datenbank gelöscht werden (Pythons `del`-Funktion kann auf das `srv`-Objekt angewendet werden) und schließlich neu erzeugt werden.

**5.6.5 Benutzerdefinierte Views mit MapReduce**

Auch was die Anfragen an die Datenbank betrifft, ist CouchDB flexibler als herkömmliche relationale Datenbanksysteme. Während man bei Anfragen in relationalen Datenbanken beschränkt ist auf SQL-SELECT-Anweisungen, kann man in CouchDB die Anfragen selbst programmieren; damit ist es potentiell möglich einen großen Teil der Anwendungslogik direkt in die Datenbank einzubetten.

Die damit gewonnene Flexibilität wird aber damit bezahlt, dass es etwas schwieriger ist, sich in den Anfrage-Mechanismus von CouchDB einzudenken. Dieser Anfrage-Mechanismus ist angelehnt (jedoch nicht identisch) an das ursprünglich von Google entworfene MapReduce-Framework [4], das speziell auf die Anforderungen verteilter und paralleler Programmierung ausgelegt ist und die effiziente Berechnung bestimmter Funktionen über viele potentiell weit verteilte Rechner erlaubt.

Die Erzeugung einer View – was in diesem Zusammenhang nichts anderes ist als eine Datenbankabfrage inklusive bestimmter benutzerdefinierter Berechnungen auf den Daten – unterteilt sich in die Aufgaben, zum einen eine passende Map-Funktion zu entwerfen und zum anderen eine passende Reduce-Funktion.

- *Map*: Erzeugt aus einem Dokument ein oder auch mehrere temporäre Schlüssel-Wert-Paare, die man auch als temporäre View bezeichnen kann. Eine Map-Funktion ist definiert in JavaScript-Syntax und hat die folgende allgemeine Form:

```
''' function(doc) {
    ...
    emit(k1,v1);
    ...
    emit(kn,vn); }'''
```

Die Key-Value-Einträge der temporären View, werden also durch die (vordefinierte) Funktion `'emit'` erzeugt.

- *Reduce*: Eine Reduce-Funktion sollte ausschließlich dazu verwendet werden, die Einträge einer (durch die Map-Funktion temporär erzeugten) View zusammenzu-

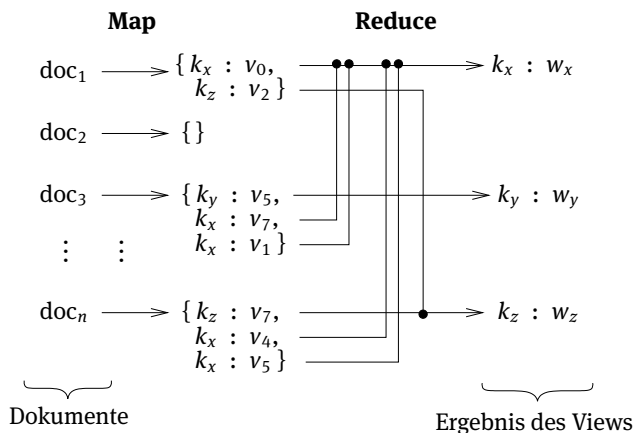
fassen bzw. zu vereinigen und eventuell durch eine bestimmte Funktion zu verknüpfen. Der Reduce-Funktion wird eine Liste von Zwischen-Werten übergeben, und Reduce fasst diese in einem einzigen Wert zusammen. Eine Reduce-Funktion muss, ebenso wie die Map-Funktion, in JavaScript-Syntax definiert werden und hat die folgende allgemeine Form:

```
''' function(keys, values) {
    ...
    return ergebnis; }'''
```

Die Reduce-Funktion sollte einen einzigen Wert zurückliefern. Da die Reduce-Funktion eventuell wiederum auf die durch sie erzeugten Werte angewendet wird, sollte dieser erzeugte Wert ...

- ...sowohl als Inhalt im Value-Feld des letztendlich erzeugten Views dienen können
- ...als auch ein Element der Value-Liste der Reduce-Funktion sein können.

Abbildung 5.7 zeigt das Zusammenspiel der Map-Funktion und der Reduce-Funktion bei der Erzeugung eines Views. Diese Darstellung nimmt an, dass bei der Anfrage `'group=True'` gesetzt ist; dies bewirkt, dass die Reduce-Funktion die Werte gruppiert nach den Schlüsseln verknüpft.



**Abb. 5.7.** Graphische Darstellung des Zusammenspiels der Map-Funktion und der Reduce-Funktion bei der Berechnung einer View in CouchDB: Ganz links im Bild sind alle Dokumente dargestellt, die sich momentan in der CouchDB befinden, ganz rechts im Bild ist der durch die kombinierte Anwendung der Map-Funktion und der Reduce-Funktion gebildete View dargestellt. Map bildet jedes Dokument auf eine Menge von Schlüssel-Wert-Paaren ab. Reduce verknüpft – zumindest dann, wenn `'group=True'` gesetzt ist – anschließend alle einem bestimmten Schlüssel zugeordneten Werte und berechnet daraus jeweils einen Wert.



