Fachhochschule Südwestfalen
University of Applied Sciences
Standort Hagen
Fachbereich Technische Betriebswirtschaft

Datenbanken

Vorlesungsskript für das 3. Semester

Andreas de Vries

Version: 17. Januar 2025

Inhaltsverzeichnis

1 Datenbankmanagementsysteme (DBMS) 6
- 1.1 Die Hierarchie der Speicherkonzepte . . . 6
- 1.2 Die ANSI-SPARC-Architektur . . . 8
- 1.3 Relationen . . . 9

2 Die Sprachelemente von SQL 11
- 2.1 * Informatischer Überblick . . . 11
- 2.2 Reservierte Wörter . . . 12
- 2.3 Kommentare . . . 12
- 2.4 Datentypen . . . 13
- 2.5 Anweisungen . . . 14
- 2.6 Funktionen . . . 16

3 Die `SELECT`-Anweisung 18
- 3.1 Einfache `SELECT`-Anweisungen . . . 18
- 3.2 Alias-Spalten mit `AS` . . . 21
- 3.3 Auswahlfilter: Die `WHERE`-Klausel . . . 22
- 3.4 Textsuche mit `LIKE` . . . 24
- 3.5 Sortierung mit `ORDER BY` . . . 25

4 `NULL`-Werte und dreiwertige Logik 28
- 4.1 Unbekannte Information . . . 28
- 4.2 Dreiwertige Logik . . . 29
- 4.3 Abfragen auf `NULL`-Werte . . . 31
- 4.4 Setzen von Standardwerten bei `NULL`-Einträgen . . . 32

5 Primärschlüssel: Identifikation von Datensätzen 33
- 5.1 Schlüssel . . . 33
- 5.2 Wahl eines natürlichen Primärschlüssels . . . 34
- 5.3 Primärschlüssel und `NULL`-Werte . . . 35
- 5.4 Künstliche Primärschlüssel . . . 36

6 Daten analysieren und zusammenfassen (`GROUP BY`) 38
- 6.1 Aggregatfunktionen . . . 38
- 6.2 Gruppieren nach Spalten mit `GROUP BY` . . . 40
- 6.3 Gruppieren mit `HAVING`: Filtern mit Aggregatfunktionen . . . 41
- 6.4 `WHERE` oder `HAVING`? . . . 42
- 6.5 * Der `OVER-PARTITION`-Ausdruck . . . 43

7 Mengenoperationen 46
- 7.1 Mengenlehre: Mathematik der Mengen . . . 46
- 7.2 Mengenoperationen in SQL . . . 47

7.3 Verschachtelte `SELECT`-Anweisungen: `SELECT` in `SELECT` 49

8 Entity-Relationship Modelle **53**
8.1 Probleme mit Daten in einer einzigen Tabelle 53
8.2 Datenmodellierung 55
8.3 ER-Diagramme 56

9 Ableitung von Tabellen aus dem ERM **61**
9.1 Fremdschlüssel 61
9.2 Grundregeln der Implementierung von Beziehungen 63
9.3 Die Hauptbeziehung 1:N 64
9.4 Die Hauptbeziehung M:N 66
9.5 Die Hauptbeziehung 1:1 69
9.6 Schlecht oder gar nicht implementierbare Beziehungen 71
9.7 Spezielle Beziehungen 72
9.8 SQL mit mehreren Tabellen 77

10 Normalisierung **81**
10.1 Anomalien 81
10.2 Die ersten vier Normalformen 84
10.3 Anwendungsfall: Die Comic-Alben 86
10.4 Zusammenfassung 88

11 Joins **89**
11.1 Inner Joins 89
11.2 Left und Right Joins 91
11.3 Joins mit mehr als zwei Tabellen 96
11.4 Self Joins 98

12 Views **100**
12.1 Was sind Views? 100
12.2 Warum Views? 101
12.3 Beispiele für Views 101
12.4 Änderungen von Daten über Views 103

13 * Rekursionen **104**
13.1 Die Grundlage: Common Table Expressions mit `WITH` 104
13.2 Was sind Rekursionen? 105
13.3 Rekursionen mit SQL 106
13.4 Anwendung auf Graphen und Netzwerke 109
13.5 * Die Turing-Vollständigkeit von SQL 110

A Anhang **114**
A.1 Übersicht der SQL-Anweisungen 114
A.2 Die 12 Codd'schen Regeln 116
A.3 Relationale Vollständigkeit 120

Bibliography **123**

Internetquellen **123**

Index **124**

Vorwort

Das vorliegende Skript ist Grundlage der Vorlesung Datenbanken, die ich seit dem Wintersemester 2019/20 am Fachbereich Technische Betriebswirtschaft der FH Südwestfalen in Hagen halte. Der Stoff basiert auf den Vorlesungen meiner Vorgänger Hermann Johannes und Stefan Böcker, deren Schwerpunkte die Datenmodellierung mit Entity-Relationship-Diagrammen und die Anzeige von Datenbankeinträgen mit SQL bildeten.

Die Inhalte habe ich im Wesentlichen beibehalten. Einige Ergänzungen gehen auf Anregungen meines Mitarbeiters Ingo Schröder und auf das Lehrbuch *Grundkurs Datenbanksysteme* von Lothar Piepmeyer[1] zurück. Allerdings wurde die Vermittlung des Stoffs in eine andere Reihenfolge als bisher üblich gebracht. Meine Vorgänger hatten die auch in realen Projekten angewendete Vorgehensweise gewählt und behandelten zunächst die Datenmodellierung mit ER-Diagrammen, ergänzt um die Theorie zur effizienten und redundanzfreien Datenspeicherung. Erst danach wurde die Umsetzung von Datenmodellen in Tabellen und die Programmierung mit SQL besprochen. Der große Vorteil dieses Zugangs ist, dass man von Beginn an mit realistischen Anwendungsfällen hoher Komplexität arbeiten kann.

Als Nachteil erscheint mir jedoch, dass die Studierenden direkt mit recht abstrakten Datenmodellen und Entitätsbeziehungen umgehen müssen, ohne die datentechnische Basis der Tabellen und deren Implementierung mit SQL zu kennen. Dreht man das Konzept jedoch einfach um und führt die Programmierung mit SQL vor der Datenmodellierung ein, so verstehen die Studierenden ohne Kenntnis des Entitätsbegriffs nur schwer das schnell recht komplexe Zusammenspiel mehrerer Tabellen.

Der Zugang dieses Skripts ist daher ein Kompromiss, der die Nachteile der beiden Vorgehensweisen vermeidet und die Vorteile erhält. Zunächst wird eine Einführung in SQL und das Arbeiten mit einzelnen Tabellen behandelt, bevor dann die Modellierung mit Entity-Relationship-Diagrammen und schließlich die Implementierung und Analyse mehrerer Tabellen mit SQL durchgenommen wird. So werden Funktionsweise und Eigenschaften von Tabellen auf elementare Weise vermittelt, bevor komplexere Konzepte der Beziehungen zwischen Tabellen angegangen werden, immer im Wechselspiel von Theorie und deren Umsetzung in SQL.

Was ist davon klausurrelevant?

In der Regel interessiert die Studentinnen und Studenten bei der Bereitstellung von Lehrmaterial vor allem die Frage: Was ist davon klausurrelevant? Die Standardantwort eines Professors darauf lautet in der Regel dann: Natürlich alles! Hier allerdings möchte ich eine etwas differenziertere Antwort geben. Diese Skript ist inhaltlich die Obermenge des Stoffs, der in der Klausur „drankommen“ kann. D.h. es werden darin manche Themen angerissen, die zum Bestehen des Faches tatsächlich nicht notwendig sind, sondern Ergänzungen und Hintergrundinformationen für die Interessierten oder Neugierigen darstellen. Sie sind in der Regel mit einem Sternchen (*) gekennzeichnet, stehen in den Fußnoten oder sind eingeklammert.

[1] Piepmeyer (2011).

Warum ein Skript?

Diese Frage wird mir oft von Kolleginnen und Kollegen gestellt; dahinter steht in der Regel die Befürchtung, dass dann ja keiner mehr in die Vorlesung kommt (für die es im Übrigen keine Anwesenheitspflicht gibt und nach meiner Überzeugung auch nicht geben sollte). Meine Erfahrungen mit ausformulierten Skripten sind jedoch etwas anders. Zwar ist spätestens ab etwa Mitte des Semesters ein Einbruch der Teilnahme auch an meinen Vorlesungen zu bemerken, jedoch gibt ein – zugegeben nur wahrgenommener und quantitativ nicht überprüfter – Vergleich mit Vorlesungen ohne Skript kein signifikant anderes Resultat. Leider ist es offenbar so, dass das Format Vorlesung als nicht wichtig oder effizient genug betrachtet wird.

In meinen Augen ist diese Einschätzung ein Fehler. Sicher ist eine Vorlesung nicht *die eine* für den Lernerfolg wichtige Lehrform. Noch wichtiger dafür ist sicher die Teilnahme an den Übungen und das Lösen der Aufgaben, bei auftretenden Problemen unter Anleitung und mit Hinweisen der Lehrenden. In den Übungen gelingt aber vornehmlich das praktische Tun, die Umsetzung von Konzepten. Zu diesen Konzepten gelangt man jedoch oft nur mit einem nichttrivialen theoretischen Hintergrund. Idealerweise gelingt die Vermittlung der Theorie in einer Vorlesung, und zwar im Dialog zwischen Lernenden und Lehrendem. Natürlich ist ein solcher Dialog in der Realität bei über 150 Teilnehmern nur mit großen Einschränkungen möglich. Aber es gar nicht erst zu versuchen, ist für mich keine Option.

Für hinreichend komplexe Inhalte ist zur Ergänzung von Vorlesung und Übungen ein ausformuliertes Skript, oder ein Lehrbuch, unerlässlich. Zwar gibt es eine ganze Reihe empfehlenswerter Lehrbücher über Datenbanken, die die in diesem Skript dargestellten Inhalte ergänzen und vertiefen, beispielsweise Heuer et al. (2020) und Piepmeyer (2011). Wenn ich ein Lehrbuch gefunden hätte, das meinen Vorstellungen dieser Lehrveranstaltung in allen Belangen entspricht, hätte ich es verwendet.

Hagen,
im Januar 2025 Andreas de Vries

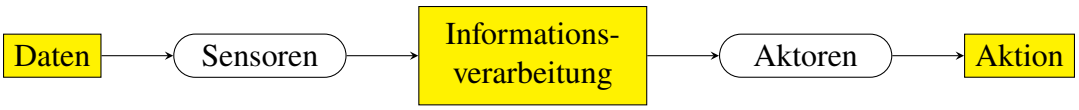
1 Datenbankmanagementsysteme (DBMS)

Kapitelübersicht

1.1 Die Hierarchie der Speicherkonzepte 6
1.2 Die ANSI-SPARC-Architektur 8
1.3 Relationen 9

1.1 Die Hierarchie der Speicherkonzepte

Ein wichtiges Merkmal aller Lebewesen ist es, Daten aus der Umwelt als Information zu verarbeiten und daraus Aktionen abzuleiten.



Bit: Einheit einer Datenmenge

Die Daten sind hierbei Reize der Umgebung, sei es in Form elektromagnetischer oder akustischer Wellen, chemischer Reaktionen oder mechanischen Drucks. In der Informatik wird eine Datenmenge in Bits gemessen, der Einheit des Informationsgehalts. Höhere Lebewesen filtern aus diesen Daten durch ihr zentrales Nervensystem Information. Beim Menschen geschieht dies zum allergrößten Teil unbewusst. Wie aus Tabelle 1.1 ersichtlich, werden von den etwa 11,2 Megabits, die unsere Sinnesorgane pro Sekunde empfangen können, nur etwa 77 Bits bewusst wahrgenommen, also etwa 0,01 ‰ = 10^{-5}. Zum Vergleich dazu hat ein Farbfernseher eine Datenrate von ungefähr 10 Mbit/s, also etwa die Aufnahmekapazität unserer Augen, ein Radio hat eine Datenrate von ca. 10 kbit/s, d.h. etwa 10 % der Kapazität unserer Ohren, und ein laut gelesener Text hat eine Datenrate von etwa 25 bit/s.[1]

An einem Tag mit 16 Stunden Helligkeit nehmen also allein unsere Augen eine Datenmenge von etwa 72 GB auf, bewusst wahrgenommen werden davon lediglich 288 kB. Die Speicherkapazität eines menschlichen Gehirns wird auf etwa 60 TB geschätzt.[2]

Datenspeicherung

Ganz ähnlich wie biologische Organismen verarbeiten auch unsere technischen Informationssysteme Daten zu Information und speichern sie. Abhängig von Menge und Komplexität

[1] vgl. Nørretranders (1997):S. 227.

