# Relationale Datenabfragen mit SQL

Gabriel Katz

23. Januar 2015

# Inhaltsverzeichnis


**1 Einleitung** **3**
- 1.1 Worum geht es hier? . . . 3
- 1.2 Relationale Datenbanken . . . 4
- 1.3 Kapiteltest . . . 7

**2 Relationale Datenbanken in SQL** **11**
- 2.1 Was ist SQL? . . . 11
- 2.2 Datentypen . . . 12
  - 2.2.1 Zahlwerte . . . 13
  - 2.2.2 Temporale Daten . . . 13
  - 2.2.3 Zeichenketten . . . 13
- 2.3 Datenbanken erstellen in SQL . . . 14
- 2.4 Daten einsetzen . . . 16
- 2.5 Kapiteltest . . . 17

**3 Datenbankabfragen** **18**
- 3.1 Aufbau einer SELECT-Abfragen . . . 18
- 3.2 Sortieren mit ORDER BY . . . 20
- 3.3 WHERE-Bedingungen . . . 21
  - 3.3.1 Textvergleich . . . 21
  - 3.3.2 Vergleich mit NULL . . . 22
- 3.4 Unterabfragen . . . 23
  - 3.4.1 Mengenabfragen . . . 23
  - 3.4.2 Unterabfragen . . . 23
- 3.5 Verschiedene JOINs . . . 24
  - 3.5.1 INNER JOIN . . . 24
  - 3.5.2 OUTER JOIN . . . 25
  - 3.5.3 JOIN mit Unterabfragen . . . 26
- 3.6 Aggregierung mit GROUP BY . . . 26
  - 3.6.1 GROUP BY mit JOINS . . . 27
  - 3.6.2 Filtern bei Gruppierungen . . . 28

3.7 Kapiteltest . . . . . 29

**4 Datenbankmanipulationen** **32**
4.1 Löschen . . . . . 32
4.1.1 Löschen mit JOINs . . . . . 33
4.2 Referenzielle Integrität . . . . . 33
4.3 Updates . . . . . 34
4.3.1 Updates mit JOINs . . . . . 35
4.4 Einfügen . . . . . 36
4.5 Kapiteltest . . . . . 36

**5 Lösungen** **38**
5.1 Kapitel 1 . . . . . 38
5.2 Kapitel 2 . . . . . 39
5.3 Kapitel 3 . . . . . 43
5.4 Kapitel 4 . . . . . 50

# Kapitel 1

# Einleitung

## 1.1 Worum geht es hier?

Eines der wichtigsten Zwecke von Computer ist die Speicherung von Daten. Wenn eine Menge von organisierten Informationen dauerhaft auf einem zentralen Computer gespeichert werden, und diese Daten leicht abgefragt, ergänzt, bearbeitet und gelöscht werden kann, ist von einer **Datenbank** die Rede. So ist zum Beispiel ein Notizblock (egal ob in Papierform oder in digitaler, wie zum Beispiel Notes oder OneNote) eine stark eingeschränkte Datenbank. Um relevante Daten zu finden, muss man eventuell Seite für Seite durchgehen. Beim Notizblock aus Papier ist auch die Überarbeitung der Daten eingeschränkt, da man z.B. auf der Seite nur begrenzt Platz hat. Daher stellt sich die Frage, welche besser zum Bearbeiten geeigneten Wege es gibt, Daten auf dem Computer zu speichern.

Die relationalen Datenbanken sind die wohl verbreitetste Art von Datenbanken. Was sie genau sind, werden wir im nächsten Abschnitt genauer betrachten. Zwar gibt es zur Zeit einen Trend zu anderen, sogenannten NoSQL-Datenbanken, doch auch für diese sind Kenntnisse in relationalen Datenbanken wichtig. Diese Unterlagen beschäftigen sich mit SQL, der Standardsprache in relationalen Datenbanken. Den Rest dieses Kapitels werden wir mit den relationalen Datenbanken und deren Aufbau beschäftigen. Kapitel 2 erklärt was SQL ist, und zeigt, wie man eine Datenbank in SQL erstellt. Kapitel 3 ist das Kernkapitel dieser Unterlagen und erklärt ausfüehrlich, wie man Daten in SQL abfragt. Schliesslich sehen wir in Kapitel 4, wie man die Anfragetechniken verwendet, um Daten in SQL zu manipulieren.

Wir arbeiten in dieser Unterrichtseinheit mit 3 verschiedenen Datenbankbeispielen. In den Kapitel selbst verwenden wir zwei minimale Datenbanken, welche nur 3 Tabellen beinhalten. Die Kapiteltests bearbeiten ein etwas grösseres Datenbankmodell.

SQL verfügt über sehr viele Funktionen, um Berechnungen mit den internen Datentypen durchzuführen. Beispielsweise gibt es eine Funktionsbibliothek, um Zeitdaten zu vergleichen und zu berechnen. Da diese Unterlagen sich nicht als SQL-Referenz versteht, werden solche Funktionen nur dann eingeführt, wenn sie verwendet werden.

## 1.2 Relationale Datenbanken

Das Konzept für relationale Datenbanken basiert auf einem Paper von Dr. E. F. Codd aus dem Jahre 1970 namens "A Relational Model of Data for Large Shared Data Banks". Dort schlug er vor, Daten in einer Menge von **Tabellen** darzustellen. Dies Tabellen haben einen Namen und mehrere **Spalten**. Eine Spalte einer Tabelle ist durch seinen Namen, seinen Datentyp, und ob sie obligatorisch oder fakultativ ist, definiert. Datentypen drücken aus, welche Art von Daten in der Spalte gespeichert werden kann. Beispielsweise hat eine Spalte, in welcher ein Datum gespeichert wird einen anderen Datentypen wie eine Spalte mit Zahl oder wie eine Spalte mit Text. Wir werden einige Datentypen, welche von SQL unterstützt werden im nächsten Kapitel etwas mehr im Detail betrachten.
Die Daten werden dann in den **Zeilen** der Tabelle gespeichert. Jede Zeile der Tabelle enthält mindestens in allen obligatorischen Spalten Daten. Wenn eine Zeile in einer Spalte keine Daten hat, sagt man auch, dass sie NULL enthält. Normalerweise ist die Anzahl der Spalten der Tabelle (des Datenschemas) überschaubar, wogegen die Anzahl der Zeilen riesig sein kann.
Eine Tabelle in einer relationalen Datenbank hat immer ein Merkmal, an welchem die Zeilen der Tabellen eindeutig identifiziert werden können. Dieses Merkmal wird **Primärschlüssel (PK)** genannt. Dieser Primärschlüssel besteht häufig aus einer einzelnen Tabellenspalte, doch er kann auch aus mehreren Tabellenspalten bestehen. In diesem Fall spricht man von einem **zusammengesetzten** Schlüssel. Keines der Tabellenspalten des Primärschlüssels darf fakultativ sein. Häufig ist der Primärschlüssel eine Spalte, welche nicht eigentliche Daten enthält, sondern nur ein Kennzeichen, zum Beispiel eine Zahl. In den hier gezeigten Beispielen ist der Primärschlüssel fast immer eine Zahl, doch dies muss nicht immer so sein.
Die Kernidee hinter den relationalen Datenbanken, ist, dass diese Tabellen verbunden sind. Doch wie sieht eine derartige Verbindung aus? Hier kommen die sogenannten **Fremdschlüssel (FK)** ins Spiel. Ein Fremdschlüssel besteht aus einer oder mehreren Spalten einer Tabelle, welche den Primärschlüssel einer anderen Tabelle referenzieren. Da zusammengesetzte Fremdschlüssel in der Praxis selten auftreten, werden wir diese in den Beispielen nicht behandeln.

Tabellen 1.1 bis 1.3 zeigen ein Beispiel für eine einfache Datenbank, wie

| CustomerId (PK) | FirstName | LastName | Email |
|---|---|---|---|
| 1 | Hans | Muster | hmuster@example.com |
| 2 | Anja | Tester | atester@example.com |
| 3 | Ferdinand | Meier | fmeier@example.com |

Tabelle 1.1: Customer

| ServiceId (PK) | Name | Duration |
|---|---|---|
| 1 | Waschen, Schneiden | 30:00 |
| 2 | Waschen, Schneiden, Fönen | 45:00 |
| 3 | Tönen | 90:00 |

Tabelle 1.2: Service

| BookingId (PK) | ServiceFk (FK(Service.ServiceId)) | CustomerFk (FK(Customer.CustomerId)) | StartDate |
|---|---|---|---|
| 1 | 1 | 3 | 21.11.2014 10:30 |
| 2 | 1 | 1 | 24.11.2014 08:30 |
| 3 | 3 | 3 | 23.11.2014 15:00 |

Tabelle 1.3: Booking

sie in einem Buchungssystem eines Friseursalons vorstellbar ist. In der Tabelle Customer sind die Kundeninformation aufgelistet. Ändert sich beispielsweise die Emailadresse eines Kunden, muss diese lediglich an einer Stelle, nämlich in der entsprechenden Spalte der Customer-Tabelle angepasst werden. Wir sehen hier, dass der Primärschlüssel, das Feld CustomerId, keine eigentlichen Information zum Benutzer enthält. Doch wieso wurde dieses Feld eingeführt? Betrachten wir die Alternativen für den Primärschlüssel: Wenn die Spalte Email Primärschlüssel wäre, entständen zwei Nachteile. Einerseits könnten die Kunden ihre Mailadresse nicht mehr wechseln, da sonst das eindeutige Merkmal der Kundenzeile verloren gehen würde. Andererseits können keine zwei Benutzer die gleiche Mailadresse teilen. Die Spalten FirstName und LastName sind nicht eindeutig, und können somit nicht als Primärschlüssel (auch zusammengenommen nicht) gewählt werden. Es gäbe noch die Möglichkeit, die Kombination FirstName, LastName und Email als Primärschlüssel zu wählen, doch wieder gäbe es das Problem, dass man die Daten nicht mehr ändern darf. Daher verzichtet man auf komplexe Primärschlüssel und führt lieber eine weitere Zeile ein.

Bei der Tabelle Service verhält es sich gleich wie bei der Customer-Tabelle: Da alle anderen Spalten sich ändern können, und eventuell nicht eindeutig sind, wurde eine zusätzliche Spalte für den Primärschlüssel eingeführt,

Die Booking-Tabelle hat zwei Fremdschlüssel: Die Spalte ServiceFk referenziert die Spalte ServiceId der Tabelle Service, und die Spalte CustomerFk refe-

| AlbumId (PK) | ArtistName | AlbumName | GenreFk (FK) |
|---|---|---|---|
| 1 | Red Hot Chilli Peppers | Californication | 1 |
| 2 | The Ramones | Rocket To Russia | 2 |
| 3 | The Beastie Boys | Ill Communication | 3 |

Tabelle 1.4: Album

| AlbumFk (PK, FK) | TrackId (PK) | Name | Duration |
|---|---|---|---|
| 3 | 1 | Sure Shot | 3:20 |
| 1 | 1 | Around the World | 3:59 |
| 2 | 2 | Rockaway Beach | 2:07 |
| 1 | 4 | Otherside | 4:15 |

Tabelle 1.5: Track

| GenreId(PK) | ParentGenreFk (FK) | Name |
|---|---|---|
| 1 | NULL | Rock |
| 2 | 1 | Punk Rock |
| 3 | NULL | Hip-Hop |

Tabelle 1.6: Genre

renziert die Spalte CustomerId der Tabelle Customer. Der erste Eintrag in der Booking-Tabelle bedeutet also, dass Ferdinand Meier (der Kunde mit der CustomerId 3) am 21. November 2014 um 10:30 einen 45-minutigen Termin zum Haare waschen und schneiden hat (der Service mit der ServiceId 1).

In diesem Beispiel enden Spalten, welche andere Tabellen referenzieren auf -Fk, und Primärschlüssel auf -Id. Es handelt sich hierbei um eine **Namenskonvention**, an welche in allen Beispielen dieser Unterlagen verwendet wird.

