Relationale Datenabfragen mit SQL

Gabriel Katz

6. April 2015

Inhaltsverzeichnis

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. 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.

Vom Leser dieser Unterlagen wird erwartet, dass er das Konzept relationaler Datenbanken bereits kennt, und bereits mit der Modellierung von relationalen Datenbanken erste Erfahrungen gemacht hat. Somit sind diese Unterlagen sind besonders gut für ein erstes Programmierlab nach der Einführung in die relationalen Datenbanken geeignet.

In diesem Kapitel repetieren wir anhand dreier Datenbankbeispielen, welche wir in den gesamten Unterlagen wieder verwenden werden, die relationalen Datenbanken. Leser, welche sich diese Repetition sparen möchten, können dies tun, wenn sie die Datenbankbeispiele verstanden haben.

Der Inhalt der weiteren Kapitel ist wie folgt aufgebaut: Kapitel 2 erklärt, was

SQL ist, und zeigt, wie man eine Datenbank in SQL erstellt. In Kapitel 3 wird gezeigt, wie man Daten in SQL aus einer einzelnen Tabelle abfragt. Danach sehen wir in Kapitel 4, wie man Abfragen über mehrere Tabellen erstellen kann. Kapitel 5 beschreibt dann, wie mehrere Datensätze in einem Datensatz zusammengefasst werden können.

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 Repetition: 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 Kennzahl. In den hier gezeigten Beispielen ist der Primärschlüssel fast immer eine Ganzzahl, doch dies muss nicht immer so sein.
Die Kernidee hinter den relationalen Datenbanken, ist, dass diese Tabellen verbunden sind. Hier kommen die sogenannten **Fremdschlüssel (FK)** ins Spiel. Ein

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

Fremdschlüssel besteht aus einer oder mehreren Spalten einer Tabelle, welche den Primärschlüssel einer anderen Tabelle referenzieren. In unseren Übungsbeispielen kommen keine zusammengesetzte Fremdschlüssel vor.

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

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

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` referenziert 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 Datenbankbeispiel. 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 `Kunde`-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. 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. Ziel dieses Kapitels ist, dass wir die Daten für die weiteren Kapitel selbst aufsetzen können.

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 INTs 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`.

Aufgabe 2.1 Wieso wählt man bei Zeichenketten nicht immer ein `VARCHAR` mit sehr grosser Höchstlänge? Dies würde doch in jedem Fall funktionieren.

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.

Beispiel 2.1 Erstelle 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_booking_service FOREIGN KEY (ServiceFk)
8      REFERENCES Service(ServiceId),
9    CONSTRAINT fk_booking_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 Leerschlägen 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_booking_service` und `fk_booking_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:

Beispiel 2.2 Erstelle die Tabelle `Track` in SQL

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_track_album FOREIGN KEY (AlbumFk)
9                  REFERENCES Album(AlbumId)
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:

Beispiel 2.3 Füge für Ferdinand Meier noch einen Termin zum Haare waschen und schneiden am 21. November 2014 um halb elf in die Datenbank ein.

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:

Beispiel 2.4 Füge mittels einer Abfrage folgende Termine in die Termindatenbank ein.

- Ferdinand Meier, Waschen und Schneiden, am 21. November 2014 um halb elf.
- Anja Teste, Waschen, Schneiden und Fönen, 26. November 2014 um 14:45.

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 im letzten Beispiel:

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 Zusammenfassung

- Wir kennen die wichtigsten Datentypen von SQL und wissen, wann welcher Datentyp eingesetzt wird.
- Wir wissen, wie wir anhand eines existierendes Tabellenmodells die dazugehörige SQL-Datenbank erstellen.
- Wir können Einträge in die SQL-Datenbank generieren.

2.6 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 über eine Tabelle

Jetzt dass wir wissen, wie wir unsere Datenbanken erstellen und Daten einfügen, können wir anfangen, Daten abzufragen. In diesem Kapitel lernen wir, Abfragen auf eine einzelne Tabelle abzusetzen. Das Ziel dieses Kapitels ist es, dass der Leser die Daten aus einer einzelnen Tabelle in einem sinnvollen Format darstellen kann. Dies lernen wir in drei Schritten: Zuerst fragen wir sämtliche Datensätze einer Tabelle in fixer Reihenfolge ab und konzentrieren uns darauf, wie wir die Zeilendaten darstellen können. Danach schauen wir, wie wir die Daten ordnen können, und schliesslich betrachten wir, wie wir gewisse Zeilen der Tabelle herausfiltern. Mit diesen drei Techniken zusammen hat man die Kontrolle über die Abfrage über eine Tabelle. Dieses Wissen ist auch für Abfragen über mehrere Tabellen essenziell.