[2] Nach Bartol Jr. et al. (2015) kann eine Synapse, also die Verbindung zwischen zwei Neuronen, Informationen in 26 Zuständen digital speichern, d.h. sie hat eine Speicherkapazität von 4,7 = $\log_2 26$ bits. Bei etwa 100 Milliarden = 10^{11} Neuronen mit jeweils etwa 1000 Synapsen ergibt dies $470 \cdot 10^{12}$ bits oder 470/8 ≈ 60 TB.

Sinnessystem	Datenrate [bit/s]	
	Kapazität	Bewusstsein
Augen	10 000 000	40
Ohren	100 000	30
Haut	1 000 000	5
Geschmack	1 000	1 (?)
Geruch	100 000	1 (?)
Gesamt	11 201 000	77

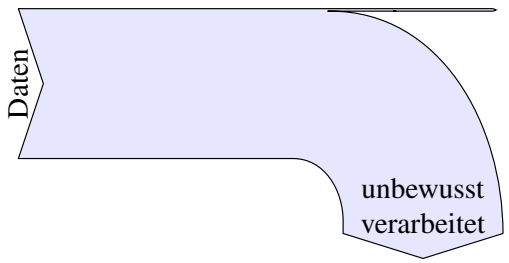


Tabelle 1.1: Informationsfluss der Sinnesorgane und des Bewusstseins. Schätzwerte sind mit "(?)" gekennzeichnet. Quelle: Zimmermann (1993)

verwendet man in der Informatik dazu verschiedene Speicherkonzepte, die eine Hierarchie hinsichtlich ihres Strukturierungsgrades bildet (Abbildung 1.1). Die einfachste Struktur sind die

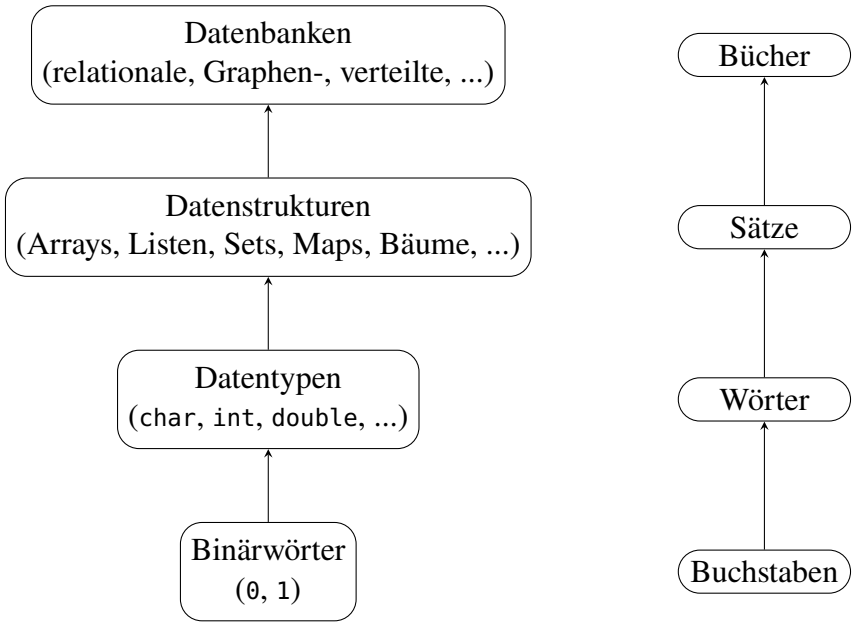


Abbildung 1.1: Die Hierarchie der Speicherkonzepte in der Informatik (links) und in der Literatur (rechts)

Binärwörter, die sich nur aus Nullen und Einsen zusammensetzen. Sie bilden die logische Grundlage aller Datenspeicher, d.h. alle Daten werden am Ende durch sie gespeichert. Das entspricht in etwa der Tatsache, dass geschriebene Texte als Basis mit Buchstaben geschrieben („gespeichert") sind. Für sich alleine genommen haben Binärwörter zwar keine eigentliche Bedeutung, dafür können sie aber auch Daten aller Art speichern. Binärwort

Die nächste Strukturierungsstufe sind die *Datentypen*, die Daten in Symbole, ganze Zahlen oder Kommazahlen unterscheiden und als Binärwörter fester Länge speichern. Datentypen sind die Basis für die Programmiersprachen und zum großen Teil standardisiert: Symbole sind üblicherweise durch den Unicode UTF-8 der Längen 1 bis 4 Byte kodiert (früher ASCII mit 7 bits), ganze Zahlen als Integer der Größe 4 Byte und Kommazahlen als Gleitkommazahlen nach dem Format IEEE 754 mit 4 Byte (`single` oder `float`) oder 8 Byte (`double`). Analog einem geschriebenen Text bilden sie als Struktureinheiten Wörter, die aus den Buchstaben gebildet sind. Datentyp

Werte von Datentypen können zu Einheiten sogenannter *Datenstrukturen* zusammengefasst werden. Die häufigste Datenstruktur der Informatik ist das Array. Die API's fast aller gängigen Programmiersprachen bieten darüber hinaus Listen und Zuordnungen („Maps") an. Datenstruktur

Zur strukturierten und vor allem persistenten – d.h. dauerhaften – Speicherung großer und komplexer Datenmengen eignen sich diese Datenstrukturen allerdings nicht. Oft sollen vorhandene Datenbestände außerdem noch für mehrere Nutzer gleichzeitig verfügbar sein, wobei die Lese- und Schreibrechte individuell festzulegen sind. Für diese Zwecke werden *Datenbanken* eingesetzt. Abhängig von der Struktur der Daten und den Zielsetzungen der Anwendung gibt es dazu verschiedene Datenbanktypen. Das Spektrum reicht zum Beispiel von relationalen Datenbanken auf zentralen Servern über verteilte Datenbanken, die auf mehreren Rechnern oder sogar auf Rechnerverbünden laufen, bis hin zu Graphendatenbanken, die sich für eine netzorientierte Datenorganisation eignen. Der für betriebswirtschaftliche Anwendungen seit den 1970er Jahren wichtigste Datenbanktyp jedoch sind die relationalen Datenbanken. Sie sind Hauptthema dieses Skriptes.

Datenbank

1.2 Die ANSI-SPARC-Architektur

Interessanterweise folgte die historische Entwicklung der Datenspeicherung im Wesentlichen genau der Speicherkonzepthierarchie in Abbildung 1.1, von unten nach oben. In den 1960er Jahren wurde Datenspeicherung allgemein auf Basis von Datenstrukturen wie Arrays, Listen und Maps realisiert. Durch die zunehmende Komplexität der anfallenden Daten und sich oft erweiternde Anforderungen an sie kam es zu immer ernsteren Problemen, zu deren Lösung schließlich 1975 die dreischichtige Referenzarchitektur des ANSI-SPARC eingeführt wurde. Dieses Modell bildete den abstrakten Entwurf der in der Folgezeit realisierten Datenbanken. Es ist in Abbildung 1.2 skizziert. Es ist ein Schichtenmodell, in dem jede Schicht ihre speziellen Aufgaben

dreischichtige Referenzarchitektur

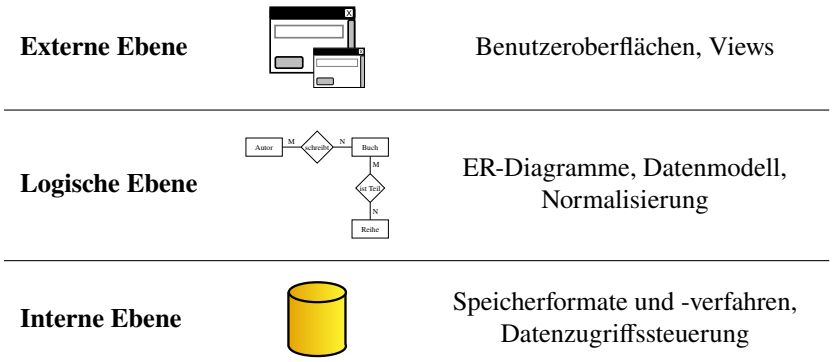


Abbildung 1.2: Die drei Schichten der ANSI-SPARC-Architektur

hat und mit den angrenzenden Schichten über festgelegte Protokollschnittstellen kommuniziert. Dadurch ist es problemlos möglich, in jeder Schicht separat Änderungen vorzunehmen, solange sie die Schnittstellen korrekt bedienen. Ein solches Schichtenmodell ist beispielsweise auch die TCP/IP-Architektur von 1973, die später die Grundlage des Internets wurde.

Die ANSI-SPARC-Architektur fordert von einer allgemeinen Datenbank, dass sie aus drei Ebenen besteht: Die externe Ebene ist die Schnittstelle zu den Anwendern und stellt die individuellen Benutzersichten bereit, beispielsweise Formulare, Eingabemasken oder Ausgabelisten für die verschiedenen Nutzergruppen. In der logischen oder konzeptionellen Ebene werden die zu speichernden Daten beschrieben und mit dem Ziel modelliert, sie effizient verwaltbar und zugreifbar zu machen und gleichzeitig den Anwenderanforderungen zu genügen. Die interne Schicht schließlich stellt die physikalische Sicht auf das Datenbanksystem dar und beschreibt, in welchen Dateien welche Daten genau gespeichert werden, wie bei Mehrbenutzersystemen die lesenden und schreibenden Zugriffe geregelt werden usw..

Die ANSI-SPARC-Architektur wurde zwar nie ein offizieller Standard. Auch kann bei der Implementierung eine klare Trennung zwischen den Ebenen kaum gelingen; insbesondere sind die externe und die logische Ebene eng verzahnt, so dass eine ideale logische Datenunabhängigkeit unrealistisch ist. Dennoch gilt ANSI-SPARC noch heute als das ideale Entwurfsmodell für ein Datenbanksystem[3]. Wir definieren daher den exakten Begriff Datenbanksystem wie folgt:

Definition 1.1. Ein *Datenbanksystem* ist ein Softwaresystem zur Speicherung und Verwaltung strukturierter Daten, das die drei Ebenen der ANSI-SPARC-Architektur realisiert. Die Menge der gespeicherten Daten heißt *Datenbank* oder auch *Datenbasis*, die Menge an Programmen zum Zugriff auf die Daten heißt *Datenbankmanagementsystem (DBMS)*.[4] □

Datenbank

DBMS

1.3 Relationen

Ende der 1960er Jahre entwickelte der IBM-Mitarbeiter Edgar F. „Ted“ Codd ein Datenbankmodell, das zur Strukturierung von Daten Tabellen nutzt. Er realisierte damit das in der Mathematik des 19. und 20. Jahrhunderts intensiv untersuchte Konzept der *Relationen*. Eine Relation R ist mathematisch eine Teilmenge des kartesischen Produkts gegebener Mengen A_1, ..., A_n,

Relation R

$$R \subseteq A_1 \times \ldots \times A_n. \tag{1.1}$$

Formal lassen sich die Elemente der Relation als n-Tupel schreiben:

$$(x_1, \ldots, x_n) \qquad \text{mit} \qquad x_j \in A_j \text{ für } j = 1, \ldots, n. \tag{1.2}$$

Beispiel 1.2. [5] Betrachten wir die folgenden zwei Mengen, die jeweils eine der Eigenschaften A_1 = „Farbe“ und A_2 = „Karte“ einer Spielkarte beschreiben.

$$A_1 = \{\text{Kreuz, Pik, Herz, Karo}\}, \quad A_2 = \{2, 3, 4, 5, 6, 7, 8, 9, 10, \text{Bube, Dame, König, Ass}\}.$$

Das kartesische Produkt $A_1 \times A_2$ beider Mengen ist dann die Menge aller $4 \cdot 13 = 52$ Kombinationen der beiden Eigenschaften, also mit anderen Worten ein komplettes Pokerblatt. Dann ist das folgende „Full House“ R eine Relation dieser beiden Mengen:

$$R = \{(\text{Pik, Ass}), (\text{Kreuz, Ass}), (\text{Herz, Ass}), (\text{Herz, König}), (\text{Karo, König})\}, \tag{1.3}$$

denn $R \subset A_1 \times A_2$ ist eine Teilmenge des kartesischen Produkts $A_1 \times A_2$. Sie kann äquivalent auch durch eine Tabelle beschrieben werden:

R

A_1	A_2
Pik	Ass
Kreuz	Ass
Herz	Ass
Herz	König
Karo	König

oder eben:

Full House

Farbe	Karte
Pik	Ass
Kreuz	Ass
Herz	Ass
Herz	König
Karo	König

Die Menge *aller* Full Houses ist ebenfalls eine Relation von $A_1 \times A_2$. □

[3] Piepmeyer (2011):S. 16.
[4] vgl. Krcmar (2015):§6.1.3.1.
[5] nach Piepmeyer (2011):S. 36f.