---

**Aufgabe 1.1** Was bedeutet es, wenn beim Eintrag mit der BookingId 2 der ServiceFK von 1 auf 2 geändert wird?

---

---

**Aufgabe 1.2** Wieso sollten die Spalten ServiceId und CustomerId keinen zusammengesetzten Primärschlüssel für die Tabelle Booking bilden?

---

Wir betrachten noch ein Beispiel für eine Datenbank. Tabellen 1.4 bis 1.6 sind Teil einer Musikbibliotheksdatenbank. In diesem Beispiel kommt in der Tabelle Track ein zusammengesetzter Primärschlüssel vor. Da der Track eines Albums

durch das Album und der Tracknummer eindeutig gegeben ist, und die Tracknummer und das Album sich nie mehr ändert, eignet sich diese Kombination hervorragend als zumsammengesetzten Primärschlüssel.

Ein weiteres häufiges Muster, dass bei relationalen Datenbanken vorkommt, ist bei der Genre-Tabelle zu finden. Wir sehen, dass die ParentGenreFk die eigene Tabelle referenziert. Im Beispiel ist Punk Rock ein Subgenre von Rock. Wir stellen ausserdem fest, dass gewisse Genres einen NULL-eintrag beim Feld ParentGenreFk haben. Dies ist die Notation, welche besagt, dass dieses Feld für einen Eintrag nicht verwendet wird. Eine Zeile kann nur in als fakultativ deklarierten Spalten NULL-einträge haben.

---

**Aufgabe 1.3** Könnte Einträge in der Spalte AlbumFk der Tabelle Track leer sein? Wieso bzw. wieso nicht?

---

Die zwei Datenbanken, welche wir in diesem Kapitel betrachteten, werden uns im auch in den nächsten Kapitel beschäftigen, und alle Übungen bis auf die Schlussprüfung wird sich mit diesen Beispielen befassen. Daher empfiehlt es sich, diese Tabellen beim Weiterlesen immer zur Seite zu haben.

## 1.3 Kapiteltest

In diesem Kapiteltest wird ein etwas Datenmodell eingeführt, welches uns im Rest dieses Kapitels begleiten wird. In dieser Datenbank werden Biere, deren Angebot in Bars, und deren Bestellungen von diversen Personen verwaltet. Tabellen 1.7 bis 1.13 zeigt die Daten dieses Systems. In der Tabelle Bar sind die Bars aufgelistet. Die Brauerei Tabelle listet die Brauereien im System, und die Kunden erscheinen in der Kunden Tabelle. Wenn eine Bar ein Bier im Sortiment hat, existiert ein entsprechenden Eintrag in der Sortiment-Tabelle mit dem entsprechenden Verkaufspreis. Da die Preisinformation in dieser Tabelle enthalten ist, können Biere nicht einfach aus dem Sortiment gelöscht werden, da sonst Informationen zu alten Bestellungen verloren gehen würden. Anstatt dessen wird der Wert in der Deleted-Spalte auf 1 gesetzt. Mehr dazu im Abschnitt 4.2. Eine Bestellung ist ein Eintrag in der Bestellungstabelle. Da jedoch die Bestellung mehrere verschiedene Biere beinhalten kann, sind diese in einer separaten Tabelle vermerkt. Jede Bestellung kann mehrere verschiedene Bestellte Biere haben. Diese Tabelle verfügt über eine Anzahl bestellter Biere, damit nicht jedes einzelne Bier separat aufgelistet werden muss.

---

**Aufgabe 1.4** Wir versuchen am Anfang, die Daten zu verstehen. Beschreibe in chronologischer Reihenfolge, welcher Kunde wann wo was bestellt hat.

| BarId (PK) | Name | Adresse |
|---|---|---|
| 1 | Vollmond Taverne | Zentralstrasse 53 |
| 2 | Si o No | Ankerstrasse 6 |
| 3 | Brasserie Federal | Bahnhofplatz 15 |

Tabelle 1.7: Bar

| BrauereiId (PK) | Name | Ort |
|---|---|---|
| 1 | Appenzeller | Appenzell |
| 2 | Wädi-Bräu | Wädenswil |
| 3 | Heineken | Amsterdam |
| 4 | Turbinenbräu | Zürich |

Tabelle 1.8: Brauerei

| BierId (PK) | BrauereiFk (FK(Brauerei.BrauereiId)) | Name | Deziliter | Einkaufpreis |
|---|---|---|---|---|
| 1 | 1 | Quöllfrisch | 5 | 1.30 |
| 2 | 1 | Vollmond | 3.3 | 1.30 |
| 3 | 1 | Brandlöscher | 3.3 | 1.20 |
| 4 | 2 | Blond Premium | 3.3 | 1.50 |
| 5 | 2 | Dunkel | 3.3 | 1.60 |
| 6 | 3 | Lager | 5 | 0.90 |
| 7 | 4 | Sprint | 3.3 | 1.80 |
| 8 | 4 | Start | 3.3 | 1.80 |
| 9 | 4 | Rekord | 3.3 | 1.80 |

Tabelle 1.9: Bier

| KundenId (PK) | Name | LieblingsbierFk (FK(Bier.BierId)) |
|---|---|---|
| 1 | Lisa | 6 |
| 2 | Urs | NULL |
| 3 | Heinz | 3 |
| 4 | Andrea | NULL |

Tabelle 1.10: Kunde

| SortimentId PK | BarFk (FK(Bar.BarId) | BierFk (FK(Bier.BierId)) | Preis | Deleted |
|---|---|---|---|---|
| 1 | 1 | 2 | 4.00 | 0 |
| 2 | 1 | 1 | 6.00 | 0 |
| 3 | 1 | 3 | 5.00 | 0 |
| 4 | 1 | 7 | 5.00 | 0 |
| 5 | 2 | 4 | 5.00 | 0 |
| 6 | 2 | 7 | 6.00 | 0 |
| 7 | 2 | 8 | 6.00 | 0 |
| 8 | 2 | 9 | 6.00 | 0 |
| 9 | 3 | 6 | 7.00 | 0 |
| 10 | 3 | 5 | 5.00 | 0 |
| 11 | 3 | 4 | 5.00 | 0 |
| 12 | 3 | 1 | 7.50 | 0 |
| 13 | 3 | 2 | 5.50 | 0 |
| 14 | 3 | 7 | 6.00 | 0 |
| 15 | 3 | 8 | 6.00 | 0 |

Tabelle 1.11: Sortiment

| BestellungId (PK) | KundenFk (FK(Kunde.KundenId)) | BarFk (FK(Bar.BarId)) | Zeit |
|---|---|---|---|
| 1 | 2 | 1 | 31.12.2014 22:00 |
| 2 | 1 | 1 | 31.12.2014 23:23 |
| 3 | 1 | 1 | 31.12.2014 23:24 |
| 4 | 3 | 2 | 31.12.2014 23:52 |
| 5 | 4 | 3 | 1.1.2015 01:21 |

Tabelle 1.12: Bestellung

| BestellungFk (PK, FK(Bestellung. BestellungId)) | SortimentFk (PK, FK(Sortiment. SortimentId)) | Anzahl |
|---|---|---|
| 1 | 3 | 2 |
| 2 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 4 | 1 |
| 4 | 6 | 3 |
| 4 | 5 | 1 |
| 5 | 12 | 3 |
| 5 | 9 | 1 |

Tabelle 1.13: BestellteBier

**Aufgabe 1.5** Zu wie viel mal dem Einkaufspreis wird das Vollmond-Bier in der Vollmond-Taverne ausgeschenkt?

**Aufgabe 1.6** Wie ändert sich die Datenbank, wenn Urs am 1. Januar 2015 um 2 Uhr im Federal noch zwei Sprint bestellt?

**Aufgabe 1.7** Die Tabelle Sortiment hat einen einfachen Primärschlüssel. Was würde passieren, wenn wir hier anstatt dessen einen zusammengesetzten Schlüssel aus BarFk und BierFk nehmen würden.

**Aufgabe 1.8** Könnte man in der Tabelle BestellteBier für den Primärschlüssel eine Id-Spalte einführen?

# Kapitel 2

# Relationale Datenbanken in SQL

In diesem Kapitel lernen wir endlich SQL etwas näher kennen. Ziel des Kapitels ist es, dass wir die Beispieltabellen in SQL erstellen können. Dafür untersuchen wir zuerst, was SQL genau ist, dann lernen wir die wichtigsten Datentypen von SQL kennen. Anschliessend sind wir bereit, das Datenmodell in SQL zu erstellen, in welches wir dan schliesslich die Daten einfügen.

## 2.1 Was ist SQL?

Dr. E. F. Codd, der wie im letzten Kapitel beschrieben die relationale Datenbanken erfunden hat, hat auch eine Sprache namens DSL/Alpha entwickelt, um Daten in relationellen Tabellen zu bearbeiten. Nach einigen Weiterentwicklungen durch Donald D. Chamberlin und Raymond F. Boyce bei IBM Research entstand dann die Sprache SEQUEL (kurz für Standard English Query Language), welche dann aus markenrechtlichen Gründen in SQL umbenannt werden musste. SQL ist seit 1986 von ANSI (American National Standard Institute) standardisiert, und dieser Standard wurde regelmässig aktualisiert, so dass heutzutage auch zum Beispiel die Integration und Abfrage von XML in SQL-Datenbanken standardisiert ist.

Im Gegensatz zu C, C#, Javascript oder Java ist SQL keine Allzwecksprache. Sie dient lediglich der Bearbeitung und Abfrage von Relationen Daten. Mit SQL werden lediglich notwendige Ein- und Ausgaben beschrieben. Wie die Daten gespeichert, bzw. wiedergegeben werden, ist Aufgabe der Datenbankengine. Diese optimiert die Ausführung der Datenbankabfrage, so dass die Daten möglichst effizient abgefragt werden.

Eine Software, welche sich umfänglich mit der Bewirtschaftung von relationalen Datenbanken befasst, nennt sich RDBMS, kurz für Relational Database Management System. Alle gängigen RDBMS verwenden SQL oder eine Erweiterung von SQL. Die gängigsten RDBMS-Systeme sind:

- MySQL, ein Open-Source-System, welches eher wenig Erweiterungen zu SQL verwendet
- SQL Server von Microsoft. Die SQL-Erweiterung, welche hier verwendet wird, nennt sich T-SQL.
- Oracle von der gleichnamigen Firma. Die Erweiterte Sprache, welche in Oracle verwendet wird, heisst PL/SQL.
- SQLite, ein sehr schlankes Open-Source-DBMS. SQLite ist so leicht, dass einige Webbrowser (Safari, Chrome und Opera) über ein eingebautes SQLite verfügen.
- PostgreSQL

SQL an sich ist keine vollständige Programmiersprache. Viele erweiterte Versionen von SQL, wie PL/SQL, MySQL oder T-SQL sind jedoch zumindest theoretisch vollständige Programmiersprachen. (Für die Informatikexperten: SQL alleine ist nicht Turing-komplett, MySQL, T-SQL, etc. hingegen schon.) Es ist jedoch nicht praktikabel, Datenbankunabhängige Anwendungen mit diesen Erweiterungen zu entwickeln. Viele Datenbankanwendungen werden aber nicht direkt mit SQL entwickelt, sondern mit einem Toolkit oder einer API aus einer Allzwecksprache heraus. Beispiele dafür sind JDBC für Java oder das Entity Framework für C# und Visual Basic.

Man kann SQL ohne irgendetwas zu installieren auf dem Web ausprobieren. Auf `http://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all` kann man die Webbrowser-interne Datenbankengine ansteuern. Um die Daten bearbeiten zu können, muss die Seite jedoch mit Chrome, Opera oder Safari aufgerufen werden, da die anderen Browser Web-SQL nicht unterstützen. Ansonsten kann man auf `http://sqlfiddle.com/` mit vielen verschiedenen Datenbankengines herumexperimentieren.