**Beispiel 1: Anzahl Personen pro Land**

Wir wollen ausgeben, wie viele in der CouchDB gespeicherte Personen sich in den jeweiligen Ländern befinden. Betrachten wir den Code aus Listing 5.3, der als Fortsetzung von Listing 5.2 gedacht ist:

#Code aus Listing 5.2

```
mapPPL = ''' function(doc) { emit(doc.land,1) }'''
reducePPL = ''' function(keys, vals) { return sum(vals); }'''
viewPPL = mydb.query(mapPPL, reducePPL, group=True)
```

**Listing 5.3.** Implementierung einer View, die die Anzahl der Personen in den jeweiligen Ländern zurückliefern soll

Die Map-Funktion `mapPPL` in Zeile 27 wird auf jedes in der CouchDB befindliche Dokument angewendet und erzeugt für jedes Dokument ein Schlüssel-Wert-Paar; daraus ergibt sich folgende temporäre Tabelle:

| Schlüssel   | Wert |
|-------------|------|
| Deutschland | 1    |
| England     | 1    |
| Deutschland | 1    |
| ...         | ...  |

Auf diese temporäre Tabelle wird nun die Reduce-Funktion `reducePPL` angewendet. Die Option `group=True` bewirkt hierbei, dass die Reduce-Funktion immer auf alle Werte eines Schlüssel angewendet wird und die Ergebnisse schlüsselweise hinterlegt werden.

Die mit der `query`-Anweisung erzeugte View ist iterierbar. Der Inhalt kann beispielsweise über eine Listenkomprehension ausgegeben werden (wir gehen davon aus, dass insgesamt 8 Personendatensätze in die CouchDB `test` eingefügt wurden):

```
>>> [(erg.key, erg.value) for erg in viewPPL]
[('Deutschland', 4), ('England', 2), ('USA', 2)]
```

**Beispiel 2: Alle Personen eines Landes**

Wir wollen nun für jedes Land, das in den Personendatensätzen aufgeführt ist, ausgeben, welche Personen in diesem Land leb(t)en. Listing 5.4 zeigt, wie die entsprechende CouchDB-View `viewPPL` konstruiert werden kann.

#Code von Listing 5.2

```
mapPPL = ''' function(doc) { emit(doc.land, doc.name) }'''
reducePPL = ''' function(keys, vals) { return vals }'''
```

```
viewPNL = mydb.query(mapPNL, reducePNL, group=True)
```

**Listing 5.4.** Implementierung einer View, ausgibt, welche Personen sich in den jeweiligen Ländern befinden bzw. befanden

Mittels der folgenden Listenkomprehension können die Ergebnisse ausgegeben werden:

```
>>> [(erg.key, erg.value) for erg in viewPIL]
[('Deutschland', ['Tobias Haeberlein', 'Bernhard Riemann',
                  'Niklas Luhmann', 'David Hilbert']),
 ('England', ['Gregory Bateson', 'Alan Turing']),
 ('USA', ['Donald Knuth', 'Haskell Brooks Curry'])]
```

### Beispiel 3: Alle Personen, die jünger als 100 sind

Diese Abfrage kommt gänzlich ohne die Aggregierungsmöglichen einer Reduce-Funktion aus; wir müssen lediglich eine entsprechende Map-Funktion definieren, wie in Listing 5.5 gezeigt:

```
#Code von Listing 5.2
mapAge = ''' function(doc) { if (doc.age && doc.name && doc.age<100)
                          {emit(null, doc.name);}
                        }'''

dieJungen = mydb.query(mapAge)
```

**Listing 5.5.** Implementierung einer View, die die Namen aller „jungen“ Personen zurückliefert.

Diese View enthält also keine Schlüssel mehr (erstes Argument von emit ist null), sondern ausschließlich Werte, nämlich die Namen der gesuchten Personen. Die Abfrage 'if (doc.age && doc.name && doc.age<100)' stellt zum einen sicher, dass das jeweilige Dokument überhaupt Felder mit Namen 'age' und 'name' besitzt; zum anderen werden nur Werte „emit“tet, falls der Wert im Feld 'age' kleiner als 100 ist.

Mittels der folgenden Listenkomprehension können die Ergebnisse ausgegeben werden:

```
>>> [erg.value for erg in dieJungen]
['Alan Turing', 'Niklas Luhmann', 'Donald Knuth']
```

### Aufgabe 5.18

Erstellen Sie jeweils eine View, die ...

- (a) alle Personen auflistet, die älter als 100 sind und die in Deutschland leben bzw. gelebt haben.

- (b) das aufsummierte Alter aller Personen zurückliefert.
- (c) die jüngste Person zurückliefert.
- (d) das aufsummierte Alter aller in England lebenden Personen zurückliefert.