Bemerkung 1.3. Eine Relation von (endlichen) Mengen mit m Elementen lässt sich wie in obigem Beispiel 1.2 immer als eine Tabelle darstellen, deren Spalten die n Mengen des kartesischen Produkts beschreiben und deren Zeilen die Kombinationen der Werte sind. Umgekehrt ist jede Tabelle eine endliche Relation, wenn man vereinbart, dass nur Tabellen ohne doppelte Zeilen betrachtet werden und zudem zwei Tabellen als gleich angesehen werden, wenn sie sich nur durch Zeilenvertauschungen unterscheiden:[6]

$$R = \{(x_{i1}, \ldots, x_{in}) : i = 1, \ldots, m \text{ und } x_{ij} \in A_j \text{ für } j = 1, \ldots, n\} \quad \Longleftrightarrow \quad \begin{array}{|cccc|} \hline A_1 & A_2 & \cdots & A_n \\ \hline x_{11} & x_{12} & \cdots & x_{1n} \\ \vdots & \vdots & & \vdots \\ x_{i1} & x_{i2} & \cdots & x_{in} \\ \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \\ \hline \end{array} \tag{1.4}$$

In der Informatik werden die Spalten solcher Tabellen auch als *Felder* oder *Attribute* bezeichnet, eine Zeile als *Datensatz*. In der folgenden Übersicht sind diese Grundbegriffe für Daten einer relationalen Datenbank tabellarisch gegenübergestellt und mit dem Objektbegriff der objektorientierten Programmierung verglichen.

Relationales Modell (Mathematik)	**Tabellenmodell** (Angewandte Informatik)	**Objektmodell** (Programmierung)
Relation *(relation)*	Tabelle *(table)*	Klasse *(class)*
Attribut *(attribute)*	Spalte *(column)* / Feld *(field)*	Attribut *(attribute)*
Tupel *(tuple)*	Zeile *(row)* / Datensatz *(record set)*	Objekt *(object)*

Das Objektmodell wird uns in diesem Skript allerdings nicht weiter beschäftigen. Es ist hier erwähnt, um die enge Verwandtschaft der Begriffe Relation, Tabelle und Objekt aus den Bereichen Mathematik, Angewandte Informatik und Programmierung zu zeigen. Da ferner die Begriffe in der Fachliteratur häufig geeignet ausgetauscht werden – z.B. wird oft gesagt, eine Tabelle habe Attribute – haben wir hier eine Art „Vokabelliste“. □

Relationentyp

Definition 1.4. Der *Relationentyp* ist definiert als die Kombination des Namens R der Relation und der Auflistung der Mengen $(A_1, \ldots, A_n)$ und wird

$$R(A_1, A_2, \ldots, A_n) \tag{1.5}$$

geschrieben. Ähnlich wie der Datentyp in der Datenhierarchie Abbildung 1.1 ist der Relationentyp also das „Format“ einer Relation. Im Zusammenhang mit Datenbanken spricht man oft auch von dem *Schema* der Relation, manchmal spricht man auch von der *Struktur der Tabelle* oder der *Tabellenstruktur*. Die Notation (1.5) wird später bei der Programmierung von Tabellen noch eine wichtige Rolle spielen. □

Schema

Tabellenstruktur

Beispiel 1.5. Der Relationentyp der Relation „Full House“ aus Beispiel 1.2 lautet

$$\text{Full House (Farbe, Karte).} \tag{1.6}$$

An dieser Notation kann man ablesen, dass die Relation ein 2-Tupel (x_1, x_2) ist, dessen erster Wert x_1 eine Farbe (A_1) und dessen zweiter, x_2, eine Karte (A_2) ist. □

In einem später berühmt gewordenen Fachartikel[7] veröffentlichte Codd 1970 seine Gedanken zu Relationen als Grundlage für ein Datenbankmodell. Das Modell fand schnell weite Beachtung. Heute werden auf Tabellen basierende Datenbanken *relational* genannt, und das Verwaltungssystem entsprechend *RDBMS* (für *relationales Datenbankmanagementsystem*). Die Begriffe der Relationen und der Tabellen beziehen sich dabei genau genommen auf die logische Ebene der ANSI-SPARC-Architektur in Abbildung 1.2.

RDBMS

[6] vgl. Piepmeyer (2011):S. 37.

[7] Codd (1970).

2 Die Sprachelemente von SQL

Kapitelübersicht

2.1 * Informatischer Überblick 11
2.2 Reservierte Wörter 12
2.3 Kommentare 12
2.4 Datentypen 13
2.5 Anweisungen 14
2.6 Funktionen 16

SQL (Structured Query Language) ist eine Programmiersprache zur Definition von Datenstrukturen und zur Datenverwaltung relationaler Datenbanken. Es erschien erstmals 1974 und wurde 1986 ein Standard des ANSI, der Normbehörde der USA, sowie ein Jahr später der ISO. Die 2020 aktuelle Version ist SQL:2016.[1] Wir werden in diesem Kapitel eine Einführung in diese Programmiersprache kennenlernen.

2.1 * Informatischer Überblick

deklarative Programmiersprache

SQL ist eine „deklarative" Programmiersprache, d.h. im Gegensatz zu „imperativen" Sprachen wie Java, C oder Visual Basic wird nicht der Kontrollfluss, also der Ablauf des Programms programmiert, sondern nur „ergebnisorientiert" die Logik und die Funktionalität des Programms. „Im Gegensatz zur imperativen Programmierung, bei der das *Wie* im Vordergrund steht, fragt man in der deklarativen Programmierung nach dem *Was*, das berechnet werden soll."[2]

Turing-vollständig

Ferner ist SQL „Turing-vollständig", d.h. eine Programmiersprache, die logische `WHILE`-Schleifen ermöglicht, also Schleifen, die grundsätzlich unendlich oft durchlaufen werden können. Damit können alle berechenbaren Funktionen implementiert werden.[3] Siehe dazu auch Abschnitt 13.5 ab Seite 110 und de Vries (2022:§5.4).

Interpreter

SQL hat im Wesentlichen die Merkmale einer Interpretersprache, d.h. der Quelltext wird zur Laufzeit direkt übersetzt und ausgeführt. Die meisten RDBMS allerdings übersetzen den Quelltext intern in einen „Syntaxbaum" oder einen „Query Execution Plan", der als temporäre Datei zunächst gespeichert und dann ausgeführt wird (also ähnlich dem Bytecode von Java oder

[1] `https://en.wikipedia.org/wiki/SQL`
[2] `https://de.wikipedia.org/wiki/Deklarative_Programmierung`
[3] `https://stackoverflow.com/questions/900055/`, [Win]

C#).[4] In diesen Fällen ist SQL also eigentlich auch eine Compilersprache, allerdings ist die kompilierte Datei nur DBMS-intern ausführbar.

2.2 Reservierte Wörter

Jede Programmiersprache besitzt ein festes „Vokabular“ von sogenannten *reservierten Wörtern*.
Syntax Sie haben eine festgelegte Bedeutung und dürfen nur nach bestimmten Grammatikregeln, der *Syntax*, kombiniert werden. Die wichtigsten reservierten Wörter von SQL, die auch auf fast allen Datenbanksystemen funktionieren, sind in Tabelle 2.1 aufgeführt. Sie umfasst ebenso

Wichtige reservierte Wörter in SQL

ABS	ADD	ALL	ALTER	AND	ANY	AS	AVG	BETWEEN	BY
CHECK	COLLATE	CONCAT	CONSTRAINT	COUNT	CREATE	DATE	DOMAIN	DOUBLE	DECIMAL
DELETE	DESC	DISTINCT	DROP	EXCEPT	EXISTS	FOREIGN	FROM	GRANT	GROUP
HAVING	IN	INNER	INSERT	INTEGER	INTERSECT	INTO	IS	JOIN	LEFT
LIKE	MAX	MIN	NOT	NULL	OF	ON	OR	ORDER	OUTER
POSITION	POWER	PRIMARY	RECURSIVE	REFERENCES	RESTRICT	REVOKE	RIGHT	ROUND	SELECT
SET	SMALLINT	SQRT	START	STDDEV	SUBSTRING	SUM	TABLE	TIME	TIMESTAMP
UNION	UNIQUE	UPDATE	USER	VALUES	VARCHAR	VARIANCE	VIEW	WHERE	WITH

Literale

NULL | 0, 1, -2 | 1.2, 3.1E8	'Abc'

Besondere Zeichen

--	*	+, - , /, %	=, !=, <>, >, <, >=, <=

Tabelle 2.1: Wichtige reservierte Wörter und Literale von SQL

Literale, also Zahlen und Zeichenfolgen, die einen festen Wert darstellen. Die in Apostrophs
String eingeklammerten Zeichenfolgen werden *Strings* genannt. Daneben gibt es Zeichenfolgen mit besonderer Bedeutung wie Wildcards, Kommentarmarkierungen oder Rechenoperationen.

SQL ist nicht schreibungssensitiv *(not case sensitive)*, d.h. es unterscheidet nicht zwischen Groß- und Kleinschreibung. (Bei einigen RDBMS, z.B. MariaDB auf Linux, wird die Schreibung von Tabellennamen allerdings sehr wohl unterschieden.) In diesem Skript wird die Konvention verwendet, reservierte Wörter in SQL in Großbuchstaben zu schreiben.

Obwohl SQL standardisiert ist, existieren viele Dialekte für die verschiedenen Datenbanksysteme, so dass ein einmal geschriebener SQL-Quelltext nicht unbedingt auf allen Datenbanken läuft. Der zentrale Grund ist, dass die reservierten Wörter zu einem gewissen Teil nicht identisch sind. Allein schon die Anzahl der reservierten Wörter variiert zwischen den verschiedenen Datenbanksystemen:

Anzahl reservierter Wörter

OpenOffice Base*	Microsoft Access	MariaDB* / MySQL	PostgreSQL*	Azure SQL / SQL Server	Oracle
61	252	238	93	185	110

* Open Source / freie Software

Übrigens zum Vergleich: Übliche (imperative) Programmiersprachen haben deutlich weniger reservierte Wörter, z.B. hat Java 49, Python nur 30.

2.3 Kommentare

Wie in jeder Programmiersprache gibt es auch in SQL die Möglichkeit, Kommentare einzufügen, um anderen Programmierer*innen (und sich selbst) Hinweise zum Zweck oder zur Funktions-

[4] https://www.quora.com/Is-SQL-interpreted-or-compiled

weise des Programms zu geben. Kommentare werden von dem Interpreter ignoriert und daher nicht ausgeführt. Es gibt zwei Arten von Kommentaren in SQL, den einzeiligen Kommentar,

```
-- Dies ist ein Kommentar
```

der durch zwei Minuszeichen für den nachfolgenden Rest der Zeile markiert wird, und der mehrzeilige (oder Block-) Kommentar,

```
/* Dies ist ein Kommentar,
   der über mehrere Zeilen geht
 */
```

dessen Bereich mit `/*` geöffnet und mit `*/` geschlossen wird.

2.4 Datentypen

Datentypen sind die elementaren Speichereinheiten einer Programmiersprache, die jeweils einen bestimmten Speicherbereich zugewiesen bekommen (vgl. auch Abbildung 1.1 auf Seite 7). Datentypen legen die Wertebereiche dieser Einheiten fest. Es gibt in der Regel drei Kategorien von Datentypen, Zeichen *(character)*, Ganzzahlen *(integer)* und Gleitkommazahlen *(float, double)*. Aus der Aneinanderreihung von Zeichen entsteht Text *(string)* als ein zusammengesetzter Datentyp. Vgl. dazu die Hierarchie der Speicherkonzepte in Abbildung 1.1 auf Seite 7.