Um mit den Abfragen herumzuexperimentieren, stehen die Datenbankbeispiele dem Leser auf sqlfiddle zur Verfügung. Das Buchungsbeispiel befindet sich auf `http://sqlfiddle.com/#!5/4f43d`, und die Musikdatenbank befindet sich auf `http://sqlfiddle.com/#!6/846d6`. Auf die Kapiteltestdatenbank kann man über `http://sqlfiddle.com/#!9/e0cb4` zugreifen.

3.1 Aufbau einer `SELECT`-Abfragen

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

Beispiel 3.1 Erstelle eine Abfrage, welche die `Track`-Tabelle zurückgibt.

Listing 3.1: Einfache SQL-Abfrage

```
1 SELECT *
```

```
2 FROM Track
```

Diese SQL-Abfragen kann man problemlos auf `http://sqlfiddle.com/#!6/846d6` ausprobieren. Dazu gibt man die Abfrage im rechten Texteditorfenster ein und drückt dann auf "Run SQL". Danach sieht man das Resultat unten an den Textfenster. 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. Im folgenden Beispiel wird sichtbar, wie wir nur bestimmte Spalten der Tabelle auswählen können:

Beispiel 3.2 Gebe die Spalten `Name` und `DurationSeconds` mittels Abfrage zurück.

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.

Beispiel 3.3 Erstelle eine Abfrage, welche den Namen und die Zeitdauer der Lieder zurückgibt, Die Spalte mit dem Namen sollte `Trackname` heissen, und in der Abfrage sollte die Tabelle `Track t` genannt werden.

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. In folgendem Beispiel verwenden wir Konstante Werte sowie Funktionen, um Spalten zu berechnen.

Beispiel 3.4 Erstelle eine Abfrage, welche die Tabelle `Track` abfragt. Die erste Spalte soll den Namen `Status` tragen und dessen Wert sollte für jede Zeile `'In Library'` sein. Die zweite Spalte soll `Trackname` heissen und den Namen in Kleinbuchstaben sein. Die dritte Spalte sollte Die (abgerundete) Anzahl Minuten des Tracks anzeigen und `DurationMinutes` heissen.