## 2.2 Datentypen

In SQL gibt es, ähnlich wie in Programmiersprachen, verschiedene Datentypen. Es gibt eine Vielzahl von komplexeren Datentypen, welche für spezifische Zwecke, wie zum Beispiel für Bilder und XML-Dateien existieren. Doch da wir uns vor allem für die Datenabfrage interessieren, richten wir unseren Fokus auf drei verschiedene Arten von Datentypen: Zahlwerte, temporale Daten und Zeichenketten.

### 2.2.1 Zahlwerte

Ein **BIT** kann nur entweder 0 (false) oder 1 (true) sein. Dieser Datentyp ist ideal für Antworten auf Ja/Nein-Fragen, welche nicht nur in der Informatik häufig auftreten. Ein **INT** ist eine Ganzzahl. Es gibt verschiedene Variationen davon für verschiedene grössen (**SMALLINT**, **BIGINT**, etc.), doch das Grundprinzip ist das Gleiche. In den vorhergehenden Beispielen wurden $INT$s vor allem für Schlüssel verwendet. Ein **FLOAT** ist eine Gleitkommazahl. Dies heisst, dass diese Zahl fast beliebig gross oder klein sein kann, und Stellen nach dem Komma aufweisen darf. Ein **DECIMAL(p,s)** ist eine Zahl mit $p$ Ziffern insgesamt, davon $s$ nach dem Komma. Dies heisst, dass z.B. $41.46$ als **DECIMAL(4,2)** gespeichert werden kann, jedoch wird $31.8734$ auf $31.87$ gerundet, und Werte über $99.99$ könnten gar nicht erst gespeichert werden. $DECIMAL$ wird häufig für finanzielle Daten (Kosten, Preise, etc.) verwendet.

### 2.2.2 Temporale Daten

**DATE** ist der Datentyp für ein Datum ohne Zeitangaben.
**TIME** ist der Datentyp für eine Zeit ohne Datumsangaben.
**DATETIME** ist der Datentyp für eine Kombination der beiden obigen.

Leider gibt es für eine Zeitdauer keinen einheitlichen Datentyp. Eine möglichkeit ist es, daür **TIME** zu nehmen. Doch dafür muss die Zeitdauer kleiner als 24 Stunden sein. Anstatt dessen kann man sonst einen Zahlwert nehmen, und die Zahleneinheit am Besten gleich im Spaltennamen der Tabelle erwähnen, damit er nicht vergessen wird. Bei der Tabelle 1.5 zum Beispiel kann man die Spalte $Duration$ in $DurationSeconds$ umbenennen, und die Werte dann z.B. als **INT**-Werte speichern (falls einem die Sekundenbruchteile nicht interessieren).

### 2.2.3 Zeichenketten

Für Zeichenketten gibt es zwei verschiedene Formate: **CHAR(l)** ist eine Zeichenkette mit fixer Länge $l$.
**VARCHAR(l)** ist eine Zeichenkette mit höchstlänge $l$.
Wenn man bei einem Feld die Höchstlänge nicht kennt, kann man auch für $l$ anstatt eines Zahlenwertes den Ausdruck $max$ nehmen. Dies kann jedoch bei Textsuchen die Abfrageeffizienz beeinträchtzigen.

---

**Aufgabe 2.1** An sich könnte man jede Zeichenkette in ein **VARCHAR(max)** schreiben. Wieso verwendet man dann nicht immer **VARCHAR(max)**?

---

**Aufgabe 2.2** Bestimme die Datentypen des Buchungsdatenbank aus dem vorhergehenden Kapitel.

**Aufgabe 2.3** Bestimme die Datentypen des Musikdatenbank aus dem vorhergehenden Kapitel.

## 2.3 Datenbanken erstellen in SQL

Mit dem Wissen, das wir uns in den letzten Kapitel angeeignet haben, ist es relativ einfach, eine Datenbank mit SQL zu erstellen. Folgendes Beispiel erstellt die Tabelle Booking von 1.3.

Listing 2.1: Erstellen der Tabelle Booking

```
1  CREATE TABLE Booking (
2          BookingId INT NOT NULL,
3          ServiceFk INT NOT NULL,
4          CustomerFk INT,
5          StartDate DATETIME NOT NULL,
6          CONSTRAINT pk_booking PRIMARY KEY (BookingId),
7          CONSTRAINT fk_service FOREIGN KEY (ServiceFk)
8                  REFERENCES Service(ServiceId),
9          CONSTRAINT fk_customer FOREIGN KEY
10                 (CustomerFk)
11                 REFERENCES Customer(CustomerKey)
12
13 );
```

Da dies der erste SQL-Code in diesen Unterlagen ist, zuerst einige Bemerkungen: Wie in vielen Programmiersprachen dürfen bei Leerschlßagen beliebig viele weitere Leerschläge oder Zeilenumbrüche geschrieben werden. Man könnte auch diesen Code auf eine Zeile packen, oder jedes Wort dieses codes auf eine separate Zeile legen. In diesem Buch ist der Code mit etwas mehr Zeilenumbrüchen versehen als sonst in der Praxis üblich, damit die Lesbarkeit auf Papier einfacher wird. In den Beispielen werden Zeilen, welche sich auf die Vorzeile beziehen eingerückt, um eine Übersicht über den Code zu schaffen. Dies ist eine gängige Praxis in allen Programmiersprachen.

Wir sehen in diesem Codebeispiel, dass gewisse Worte in Grossbuchstaben geschrieben sind. Das sind die SQL-Schlüsselwörter. SQL unterscheidet eigentlich gar nicht zwischen Klein-und Grossbuchstaben. SQL ist also **case-insensitive** (Es gibt jedoch Ausnahmen, wie zum Beispiel der Textvergleich von Datensätzen). Es hat sich jedoch eingebürgert SQL-Schlüsselwörter in Grossbuchstaben zu schreiben. Dem wird auch in diesen Unterlagen Folge geleistet.

Die erste Zeile beschreibt, was wir tun wollen, nämlich eine Tabelle erstellen. Die folgenden Felder sind die Spalten der Tabelle. Zuerst wird jeweils der Name der Spalte geschrieben, dann der Datentyp. Wenn eine Spalte nicht leer sein darf, dann wird dies mit NOT NULL gekennzeichnet. In unserem Beispiel dürfen auch Buchungen getätigt werden, bei welchen der Kunde fehlt, da er zum Beispiel nicht bekannt ist. Primärschlüssel und Fremdschlüssel werden mit dem CONSTRAINT-Keyword erstellt. Solche Constraints erhalten einen Namen, damit man sie später auch noch modifizieren oder löschen kann. Die Namen pk_booking, fk_service und fk_customer stammen aus Konventionen, wenn man möchte, kann man Constraints beliebig nennen. Auf den Namen folgt die Art des Constraints. In dieser Lektion werden nur die Contraints “Primary Key” und “Secondary Key” behandelt. Nach der Art des schlüssels wird deklariert, auf welche Spalten der Constraint wirkt. Alle Constraints, welche wir hier eignefügt haben, gelten nur für eine einzelne Spalte. Zeilen 7 bis 12 beschäftigen sich mit den Fremdschlüssel. Zusätzlich zur Information, welche Spalte teil des Schlüssels ist, wird auch noch beschrieben, welche Spalten welcher Tabelle referenziert werden.

Hier noch ein Beispiel, wie die Tabelle 1.5 in SQL aussieht:

Listing 2.2: Erstellen der Tabelle Track

```
1  CREATE TABLE Track (
2          AlbumFk INT NOT NULL,
3          TrackId INT NOT NULL,
4          Name NVARCHAR(200),
5          DurationSeconds INT NOT NULL,
6          CONSTRAINT pk_track PRIMARY KEY
7                  (AlbumFk, TrackId)
8          CONSTRAINT fk_album FOREIGN KEY (AlbumFk)
9                  REFERENCES Album(AlbumKey)
10 );
```

**Aufgabe 2.4** Erstelle die zwei anderen Tabellen dieser Datenbank in SQL.

**Aufgabe 2.5** Erstelle die restlichen zwei Tabellen der Buchungsdatenbank in SQL.

## 2.4 Daten einsetzen

Eine Datenbank abzufragen macht erst Sinn, wenn Daten drin sind. Dies geschieht mit dem INSERT INTO-Statement. Dieses Statement besteht aus zwei Teilen. Im ersten Teil wird angegeben, in welche Spalten welcher Tabelle Werte eingesetzt werden. Im zweiten-Teil, nach dem VALUES-Keyword, werden die Werte eigesetzt. Das Ganze sieht zum Beispiel für die Booking-Tabelle so aus:

Listing 2.3: Einfügen eines Datensatzes

```
1 INSERT INTO Booking
2         (BookingKey, ServiceFk, CustomerFk, StartDate)
3 VALUES (4, 1, 3, '2014-11-21 10:30')
```

In diesem Beispiel sehen wir auch, wie wir in SQL mit Datentypen umgehen müssen: Zahlenwerte schreiben wir ganz gewöhnlich aus. Alle anderen Datentypen, also Zeitdaten und Zeichenketten schreiben wir mit Hochkommas. Zeitdaten werden normalerweise im Format yyyy-MM-dd HH:mm(:ss), also zuerst Jahr, dann Monat, anschliessend Tag geschrieben.

Wir können auch gleich mehrere Datensätze in einem Befehl in eine Tabelle einfügen. Dafür muss man die verschiedenen Datensätze mit Komma voneinander trennen:

Listing 2.4: Einfügen mehrerer Datensätze

```
1 INSERT INTO Booking
2         (BookingKey, ServiceFk, CustomerFk, StartDate)
3 VALUES (4, 1, 3, '2014-11-21 10:30'),
4        (5, 2, 2, '2014-11-26 14:45')
```

In der Praxis ist es häufig der Fall, dass Spalten wie BookingKey automatisch generiert werden. Wie man in der Datenbank spezifiziert, dass ein Feld automatisch generiert werden soll, variiert von System zu System. Daher gehen wir hier nicht näher darauf ein. Wenn eine Spalte freiwillig ist, oder automatisch generiert wird, dann muss man die Spalte im INSERT-Statement nicht angeben. Wenn wir also wissen, dass der BookingKey automatisch generiert wird, reicht auch schon folgendes zum Einfügen der beiden Datensätze:

Listing 2.5: Einfügen mit automatisch generiertem Feld

```
1 INSERT INTO Booking
2         (ServiceFk, CustomerFk, StartDate)
3 VALUES (1, 3, '2014-11-21 10:30'),
4        (2, 2, '2014-11-26 14:45')
```

Wenn man Daten in SQL-Tabellen einfügt, müssen die Referenzen existieren. Wenn wir versuchen würden, eine Buchung mit CustomerFk 4 einzufügen, würden wir einen Fehler zurückerhalten, selbst wenn wir in der nächten Zeile den Kunden mit der CustomerId 4 erstellen würden. Die Erstellungsreihenfolge ist also wichtig.

---

**Aufgabe 2.6** Erstelle sie die Beispielsdatensätze der Tracks-Datenbank mittels SQL-Skript.

---

## 2.5 Kapiteltest

Wir arbeiten wieder mit der Datenbank aus Tabellen 1.7 bis 1.13.

---

**Aufgabe 2.7** Erstelle die Datenbanktabellen Bestellung und BestellteBier mittels SQL.

---

---

**Aufgabe 2.8** Urs hat am 1. Januar 2015 um 2 Uhr im Federal noch zwei Sprint bestellt. Bilde dies mittels Skript in die Datenbank ab.