In SQL werden je nach RDBMS unterschiedliche Datentypen festgelegt. Aber gängige und in fast allen Dialekten verwendete sind diejenigen in Tabelle 2.2. Die Tabelle ist in die vier Bereiche

Datentyp	Speicher	Beschreibung
`varchar(`n`)`	n Byte	Text (String) mit Maximallänge n; wird mit Apostrophs `'Abc'` gekennzeichnet
`text`	x Byte	Text beliebiger Länge; wird mit Apostrophs `'Abc'` gekennzeichnet
`smallint`	2 Byte	Ganzzahl im Bereich von −32 768 bis 32 767 (= -2^{15} bis $2^{15}-1$)
`int`	4 Byte	Ganzzahl im Bereich von −2 147 483 648 bis 2 147 483 647 (= -2^{31} bis $2^{31}-1$)
`bigint`	8 Byte	Ganzzahl im Bereich von −9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807 (= -2^{63} bis $2^{63}-1$)
`float`	4 Byte	Gleitkommazahl im Bereich von `1.0E-38` bis `3E+38` (= 2^{-126} bis 2^{128}), auf bis zu 8 Dezimalstellen genau
`double`	8 Byte	Gleitkommazahl mit einem Betrag von `2E-308` bis `2E+308` (= 2^{-1022} bis 2^{1024}) auf bis zu 16 Dezimalstellen genau
`decimal`(p,s)	$f(p)$ Byte*	Festkommazahl mit maximal p Stellen, davon s Nachkommastellen; Restriktionen: $p \leqq 38$ bei SQL Server ($p \leqq 65$ bei MariaDB) und $0 \leqq s \leqq p$
`date`	3 Byte	Datum, je nach RDBMS im Format `'yyyy-mm-dd'` oder `#mm/dd/yyyy#`
`time`	4 Byte	Zeit, je nach RDBMS meist im Format HH:MM:SS
`timestamp`	8 Byte	Datum und Uhrzeit im Format `'yyyy-mm-dd hh:mm:ss'` (SQL Server: `datetime`)

Tabelle 2.2: Gängige Datentypen in SQL. * Erläuterung: $f(p) = 1 + 4 \cdot \left\lceil \frac{p}{9{,}5} \right\rceil$

Strings, Ganzzahlen, Gleitkommazahlen und Datum/Zeit unterteilt. Zeichen und Text werden in SQL mit Apostrophs `'Abc'` gekennzeichnet, Kommazahlen mit einem Punkt als Dezimaltrenner. Siehe dazu auch die Übersicht über die Literale in der obigen Tabelle 2.1.

Besonderheiten beispielsweise von MS Access sind, dass dort der Datentyp `Integer` nur 2 Byte hat und der Datentyp `Long Integer` 4 Byte (wie hier `int`); ferner heißt der Datentyp `float` dort `Single`.[5] Für weitere Information zu den Datentypen seien die Dokumentationen

`https://de.wikibooks.org/wiki/Einf%C3%BChrung_in_SQL:_Datentypen`

[5] `https://support.office.com/en-us/article/30ad644f-946c-442e-8bd2-be067361987c`; im Widerspruch dazu jedoch `https://docs.microsoft.com/en-us/office/client-developer/access/desktop-database-reference/equivalent-ansi-sql-data-types`

und

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

empfohlen.

2.5 Anweisungen

Ein SQL-Programm besteht aus einer oder mehreren Anweisungen, auch Befehle oder Statements genannt. Zwei Anweisungen werden in SQL mit einem Semikolon getrennt. Für die letzte auszuführende Anweisung darf es weggelassen werden.

SQL ist, anders als der Name vielleicht vermuten lässt, keine reine Abfragesprache für Daten, sondern kann zusätzlich sowohl die Struktur der Datenbank als auch Zugriffsrechte verwalten. Entsprechend ist der Befehlssatz von SQL in drei Teilbereiche gegliedert, die DCL, die DDL und die DML. Die *Data Control Language (DCL)* ist der Sprachteil von SQL, der für die Verwaltung von Zugriffsrechten auf die Datenbank zuständig ist. Wir werden uns in diesem Skript damit nicht beschäftigen, daher sei dazu auf Anhang A.1 ab Seite 114 verwiesen.

Data Definition Language (DDL) Mit der Data Definition Language können eine Datenbank und die Struktur ihrer Tabellen verwaltet werden. Konkret können Datenbanken, Tabellen und User angelegt, verändert und gelöscht werden. Dazu stehen die Ausdrücke **`CREATE`**, **`ALTER`** und **`DROP`** zur Verfügung, die kombiniert werden können mit den Ausdrücken **`DATABASE`**, **`TABLE`** und **`USER`**:

$$\left\{\begin{array}{c}\texttt{CREATE}\\ \texttt{ALTER}\\ \texttt{DROP}\end{array}\right\} + \left\{\begin{array}{c}\texttt{DATABASE}\\ \texttt{TABLE}\\ \texttt{USER}\end{array}\right\} + \textit{Name} + \textit{[Zusatzoptionen]};\tag{2.1}$$

Relationentyp

Wir werden in diesem Skript lediglich die Verwaltung von Tabellen behandeln, für weitere Informationen sei auf den Anhang A.1 verwiesen. Um eine Tabelle zu erzeugen, müssen wir uns zunächst über ihren Relationentyp nach Definition 1.4 auf Seite 10 im Klaren sein. Der Relationentyp wird in SQL definiert, indem die Spaltennamen mit ihrem Datentyp aufgeführt werden:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
   spalte_2 datentyp_2,
     ...
   spalte_n datentyp_n
);
```

Auf diese Weise wird also der Relationentyp

`tabelle(datentyp_1, ..., datentyp_n)`

Schema, Tabellenstruktur

definiert. Der Relationentyp einer Tabelle wird oft auch *Schema* oder *Struktur* der Tabelle genannt. In den meisten Datenbanksystemen kann man den Zeichensatz festlegen, in dem die Werte in Spalten mit Textformaten gespeichert werden. Zu empfehlen ist hier der Zeichensatz UTF-8:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
     ...
) DEFAULT CHARSET=utf8;
```

Dieser Zeichensatz codiert den Unicode und ermöglicht die Buchstaben aller Verkehrssprachen der Welt sowie zahlreiche technische und mathematische Symbole.

Möchte man für eine Spalte nur bestimmte Werte ermöglichen, so kann man mit dem reservierten Wort **CHECK** eine Gültigkeitsregel für die Werte definieren:

Gültigkeitsregeln mit CHECK

```
CREATE TABLE tabelle (
   ...
   spalte_x datentyp_x CHECK (spalte_x IN (wert_1, ..., wert_n)),
   ...
) DEFAULT CHARSET=utf8;
```

ähnlich können für numerische Datentypen auch Zahlbereiche angegeben werden, beispielsweise:

```
   ...
   preis decimal(10,2) CHECK (preis > 0),
   ...
```

Die Struktur einer Tabelle wird durch **ALTER** geändert, mit **DROP** wird eine Tabelle vollständig gelöscht. Da wir diese Befehle nicht näher behandeln, sei für ihre genaue Syntax und weitere Informationen wieder auf den Anhang A.1 verwiesen.

Data Manipulation Language (DML) Mit der *Data Manipulation Language (DML)* werden Anweisungen für die Verwaltung der Daten selbst bereitgestellt, d.h. man kann damit Datensätze anlegen, lesen, aktualisieren und löschen. Diese vier zentralen Funktionen auf Daten werden mit dem Kürzel CRUD (für *create*, *read*, *update* und *delete*) zusammengefasst. Beispielsweise werden in eine existierende Tabelle Datensätze eingefügt, indem der Ausdruck **INSERT** verwendet wird:

Datenverwaltung mit den 4 Funktionen CRUD

```
INSERT INTO tabelle (spalte_1, ..., spalte_n) VALUES
  (wert_11,  ...,  wert_1n),
      ...
  (wert_m1,  ...,  wert_mn);
```

Hiermit werden m Datensätze mit ihren Werten für die n Spalten eingefügt. Wichtig ist, dass die im ersten Klammerpaar angegebene Reihenfolge der Spalten der Reihenfolge der Werte entsprechen muss. Bei manchen RDBMS kann man pro INSERT nur einen Datensatz einfügen, z.B. bei Oracle. Will man alle Spalten der Tabelle füllen, kann man die Klammern mit den Spaltennamen vor **VALUES** auch weglassen. Dazu muss allerdings die genaue Anzahl und Anordnung der Spalten in dem Tabellenschema eingehalten werden.

Daten können in SQL mit **SELECT** gelesen werden. Dieser Befehl ist der komplexeste in SQL. Die vollständige Syntax ist im Anhang A.1 aufgeführt. Wir werden die Anweisung im Laufe dieses Skripts genauer kennenlernen. Die einfachste Variante lautet wie folgt:

```
SELECT * FROM tabelle
```

Sie zeigt die gesamte Tabelle mit dem Namen `tabelle` an, d.h. all ihre Spalten und Datensätze.

Die Ausdrücke **UPDATE** zum Ändern eines Datensatzes und **DELETE** zum Löschen von Datensätzen dagegen werden wir nicht näher behandeln, aber auch sie sind im Anhang A.1 erläutert.

Beispiel 2.1. Wir werden in diesem Beispiel als erste Anwendung der betrachteten SQL-Anweisungen die Realisierung des Relationentyps Full House aus Beispiel 1.2 auf Seite 9 als Tabelle betrachten:

```
CREATE TABLE full_house (
  farbe varchar(5),
```

```
  karte varchar(5)
) DEFAULT CHARSET=utf8;
```

Mit dem Befehl

```
INSERT INTO full_house (farbe,karte) VALUES
  ('Kreuz','Ass'),
  ('Pik','Ass'),
  ('Herz','König'),
  ('Herz','Ass'),
  ('Karo','König');
```

werden die Daten darin gespeichert. Die Tabelle können wir uns dann mit der Anweisung

```
SELECT * FROM full_house;
```

ausgeben lassen:

farbe	karte
Kreuz	Ass
Pik	Ass
Herz	König
Herz	Ass
Karo	König

□

2.6 Funktionen

Berechnung von Tabellenausgaben

In SQL sind einige Standardfunktionen definiert, die als Parameter geeignete Attributwerte der einzelnen Datensätze einer Tabelle erhalten können und einen Wert zurückliefern. Sie werden oft auch *Single-Row-Funktionen* genannt, da sie auf einzelnen Datensätzen einsetzbar sind. Einige Beispiele sind die folgenden mathematischen und stringbearbeitenden Funktionen:

Elementare mathematische Funktionen:

`ABS`(x)	$\lvert x\rvert$
`POWER`(x, y)	x^y
`ROUND`(x, n)	x auf n Nachkommastellen runden
`SQRT`(x)	$\sqrt{x}$

Wichtige Stringfunktionen:

`CONCAT('s1', ..., 'sn')`	Hängt die Strings s1 bis sn aneinander
`POSITION('teil' IN 'String')`	Position des Teilstrings im Text; 0, wenn nicht enthalten
`REPLACE('Text', 'ae', 'ä')`	In ‘Text’ wird überall ‘ae’ durch ‘ä’ ersetzt
`SUBSTRING('Text',` x [, y]`)`	Teilstring ab Position x [optional nur bis Position $x + y$]
`TRIM(' Text ')`	entfernt führende und hängende Leerzeichen

Hierbei stellen alle von Apostrophs eingeklammerten Zeichenfolgen einen beliebigen String dar, die Variablen x, y und n dagegen Zahlen. Die Zeichenfolgen und Zahlen können Konstanten sein oder Attributwerte eines Datensatzes annehmen. Dabei ist zu beachten, dass in Funktion mit mehreren Parametern nur Attributwerte aus demselben Datensatz eingesetzt werden können.