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

Tabelle 3.1: Erwünschtes Resultat aus Aufgabe 3.1

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
```

Wir sehen in der ersten Zeile, dass wir mit Selektieren eines konstanten Werts, hier `'In Library'`, bewirken kann, dass jede Zeile der Resultattabelle den Wert aufweist. Mit der String-Funktion `LOWER()` können wir eine Zeichenkette in Kleinbuchstaben augeben. Und mit einer Ganzzahldivision, wie wir es schon von anderen Programmiersprachen her kennen, errechnen wir die Zeitdauer in Minuten.

Weiter gibt es noch das Schlüsselwort `DISTINCT` zu erwähnen. mit diesem Schlüsselwort werden doppelte Einträge eliminiert.

Beispiel 3.5 Wieviel Minuten (abgerundet) können die Lieder der `Track`-Tabelle dauern?

Listing 3.5: DISTINCT

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

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

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.

Beispiel 3.6 Ordnen sie die Tabelle Track absteigend nach Albumsfremdschlüssel und dann aufsteigend nach Namen.

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.

Beispiel 3.7 Suche alle 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.

Beispiel 3.8 Suche nach allen Tracks, welche nicht am Anfang des Albums sind und länger als 4 Minuten dauern.

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.

Beispiel 3.9 Suche 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. Daher ist die Lösung des folgenden Beispiels nicht ganz trivial:

Beispiel 3.10 Finde alle Genre, deren Elterngenre nicht den Schlüssel 1 haben.

Intuitiv würde man diese Aufgabe vielleicht wie folgt lösen, doch das Resultat der folgenden Abfrage ist 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 Zusammenfassung

- Wir kennen die Grundstruktur einer `SELECT`-Abfrage, und wissen, wie wir z.B. auf Sqlfiddle eine Abfrage absetzen können.
- Wir können eine Tabelle nach bestimmten Tabellenspalten abfragen.

- Wir können sowohl die Spalten wie auch die Tabelle einer Abfrage benennen.
- Wir wissen, wie wir kleinere Berechnungen direkt in der `SELECT`-Abfrage durchführen können.
- Wir wissen, wie wir Mehrfachresultate in einer Abfrage vermeiden können.
- Wir wissen, wie wir die Abfrageresultate ordnen können.
- Wir wissen, wie wir Abfrageresultate filtern können.
- Wir sind uns der Probleme, welche auftreten können, wenn nach `NULL` gefiltert wird, bewusst, und wissen, wie wir diese Probleme vermeiden können.

3.5 Kapiteltest

Aufgabe 3.5 Erstelle eine Abfrage, welche die vollständige Tabelle Bar zurückgibt

Aufgabe 3.6 Erstelle eine Abfrage, welche die Biernamen in Kleinbuchstaben, die Anzahl Milliliter der Flaschen, und den Einkaufspreis in Euro (bei einem Wechselkurs von 1.2) zurückgibt.

Aufgabe 3.7 Welche Flaschengrössen gibt es in dieser Datenbank? Erstelle dazu eine Abfrage.

Aufgabe 3.8 Zeige, zu welchen Preisen das Bier mit der Id 7 verkauft wird.

Aufgabe 3.9 Erstelle eine Abfrage, welche die Tabelle Sortiment nach Bier geordnet zurückgibt.

Aufgabe 3.10 Erstelle eine Abfrage, welche alle Biere, die im Namen ein Umlaut (ä, ö oder ü) enthalten, anzeigt.

Aufgabe 3.11 Zeige alle Biere der Brauerei mit der Id 1 mittels Datenbankabfrage an.

Aufgabe 3.12 Zeige alle Biere an, deren Einkaufspreis pro Deziliter unter 40 Rappen ist.

Kapitel 4

Abfragen über mehrere Tabellen

Jetzt, da wir wissen, wie wir die Daten aus einer Tabelle sinnvoll abfragen, können wir sehen, wie wir Daten aus mehreren Tabellen zusammenfügen können. Dadurch können wir endlich die Macht der relationalen Datenbanken nutzen, denn so haben wir die Möglichkeit, zusammenhängende Informationen zusammen darzustellen. Eine Möglichkeit besteht darin, das Resultat einer Abfrage gleich für eine weitere Abfrage zu verwenden. Eine derartige verschachtelte Abfrage nennt sich **Unterabfrage**. Eine der häufigsten Anwendungen von Unterabfragen ist bei einer Abfrage über eine Menge von Werten. Daher betrachten wir zuerst solche sogenannten **Mengenabfragen**, bevor wir dann mit Hilfe dieser die Unterabfragen einführen. Es gibt auch andere Arten von Unterabfragen, doch auf diese werden wir hier nicht eingehen. Die zweite Möglichkeit, Tabellen miteinander verbindet, ist, indem man Tabellen bei den Schlüssel miteinander verbindet. Man kann es sich wie folgt vorstellen: Eine Zeile einer Tabelle, welche einen Fremdschlüssel hat, wird mitsamt den Daten, welche in der referenzierten Zeile der referenzierten Tabelle sind, wiedergegeben. Ein derartiges Hineinladen von Datensätzen nennt man **`JOIN`**.

4.1 Unterabfragen

4.1.1 Mengenabfragen

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

Beispiel 4.1 Finde alle Kunden, welche Tester, Muster oder Test heissen.

Listing 4.1: 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 4.2: Mengenabfrage

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

4.1.2 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.

Beispiel 4.2 Gebe alle Lieder von Bands, welche mit “The” anfangen, zurück.

Listing 4.3: Unterabfrage

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

Diese Lösung ist offensichtlich zweistufig: In der Unterabfrage werden alle Ids der Alben, deren Künstler mit “The” anfangen abgefragt. Danach wird die AlbumFk-Spalte mit diese Ids verglichen.

Aufgabe 4.1 Erstelle eine Abfrage, welche alle Buchungen zurückgibt, deren Service länger als 30 Minuten dauert.

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 4.1: `JOIN` über Customer und Booking

4.2 JOIN

4.2.1 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.

Beispiel 4.3 Erstellen wir so für den Frisör eine Ansicht, bei welchem er den Namen des Kunden und der Zeitpunkt des Termins sieht:

Listing 4.4: 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 `JOIN`s steht. So können wir auch mehrere `INNER JOINS` aneinanderreihen, um beliebig viele Tabellen zusammenfügen.

Beispiel 4.4 Erstelle eine Abfrage, welche Name von Track, Künstler, Album und Genre zu jedem Eintrag der `Tracks`-Tabelle zurückgibt.

Listing 4.5: Mehrere `INNER JOIN`s