---

# Kapitel 3

# Datenbankabfragen

Jetzt dass wir wissen, wie wir unsere Datenbanken erstellen und Daten einügen, können wir anfangen, Daten abzufragen. Wir beginnen damit, Abfragen auf eine einzelne Tabelle abzusetzen. Zuerst betrachten wir den Aufbau einer einfachen Abfrage. Anschliessend lernen wir, wie wir unser Suchresultat ordnen können, und wie wir Daten filtern können, um uns nur auf relevante Informationen zu beschränken. Dann lernen wir zwei Methoden kennen, um Daten aus mehreren Tabellen miteinander zu verknüpfen: Unterabfragen und JOINs. Schliesslich lernen wir, wie wir mehrere Tabellenzeilen in ein Resultatfeld zusammen gruppieren können.

## 3.1 Aufbau einer SELECT-Abfragen

Um Daten abzufragen, verwendet man das SELECT-Keyword. Die einfachste Datenabfrage sieht wie folgt aus:

Listing 3.1: Einfache SQL-Abfrage

```
1 SELECT *
2 FROM Track
```

Eine SQL-Abfrage in der Datenbank gibt immer eine Tabelle zurück. Diese Datenbankabfrage gibt uns die ganze Tabelle Track zurück. Nach dem SELECT-Keyword muss man eingeben, welche Daten man im Resultat erhalten möchte. * steht für sämtliche Spalten. Nach dem FROM steht, von welcher Tabelle man die Felder nehmen will. Wenn wir zum Beispiel nur die Spalten Name und Duration bräuchten, sähe die Abfrage wie folgt aus:

Listing 3.2: Spezifische Spalten abfragen

```
1 SELECT Name, DurationSeconds
```

```
2 FROM Track
```

Da ein wichtiger Zweck von SQL ist, Daten über mehrere Tabellen abzufragen, ist es oft praktisch, Tabellen in den Abfragen einen neuen Namen zu geben. Dann kann es auch nötig sein, Spalten zusammenzufassen, denn wenn mehrere Tabellen zusammengefasst werden, kann es vorkommen, dass gewisse Spalten gleich heissen. Dieses Beispiel macht das gleiche wie das letzte, die Tabelle Track wird jedoch t genannt, und die Track-Spalte wird zu Trackname umbenannt.

Listing 3.3: Tabelle und Spalten benennen

```
1 SELECT t.Name AS Trackname, t.DurationSeconds
2 FROM Track t
```

Bei SELECT-Abfragen können nicht nur Spalten aus der Tabelle, sondern auch Konstanten oder berechnete Werte abgefragt werden. So gibt zum Beispiel diese Abfrage eine Tabelle zurück, bei welcher alle Einträge den Status “In Library” haben. Weiter sind in der Tabelle die Spieldauer in Minuten und der Track-Name in Kleinbuchstaben aufgelistet. Um den Track-Namen in Kleinbuchstaben zu erhalten, wird die SQL-Funktion LOWER() verwendet.

Listing 3.4: Berechnete Spalten

```
1 SELECT 'In Library' AS Status,
2        LOWER(t.Name) AS Trackname,
3        t.DurationSeconds/60 AS DurationMinutes
4 FROM Track t
```

Weiter gibt es noch das Schlüsselwort DISTINCT zu erwähnen. mit diesem Schlüsselwort werden doppelte Einträge eliminiert. Beispielsweise gibt uns diese Abfrage zurück, wieviel Minuten die Lieder in der Tabelle dauern.

Listing 3.5: DISTINCT

```
1 SELECT DISTINCT t.DurationSeconds/60 AS DurationMinutes
2 FROM Track t
```

---

**Aufgabe 3.1** Finde eine Datenabfrage über die bestehenden Tabellen, welche die Werte aus Tabelle 3.1 zurückgibt

---

| NameLower | DurationsMilliseconds |
|---|---|
| waschen, schneiden | 1800000 |
| waschen, schneiden, fönen | 2700000 |
| tönen | 5400000 |

Tabelle 3.1: Erwünschtes Resultat aus Aufgabe 3.1

## 3.2 Sortieren mit ORDER BY

Grundsätzlich ist das Resultat einer SQL-Abfrage ungeordnet. Zwar werden in Praxis die Resultate in den meisten Datenbank-Engines nach Primärschlüssel geordnet, doch davon kann man nicht immer ausgehen. Daher existiert die ORDER BY-Klausel, mit welcher man bestimmen kann, nach welchen Spalten das Resultat geordnet wird. Diese Spalten müssen dabei gar nicht unbedingt im Resultat vorkommen. Die Datenbankengine sortiert zuerst nach der ersten Spalte, dann nach der zweiten, etc. Wenn nicht anders angegeben, wird aufsteigend sortiert. Mit dem DESC-Schlüsselwort nach dem Spaltennamen kann man absteigend sortieren. Das folgende Beispiel liefert das Resultat aus Tabelle 3.2.

Listing 3.6: Order By

```
1 SELECT *
2 FROM Track t
3 ORDER BY AlbumFk DESC, Name
```

| AlbumFk (PK, FK) | TrackId (PK) | Name | Duration |
|---|---|---|---|
| 3 | 1 | Sure Shot | 3:20 |
| 2 | 2 | Rockaway Beach | 2:07 |
| 1 | 1 | Around the World | 3:59 |
| 1 | 4 | Otherside | 4:15 |

Tabelle 3.2: Tracks nach Album und Namen sortiert

---

**Aufgabe 3.2** Sortiere die Buchungen von Buchungen absteigend nach Termin.

---

---

**Aufgabe 3.3** Sortiere die Alben nach Genre, und dann absteigend nach Künstlername.

---

## 3.3 WHERE-Bedingungen

Wie wir bereits erwähnt haben, können Tabellen in SQL riesig werden. Um den Überblick zu behalten, möchte man meistens nur einen Teil der Resultate zurückbekommen. Um dies zu erreichen, kann man die Daten filtern. Der einfachste Weg, die Daten zu beschränken, ist es, nur die obersten n Datensätze zu holen. Leider ist diese Klausel nicht standardisiert, und der Syntax ist in allen gängigen Systemen anders (TOP ins T-SQL, LIMIT in MySQL und noch mal anders in Oracle). Daher verzichten wir hier auf eine Einführung dieser Klausel.

Um gezielt nach Bedingungen zu filtern, braucht man in allen Versionen von SQL die WHERE-Klausel. In folgendem Beispiel sucht man nach Lieder, welche länger als 4 Minuten dauern:

Listing 3.7: Einfacher WHERE-Filter

```
1 SELECT *
2 FROM Track t
3 WHERE DurationSeconds > 4*60
```

Wenn man mehrere Bedingungen zusammen verbinden möchte, tut man dies entweder mit OR oder mit AND. Mit diesen zwei Verbindungsmöglichkeiten, dem NOT-Ausdruck und entsprechender Klammerung kann man beliebig komplexe logische Verbindungen zwischen Bedingungen bauen. Da Auslagenlogik jedoch nicht Teil dieses Kurses ist, betrachten wir ein ganz einfaches Beispiel, in welchem wir nach allen Tracks, welche nicht am Anfang des Albums sind, und länger als 4 Minuten dauern, suchen:

Listing 3.8: Mehrere Bedingungen mit AND

```
1 SELECT *
2 FROM Track t
3 WHERE DurationSeconds > 4*60 AND TrackId <> 1
```

---

**Aufgabe 3.4** Frage alle Termine in der Woche vom 24. bis 30. November 2014 ab. Bemerkung: Daten kann man gleich wie Zahlen mit den Symbolen kleiner als (¡) und grösser als (¿) vergleichen.

---

### 3.3.1 Textvergleich

Bei Textfelder gibt es die Möglichkeit, nach Teilstrings zu filtern. Mit dem Schlüsselwort LIKE kann man nach Anfang, Ende oder innerer Teil filtern. Das %-Zeichen

dient dabei als Wildcard. Kommt es nur nach dem Suchtext vor ('Text%'), so wird der Text nach dem Anfang gefiltert, kommt es nur vor dem Suchtext vor, so wird nach dem Ende gefiltert ('%Text'). Wenn das %-Zeichen auf beiden Seiten des Suchtexts steht, so darf der Suchtext an einer beliebigen Stelle vorkommen. In diesem Beispiel filtern wir nach Lieder, deren Name mit "Sure" anfangen

Listing 3.9: Textvergleich mit LIKE

```
1 SELECT *
2 FROM Track t
3 WHERE Name LIKE 'Sure%'
```

### 3.3.2 Vergleich mit NULL

Wenn ein Feld leer sein darf, muss man mit dem Filtern aufpassen: Felder, welche NULL sind, können nicht verglichen werden, und werden deshalb nicht retourniert. So ist zum Beispiel das Resultat folgender Abfrage leer:

Listing 3.10: Vergleich gibt keine NULL-Werte zurück

```
1 SELECT *
2 FROM Genre
3 WHERE ParentGenreFk <> 1
```

Wir wollten eigentlich nach allen Genres suchen, welche nicht Subgenres von Rock sind. Doch leider wird das NULL nicht verglichen, bzw. scheitert der Vergleich. Man ist geneigt, das Problem folgendermassen zu beheben, doch dies nützt nichts:

Listing 3.11: Wirkungsloser NULL-Vergleich

```
1 SELECT *
2 FROM Genre
3 WHERE ParentGenreFk <> 1 OR ParentGenreFk = NULL
```

Das Problem ist nach wie vor, dass der Vergleich mit NULL nicht funktioniert. Wenn man mit einem NULL-Wert vergleicht, muss man immer IS NULL verwenden, wie folgt:

Listing 3.12: Funktionierender NULL-Vergleich

```
1 SELECT *
2 FROM Genre
3 WHERE ParentGenreFk <> 1 OR ParentGenreFk IS NULL
```

## 3.4 Unterabfragen

### 3.4.1 Mengenabfragen

Anhand unserer jetztigen Kenntnisse, würden wir nach den Kunden, die entweder Tester, Muster oder Test heissen, wie folgt suchen:

Listing 3.13: Vergleich mit mehreren Werten

```
1 SELECT *
2 FROM Customer
3 WHERE LastName = 'Muster' OR LastName = 'Tester'
4         OR LastName = 'Test'
```

Dies funktioniert bei nur drei Vergleichen relativ gut. Mit mehr als ca. 10 Namen wird diese Notation jedoch mühsam. Daher kann man hier das Keyword IN verwenden, um die Abfrage kürzer und lesbarer zu machen:

Listing 3.14: Mengenabfrage

```
1 SELECT *
2 FROM Customer
3 WHERE LastName IN ('Muster', 'Tester', 'Test')
```

### 3.4.2 Unterabfragen

Eine einfache Art, Tabellen miteinander zu verbinden, sind Unterabfragen. Unterabfragen sind den Mengenabfragen aus dem vorhergehenden Abschnitt ähnlich. Lediglich ist die Vergleichsmenge nicht explizit aufgeschrieben, sondern ebenfalls das Resultat einer anderen Abfrage. Das folgende Beispiel gibt alle Lieder von Bands, welche mit “The” anfangen, zurück. Dies wird zweistufig gemacht: In der Unterabfrage werden alle Ids der Alben, deren Künstler mit “The” anfangen abgefragt. Danach wird die AlbumFk-Spalte mit diese Ids verglichen:

Listing 3.15: Unterabfrage

```
1 SELECT *
2 FROM Tracks
3 WHERE AlbumFk IN (
4         SELECT AlbumId
5         FROM Album
6         WHERE ArtistName LIKE 'The%'
7 )
```

| FirstName | LastName | StartDate |
|---|---|---|
| Ferdinand | Meier | 21.11.2014 10:30 |
| Hans | Muster | 24.11.2014 08:30 |
| Ferdinand | Meier | 23.11.2014 15:00 |

Tabelle 3.3: JOIN über Customer und Booking

---

**Aufgabe 3.5** Erstelle eine Abfrage, welche alle Buchungen zurückgibt, deren Service länger als 30 Minuten dauert.

---

## 3.5 Verschiedene JOINs

### 3.5.1 INNER JOIN

Bisher haben alle unsere Datenbankabfragen nur Werte aus einer Tabelle zurückgegeben. Doch wirklich mächtig wird SQL erst, wenn wir Tabellen zusammenfügen. Dies geschieht zum Beispiel mit INNER JOIN. Die INNER JOIN-Klausel kommt im FROM-Teil der Abfrage zum Einsatz, und fügt jeweils zwei Tabellen zusammen. Zu einem JOIN-Befehl gehört immer auch ein ON-Befehl, welcher beschreibt, unter welcher Bedingung zwei Datensätze zusammengefügt werden. Zum Beispiel erstellen wir so für den Frisör eine Ansicht, bei welchem er den Namen des Kunden und der Zeitpunkt des Termins sieht:

Listing 3.16: Einfacher INNER JOIN

```
1 SELECT c.FirstName, c.LastName, b.StartDate
2 FROM Customer c INNER JOIN Booking b
3         ON b.CustomerFK = c.CustomerId
```

Das Resultat dieser Abfrage ist in Tabelle zu sehen. Dabei können Einträge von gewissen Tabellen mehrfach auftreten. In unserem Beispiel taucht Ferdinand Meier in zwei Einträgen auf. Beim INNER JOIN spielt es, im Gegensatz zu anderen JOINS, welche wir später sehen werden, keine Rolle, welche Tabelle links und welche rechts des JOINs steht. So können wir auch mehrere INNER JOINS aneinanderreihen, um beliebig viele Tabellen zusammenfügen. Dafür hier ein Beispiel, bei welchem in der Musikdatenbank sämtliche Informationen zu den Tracks zusammengetragen werden.