Beispiel 2.2. Wenden wir auf unsere Tabelle `full_house` in Beispiel 2.1 die Anweisung

```
SELECT SUBSTRING(farbe, 1, 2) FROM full_house;
```

SUBSTRING

an, so werden für von der Farbe jedes der 5 Datensätze wegen $x = 1$ und $y = 2$ als Konstanten die ersten zwei Buchstaben angezeigt:

SUBSTRING(farbe, 1, 2)
Kr
Pi
He
He
Ka

Das Ergebnis ist also eine neue Tabelle mit nur einer Spalte, aber fünf Datensätzen. Wir können auch mehrere Attributwerte eines Datensatzes miteinander kombinieren. Beispielsweise erhalten wir mit

CONCAT

```
SELECT CONCAT('{', farbe, ' - ', karte, '}') FROM full_house;
```

die Aneinanderreihung („Konkatenation“) der Farbe und Karte jedes Datensatzes, abgeschlossen und verbunden mit konstanten Zeichen:

CONCAT(‘{’, farbe, ‘ - ’, karte, ‘}’)
{Kreuz - Ass}
{Pik - Ass}
{Herz - König}
{Herz - Ass}
{Karo - König}

Wir können darüber hinaus Funktionen ineinander verschachteln. □

Zwischenfrage 2.3. Betrachten wir die Tabelle `full_house` aus dem obigen Beispiel 2.1 und die folgende Anweisung mit verschachtelten Funktionen:

```
SELECT CONCAT(SUBSTRING(farbe,1,1), ' - ', SUBSTRING(karte,1,1)) FROM full_house;
```

Was ist die Ausgabe? [6]

[6] Von jedem Datensatz werden zwei Teilstrings konkateniert: K - A, P - A, H - K, H - A, K - K.

3

Die SELECT-Anweisung

Kapitelübersicht

3.1 Einfache SELECT-Anweisungen 18
3.2 Alias-Spalten mit AS 21
3.3 Auswahlfilter: Die WHERE-Klausel 22
3.4 Textsuche mit LIKE 24
3.5 Sortierung mit ORDER BY 25

3.1 Einfache SELECT-Anweisungen

Die wichtigste und mächtigste Anweisung in SQL ist SELECT. Sie dient der Abfrage auf die in einer Tabelle gespeicherten Daten. Eine solche Abfrage kann beliebig komplex werden. Beginnen wir mit dem Einfachen, um zunächst die grundsätzliche Funktionsweise zu verstehen.

Die SELECT-Anweisung ist eine Auswahl der Daten von Spalten einer Tabelle, also eine *Selektion* von Daten. Die Syntax lautet:

```
SELECT spalte_1, ..., spalte_n FROM tabelle;
```

SELECT

Die Spalten (`spalte_1`, . . . , `spalte_n`) müssen dabei eine Auswahl der Spalten der betrachteten Tabelle sein. Die Ausgabe eines SELECTS ist eine Tabelle mit den Spalten `spalte_1`, . . . , `spalte_n` und aus den Datensätzen der Tabelle besteht, in der die nicht ausgewählten Spalten also weggelassen sind:

spalte_1	. . .	spalte_n
$\text{wert}_{1,1}$	. . .	$\text{wert}_{1,n}$
⋮		⋮
$\text{wert}_{m,1}$	. . .	$\text{wert}_{m,n}$

Ergebnismenge

Diese Ausgabe heißt *Ergebnismenge (result set)*. Die Reihenfolge der Spalten der Ergebnismenge ist dabei durch die Reihenfolge der Spaltenauswahl im SELECT festgelegt. Will man alle Spalten der Tabelle auswählen, so kann man den Stern $*$ als Wildcard verwenden:

```
SELECT * FROM tabelle;
```

Beispiel 3.1. Betrachten wir unsere Tabelle `full_house` in Beispiel 2.1. Die Ergebnisse dreier möglicher SELECT-Anweisungen sind dann beispielsweise jeweils:

`SELECT farbe, karte FROM full_house;`	`SELECT farbe FROM full_house;`	`SELECT karte FROM full_house;`

farbe	karte
Kreuz	Ass
Pik	Ass
Herz	König
Herz	Ass
Karo	König

farbe
Kreuz
Pik
Herz
Herz
Karo

karte
Ass
Ass
König
Ass
König

Die erste Tabellenausgabe hätte man ebenso mit dem Befehl

```
SELECT * FROM full_house;
```

erreicht, da der Relationentyp `full_house(farbe, karte)` die Attribute `farbe` und `karte` genau in dieser Reihenfolge besitzt. □

Bemerkung 3.2. Mathematisch betrachtet bildet eine Selektion

$$\texttt{SELECT}\ A_{j_1}, A_{j_2}, \ldots, A_{j_k}\ \texttt{FROM}\ R;$$

die Tabelle $R(A_1, \ldots, A_n)$ mit n Attributen auf eine Ergebnismenge ab, die eine neue Tabelle $R(A_{j_1}, \ldots, A_{j_k})$ mit nur k Attributen ist, wobei $k \leq n$ und $j_1, \ldots, j_k \in \{1, \ldots, n\}$ gilt:

Ergebnismengen sind Tabellen

$$\begin{array}{|cccccc|}\hline A_1 & A_2 & A_3 & \cdots & A_n \\ \hline x_{11} & x_{12} & x_{13} & \cdots & x_{1n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{i1} & x_{i2} & x_{i3} & \cdots & x_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \cdots & x_{mn} \\ \hline \end{array} \quad \xmapsto{\texttt{SELECT}} \quad \begin{array}{|cccc|}\hline A_{j_1} & A_{j_2} & \cdots & A_{j_k} \\ \hline x_{1j_1} & x_{1j_2} & \cdots & x_{1j_k} \\ \vdots & \vdots & & \vdots \\ x_{ij_1} & x_{ij_2} & \cdots & x_{ij_k} \\ \vdots & \vdots & & \vdots \\ x_{mj_1} & x_{mj_2} & \cdots & x_{mj_k} \\ \hline \end{array} \tag{3.1}$$

Ein **SELECT** „projiziert“ also eine Tabelle auf weniger (oder gleich viel) Spalten. Die Ergebnismenge kann jedoch doppelte Zeilen (Dubletten) enthalten, d.h. sie muss keine Relation sein, wie in Bemerkung 1.3 auf Seite 10 erläutert. Um dies zu verhindern, kann man das reservierte Wort **DISTINCT** verwenden:

$$\texttt{SELECT DISTINCT}\ A_{j_1}, A_{j_2}, \ldots, A_{j_k}\ \texttt{FROM}\ R;$$

Ergebnismengen mit DISTINCT sind Relationen

Ist die ursprüngliche Tabelle eine Relation, so haben wir mit **SELECT DISTINCT** also eine Abbildung von einer Relation $R(A_1, \ldots, A_n)$ mit n Attributen auf eine neue Relation $R'(A_{j_1}, \ldots, A_{j_k})$ mit k Attributen, aber nun mit nur noch r statt m Tupeln („Datensätzen“):

R

$$\begin{array}{|cccccc|}\hline A_1 & A_2 & A_3 & \cdots & A_n \\ \hline x_{11} & x_{12} & x_{13} & \cdots & x_{1n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{i1} & x_{i2} & x_{i3} & \cdots & x_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \cdots & x_{mn} \\ \hline \end{array} \quad \xmapsto{\texttt{SELECT DISTINCT}} \quad \begin{array}{|cccc|}\hline A_{j_1} & A_{j_2} & \cdots & A_{j_k} \\ \hline x_{q_1 j_1} & x_{q_1 j_2} & \cdots & x_{q_1 j_k} \\ x_{q_2 j_1} & x_{q_2 j_2} & \cdots & x_{q_2 j_k} \\ \vdots & \vdots & & \vdots \\ x_{q_r j_1} & x_{q_r j_2} & \cdots & x_{q_r j_k} \\ \hline \end{array} \tag{3.2}$$

R'

$(q_1, \ldots, q_r \in \{1, \ldots, m\})$. Ein **SELECT** projiziert also eine Tabelle auf weniger Spalten, ein **SELECT DISTINCT** auf weniger Spalten *und* weniger Zeilen. Daher spricht man in der Theorie der Relationen bei der Selektion mit **DISTINCT** auch von einer *Projektion* der Relation R auf die Relation R', d.h. in Symbolen

$$p : R \to R'. \tag{3.3}$$

Sie ist bezüglich der Mengen der Relationen „abgeschlossen“.[1] □

Beispiel 3.3. Betrachten wir wieder unser Beispiel 2.1, so erhalten wir mit SELECT DISTINCT die folgende Ergebnismenge:

```
SELECT DISTINCT karte FROM full_house;
```

karte
Ass
König

Im Gegensatz zur dritten Ergebnismenge aus Beispiel 3.1 erhalten wir nun also eine Ergebnismenge ohne doppelte Einträge, d.h. mit Bemerkung 3.2 eine neue Relation. □

Beispiel 3.4. Betrachten wir die folgende Tabelle, die verschiedene Comic-Alben speichert.

```
-- Tabellenstruktur:
CREATE TABLE alben (
  titel varchar(50),
  reihe varchar(50),
  band  smallint,
  preis decimal(4,2),
  jahr  smallint
);
-- Daten:
INSERT INTO alben (titel, reihe, band, preis, jahr) VALUES
  ('Asterix, der Gallier', 'Asterix', 1, 2.80, 1968),
  ('Asterix und Kleopatra', 'Asterix', 2, 2.80, 1968),
  ('Gespenster Geschichten', 'Gespenster Geschichten', 1, 1.20, 1974),
  ('Die Trabantenstadt', 'Asterix', 17, 3.80, 1974),
  ('Der große Graben', 'Asterix', 25, 5.00, 1980),
  ('Das Kriminalmuseum', 'Franka', 1, 8.80, 1985),
  ('Das Meisterwerk', 'Franka', 2, 8.80, 1986);
```

Das ergibt die Tabelle 3.1. Die Auswahl der Spalten `reihe, preis` liefert dann, jeweils ohne und

alben

titel	reihe	band	preis	jahr
Asterix, der Gallier	Asterix	1	2.80	1968
Asterix und Kleopatra	Asterix	2	2.80	1968
Gespenster Geschichten	Gespenster Geschichten	1	1.20	1974
Die Trabantenstadt	Asterix	17	3.80	1974
Der große Graben	Asterix	25	5.00	1980
Das Kriminalmuseum	Franka	1	8.80	1985
Das Meisterwerk	Franka	2	8.80	1986

Tabelle 3.1: Beispieldaten der Tabelle `alben`. Vgl. Piepmeyer (2011:S. 172)

mit `DISTINCT`, die folgenden Ausgaben:

[1] Piepmeyer (2011):§4.1.

```
SELECT reihe, preis FROM alben;
```

reihe	preis
Asterix	2.80
Asterix	2.80
Gespenster Geschichten	1.20
Asterix	3.80
Asterix	5.00
Franka	8.80
Franka	8.80

```
SELECT DISTINCT reihe, preis FROM alben;
```

reihe	preis
Asterix	2.80
Gespenster Geschichten	1.20
Asterix	3.80
Asterix	5.00
Franka	8.80

Hier ergibt also nur die Selektion mit DISTINCT eine Relation. □

Zwischenfrage 3.5. Wendet man Funktionen wie in Beispiel 2.2 auf eine Tabelle an, so erhält man eine neue Tabelle. Ist die Selektion in diesen Fällen abgeschlossen? [2]

Bemerkung 3.6. * Da die Anweisung `SELECT DISTINCT` mit den Vereinbarungen aus Bemerkung 1.3 eine Relation auf eine Relation abbildet, nennt man sie „abgeschlossen“ bezüglich Relationen.[3] Dennoch gibt es von theoretischen Informatikern oft die Kritik, dass SQL sich nicht wirklich an das relationale Modell hält und die Programmiererin oder der Programmierer selbst entsprechende „Integritätsregeln“ herstellen muss.[4] Die britischen Informatiker Christopher J. Date und Hugh Darwen entwarfen in ihrem Buch *The Third Manifesto* ein konsequent relationale Programmiersprachenmodell und nannten es D. Einige Programmiersprachen wie Rel oder D4 der Datenbank Dataphor implementieren dieses Sprachenmodell. Siehe für weitere Links die englische Wikipedia-Seite `https://en.wikipedia.org/wiki/The_Third_Manifesto`. □

3.2 Alias-Spalten mit `AS`

Mit eine SELECT-Abfrage kann man nicht nur bestehende Spalten einer Tabelle auswählen, sondern auch völlig neue Spalten aus anderen Spalten oder mit Funktionen berechnete Werte erstellen. Der Name solcher Spalten heißt *Alias* und wird mit dem reservierten Wort `AS` vergeben:

```
SELECT ... ausdruck AS alias ... FROM ...
```

Hierbei ist `ausdruck` ein ein aus aus Spaltenwerten bestimmter Wert oder das Ergebnis einer Funktion. Das Wort `AS` kann auch entfallen. (Es sollte aber m.E. aus Gründen der Lesbarkeit einer Abfrage immer verwendet werden.)