```
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 4.2 Erweitere Listing 4.4 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 4.3 Verbinde die `Genre`-Tabelle mit sich selbst, so dass in der ersten Spalte das Untergenre und in der zweiten Spalte das Genre steht.

4.2.2 `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 4.1 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 4.1 das `INNER JOIN` durch ein `LEFT OUTER JOIN` ersetzen würde, so erhälte man Tabelle 4.2.

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 4.2: `LEFT OUTER JOIN`

Aufgabe 4.4 Führe von Hand die vier `JOIN`s (`INNER` / `LEFT OUTER` / `RIGHT OUTER` / `FULL OUTER`)-Operationen auf die Tabelle Genre und sich selbst (wie in Aufgabe 4.3) aus.

4.2.3 `JOIN` mit Unterabfragen

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

Beispiel 4.5 Ergänze die Abfrage aus dem Beispiel 3.4 um die Felder der Tabelle `Album`.

Listing 4.6: `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.

4.3 Zusammenfassung

- Wir können nach Zugehörigkeit einer Menge mittels `IN` filtern.
- Wir wissen, wie wir eine Unterabfrage in ein `IN` verpacken können.
- Wir wissen, was `INNER JOIN` und `OUTER JOIN` macht, und können diese Befehle anwenden.

4.4 Kapiteltest

Aufgabe 4.5 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 4.6 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 4.7 Erstelle eine Abfrage, welche für jeden Eintrag in der Tabelle BestellteBier den Kundennamen, den Biernamen und das Datum retourniert.

Kapitel 5

Aggregierung von Daten

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. 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 Aggregierungsfunktion zusammengetragen. Wir lernen am Anfang dieses Kapitel damit, die wichtigsten Aggregierungsfunktionen einzusetzen. Anschliessend betrachten wir, wie wir Aggregierte Tabellen mit anderen Tabellen verbinden können. Schliesslich sehen wir noch, wie wir auch über aggregierte Tabellenfelder filtern können.

5.1 Aggregierungsfunktionen

Der zur Aggregation gehörige Aggregierungsbefehl nennt sich `GROUP BY`. mit dem `GROUP BY`-Befehl. Jede Spalte des Resultats muss bei einer `GROUP BY`-Abfrage entweder ein Gruppierungsfeld oder eine Aggregationsfunktion sein. Mögliche Aggregierungsfunktionen 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

Beispiel 5.1 Zeige die Anzahl der Buchungen pro Service an. Zeige für jeden Service den Key an.

Listing 5.1: Zählen der Buchungen pro Service

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

Aufgabe 5.1 Erstelle eine Abfrage, welche in der Tabelle Tracks die AlbumFk und die Gesamtlänge des Albums auflistet.

5.2 `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:

Beispiel 5.2 Zeige die Anzahl der Buchungen für jeden Service, welcher gebucht wurde, an. Zeige fr jeden Service den Namen an

Listing 5.2: Versuch, GROUP BY mit JOIN zu kombinieren.

```
1 SELECT ?, COUNT(*)
2 FROM Booking b LEFT 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 `ServiceFk`s 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 eleganter und klarer. Die Abfrage sieht dann so aus:

Listing 5.3: `JOIN` mit gruppierter Tabelle

```
1 SELECT s.Name, BookingCount
2 FROM Service s LEFT 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 5.2 Erweitere die Abfrage aus Aufgabe 5.1, so dass der Künstlername und der Albumname neben der Gesamtspieldauer des Albums steht.