Listing 3.17: Mehrere INNER JOINs

```
1 SELECT t.Name AS TrackName, a.ArtistName,
2         a.AlbumName, g.GenreName
3 FROM Track t
4         INNER JOIN Album a ON t.AlbumFk = a.AlbumId
5         INNER JOIN Genre g ON a.GenreFk = g.GenreId
```

**Aufgabe 3.6** Erweitere Listing 3.16 um den Namen des Services und dem Enddatum. Um das Enddatum zu berechnet, kannst du auf die eingebaute SQL-Funktion DATEADD(minute, x, StartDate) zurückgreifen. So erhältst Du das Datum, welches x Minuten nach dem Startdatum ist.

**Aufgabe 3.7** Verbinde die Genre-Tabelle mit sich selbst, so dass in der ersten Spalte das Untergenre und in der zweiten Spalte das Genre steht.

### 3.5.2 OUTER JOIN

Wie der aufmerksame Leser vielleicht schon bemerkt hat, sind bei einem JOIN nicht alle Einträge der verknüpften Tabellen vorhanden. Zum Beispiel taucht in der Tabelle 3.3 Anja Tester nicht mehr auf. Es kann sein, dass man sämtliche Einträge einer Tabelle erhalten möchte. Dafür ist der OUTER JOIN. Ein LEFT, bzw. ein RIGHT OUTER JOIN stellt sicher, dass jeder Eintrag der Tabelle links, bzw. rechts des Schlüsselworts mindestens einmal auftritt. Ein FULL OUTER JOIN stellt sicher, dass jeder Eintrag beider Tabellen vorkommt. Wenn man bei der Abfrage zu Tabelle 3.3 das INNER JOIN durch ein LEFT OUTER JOIN ersetzen würde, so erhälte man Tabelle 3.4.

| FirstName | LastName | StartDate |
|---|---|---|
| Ferdinand | Meier | 21.11.2014 10:30 |
| Hans | Muster | 24.11.2014 08:30 |
| Ferdinand | Meier | 23.11.2014 15:00 |
| Anja | Tester | NULL |

Tabelle 3.4: LEFT OUTER JOIN

**Aufgabe 3.8** Führe von Hand die vier JOINs (INNER / LEFT OUTER / RIGHT OUTER / FULL OUTER)-Operationen auf die Tabelle Genre und sich selbst (wie in Aufgabe 3.7) aus.

### 3.5.3 JOIN mit Unterabfragen

Joins sind nicht nur mit bereits existierenden Tabellen möglich. Wir können JOINs auch mit berechneten Tabellen ausführen. Hier ein Beispiel:

Listing 3.18: JOIN mit berechneter Tabelle

```
1 SELECT *
2 FROM Album a INNER JOIN
3   (SELECT AlbumFk
4     'In Library' AS Status,
5     LOWER(t.Name) AS Trackname,
6     t.DurationSeconds/60 AS DurationMinutes
7   FROM Track t) t2
8   ON t2.AlbumFk = a.AlbumKey
```

Wenn man mit vielen Unterabfragen arbeitet, kann man sehr schnell die Übersicht über die verschiedenen Tabellennamen, bzw. Spaltennamen zu verlieren. Daher empfiehlt es sich dann, selbsterklärende Namen einzuführen.

## 3.6 Aggregierung mit GROUP BY

Alle Datenbankabfragen, welche wir bisher abgesetzt haben, lieferten Rohdaten zurück. Manchmal reicht es jedoch, Statistiken über die Daten zu sehen. Beispielsweise möchte man nur die Anzahl Buchungen pro Service sehen, oder man möchte nur die Gesamtlange der Alben wissen. Um dies zu tun, gruppiert man die Einträge nach gewissen Spalten mit dem GROUP BY-Befehl. Die Datenbankengine fasst dann sämtliche Zeilen, welche die gleichen Gruppierungsfelder haben, in eine Zeile zusammen. Dabei werden Spalten, nach welchen nicht gruppiert werden, mit einer Aggregationsfunktion zusammengetragen. Jede Spalte des Resultats muss bei einer GROUP BY-Abfrage entweder ein Gruppierungsfeld oder eine Aggregationsfunktion sein. Mögliche Aggregationsfunktionen sind:

- COUNT(*), gibt die Anzahl zusammengefassten Datensätze zurück
- MAX(field) und MIN(field), retourniert den maximalen, bzw. minimalen Wert einer Spalte
- AVG(field), retourniert den Durchschnitt einer Spalte

So sieht die Abfrage für die Anzahl Buchungen pro Service aus:

Listing 3.19: Zählen der Buchungen pro Service

```
1 SELECT ServiceFk, COUNT(*)
2 FROM Booking
3 GROUP BY ServiceFk
```

**Aufgabe 3.9** Erstelle eine Abfrage, welche in der Tabelle Tracks die AlbumFk und die Gesamtlänge des Albums auflistet.

### 3.6.1 GROUP BY mit JOINS

Die letzte Abfrage wäre viel schöner, wenn wir gleich den Servicenamen dazu nehmen würden. Dies können wir mit JOIN zusammenfügen. Die Abfrage sieht etwa so aus:

Listing 3.20: Versuch, GROUP BY mit JOIN zu kombinieren.

```
1 SELECT ?, COUNT(*)
2 FROM Booking b INNER JOIN Service s
3         ON s.ServiceKey = b. ServiceFk
4 GROUP BY b.ServiceFk
```

Jetzt stellt sich die Frage, was beim Fragezeichen stehen sollte. Da s.Name noch kein Gruppierungsfeld ist, müsste es aggregiert werden. Es gibt hier 3 Möglichkeiten, weiterzufahren. Alle Möglichkeiten führen zum Ziel, doch die letzte ist zu bevorzugen.

1. Wir verwenden einfach eine Aggregierung, welche etwas Sinnvolles zurückliefert, wie zum Beispiel MAX(s.Name). Dies funktioniert, da Zeichenketten in SQL auch geordnet werden können, und das Maximum von gleichen Werten den gleichen Wert hat. Dennoch ist diese Lösung eher ein Hack, der für Nichteingeweihte schwer verständlich ist.

2. Wir gruppieren einfach auch nach s.Name. Auch diese Lösung funktioniert einwandfrei. Da alle Einträge mit dem gleichen ServiceFk den gleichen Servicenamen haben, macht diese zusätzliche Gruppierung nichts Sichtbares. Diese Lösung macht jedoch in den meisten Datenbankengines die Abfrage ineffizient, und ist auch unschön, da nach so wenig Felder wie möglich gruppiert werden soll.

3. Wir gruppieren in einer Unterabfrage. In einer Unterabfrage erstellen wir eine Zwischentabelle bookingByService, welche die ServiceFks und die Anzahl als Spalten hat, dann JOINen wir diese Tabelle mit der Booking Tabelle. Dies sieht zwar auf den ersten Blick komplizierter aus, ist jedoch viel deutlicher. Die Abfrage sieht dann so aus:

Listing 3.21: JOIN mit gruppierter Tabelle

```
1 SELECT s.Name, BookingCount
2 FROM Service s INNER JOIN
3   (SELECT ServiceFk, COUNT(*) AS BookingCount
4   FROM Booking
5   GROUP BY ServiceFk) bookingByService
6   ON bookingByService.ServiceFk = s.ServiceKey
```

Eine kleine Anmerkung: Diese Technik muss nur dann angewendet werden, wenn die Aggregation eigentlich nur über eine Tabelle stattfindet. Es gibt Fälle, in welchen Daten aus mehreren Tabellen zusammengerechnet und dann aggregiert werden. Dann braucht man diese Technik nicht.

---

**Aufgabe 3.10** Erweitere die Abfrage aus Aufgabe 3.9, so dass der Künstlername und der Albumname neben der Gesamtspieldauer des Albums steht.

---

### 3.6.2 Filtern bei Gruppierungen

Bei einer Gruppierungsabfrage kann man wie gewohnt mit einer WHERE-Klausel über die Datensätze filtern. Diese Filterung geschieht dann vor der Aggregation. Zum Beispiel kann man wie folgt nur die Buchungen ab dem 24. November für die Statistik berücksichtigen:

Listing 3.22: Filtern vor Gruppierung

```
1 SELECT ServiceFk, COUNT(*)
2 FROM Booking
3 WHERE StartDate > '2014-11-24'
4 GROUP BY ServiceFk
```

Es gibt jedoch auch die Möglichkeit, über Aggregationen zu filtern. Dies geschieht mit der HAVING-Klausel. Diese zusätzliche Filterung steht nach der GROUP BY-Klausel. In diesem Beispiel suchen wir alle Services, welche ab dem 24. November genau ein mal gebucht wurden:

Listing 3.23: Filtern vor und nach Gruppierung

```
1 SELECT ServiceFk, COUNT(*)
2 FROM Booking
3 WHERE StartDate > '2014-11-24'
4 GROUP BY ServiceFk
5 HAVING COUNT(*) = 1
```

Die Aggregation, welche in der HAVING-Klausel erwähnt wird, muss nicht zwingend die gleiche sein wie die im Resultat, wie sie in der nächsten Übung sehen werden

---

**Aufgabe 3.11** Erstelle eine Abfrage, welche die Gesamtdauer der Alben, bei welchen die Durchschnittsdauer der Tracks weniger als 3 Minuten beträgt, berechnet.

---

---

**Aufgabe 3.12** Gebe bei den folgenden Bedingungen an, ob eine WHERE oder eine HAVING-Klausel gebraucht wird. Die Abfrage muss nicht erstellt werden.

- Finde die Anzahl Buchungen aller Kunden, welche schon mindestens zwei Buchungen getätigt haben.
- Finde die Anzahl Buchungen des "Tönen"-Services aller Kunden.
- Finde die Alben, welche mehr als 10 Tracks haben.
- Finde die durchschnittliche Dauer der ersten fünf Tracks jedes Albums.

---

---

**Aufgabe 3.13** Ein Bonustrack in einem Album ein Track, deren TrackId grösser ist als die Anzahl Tracks des Albums. In unserem Beispiel sind Rockaway Beach und Otherside Bonustracks, da in unserem unvollständigen Beispiel Album 2 nur ein Track hat, bzw. Album 2 nur zwei Tracks hat. Erstelle eine Abfrage, welche die Alben-Ids aller Alben mit Bonustracks zurückgibt.

---

## 3.7 Kapiteltest

Wieder arbeiten wir mit der Datenbank aus Tabellen 1.7 bis 1.13.