Beispiel 3.7. Wollen wir uns für jedes unserer Comic-Alben die Mehrwertsteuer und die Nettopreise berechnen und anzeigen lassen, so können wir dazu Aliasse verwenden:

```
SELECT titel,
       preis AS brutto,
       ROUND(preis/1.19, 2) AS netto,
       ROUND(0.19/1.19*preis, 2) AS mwst
FROM alben;
```

Diese Abfrage ergibt dann:

---

[2] Ein SELECT nicht unbedingt, wie man an der Ausgabe der ersten Teilstringausgabe erkennt, aber ein SELECT DISTINCT immer! (Probieren Sie es aus!)

[3] vgl. Piepmeyer (2011):§4.2.

[4] vgl. Piepmeyer (2011):S. 88ff.

titel	brutto	netto	mwst
Asterix, der Gallier	2.80	2.35	0.45
Asterix und Kleopatra	2.80	2.35	0.45
Gespenster Geschichten	1.20	1.01	0.19
Die Trabantenstadt	3.80	3.19	0.61
Der große Graben	5.00	4.2	0.8
Das Kriminalmuseum	8.80	7.39	1.41
Das Meisterwerk	8.80	7.39	1.41

□

Ebenso wie für Spaltennamen können wir auch für Tabellennamen Aliasse verwenden. Zum Beispiel könnten wir die Abfrage aus Beispiel 3.7 äquivalent formulieren als:

```
SELECT a.titel,
       a.preis AS brutto,
       ROUND(a.preis/1.19, 2) AS netto,
       ROUND(0.19/1.19*a.preis, 2) AS mwst
FROM alben AS a;
```

Hier wird also die Tabelle `alben` smit dem Alias `a` versehen, das dann innerhalb der Abfrage als gültiger Tabellenname verwendet werden kann, wie hier als Referenzen `a.titel` und `a.preis` für die Spalten `titel` und `preis` in der Tabelle `alben`. In diesem Beispiel mit nur einer Tabelle macht ein Alias für Tabellen keinen Sinn. Für das Arbeiten mit mehren Tabellen können Aliasse die Schreibarbeit bei Abfragen erleichtern.

Zwischenfrage 3.8. Informatisch ist die Abfrage in Beispiel 3.7 zwar völlig korrekt, aber leider ist die Mehrwersteuer auf Bücher ermäßigt auf 7 %. Wie muss dieAbfrage also lauten, um sich die Titel und die wirtschaftlich korrekten Mehrwertsteuerbeträge anzeigen zu lassen? [5]

3.3 Auswahlfilter: Die `WHERE`-Klausel

Mit der Konstruktion

```
SELECT spalte_1, ..., spalte_n FROM tabelle WHERE bedingung;
```

können wir eine Auswahl von Spalten einer Tabelle treffen, wobei nur Zeilen berücksichtigt werden, die der nach dem reservierten Wort `WHERE` Bedingung genügen. Die Bedingung wird auch `WHERE`-Klausel genannt und filtert also die gewünschten Datensätze aus allen Datensätzen der Tabelle. In der Terminologie der Relationenalgebra heißt die Bedingung *Prädikat*.[6] Meist wird die WHERE-Klausel mit einem Vergleich zu bestimmten Werten gebildet, d.h. mit einem der folgenden *Vergleichsoperatoren* gebildet:

=	<>, !=	>	<	>=	<=
gleich	ungleich	echt größer	echt kleiner	größer gleich	kleiner gleich

(3.4)

In Microsoft Access funktioniert der Operator `!=` nicht, hier muss der Operator `<>` verwendet werden!

Beispiel 3.9. Betrachten wir unsere Full House Tabelle aus Beispiel 2.1. (a) Wie finden wir heraus, welche Karten in dem Full House haben die Farbe Herz? (b) Wie lauten Farbe und Karte der Spielkarten des Full House, die nicht die Farbe Herz haben?

[5] `SELECT titel, ROUND(0.07/1.07*preis, 2) AS mwst FROM alben;`

[6] Piepmeyer (2011):§§4.4, 4.7.

```
SELECT karte FROM full_house
WHERE farbe = 'Herz';
```

karte
König
Ass

```
SELECT farbe, karte FROM full_house
WHERE farbe <> 'Herz';
```

farbe	karte
Kreuz	Ass
Pik	Ass
Karo	König

Die zweite WHERE-Klausel hätten wir auch mit `!=` statt `<>` versehen können. □

Beispiel 3.10. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, für die weniger als 1 € Mehrwertsteuer anfallen? Wir können dazu die Abfrage aus Beispiel 3.7 modifizieren und mit der WHERE-Klausel die geeigneten Datensätze filtern:

```
SELECT titel, ROUND(0.19/1.19*preis, 2) AS mwst FROM alben
WHERE 0.19/1.19*preis < 1;
```

(Leider ermöglicht SQL nicht die Verwendung von Aliassen desselben SELECTs in der WHERE-Klausel!) Die Abfrage liefert:

titel	mwst
Asterix, der Gallier	0.45
Asterix und Kleopatra	0.45
Gespenster Geschichten	0.19
Die Trabantenstadt	0.61
Der große Graben	0.8

□

In der klassischen Logik ist eine Bedingung eine logische Aussage, die wahr oder falsch sein kann. Die WHERE-Klausel filtert in diesem Sinne nur diejenigen Datensätze, für die diese Aussage wahr ist. In SQL können wir, wie in der Aussagenlogik und somit auch in anderen Programmiersprachen bekannt, mehrere Aussagen logisch miteinander verknüpfen. Die mathematische Grundlage dafür ist die Boole'sche Algebra. Die wichtigsten dafür notwendigen logischen Operatoren sind **NOT**, **AND** und **OR**, also die logische Verneinung, das logische ODER und das logische UND.

Beispiel 3.11. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, für die weniger als 1 € Mehrwertsteuer anfallen, aber nach 1975 erschienen sind? Wir müssen dazu nur die WHERE-Klausel aus Beispiel 3.10 mit **AND** um eine weitere Filterbedingung erweitern:

```
SELECT titel FROM alben WHERE 0.19/1.19*preis < 1 AND jahr > 1975;
```

Die Abfrage liefert in unserem Fall nur eine einzige Zeile:

titel
Der große Graben

wie wir durch Vergleich der Tabelle 3.1 erkennen können. □

Wollen wir Zahlen- oder Datumswerte einer Spalte x aus einem Wertebereich $x \in$ [`wert_1`, `wert_2`] filtern, so können wir dies mit einer AND-Verknüpfung der beiden Ungleichungen $x \geqq$ `wert_1` und $x \leqq$ `wert_2` erreichen:

```
SELECT ... FROM tabelle WHERE x >= wert_1 AND x <= wert_2
```

Wir können in SQL auf Wertebereiche von Zahlen oder Datumsangaben auch den Ausdruck **BETWEEN** ... **AND** ... verwenden:

```
SELECT ... FROM tabelle WHERE x BETWEEN wert_1 AND wert_2
```

Hierbei muss `wert_1` kleiner gleich `wert_2` sein. Näheres siehe z.B. unter `https://www.w3schools.com/sql/sql_between.asp`.

Beispiel 3.12. Wie finden wir in unserer Comics-Datenbank aus Beispiel 3.4 diejenigen Alben heraus, die von 1970 bis 1980 erschienen sind? Nun ja, mit dem Ausdruck `between` ist die Abfrage sehr einfach:

```
SELECT titel FROM alben WHERE jahr BETWEEN 1970 AND 1980;
```

Die Abfrage liefert in unserem Fall drei Datensätze:

titel
Gespenster Geschichten
Die Trabantenstadt
Der große Graben

wie in Tabelle 3.1 abzulesen ist. □

Wir werden die logischen Operatoren von SQL später in Bemerkung 4.4 genauer untersuchen. Nur soviel sei hier schon verraten: SQL bietet mehr als die klassische Logik!

3.4 Textsuche mit `LIKE`

Der Operator `LIKE` dient in Suchbedingungen zum Abfragen von Textmustern über Schablonen und Wildcards. Die Syntax lautet:

```
... WHERE ausdruck LIKE textmuster ...
```

Hierbei ist `ausdruck` ein Spaltenname oder ein String `'abc'`, und `textmuster` ein String mit der Wildcard `%` oder der Schablone `_`. Durch den Ausdruck werden diejenigen Datensätze gefiltert, die dem Muster entsprechen. Die folgenden Beispiele sollen die Wirkung von `LIKE` verdeutlichen:

Fragestellung	**SQL-Abfrage**
Welche Albentitel beginnen mit ‘D’?	`SELECT titel FROM alben WHERE titel LIKE 'D%';`
Welche Albentitel enden mit ‘n’?	`SELECT titel FROM alben WHERE titel LIKE '%n';`
Welche Albentitel enthalten ein ‘n’?	`SELECT titel FROM alben WHERE titel LIKE '%n%';`
Welche Albentitel haben als 3. Buchstaben ein ‘r’?	`SELECT titel FROM alben WHERE titel LIKE '__r%';`

Die Wildcard beinhaltet insbesondere den leeren String. Die Überprüfung auf Treffer ist für die meisten RDBMS schreibungsinsensitiv *(case insensitive)*, unterscheidet also nicht zwischen Groß- und Kleinschreibung der Buchstaben. Beachte: Bei MS Access ist die Wildcard durch $*$ und die Schablone durch `?` gegeben:[7]

Bezeichnung	**Art der Treffer**	**SQL-Standard**	**Access**
Wildcard	mehrere beliebige Zeichen	`%`	$*$
Schablone	einzelnes beliebiges Zeichen	`_`	`?`

(3.5)

[7] `https://support.office.com/de-de/article/b2f7ef03-9085-4ffb-9829-eef18358e931`

Vgl. https://www.w3schools.com/sql/sql_like.asp. Die Verneinung der LIKE-Operators ist NOT LIKE.

Beispiel 3.13. Um herauszufinden, welche Albentitel einer Reihe angehören, die ein 'x' enthält, können wir den LIKE-Operator verwenden. Um das Gegenteil herauszufinden, welche Albentitel einer Reihe angehören, die kein 'x' enthält, kann man ihn mit NOT kombinieren:

```
SELECT titel FROM alben
WHERE reihe LIKE '%x%';
```

titel
Asterix, der Gallier
Asterix und Kleopatra
Die Trabantenstadt
Der große Graben

```
SELECT titel FROM alben
WHERE reihe NOT LIKE '%x%';
```

titel
Gespenster Geschichten
Das Kriminalmuseum
Das Meisterwerk

Insbesondere wird also mit % auch der leere String als Zeichen erfasst, so dass hier das 'x' als Endbuchstabe berücksichtigt wird. Verwenden wir eine MS Access Datenbank, so müssen wir stattdessen mit `LIKE '*x*'` die geeignete Wildcard verwenden.

Wollen wir uns alle Titel anzeigen lassen, deren dritter Buchstabe ein 's' ist, so lautet die Abfrage

```
SELECT titel FROM alben WHERE titel LIKE '__s%';
```

(also mit zwei Unterstrichen!) Sie liefert die Ergebnismenge

titel
Gespenster Geschichten
Das Kriminalmuseum
Das Meisterwerk

Bei einer MS Access Datenbank muss die Schablone entprechend `LIKE '??s'` lauten. □

3.5 Sortierung mit ORDER BY

Häufig ist es erwünscht, sich die Ergebnismenge einer Datenbankabfrage sortiert anzeigen zu lassen. In SQL gibt es dazu die Komponente ORDER BY, die an das Ende der zu sortierenden SELECT-Anweisung angefügt wird:

```
SELECT spalte_1, ..., spalte_n FROM tabelle ORDER BY spalte_j1, ..., spalte_jk;
```

Hinter BY werden als Sortierkriterium die Spalten aufgelistet, nach denen sortiert werden soll. Das bedeutet, dass zunächst nach Spalte spalte_j1 sortiert wird, falls Einträge den gleichen Wert für spalte_j1 haben, wird nach der nächsten Spalte spalte_j2 der Liste sortiert, usw. ..., bis zur letzten Spalte spalte_jk der Liste. Die Spalten dieser Liste müssen dabei nicht unbedingt auch in der SELECT-Liste enthalten sein.