5.3 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.

Beispiel 5.3 Zeige die Anzahl Buchungen ab dem 24. November für jeden Service, der in dieser Zeit gebucht wurde, an. Zeige für jeden Service den Key an.

Listing 5.4: 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.

Beispiel 5.4 Suche die Schlüssel aller Services, welche ab dem 24. November genau ein mal gebucht wurden:

Listing 5.5: Filtern vor und nach Gruppierung

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

Wir sehen, dass die Aggregation, welche in der `HAVING`-Klausel erwähnt wird, gar nicht in der `SELECT`-Klausel vorkommt. Wie wir in der nächsten Übung sehen werden, können in `HAVING`-Klausel und in der `SELECT`-Klausel völlig verschiedene Aggregationen vorkommen.

Aufgabe 5.3 Erstelle eine Abfrage, welche die Gesamtdauer der Alben, bei welchen die Durchschnittsdauer der Tracks weniger als 3 Minuten beträgt, berechnet.

Aufgabe 5.4 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 5.5 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.

5.4 Zusammenfassung

- Wir kennen die wichtigsten Aggregierungsfunktionen.
- Wir wissen, wie man das `GROUP BY` über eine Tabelle einsetzt.
- Wir wissen, wie man das `GROUP BY` über mehrere Tabellen einsetzt, und wie man die Probleme vermeiden kann, welche dabei auftreten können.
- Wir wissen, wie wir bei Gruppierungsabfragen vor der Gruppierung, und wie wir nach der Gruppierung filtern können.

5.5 Kapiteltest

Aufgabe 5.6 Erstelle eine Abfrage, welche zurückgibt, welcher Kunde wieviel Bestellungen getätigt hat.

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

Aufgabe 5.7 Erstelle eine Abfrage, welche bei jeder Bestellung KundenId, BarId und Gesamtpreis zurückgibt.

Aufgabe 5.8 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 5.9 Zeige die Namen der Personen an, welche in der Silvesternacht mehr als 20 Franken für Bier ausgegeben haben.

Aufgabe 5.10 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 6

Lösungen

6.1 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 beinhaltet auch das Fönen nach dem Haarschnitt.

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 `BestellteBier` werden folgende Zeilen eingefügt:

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

Tabelle 6.1: Bestellung

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

Tabelle 6.2: 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.

6.2 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 Daten, welche nicht länger als eine gewisse Anzahl Zeichen

haben kann. Zum Beispiel gibt es weltweit keine Ortsnamen, welche länger als 150 Zeichen lang sind. Daher kann man, um die Qualität der Daten zu gewährleisten, die Länge eines Ortsnamenfeldes auf 150 Zeichen (etwas mehr, falls man für noch nicht existierende Ortsnamen gewappnet sein möchte) zu beschränken.

2. In der präsentierten Lösung sind die Zeichenkettenlängen frei gewählt. Abweichungen in den Längen können durchaus vorkommen. Auch sind andere Formate für die Zeitdauerwerte vorstellbar.

   - Customer:
     CustomerId INT
     FirstName VARCHAR(200)
     LastName VARCHAR(200)
     Email VARCHAR(300)
   - Service:
     ServiceId INT
     Name VARCHAR(1000)
     DurationMinutes INT
   - Booking:
     BookingId INT
     ServiceFk INT
     CustomerFk INT
     StartDate DATETIME

3. - Album:
     AlbumId INT
     ArtistName VARCHAR(1000)
     AlbumName VARCHAR(1000)
     GenreFk INT
   - Track:
     AlbumFK INT
     TrackId INT
     Name VARCHAR(1000)
     DurationSeconds INT
   - Genre:
     GenreId INT
     ParentGenreFk INT
     Name VARCHAR(1000)

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

Listing 6.1: Erstellen der Tabelle Album

```
CREATE TABLE Album (
  AlbumId INT NOT NULL,
  ArtistName VARCHAR(200),
  AlbumName VARCHAR(200),
  GenreFk INT,
  CONSTRAINT pk_album PRIMARY KEY
    (AlbumId),
  CONSTRAINT fk_album_genre FOREIGN KEY
    (GenreFk)
    REFERENCES Genre(GenreId)
);
```