---

**Aufgabe 3.14** Erstelle eine Abfrage, welche die vollständige Tabelle Bar zurückgibt

---

---

**Aufgabe 3.15** Erstelle eine Abfrage, welche die Tabelle x zurückgibt.

**Aufgabe 3.16** Welche Flaschengrössen gibt es in dieser Datenbank? Erstelle dazu eine Abfrage.

**Aufgabe 3.17** Erstelle eine Abfrage, welche die Tabelle Sortiment nach Bier geordnet zurückgibt.

**Aufgabe 3.18** Erstelle eine Abfrage, welche alle Biere, die im Namen ein Umlaut (ä, ö oder ü) enthalten, anzeigt.

**Aufgabe 3.19** Zeige alle Biere der Brauerei mit der Id 1 mittels Datenbankabfrage an.

**Aufgabe 3.20** Zeige alle Biere an, deren Einkaufspreis pro Deziliter unter 40 Rappen ist.

**Aufgabe 3.21** Erstelle eine Abfrage, welche den Namen jedes Kunden und den Namen seines Lieblingsbier anzeigt. Dabei sollen auch Kunden angezeigt werden, welche kein Lieblingsbier haben.

**Aufgabe 3.22** Erstelle eine Abfrage, welche den Namen jedes Kunden und den Namen seines Lieblingsbier anzeigt. Dabei sollen auch Kunden angezeigt werden, welche kein Lieblingsbier haben. Ausserdem sollen auch gleich die Biere, welche von keinem Kunden favorisiert werden, ausgegeben werden

**Aufgabe 3.23** Erstelle eine Abfrage, welche für jeden Eintrag in der Tabelle BestellteBier den Kundennamen, den Biernamen und das Datum retourniert.

---

**Aufgabe 3.24** Erstelle eine Abfrage, welche bei jeder Bestellung KundenId, BarId und Gesamtpreis zurückgibt.

---

**Aufgabe 3.25** Erstelle eine Abfrage, welche für jede BarId, welche von Bestellungen referenziert wird, den Umsatz für die Silvesternacht (31.12.2014 18:00 bis 1.1.2015 06:00) zurückgibt.

---

**Aufgabe 3.26** Zeige die Namen der Personen an, welche in der Silvesternacht mehr als 20 Franken für Bier ausgegeben haben.

---

**Aufgabe 3.27** Erstelle eine Abfrage, welche den Biernamen und dessen günstigsten Verkaufspreis zurückgibt. Tipp: Die Abfrage muss berücksichtigen, ob das Bier im Sortiment als gelöscht markiert wurde.

---

# Kapitel 4

# Datenbankmanipulationen

Wir haben jetzt gelernt, wie man Datenbanken erstellt, wie man Datensätze einfügt, und wie man diese Daten abfragt. Was jetzt noch fehlt ist das eigentliche Arbeiten mit den Daten, also wie man Datensätze bearbeitet und löscht. Diese Operationen sind einfacher zu verstehen, wenn man das Abfragen von Daten bereits beherrscht. Daher wurden sie bis jetzt vorenthalten. In diesem Kapitel sehen wir auch, wie wir anhand bereits vorhandener Daten weitere Daten generieren kann.

## 4.1 Löschen

Die Löschabfrage ist der einfachen SELECT-Abfrage sehr ähnlich. Mit diesem Befehl würden sämtliche Datensätze der Tabelle Booking gelöscht:

Listing 4.1: Löschen aller Zeilen einer Tabelle

```
1 DELETE
2 FROM Booking
```

Natürlich kommt es sehr selten vor, dass man alle Daten aus einer Tabelle löschen muss. Daher kommt der WHERE-Filter beim löschen häufig zum Einsatz.

Listing 4.2: Löschen von gefilterten Zeilen

```
1 DELETE
2 FROM Booking
3 WHERE BookingId = 3
```

---

**Aufgabe 4.1** Lösche alle Tracks, welche mit einem Leerzeichen enden.

---

### 4.1.1 Löschen mit JOINs

Analog zu den SELECT-Abfragen kann es auch bei DELETEs notwendig sein, die Daten über mehrere Tabellen herauszufiltern, bevor man sie löscht. In diesem Fall muss man spezifizieren, von welchen Tabellen man die Daten löschen möchte. In folgendem Beispiel werden sämtliche Buchungen von Kunden gelöscht, die Tester heissen. Die Kunden selbst werden jedoch nicht aus der Datenbank gelöscht:

Listing 4.3: Löschen mit JOIN

```
1 DELETE b
2 FROM Booking b INNER JOIN Customer c
3   ON b.CustomerFk = c.CustomerKey
4 WHERE c.LastName = 'Tester'
```

---

**Aufgabe 4.2** Lösche alle Tracks vom Künstler 'Various Artists'

---

## 4.2 Referenzielle Integrität

Im SQL muss man beim Modifizieren und Löschen aufpassen. Eine Änderung in der Datenbank kann nicht vorgenommen werden, wenn ein Fremdschlüssel danach ins Leere zeigt. So kann man zum Beispiel in der Tabelle Service nicht plötzlich den Service mit der Id 1 löschen, da dieser in der Tabelle Booking bereits referenziert wird. Wenn man einen referenzierten Datensatz löschen möchte, muss man sämtliche referenzierende Datenstze auch gleich löschen. Die Datenbank kann so eingestellt sein, dass sie dies gleich automatisch tut. Diese Einstellung heisst CASCADE. Wir betrachten die Option hier nicht näher.

Manchmal möchte man die referenzierende Datensätze gar nicht löschen. Es gibt zwei Möglichkeiten, wie der Datenbankdesigner dies zulassen kann. Die erste Option ist das sogenannte SET NULL. Auf diese kann nur zurückgegriffen werden, wenn die Referenzspalte leer, bzw. NULL sein darf. Wird bei dieser Option ein Datensatz gelöscht, werden alle Referenzen auf NULL gesetzt. Die zweite Möglichkeit, wie mit gelöschten Datensätzen umgegangen werden kann, ist mit dem sogenannten Deleted-Flag. Dies ist eine zusätzliche Spalte in der Tabelle, welche meistens 'Deleted' oder ähnlich heisst und ein BIT beinhaltet. Wenn man Daten löschen möchte, wird in diesem Fall der Datensatz nicht aus der Datenbank entfernt. Statt dessen wird die Deleted Spalte einfach auf 1 gesetzt. Dies muss dann bei der Datenbankabfrage berücksichtigt werden. Wenn wir bei der Tabelle Service ein Deleted verwenden würden, müsste man zum Beispiel wie folgt die Anzahl Services zählen:

Listing 4.4: Delete-Flag

```
1 SELECT COUNT(*)
2 FROM Service
3 WHERE Deleted = 0
```

## 4.3 Updates

Der UPDATE-Befehl funktioniert analog zum SELECT- und zum DELETE-Befehl. Der einzige Unterschied ist, dass man spezifizieren muss, welche Felder wie aktualisiert werden müssen. In diesem Beispiel verbessern wir endlich einen Schreibfehler in der ursprünglichen Datenbank:

Listing 4.5: Einfaches Update

```
1 UPDATE Album
2 SET ArtistName = 'Red Hot Chili Peppers'
3 WHERE AlbumId = 1
```

Man kann auch gleich mehrere Spalten gleichzeitig in der Datenbank bearbeiten. Dazu reiht man die Felder mit Komma getrennt aneinander, wie in folgendem Beispiel. Anmerkung: Dieser Syntax funktioniert nicht in PL/SQL.

Listing 4.6: Update mehrerer Spalten

```
1 UPDATE Service
2 SET Name = 'Nur Waschen',
3   DurationMinutes = DurationMinutes - 5
4 WHERE ServiceId = 1
```

Dieses Skript setzt die Dauer des Services um fünf Minuten hinunter. Dieser Syntax ist für nichtprogrammierer etwas gewöhnungsbedürftig. Wenn ein Spaltenname auf der rechten Seite des Gleichs steht, repräsentiert der Spaltenname den Wert dieser Spalte vor des Updates.

---

**Aufgabe 4.3** Angenommen, wir erweitern die Tabelle Customer um eine Spalte FullName, welche aus FirstName, LastName und einem Leerzeichen besteht. Schreibe eine UPDATE-Skript, welches diese Spalte befüllt. Tipp: Um Zeichenketten aneinaderzuhängen, kann man die SQL- Funktion CONCAT(,) verwenden. Diese Funktion kann mit beliebig vielen Parameter aufgerufen werden.

---

### 4.3.1 Updates mit JOINs

Um Daten von anderen Tabellen der Datenbank für ein Update zu verwenden, empfiehlt es sich, JOINS zu verwenden. Leider ist dies nicht im SQL-Standard enthaten. Daher variiert der Syntax dieser sehr nützlichen Funktionalität sehr stark zwischen den verschiedenen SQL-Erweiterungen. Daher hier ein Beispiel, welches zumindest in MySQL funktioniert. In den meisten Erweiterungen sieht diese Abfrage ähnlich (jedoch etwas komplizierter) aus. In diesem Beispiel nehmen wir an, dass die Track-Tabelle um die Spalte ArtistName erweitert worden ist, und füllen nun diese Spalte mit den Daten aus der Album-Tabelle ab:

Listing 4.7: Update mit JOIN in MySQL

```
1 UPDATE Album a INNER JOIN Track t
2   ON a.AlbumKey = t.AlbumFk
3 SET t.ArtistName = a.ArtistName
```

Der Vollständigkeit halber zeigen wir hier noch wie das Gleiche im SQL-Standard aussieht. Hier verwendet man eine Unterabfrage, um den Wert des Künstlernamen zu erhalten:

Listing 4.8: Standardisierter Update mit JOIN

```
1 UPDATE Track t
2 SET t.ArtistName = (
3   SELECT a.ArtistName
4   FROM Album a
5   WHERE a.ArtistKey = t.AlbumFk
6 )
```

---

**Aufgabe 4.4** Angenommen, wir erweitern die Tabelle Booking um eine Spalte Comment, in welcher Text geschrieben werden kann. Schreibe eine UPDATE-Skript, welches diese Spalte mit Vornamen, Nachnamen und Servicenamen befüllt. Tipp: Wie in der letzten Aufgabe wird hier auch die CONCAT()-Funktion benötigt. Dazu dürfen wir annehmen, dass wir MySQL verwenden.

---

---

**Aufgabe 4.5** Angenommen, wir erweitern die Tabelle Booking um eine Spalte EndDate, in welcher das Enddatum des Termins geschrieben werden soll. Schreibe eine UPDATE-Skript, welches diese Spalte füllt.

---

## 4.4 Einfügen

Wir können auch mittels Unterabfrage das Generieren von Daten erleichtern. Als Beispiel nehmen wir hier wieder das Frisörunternehmen. Dieses möchte nun seine Kunden über das Internet Termine buchen lassen, und möchte für jedes seiner Kunden ein Login erstellen. Diese Logins werden in eine Tabelle User gespeichert, welche über die Pflichtfelder CustomerFk, Login und IsPasswordSet. Folgendes Skript erstellt einen Benutzer für jeden Kunden, bei welchem das Passwort noch nicht gesetzt ist, und dessen Login der Emailadresse des Benutzers entspricht.

Listing 4.9: Daten generieren

```
1 INSERT INTO User (CustomerFk, Login, IsPasswordSet)
2 SELECT (CustomerId, Email, 0)
3 FROM Customer
```

## 4.5 Kapiteltest

Wieder arbeiten wir mit der Datenbank aus Tabellen 1.7 bis 1.13.

---

**Aufgabe 4.6** Erstelle ein Skript aus der Tabelle BestellteBier alle Zeilen mit Anzahl grösser als 1 löscht.

---

---

**Aufgabe 4.7** Erstelle ein Skript, welch die Bestellung 4 mit sämtlichen Bestellungsdetails aus der Datenbank löscht.

---

---

**Aufgabe 4.8** Wann darf man Datensätze aus der Tabelle Bier löschen?

---

---

**Aufgabe 4.9** Die Bestellung 5 wurde falsch erfasst. es wurden nicht drei Bier mit SortimentId 12 und ein Bier mit SortimentId 9 bestellt, sondern je zwei Bier. Passe dies mittels SQL-Skript an.