Beispiel 3.14. Wollen wir uns die Reihen der Comics aus Beispiel 3.4 sortiert anzeigen und danach, bei Alben der gleichen Reihe, nach dem Jahr, so schreiben wir folgende Anweisung:

```
SELECT reihe, titel, jahr FROM alben ORDER BY reihe, jahr;
```

Die Ergebnismenge lautet dann

reihe	titel	jahr
Asterix	Asterix, der Gallier	1968
Asterix	Asterix und Kleopatra	1968
Asterix	Die Trabantenstadt	1974
Asterix	Der große Graben	1980
Franka	Das Kriminalmuseum	1985
Franka	Das Meisterwerk	1986
Gespenster Geschichten	Gespenster Geschichten	1974

Wie erwartet erfolgt die Sortierung nach Strings alphabetisch, nach Zahlen numerisch. □

Wie wir an dem obigen Beispiel erkennen, sortiert SQL die Daten in aufsteigender Reihenfolge. Wollen wir dagegen für ein (oder mehrere) Spalten eine Sortierung in *absteigender* Reihenfolge, so müssen wir hinter die betreffende Spalte das reservierte Wort `DESC` schreiben, also

```
SELECT ... FROM tabelle ORDER BY ..., spalte_ji DESC, ...
```

Um ausdrücklich zu betonen, dass aufsteigend sortiert werden soll, kann man den Ausdruck `ORDER BY` auch durch `ASC` für *ascending* abschließen. (Strenggenommen ist ASC aber überflüssig.)

Beispiel 3.15. Wollen wir uns die Reihen der Comics aus Beispiel 3.4 sortiert anzeigen und danach, bei gleicher Reihe, nach dem Jahr, so erreichen wir dies mit der folgenden Anweisung:

```
SELECT reihe, titel, jahr FROM alben ORDER BY reihe, jahr DESC;
```

Die Ergebnismenge lautet dann

reihe	titel	jahr
Asterix	Der große Graben	1980
Asterix	Die Trabantenstadt	1974
Asterix	Asterix und Kleopatra	1968
Asterix	Asterix, der Gallier	1968
Franka	Das Meisterwerk	1986
Franka	Das Kriminalmuseum	1985
Gespenster Geschichten	Gespenster Geschichten	1974

Die Sortierung nach der Reihe bleibt also alphabetisch aufsteigend, die Sortierung bei gleicher Reihe nach Jahren dagegen absteigend. □

Bemerkung 3.16. *(Kollation)* Eigentlich ist ja sonnenklar, was „alphabetische Sortierung“ bedeutet, nicht wahr? Bei genauerem Hinsehen gibt es da aber viele Mehrdeutigkeiten. Was ist mit Groß- und Kleinbuchstaben? Ist ‘a’ vor oder nach ‘A’ oder ist die Schreibung groß/klein egal? Was ist mit Umlauten? Wie verhält sich also zum Beispiel ‘Ä’ zu ‘A’? Sind Ziffern vor Buchstaben einzuordnen? Was ist mit Buchstaben aus dem türkischen, griechischen, chinesischen, arabischen oder hebräischen Alphabet?

Die genaue Sortierungsregel für Buchstaben und Zeichen in Einträgen einer Datenbank heißt *Kollation*. Sie kann bei vielen RDBMS insgesamt oder individuell für einzelne Tabellen konfiguriert werden. In MS Access zum Beispiel wird die Kollation über die allgemeinen Einstellungen konfiguriert, im SQL-Standard ist sie bei der Erstellung einer Datenbank mit dem reservierten Wort `COLLATE` möglich. Um möglichst viele Schriftzeichen zu erlauben, wird üblicherweise der Zeichensatz Unicode verwendet, und hier die speichereffiziente Variante UTF-8. In MariaDB und MySQL kann man z.B. mit

```
CREATE DATABASE datenbank DEFAULT CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

eine gesamte Datenbank so konfigurieren, dass UTF-8 als Zeichensatz und eine schreibungsinsensitive *(ci = case insensitive)* Sortierung eingerichtet wird. In PostgreSQL lautet der entsprechende Befehl

```
CREATE DATABASE datenbank ENCODING='UTF8' LC_COLLATE='und-x-icu'
```

Hier können auch länderspezifische Kodierungen festgelegt werden, z.B. für einen deutschen Schriftsatz mit der Option `LC_COLLATE='de_DE'`. □

4

NULL-Werte und dreiwertige Logik

Kapitelübersicht

4.1 Unbekannte Information 28
4.2 Dreiwertige Logik 29
4.3 Abfragen auf NULL-Werte 31
4.4 Setzen von Standardwerten bei NULL-Einträgen 32

4.1 Unbekannte Information

In der Praxis haben wir es oft mit unvollständiger oder unbekannter Information zu tun. So kann es beim Erfassen von Personendaten vorkommen, dass wir das Geburtsdatum oder den Wohnort nicht kennen; bei einem Artikel wissen wir das Verkaufsdatum noch nicht, solange er noch auf Lager ist. Es gibt grundsätzlich verschiedene Möglichkeiten, in einer Datenbank mit fehlender Information umzugehen. Codd löste das Problem für relationale Datenbanken als Teil seiner „3. Regel“:

Definition 4.1. Fehlende Information eines Attributs wird durch den speziellen Wert `NULL` dargestellt, unabhängig vom Datentyp des Attributs. □

Der Wert `NULL` ist also ein *einheitlicher* Repräsentant für unbekannte Information.

Beispiel 4.2. Nehmen wir für das Beispiel 3.4 unserer Comic-Datenbank an, wir hätten ein Album entdeckt, das wir eintragen möchten, ohne allerdings den Band und den Preis zu kennen. Mit der folgenden `INSERT`-Anweisung können wir diesen Eintrag in der Tabelle `alben` speichern:

```
INSERT INTO alben (titel, reihe, jahr) VALUES ('Lucky Luke', 'Lucky Luke', 1976);
```

Mit

```
SELECT * FROM alben;
```

erhält man dann:

reihe	titel	band	preis	jahr
Asterix, der Gallier	Asterix	1	2.80	1968
Asterix und Kleopatra	Asterix	2	2.80	1968
Gespenster Geschichten	Gespenster Geschichten	1	1.20	1974
Die Trabantenstadt	Asterix	17	3.80	1974
Der große Graben	Asterix	25	5.00	1980
Das Kriminalmuseum	Franka	1	8.80	1985
Das Meisterwerk	Franka	2	8.80	1986
Lucky Luke	Lucky Luke	NULL	NULL	1976

SQL verwendet also tatsächlich für unbekannte Information den Wert `NULL`. (Leider verwendet MS Access jedoch oft abhängig vom Datentyp einen vorgegebenen Wert, bei Zahlen beispielsweise 0.) □

Will man explizit verhindern, dass ein Attribut `NULL` sein kann, so müssen wir es bei der Erzeugung der Tabelle mit der Einschränkung `NOT NULL` versehen, also zum Beispiel:

```
CREATE TABLE personen (
  name varchar(20) NOT NULL;
  alter int
);
```

Ein Eintragsversuch mit

```
INSERT INTO personen (alter) VALUES (24);
```

oder

```
INSERT INTO personen (name, alter) VALUES (NULL, 24);
```

führt dann jeweils zu einer Fehlermeldung. Die Tabelle ist damit so konstruiert, dass sie einen Eintrag ohne eine Wertangabe für die spezifizierten Attribute gar nicht erst ermöglicht.

Die Erstellung eines Tabellenattributs mit der Einschränkung `NOT NULL` ist eine von mehreren Möglichkeiten in SQL, sogenannte *Integritätsregeln*[1] zu implementieren. Integritätsregeln spielen eine wichtige Rolle bei den Beziehungen von Datensätzen unterschiedlicher Tabelle. Wir werden darauf insbesondere in den Abschnitten 9.5.1 und 9.5.2 ab Seite 69 zurückkommen.

4.2 Dreiwertige Logik

Mit dem Wert `NULL` für unbekannte Information ist zwar ein Problem der realen Welt gelöst. Welchen Wahrheitswert jedoch soll eine Abfrage auf den Wert `NULL` ergeben? Ist die logische Aussage

```
'Otto' = NULL
```

wahr oder falsch? Um das herauszufinden, betrachten wir anhand des folgenden Beispiels, wie SQL hier entscheidet.

Beispiel 4.3. Betrachten wir zur Verdeutlichung das folgende vereinfachte Beispiel einer Tabelle *logik(aussage, wahrheitswert)* mit zwei Attributen *aussage* und *wahrheitswert*, die logische Aussagen und ihre jeweiligen Wahrheitswerte speichern sollen. Also beispielsweise die Aussage „1 + 2 = 3“ mit ihrem Wahrheitswert `true`. Die Tabellenstruktur wird dann mit der Anweisung

```
CREATE TABLE logik (aussage varchar(100), wahrheitswert boolean);
```

[1] Piepmeyer (2011):S. 88ff.

erzeugt. Speichern wir nun die Ausagen „1 + 2 = 3“ und „2 + 3 = 4“ mit ihren jeweiligen Wahrheitswerten:

```
INSERT INTO logik (aussage, wahrheitswert) VALUES
  ('1+2 = 3', true),
  ('2+3 = 4', false);
```

Wie speichern wir jedoch die Aussage „Ich werde eine 6 würfeln“? Wir wissen nicht, ob sie wahr oder falsch ist, also speichern wir sie ohne ihren Wahrheitswert:

```
INSERT INTO logik (aussage) VALUES ('Ich werde eine 6 würfeln');
```

Setzen wir die `SELECT`-Anweisungen

```
SELECT * FROM logik;
SELECT * FROM logik WHERE wahrheitswert OR NOT wahrheitswert;
SELECT * FROM logik WHERE (wahrheitswert = NULL) OR NOT (wahrheitswert = NULL);
```

darauf ab, so erhalten wir jeweils die folgenden Ergebnismengen:

`SELECT * FROM logik;`

aussage	wahrheitswert
1+2 = 3	TRUE
2+3 = 4	FALSE
Ich werde eine 6 würfeln	NULL

`SELECT * FROM logik WHERE wahrheitswert OR NOT wahrheitswert`

aussage	wahrheitswert
1+2 = 3	TRUE
2+3 = 4	FALSE

`SELECT * FROM logik WHERE wahrheitswert = NULL OR NOT wahrheitswert = NULL`

aussage	wahrheitswert

Das ist verblüffend. In der Boole’schen Algebra ist eine Aussage stets wahr oder falsch, d.h. die `OR`-Verknüpfung eines beliebigen Wahrheitswert mit seiner Verneinung, also

```
wahrheitswert OR NOT wahrheitswert
```

ist in der klassischen Logik immer wahr. Das hatte schon Shakespeares Hamlet gewusst: „to be `OR NOT` to be“![2] Bei allen drei `SELECT`-Anweisungen dürften wir also erwarten, dass sie jeweils *alle* Datensätze der Tabelle anzeigen. Haben wir also einen schwerwiegenden Fehler in SQL gefunden? □

Tatsächlich sind die Abfrageergebnisse mit den WHERE-Klauseln aus dem obigen Beispiel nur konsequent, was wir mit der dritten Abfrage sofort herleiten können:

Regel 1. Jeder *Vergleich eines Wertes mit `NULL` ist weder wahr noch falsch, sondern unbekannt, also `NULL`.*

Warum ist das nur konsequent? Gegenfrage: Was soll ein Vergleich eines Wertes mit einem unbekannten Wert denn sonst ergeben? Der Wert `NULL` ist weder gleich noch ungleich einem anderen Wert, insbesondere aber kann `NULL` nach Definition 4.1 als unbekannter Wert weder gleich noch ungleich sich selbst (eigentlich ja einem anderen unbekannten Wert) sein.

Bemerkung 4.4. Da `NULL` unbekannte Information darstellt, muss konsequenterweise auch jeder Vergleich mit `NULL` das Resultat „unbekannt“ ergeben. Weder wahr noch falsch sind daher korrekt! Vergleiche in SQL basieren daher nicht auf der klassischen Logik mit zwei logischen Werten, sondern auf einer *drei*wertigen Logik mit den drei Wahrheitswerten w (wahr), f (falsch) und

[2] William Shakespeare (1604): *The Tragedy of Hamlet, Prince of Denmark*, Act 3, Scene 1.

u (unbekannt). Der polnische Mathematiker Jan Łukasiewicz hat eine solche Logik, heute $_3$ genannt, bereits 1920 entwickelt.[3] Sie ist durch die folgenden Wahrheitstabellen gegeben:

x	NOT x
f	w
u	u
w	f

AND	f	u	w
f	f	f	f
u	f	u	u
w	f	u	w

OR	f	u	w
f	f	u	w
u	u	u	w
w	w	w	w

(4.1)

Die Verneinung NOT x eines Wahrheitswertes x ist hier in der ersten Tabelle zeilenweise durch die zweite Spalte angegeben, während die zweite und dritte Tabelle jeweils so zu lesen sind, dass die die beiden Argumente im Spalten- und Zeilenkopf stehen und ihre Verknüpfung im Innern der Tabelle. (Also beispielsweise u AND $w = u$.) □

Bemerkung * 4.5. *(Numerische Algebra der Wahrheitswerte)* Man kann mit Wahrheitswerten rechnen wie mit Zahlen. Mit den Entsprechungen $0 \leftrightarrow$ falsch, $\frac{1}{2} \leftrightarrow$ unbekannt und $1 \leftrightarrow$ wahr:

x	NOT x
0	1
$\frac{1}{2}$	$\frac{1}{2}$
1	0

AND	0	$\frac{1}{2}$	1
0	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$
1	0	$\frac{1}{2}$	1

OR	0	$\frac{1}{2}$	1
0	0	$\frac{1}{2}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
1	1	1	1

(4.2)

(Der Wahrheitswert u liegt hier also *zwischen* f und w.) Ein Vorteil dieser Darstellung der Wahrheitswerte ist, dass die logischen Operatoren als numerische Verknüpfungen aufgefasst werden können:

$$\text{NOT } a \;=\; 1 - a, \qquad a \text{ AND } b \;=\; \min(a, b), \qquad a \text{ OR } b \;=\; \max(a, b), \tag{4.3}$$

Diese Version der Łukasiewicz'schen Logik L_3 ist nicht unüblich.[4] □

Zwischenfrage 4.6. Welche Ergebnismengen liefern die folgenden Anweisungen für die Tabelle `logik` aus Beispiel 4.3? [5]