Listing 6.2: Erstellen der Tabelle Genre

```
CREATE TABLE Genre (
  GenreId INT NOT NULL,
  ParentGenreFk INT,
  Name VARCHAR(1000),
  CONSTRAINT pk_genre PRIMARY KEY
    (GenreId),
  CONSTRAINT fk_genre_genre FOREIGN KEY
    (ParentGenreFk)
    REFERENCES Genre(GenreId)
);
```

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

Listing 6.3: Erstellen der Tabelle Album

```
CREATE TABLE Customer (
        CustomerId INT NOT NULL,
        FirstName VARCHAR(200),
        LastName VARCHAR(200),
        Email VARCHAR(300),
        CONSTRAINT pk_customer PRIMARY KEY
                (CustomerId)

);
```

Listing 6.4: Erstellen der Tabelle Genre

```
CREATE TABLE Service (
        ServiceId INT NOT NULL,
        Name VARCHAR(1000),
        DurationMinutes INT NOT NULL,
        CONSTRAINT pk_service PRIMARY KEY
                (ServiceId)
);
```

6. Erstellen der Beispieldaten:

Listing 6.5: 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 6.6: 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
```

```
7     (BestellungId),
8   CONSTRAINT fk_bestellung_kunde FOREIGN KEY
9     (KundenFk)
10    REFERENCES Kunde(KundenId),
11  CONSTRAINT fk_bestellung_bar FOREIGN KEY(BarFk)
12    REFERENCES Bar(BarId)
13 );
14 CREATE TABLE BestellteBier (
15   BestellungFk INT NOT NULL,
16   SortimentFk INT NOT NULL,
17   Anzahl INT NOT NULL,
18   CONSTRAINT pk_bestellte_bier PRIMARY KEY
19     (BestellungFk, SortimentFk),
20   CONSTRAINT fk_bestellteBier_bestellung
21     FOREIGN KEY (BestellungFk)
22     REFERENCES  Bestellung(BestellungId),
23   CONSTRAINT fk_bestellteBier_sortiment
24     FOREIGN KEY (SortimentFk)
25     REFERENCES Sortiment(SortimentId)
26 );
```

8. Die Daten werden wie folgt eingefügt:

Listing 6.7: 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)
```

6.3 Kapitel 3

1. Die Abfrage sieht wie folgt aus:

Listing 6.8: Abfrage zu Tabelle 3.1

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

2. Buchungen sortiert:

Listing 6.9: Buchungen sortiert

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

3. Alben sortiert:

Listing 6.10: Alben sortiert

```
1 SELECT *
2 FROM Album
3 ORDER BY GenreFk, ArtistName DESC
```

4. Termine gefiltert:

Listing 6.11: Termine gefiltert

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

5. Tabelle Bar:

Listing 6.12: Tabelle Bar

```
1 SELECT *
2 FROM Bar
```

6. Tabelle Bier:

Listing 6.13: Bier

```
1 SELECT
2   LOWER(Name) AS NameLower,
3   Deziliter * 100 AS Milliliter,
4   Einkaufpreis / 1.2 AS EinkaufpreisEuro
5 FROM Bier
```

7. Flaschengrössen:

Listing 6.14: Flaschengrössen

```
1 SELECT DISTINCT Deziliter
2 FROM Bier
```

8. Preise des Biers:

Listing 6.15: Preis von Bier 7

```
1 SELECT Preis
2 FROM Sortiment
3 WHERE BierFk = 7
```

9. Sortiment nach Bier:

Listing 6.16: Sortiment nach Bier

```
1 SELECT *
2 FROM Sortiment
3 ORDER BY BierFk
```

10. Bier mit Umlaut:

Listing 6.17: Bier mit Umlaut

```
1 SELECT *
2 FROM Bier
3 WHERE Name LIKE '%ä%'
4   OR Name LIKE '%ö%'
5   OR Name LIKE '%ü%'
```