---

---

**Aufgabe 4.10** Die Preise der Bar mit der Id 2 waren falsch in der Datenbank: Sie sind eigentlich 50 Rappen teurer. Dies wurde beim Kassieren bereits berücksichtigt. Erstelle dafür ein SQL-Skript.

---

---

**Aufgabe 4.11** Die Bar mit der Id 2 ändert ihre Preise: Alle Biere werden 50 Rappen teurer. Erstelle dafür ein SQL-Skript. Wir dürfen annehmen, dass die Bar noch keine gelöschte Einträge im Sortiment hat. Ausserdem wird die Spalte SortimentId automatisch generiert und muss beim Einfügen nicht berücksichtigt werden.

---

# Kapitel 5

# Lösungen

## 5.1 Kapitel 1

1. Wenn die ServiceFk sich ändert, wird ein anderer Service referenziert. Der Service 2 dauert 15 Minuten länger als der Service 1 und dauert dafür 15 Minuten länger.

2. Der Primärschlüssel ist ein eindeutiges Merkmal einer Zeile. Wenn wir ServiceId und CustomerId als zusammengesetzten Primärschlüssel verwenden würden, dürfte für ein Kunde ein Service nur genau ein Mal im System erfasst sein. Dies heisst also, dass ein Kunde nicht mehrere Termine für einen Service gleichzeitig haben kann. Sogar wenn dies akzeptabel wäre, gibt es jedoch einen anderen Grund, welcher dagegen spricht: Beim nächsten Termin des gleichen Typs müsste man in der Datenbank die alte Buchung überschreiben. Somit wären die historischen Daten unbrauchbar.

3. Obwohl es in einer Musiksammlung vorkommen kann, dass das Album bei gewissen Tracks nicht bekannt sind, wird dies in dieser Datenbank keine Rechnung getragen. Die Spalte AlbumFk ist Teil des Primärschlüssels, und darf somit nicht fakultativ sein, also keine NULL-einträge enthalten.

4. - Am 31.12. um 22 Uhr hat Urs in der Vollmond Taverne 2 Brandlöscher bestellt.
   - Um 23 Uhr 23 hat Lisa auch in der Vollmond Taverne 2 Brandöscher und ein Quöllfrisch bestellt.
   - Gleich danach hat Lisa am gleichen Ort noch ein Sprint bestellt.
   - Um 23 Uhr 52 hat Heinz im Si o No 3 Sprint und ein Wädi-Bräu Blond Premium bestellt.

- Am Neujahrstag um 1 Uhr 21 hat Andrea in der Brasserie Federal 3 Quöllfrisch und ein Heineken Lager bestellt.

5. Das Vollmond-Bier kostet im Einkauf 1.30 Franken, und wird in der Vollmond Taverne zu 4 Franken ausgeschenkt. Also wird das Bier zu etwas mehr als 3 mal dem Einkaufspreis verkauft.

6. In die Tabellen Bestellung und Bestellte Bier werden folgende Zeilen eingefügt:

| BestellungId (PK) | KundenId (FK(Kunde.KundenId)) | BarId (FK(Bar.BarId)) | Zeit |
|---|---|---|---|
| 6 | 2 | 3 | 1.1.2015 02:00 |

Tabelle 5.1: Bestellung

| BestellungFk (PK, FK(Bestellung. BestellungId)) | SortimentFk (PK, FK(Sortiment. SortimentId)) | Anzahl |
|---|---|---|
| 6 | 14 | 2 |

Tabelle 5.2: BestellteBier

7. Wenn der Schlüssel zusammengesetzt wäre, würde eine Bar pro Bier genau einen Preis haben. Auf den ersten Blick macht das Sinn, jedoch nicht in Kombination mit der Delete-Spalte. Die Delete-Spalte ermöglicht es uns, den Preis jedes bestellten Bieres nachzuvollziehen. Diese Möglichkeit geht jedoch verloren, wenn wir nur einen Preis pro Bier und Bar speichern können.

8. Eine Spalte BestellteBierId könnte man einführen und als alleiniger Primärschlüssel wählen. Die Datenbank würde jedoch eine sinnvolle Einschiänkung verlieren. Pro Bestellung könnte es nämlich mehrere Einträge mit dem gleichen Bier, bzw. dem gleichen Eintrag im Sortiment geben. Das kann gewisse Berechnungen etwas komplexer machen, würde jedoch der Funktionalität der Datenbank nicht schaden.

## 5.2 Kapitel 2

1. Es macht durchaus Sinn, wenn man die Länge einer Zeichenkette einschränkt. Einerseits, weil andere Längen keinen Sinn ergeben. Beispielsweise haben die Kantonskürzel (z.b. ZH, ZG, SG, etc.) immer genau zwei Buchstaben. Auch gibt es weltweit keine Ortsnamen, welche länger als 150 Zeichen lang

sind. Andererseits kann man somit auf der Datenbank die Länge gewisser Textgrössen einschränken. Nicht jeder Text soll beliebig lange sein dürfen.

2. In dieser Lösung verzichten wir darauf, Zeichenketten zu beschränken. Auch sind andere Formate für die Zeitdauerwerte vorstellbar.

   - Customer:
     CustomerId INT
     FirstName VARCHAR(MAX)
     LastName VARCHAR(MAX)
     Email VARCHAR(MAX)
   - Service:
     ServiceId INT
     Name VARCHAR(MAX)
     DurationMinutes INT
   - Booking:
     BookingId INT
     ServiceFk INT
     CustomerFk INT
     StartDate DATETIME

3. - Album:
     AlbumId INT
     ArtistName VARCHAR(MAX)
     AlbumName VARCHAR(MAX)
     GenreFk INT
   - Track:
     AlbumFK INT
     TrackId INT
     Name VARCHAR(MAX)
     DurationSeconds INT
   - Genre:
     GenreId INT
     ParentGenreFk INT
     Name VARCHAR(MAX)

4. Bei vielen Zeilen steht es uns frei, ob sie NULL sein dürfen oder nicht.

Listing 5.1: Erstellen der Tabelle Album

```
1 CREATE TABLE Album (
2         AlbumId INT NOT NULL,
```

```
3         ArtistName VARCHAR(200),
4         AlbumName VARCHAR(200),
5         GenreFk INT,
6         CONSTRAINT pk_album PRIMARY KEY
7                 (AlbumId),
8         CONSTRAINT fk_genre FOREIGN KEY (GenreFk)
9                 REFERENCES Genre(GenreId)
10 );
```

Listing 5.2: Erstellen der Tabelle Genre

```
1  CREATE TABLE Genre (
2         GenreId INT NOT NULL,
3         ParentGenreFk INT,
4         Name VARCHAR(MAX),
5         CONSTRAINT pk_genre PRIMARY KEY
6                 (GenreId),
7         CONSTRAINT fk_genre FOREIGN KEY (ParentGenreFk)
8                 REFERENCES Genre(GenreId)
9  );
```

5. Bei vielen Zeilen steht es uns frei, ob sie NULL sein dürfen oder nicht.

Listing 5.3: Erstellen der Tabelle Album

```
1  CREATE TABLE Customer (
2         CustomerId INT NOT NULL,
3         FirstName VARCHAR(200),
4         LastName VARCHAR(200),
5         Email VARCHAR(MAX),
6         CONSTRAINT pk_customer PRIMARY KEY
7                 (CustomerId)
8
9  );
```

Listing 5.4: Erstellen der Tabelle Genre

```
1  CREATE TABLE Service (
2         ServiceId INT NOT NULL,
3         Name VARCHAR(MAX),
4         DurationMinutes INT NOT NULL,
5         CONSTRAINT pk_service PRIMARY KEY
6                 (ServiceId)
```

```
);
```

6. Erstellen der Beispieldaten:

Listing 5.5: Erstellen der Beispielsdaten

```
INSERT INTO Genre (GenreId, ParentGenreId, Name)
VALUES (1, NULL, 'Rock'),
  (2, 1, 'Punk Rock'),
  (3, NULL, 'Hip-Hop')

INSERT INTO Album
        (AlbumKey, ArtistName, AlbumName, GenreFk)
VALUES
  (1, 'Red Hot Chilli Peppers',
    'Californication', 1),
  (2, 'The Ramones', 'Rocket To Russia', 2),
  (3, 'The Beastie Boys', 'Ill Communication', 3)

INSERT INTO Track
        (AlbumFk, TrackId, Name, DurationSeconds)
VALUES (3, 1, 'Sure Shot', 200),
  (1, 1, 'Around the World', 239),
  (2, 2, 'Rockaway Beach', 127),
  (1, 4, 'Otherside', 255)
```

7. So werden die Tabellen erstellt.

Listing 5.6: Erstellen der Tabelle Bestellung und BestellteBier

```
CREATE TABLE Bestellung (
        BestellungId INT NOT NULL,
        KundenFk INT NOT NULL,
        BarFk INT NOT NULL,
        Zeit DATETIME,
        CONSTRAINT pk_bestellung PRIMARY KEY
                (BesellungId),
        CONSTRAINT fk_kunde FOREIGN KEY(KundenFk)
        REFERENCES Kunde(KundenId),
        CONSTRAIN fk_bar FOREIGN KEY(BarFk)
        REFERENCES Bar(BarId)
);
```

```
13 CREATE TABLE BestellteBier (
14         BestellungFk INT NOT NULL,
15         SortimentFk INT NOT NULL,
16         Anzahl INT NOT NULL,
17         CONSTRAINT pk_bestellte_bier PRIMARY KEY
18                 (BesellungFk, SortimentFk),
19         CONSTRAINT fk_bestellung FOREIGN KEY
20           (BestellungFk) REFERENCES
21           Bestellung(BestellungId),
22         CONSTRAINT fk_sortiment FOREIGN KEY
23         (SortimentFk)REFERENCES
24          Sortiment(SortimentId)
25 );
```

8. Die Daten werden wie folgt eingefügt:

Listing 5.7: Einfügen der Bestellung

```
1 INSERT INTO Bestellung
2   (BestellungId, KundenId, BarId, Zeit)
3  VALUES (6, 2, 3, '2015-1-1 02:00')
4 INSERT INTO BestellteBier
5   (BestellungFk, SortimentFk, Anzahl)
6 VALUES (6, 14, 2)
```

## 5.3 Kapitel 3

1. Die Abfrage sieht wie folgt aus:

Listing 5.8: Abfrage zu Tabelle 3.1

```
1 SELECT LOWER(Name) AS NameLower,
2   DurationMinutes*60*1000 AS DurationMilliseconds
3 FROM Service
```

2. Buchungen sortiert:

Listing 5.9: Buchungen sortiert

```
1 SELECT *
2 FROM Booking
3 ORDER BY StartDate DESC
```

3. Alben sortiert:

Listing 5.10: Alben sortiert

```
SELECT *
FROM Album
ORDER BY GenreFk, ArtistName DESC
```

4. Termine gefiltert:

Listing 5.11: Termine gefiltert

```
SELECT *
FROM Booking
WHERE BookingStart > '2014-11-24'
  AND BookingStart < '2014-12-01'
```

5. Termine, welche länger als 30 Minuten dauern:

Listing 5.12: Termine, welche länger als 30 Minuten dauern

```
SELECT *
FROM Booking
WHERE ServiceFk IN (
  SELECT ServiceId
  FROM Service
  WHERE DurationMinutes > 30
)
```

6. Erweiterte Tabelle:

Listing 5.13: Erweiterte Tabelle

```
SELECT c.FirstName, c.LastName, s.Name, b.StartDate,
  DATEADD(MINUTE, s.DurationMinutes,
    b.StartDate) AS EndDate
FROM Customer c
  INNER JOIN Booking b
    ON b.CustomerFk = c.CustomerId
  INNER JOIN Service s
    ON b.ServiceFk = s.ServiceId
```