11. Biere der Brauerei 1:

Listing 6.18: Biere der Brauerei 1

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

12. Günstige Biere:

Listing 6.19: Günstige Biere

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

6.4 Kapitel 4

1. Termine, welche länger als 30 Minuten dauern:

Listing 6.20: Termine, welche länger als 30 Minuten dauern

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

2. Erweiterte Tabelle:

Listing 6.21: 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
```

3. Selbst-JOIN:

Listing 6.22: Erweiterte Tabelle

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

4. Die vier Tabellen sehen wie folgt aus:

SubGenreName	GenreName
Punk Rock	Rock

Tabelle 6.3: INNER JOIN

SubGenreName	GenreName
Rock	NULL
Punk Rock	Rock
Hip-Hop	NULL

Tabelle 6.4: LEFT JOIN

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

Tabelle 6.5: RIGHT JOIN

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

Tabelle 6.6: FULL OUTER JOIN

5. Kunde mit Lieblingsbier:

Listing 6.23: Kunde mit Lieblingsbier

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

6. Kunde mit Lieblingsbier sowie unfavorisierte Biere:

Listing 6.24: 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
```

7. Bestellte Bier in lesbarer Form:

Listing 6.25: Bestellte Bier in lesbarer Form

```
SELECT k.Name, bi.Name, b.Zeit
FROM BestellteBier bb
  INNER JOIN Bestellung b
    ON bb.BestellungFk = b.BestellungId
  INNER JOIN Kunde k
    ON  b.KundenFk = k.KundenId
  INNER JOIN Sortiment s
    ON bb.SortimentFk = s.SortimentId
  INNER JOIN Bier bi
    ON s.BierFk = bi.BierId
```

6.5 Kapitel 5

1. Aggregierte Albumlänge:

Listing 6.26: Albumlänge

```
SELECT AlbumFk,
  SUM(DurationMinutes) AS AlbumLength
FROM Tracks
GROUP BY AlbumFk
```

2. Mit Albumnamen:

Listing 6.27: 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
```

3. HAVING-Filter:

Listing 6.28: Albumlänge mit HAVING-Filter

```
SELECT AlbumFk,
   SUM(DurationMinutes) AS AlbumLength
```

```
3 FROM Tracks
4 GROUP BY AlbumFk
5 HAVING AVG(DurationMinutes) < 180
```

4. 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.

5. Bonustrack-Abfrage:

Listing 6.29: Bonustrack-Abfrage

```
1 SELECT AlbumFk
2 FROM Tracks
3 GROUP BY AlbumFk
4 HAVING COUNT(*) < MAX(TrackId)
```

6. Bestellungen pro Kunde:

Listing 6.30: Bestellungen pro Kunde

```
1 SELECT b.KundenFk, COUNT(*)
2 FROM Bestellung
3 GROUP BY KundenFk
```

7. 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.

Listing 6.31: Bestellungen

```
SELECT b.KundenFk, b.BarFk, bbGrouped.TotalPrice
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
```

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

Listing 6.32: Umsatz pro Bar

```
SELECT b.BarFk,
  SUM(bbGrouped.TotalPrice) AS Neujahrsumsatz
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.BarFK
```

9. Und wieder können wir die vorhergehende Abfrage wiederverwenden:

Listing 6.33: Trinkfreudige Kunden

```
SELECT k.Name
FROM Kunde k
INNER JOIN
( SELECT b.KundenFk,
    SUM(bbGrouped.TotalPrice) AS Neujahrsausgaben
  FROM Bestellung b INNER JOIN
```

```
7     (SELECT bb.BestellungFk,
8       SUM(bb.Anzahl * s.Preis) AS TotalPrice
9     FROM BestellteBier bb
10      INNER JOIN Sortiment s
11        ON s.SortimentId = bb.SortimentFk
12    GROUP BY bb.BestellungFk
13    ) bbGrouped
14  ON bbGrouped.BestellungFk = b.BestellungId
15  WHERE b.Zeit > '2014-31-12 18:00'
16    AND b.Zeit < '2015-01-01 06:00'
17  GROUP BY b.KundenFk) ka
18 ON ka.KundenFk = k.KundenId
19 WHERE ka.Neujahrsausgaben > 20
```

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

Listing 6.34: Günstigster Preis

```
1 SELECT b.Name, billigBier.BilligPreis
2 FROM Bier b
3 INNER JOIN
4   (SELECT BierFk, MIN(Preis) AS BilligPreis
5   FROM Sortiment
6   WHERE Deleted = 0
7   GROUP BY BierFk) billigBier
8   ON billigBier.BierFk = b.BierId
```