7. Selbst-JOIN:

Listing 5.14: Erweiterte Tabelle

```
1 SELECT g1.Name AS SubGenreName, g2.Name AS ParentGenreName
2 FROM Genre g1
3   INNER JOIN Genre g2
4     ON g1.ParentGenreFk = g2.GenreId
```

8. Die vier Tabellen sehen wie folgt aus:

| SubGenreName | GenreName |
|---|---|
| Punk Rock | Rock |

Tabelle 5.3: INNER JOIN

| SubGenreName | GenreName |
|---|---|
| Rock | NULL |
| Punk Rock | Rock |
| Hip-Hop | NULL |

Tabelle 5.4: LEFT JOIN

| SubGenreName | GenreName |
|---|---|
| Punk Rock | Rock |
| NULL | Punk Rock |
| NULL | Hip-Hop |

Tabelle 5.5: RIGHT JOIN

| SubGenreName | GenreName |
|---|---|
| Rock | NULL |
| Hip-Hop | NULL |
| Punk Rock | Rock |
| NULL | Punk Rock |
| NULL | Hip-Hop |

Tabelle 5.6: FULL OUTER JOIN

9. Aggregierte Albumlänge:

Listing 5.15: Albumlänge

```
1 SELECT AlbumFk,
2   SUM(DurationMinutes) AS AlbumLength
3 FROM Tracks
4 GROUP BY AlbumFk
```

10. Mit Albumnamen:

Listing 5.16: Albumlänge mit Albumnamen

```
SELECT a.ArtistName, a.AlbumName,
  albumLengths.AlbumLength
FROM Album a INNER JOIN
  ( SELECT AlbumFk,
      SUM(DurationMinutes) AS AlbumLength
    FROM Tracks
    GROUP BY AlbumFk) albumLengths
  ON a.ArtistId = albumLengths.ArtistFk
```

11. HAVING-Filter:

Listing 5.17: Albumlänge mit HAVING-Filter

```
SELECT AlbumFk,
   SUM(DurationMinutes) AS AlbumLength
FROM Tracks
GROUP BY AlbumFk
HAVING AVG(DurationMinutes) < 180
```

12. Zur Repetition: Eine WHERE-Klausel ist ein Filter für die Daten vor der Aggregation. Die HAVING-Klausel filtert Daten, nachdem sie aggregiert wurden.

    - Die Anzahl Buchungen ist ein aggregierter Wert. Daher steht diese Bedingung in der HAVING-Klausel.
    - Nach Service muss vor der Aggregation, also im WHERE-Teil gefiltert werden.
    - HAVING-Klausel
    - WHERE-Klausel. Zuerst suchen wir die ersten fünf Tracks des Albums heraus. Der Durchschnitt wird anschliessend berechnet.

13. Bonustrack-Abfrage:

Listing 5.18: Bonustrack-Abfrage

```
SELECT AlbumFk
FROM Tracks
GROUP BY AlbumFk
HAVING COUNT(*) < MAX(TrackId)
```

14. Tabelle Bar:

Listing 5.19: Tabelle Bar

```
SELECT *
FROM Bar
```

15. Tabelle Bier:

Listing 5.20: Bier

```
SELECT
  LOWER(Name) AS NameLower,
  Deziliter * 10 AS Milliliter,
  Einkaufpreis / 1.2 AS EinkaufpreisEuro
FROM Bier
```

16. Flaschengrössen:

Listing 5.21: Flaschengrössen

```
SELECT DISTINCT Deziliter
FROM Bier
```

17. Sortiment nach Bier:

Listing 5.22: Sortiment nach Bier

```
SELECT *
FROM Sortiment
ORDER BY BierFk
```

18. Bier mit Umlaut:

Listing 5.23: Bier mit Umlaut

```
SELECT *
FROM Bier
WHERE Name LIKE '%ä%'
  OR Name LIKE '%ö%'
  OR Name LIKE '%ü%'
```

19. Biere der Brauerei 1:

Listing 5.24: Biere der Brauerei 1

```
1 SELECT *
2 FROM Bier
3 WHERE BrauereiFk = 1
```

20. Günstige Biere:

Listing 5.25: Günstige Biere

```
1 SELECT *
2 FROM Bier
3 WHERE Einkaufspreis / Deziliter < 0.40
```

21. Kunde mit Lieblingsbier:

Listing 5.26: Kunde mit Lieblingsbier

```
1 SELECT k.Name, b.Name
2 FROM Kunde k LEFT JOIN Bier b
3   ON k.LieblingsbierFk = b.BierId
```

22. Kunde mit Lieblingsbier sowie unfavorisierte Biere:

Listing 5.27: Kunde mit Lieblingsbier sowie unfavorisierte Biere

```
1 SELECT k.Name, b.Name
2 FROM Kunde k FULL OUTER JOIN Bier b
3   ON k.LieblingsbierFk = b.BierId
```

23. Bestellte Bier in lesbarer Form:

Listing 5.28: Bestellte Bier in lesbarer Form

```
1  SELECT k.Name, bi.Name, b.Zeit
2  FROM BestellteBier bb
3    INNER JOIN Bestellung b
4      ON bb.BestellungFk = b.BestellungId
5    INNER JOIN Kunde k
6      ON  b.KundenFk = k.KundenId
7    INNER JOIN Sortiment s
8      ON bb.SortimentFk = s.SortimentId
9    INNER JOIN Bier bi
10     ON s.BierFk = bi.BierId
```

24. Diese Abfrage ist etwas komplizierter. Hier Joinen wir mit einer Unterabfrage. Die Unterabfrage gruppiert die Bestellungen nach Gesamtpreis. Dazu müssen zwei Spalten aus zwei verschiedenen Tabellen zusammen verrechnet und dann aggregiert werden. Dies ist ohne weitere Unterabfrage möglich.

    Die ussere Abfrage sammelt weitere Informationen zur Bestellung.

    Die Innere

Listing 5.29: Bestellungen

```
1  SELECT b.KundenFk, b.BarFk, bbGrouped.TotalPrice
2  FROM Bestellung b INNER JOIN
3    (SELECT bb.BestellungFk,
4      SUM(bb.Anzahl * s.Preis) AS TotalPrice
5    FROM BestellteBier bb
6      INNER JOIN Sortiment s
7        ON s.SortimentId = bb.SortimentFk
8    GROUP BY bb.BestellungFK
9    ) bbGrouped
10   ON bbGrouped.BestellungFk = b.BestellungId
```

25. Wir entdecken, das wir die letzte Aufgabe gröstenteils wiederverwenden können. Die Abfrage wird nun noch etwas komplizierter:

Listing 5.30: Umsatz pro Bar

```
1  SELECT b.BarFk,
2    SUM(bbGrouped.TotalPrice) AS Neujahrsumsatz
3  FROM Bestellung b INNER JOIN
4    (SELECT bb.BestellungFk,
5      SUM(bb.Anzahl * s.Preis) AS TotalPrice
6    FROM BestellteBier bb
7      INNER JOIN Sortiment s
8        ON s.SortimentId = bb.SortimentFk
9    GROUP BY bb.BestellungFK
10   ) bbGrouped
11   ON bbGrouped.BestellungFk = b.BestellungId
12 WHERE b.Zeit > '2014-31-12 18:00'
13   AND b.Zeit < '2015-01-01 06:00'
14 GROUP BY b.BarFK
```

26. Und wieder können wir die vorhergehende Abfrage wiederverwenden:

Listing 5.31: Trinkfreudige Kunden

```
SELECT k.Name
FROM Kunde k
INNER JOIN
( SELECT b.KundenFk,
    SUM(bbGrouped.TotalPrice) AS Neujahrsausgaben
  FROM Bestellung b INNER JOIN
    (SELECT bb.BestellungFk,
      SUM(bb.Anzahl * s.Preis) AS TotalPrice
    FROM BestellteBier bb
      INNER JOIN Sortiment s
        ON s.SortimentId = bb.SortimentFk
    GROUP BY bb.BestellungFk
    ) bbGrouped
  ON bbGrouped.BestellungFk = b.BestellungId
  WHERE b.Zeit > '2014-31-12 18:00'
    AND b.Zeit < '2015-01-01 06:00'
  GROUP BY b.KundenFk) ka
ON ka.KundenFk = k.KundenId
WHERE ka.Neujahrsausgaben > 20
```

27. Dies ist wieder ein einfacher JOIN mit einem GROUP BY

Listing 5.32: Günstigster Preis

```
SELECT b.Name, billigBier.BilligPreis
FROM Bier b
INNER JOIN
  (SELECT BierFk, MIN(Preis) AS BilligPreis
  FROM Sortiment
  WHERE Deleted = 0
  GROUP BY BierFk) billigBier
  ON billigBier.BierFk = b.BierId
```

## 5.4 Kapitel 4

1. Löschen:

Listing 5.33: Löschen von Einträgen

```
DELETE
```

```
2 FROM Tracks
3 WHERE Name LIKE '% '
```

2. Löschen von 'Various Artists:

Listing 5.34: Löschen von Einträgen

```
1 DELETE t
2 FROM Tracks t INNER JOIN
3   Album a ON a.AlbumKey = t.AlbumFk
4 WHERE a.ArtistName = 'Various Artists'
```

3. FullName:

Listing 5.35: Einfägain eines Kommentars

```
1 UPDATE Customer c
2 SET c.FullName = CONCAT(FirstName, ' ', LastName)
```

4. In MySql:

Listing 5.36: Einfägain eines Kommentars

```
1 UPDATE Booking b
2   INNER JOIN Customer c
3     ON c.CustomerKey = b.CustomerFk
4   INNER JOIN Service s
5    ON s.ServiceKey = b.ServiceFk
6 SET b.Comment = CONCAT(c.FirstName, ' ',
7     c.LastName, ' ', s.Name)
```

5. Enddatum in MySQL:

Listing 5.37: Einfügen des Enddatums

```
1 UPDATE Booking b
2   INNER JOIN Service s
3    ON s.ServiceKey = b.ServiceFk
4 SET b.startDate =
5   DATEADD(MINUTE, s.DurationMinutes, b.StartDate)
```

6. Löschen:

Listing 5.38: Löschen

```
1 DELETE FROM BestellteBier
2 WHERE Anzahl > 1
```

7. Löschen von zusammenhängenden Einträgen:

Listing 5.39: Löschen

```
1 DELETE b, bb FROM BestellteBier bb
2   INNER JOIN Bestellung b
3     ON b.BestellungId = bb.BestellungFk
4 WHERE b.BestellungId = 4
```

8. Um die Referenzielle Integrität zu bewahren, darf ein Bier nur dann gelöscht werden, wenn es nicht referenziert wird. In diesem konkreten Fall gibt es zwei Tabellen, welche das Bier referenzieren: Kunden und Sortiment. Dies heisst, dass das Bier nicht das Lieblingsbier eines Kunden sein darf, und auch in keinem Sortiment vorkommen darf. Da Sortimentseinträge in dieser Datenbank nicht gelöscht werden, heisst das, dass das Bier nie in einem Sortiment vorhanden war.

9. Update der Bestellung:

Listing 5.40: Update der Bestellung

```
1 UPDATE BestellteBier
2 SET Anzahl = 2
3 WHERE BestellungId = 5
```

10. Rückwirkendes Update der Preise:

Listing 5.41: Rückwirkendes update der Preise

```
1 UPDATE Sortiment
2 SET Preis = Preis + 0.5
3 WHERE BarFk = 2
```

11. Update der Preise:

Listing 5.42: Rückwirkendes update der Preise

```
1 UPDATE Sortiment
2 SET Deleted = 1
3 WHERE BarFk = 2
```

```
4
5 INSERT INTO Sortiment
6   (BarFk, BierFk, Preis, Deleted)
7 SELECT(BarFk, BierFk, Preis + 0.50, 0)
8 FROM Sortiment
9 WHERE BarFk = 2
```