```
SELECT wahrheitswert FROM logik WHERE (1 = 1) OR (NULL = NULL);
SELECT wahrheitswert FROM logik WHERE (1 = 1) AND (NULL = NULL);
```

4.3 Abfragen auf NULL-Werte

Wie können wir nun herausfinden, ob ein Attribut eines Datensatzes den Wert NULL hat? Mit dem Gleichheitszeichen = geht es ja nun nicht. Für diesen Zweck können wir aber das reservierte Wort IS verwenden. Für den umgekehrten Fall steht uns IS NOT zur Verfügung. Beispielsweise ergeben die folgenden Anfragen für unser Beispiel 4.3:

```
SELECT aussage FROM logik
WHERE wahrheitswert IS NULL;
```

aussage
Ich werde eine 6 würfeln

```
SELECT aussage FROM logik
WHERE wahrheitswert IS NOT NULL;
```

aussage
1+2 = 3
2+3 = 4

[3] vgl. de Vries (2007):§4.2.
[4] de Vries (2007):§4.2.
[5] Die erste Ergebnismenge lautet {TRUE, FALSE, NULL}, die zweite Ergebnismenge ist leer.

4.4 Setzen von Standardwerten bei `NULL`-Einträgen

Oft benötigt man Datenauswertungen,in denen ein NULL-Wert einen berechenbaren Wert, z.B. 0, erhält. Dazu gibt es die Funktion

$$\texttt{IFNULL}(x, \texttt{neu}),$$

die einen NULL-Wert durch den Wert `neu` ersetzt, aber ansonsten den originalen Wert x zurückgibt. Bei MS Access heißt diese Funktion `Nz`, bei Oracle `NVL`.

Beispiel 4.7. Wollen wir uns in der Ausgabe unserer Logiktabelle in Bemerkung 4.5 für unbekannte Wahrheitswerte den Wert $\frac{1}{2}$ anzeigen, so erhalten wir

```
SELECT aussage, IFNULL(wahrheitswert, 0.5) AS wahrheitswert FROM logik;
```

die Ergebnismenge

aussage	wahrheitswert
1+2 = 3	1
2+3 = 4	0
Ich werde eine 6 würfeln	0.5

wie in der Łukasiewicz-Logik. □

5 Primärschlüssel: Identifikation von Datensätzen

Kapitelübersicht

5.1 Schlüssel 33
5.2 Wahl eines natürlichen Primärschlüssels 34
5.3 Primärschlüssel und `NULL`-Werte 35
5.4 Künstliche Primärschlüssel 36

5.1 Schlüssel

Ein wichtiges Merkmal einer relationalen Datenbank ist, dass sie keine völlig identischen Datensätze enthalten darf. Jeder Datensatz muss also irgendeine Information enthalten, die sonst kein anderer Datensatz der Tabelle hat. Um dies für eine Datenbank technisch zu gewährleisten, gibt es den Begriff des „Schlüssels“: Ein *Schlüssel* ist eine Kombination von Attributen einer Tabelle, die *eindeutig* ist in dem Sinne, dass sie für jeden Datensatz eine andere Kombination von Attributwerten aufweist. Im Extremfall ist nur die Kombination aller Attributwerte eindeutig, d.h. der Schlüssel muss dann aus allen Attributen zusammengesetzt sein. Durch seinen Schlüssel ist also jeder Datensatz einer Tabelle eindeutig identifizierbar. In unserer „Full House“-Tabelle aus Beispiel 2.1 wäre z.B. die Kombination `(farbe, karte)` ein Schlüssel, nicht aber eines der Attribute `farbe` oder `karte` alleine:

farbe	karte
Kreuz	Ass
Pik	Ass
Herz	König
Herz	Ass
Karo	König

Ein Schlüssel ist *reduzierbar*, falls er für die Tabelle eindeutig bleibt, auch wenn ein Feld aus ihm entfernt wird. Z.B. ist die Kombination `(reihe, titel)` unserer Comicalben-Tabelle reduzierbar, denn das Attribut `reihe` darf weggelassen werden, ohne dass die Eindeutigkeit verloren geht. Damit kann der für relationale Datenbanken wichtigste Schlüssel, der sogenannte Primärschlüssel, definiert werden:

Definition 5.1. Der *Primärschlüssel* einer Tabelle ist ein nicht reduzierbarer Schlüssel, der zur Identifikation der Datensätze ausgewählt wurde. Er heißt *zusammengesetzt*, wenn er aus mehreren Attributen besteht. In der Praxis kann ein Primärschlüssel nachträglich meist nur schwer geändert werden. □

Die Beziehung der verschiedenen Schlüsselbegriffe wird durch die folgende Grafik illustriert:

Es gibt zwei Möglichkeiten in SQL, den Primärschlüssel einer Tabelle schon bei ihrer Erzeugung bestimmen, zum Einen mit der Anweisung:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
   spalte_2 datentyp_2,
     ...
   spalte_n datentyp_n,
   PRIMARY KEY(spalte_x, ..., spalte_y)
);
```

zum Anderen mit der Anweisung

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1 PRIMARY KEY,
   spalte_2 datentyp_2,
     ...
   spalte_n datentyp_n
);
```

In der ersten Variante bestimmt die in den Klammern nach `PRIMARY KEY` aufgeführte Spaltenauswahl diejenigen Spalten der Tabelle, die ihren Primärschlüssel bilden. In der zweiten Variante ist die Spalte, in der der Ausdruck `PRIMARY KEY` steht, gleichzeitig ihr Primärschlüssel.

Bei der Darstellung des Typs der Tabelle in der Notation (1.5) für den Relationentyp (Seite 10) werden die Attribute des Primärschlüssels unterstrichen:

tabelle (spalte_x, ..., spalte_y, spalte_z, ...)

5.2 Wahl eines natürlichen Primärschlüssels

Bildet man den Primärschlüssel aus einer Attributkombination (oder einem einzelnen Attribut) einer Tabelle, so spricht man auch von einem „natürlichen“ Primärschlüssel. Obwohl die Eigenschaften eines Primärschlüssels wohldefiniert sind, ist die Festlegung des Primärschlüssels einer gegebenen Tabelle nicht immer einfach und hängt von dem konkreten Anwendungsfall und seinem Kontext ab.

Beispiel 5.2. Betrachten wir ein einfaches Beispiel, eine Tabelle namens *artikel*. Sie soll alle Artikel eines Werkzeuge produzierenden Betriebes speichern, als Grundlage für einen Angebotskatalog. Ein Artikel ist spezifiziert durch seine Bezeichnung, seinen Preis und seine Farbe:

artikel

bezeichnung	preis	farbe
Hammer	15,– €	braun
Hammer	15,20 €	rot
Mottek	45,10 €	schwarz
Säge	25,– €	blau
Säge	24,90 €	grün
Zange	5,49 €	rot

Die Spalten sind die Felder der Tabelle, und jede Zeile ein Datensatz, also ein spezieller Artikel. In unserer Artikeltabelle ist z.B. die Kombination `(bezeichnung, preis, farbe)` ein Schlüssel, denn diese Kombination ist bei allen Artikeln verschieden. Dieser Schlüssel ist aber reduzibel, der Preis allein ist schon ein Schlüssel. Der Preis kann also in unserem Beispiel als ein Primärschlüssel unserer Tabelle gewählt werden — allerdings ein schlechter! Denn wer garantiert, dass nicht morgen ein Akkuschrauber für 15,20 € angeboten wird, und der Preis ist plötzlich kein Primärschlüssel mehr? Oder was passiert, wenn der Mottek plötzlich in einer Werbungsaktion den Sonderpreis von 30,– € erhält? Ändert er dann seine „Identität“? □

Ein Primärschlüssel sollte zeitlich stabil sein, d.h. seine Werte sollten sich während des Lebenszyklus der Tabelle nicht ändern. Da in der Praxis häufig Datensätze neu in Tabellen eingefügt werden, muss ein Primärschlüssel nicht nur für den aktuellen Datenbestand, sondern auch langfristig eindeutig bleiben. Einen Primärschlüssel zu finden kann also durchaus schwierig werden!

Beispiel 5.3. Unsere Comicsammlung aus Beispiel 3.4 haben wir ohne einen Primärschlüssel angelegt. Was wäre ein sinnvoller Schlüssel? Als erster Kandidat käme da vielleicht der Titel in Frage. Er ist auf jeden Fall stabil, denn er wird sich für ein Album nicht ändern. Aber ist er sicher immer eindeutig? Eigentlich nicht, denn vielleicht erscheint morgen ein Asterix-Album „Das Meisterwerk“, und das könnten wir dann nicht mehr speichern. Dagegen ist die Kombination (Reihe, Band) sicher eindeutig, also auch in Zukunft, denn die Verlage nummerieren ihre Reihen ja systematisch durch. Unser Relationentyp wäre damit also

alben (titel, reihe, band, preis, jahr),

d.h. in SQL:

```sql
CREATE TABLE alben (
  titel varchar(50),
  reihe varchar(50),
  band  smallint,
  preis decimal(4,2),
  jahr  smallint,
  PRIMARY KEY (reihe, band)
);
```

□

5.3 Primärschlüssel und `NULL`-Werte

Eine Besonderheit ergibt sich aus Regel 1 für Attribute eines Primärschlüssels. Denn für die Eindeutigkeit muss der Vergleich der Werte zweier Primärschlüssel entweder wahr oder falsch sein. Das hat die folgende Regel zur „Entitätsintegrität“ zur Konsequenz:

Definition 5.4. *(Entitätsintegrität)* Für ein Attribut eines Primärschlüssels ist der Wert `NULL` nicht zulässig. □

Schon bei der Entwicklung des Konzepts der relationalen Datenbanken schränkte Codd die Möglichkeiten von Attributwerten mit unbekannter Information durch seine „3. Regel“ ein. Entsprechend wird die Entitätsintegrität von SQL (in allen Dialekten) unterstützt. Geben wir in Beispiel 5.3 beispielsweise die Anweisung

```sql
INSERT INTO alben (titel, reihe, jahr) VALUES ('Lucky Luke', 'Lucky Luke', 1976);
```

ein, so erhalten wir eine Fehlermeldung, denn das Attribut `reihe` darf als Bestandteil des Primärschlüssels nicht `NULL` sein. (Vgl. im Gegensatz dazu Beispiel 4.2 der Albentabelle ohne Primärschlüssel.)

5.4 Künstliche Primärschlüssel

Wegen der oben in Abschnitt 5.2 genannten Schwierigkeiten der Wahl eines „natürlichen“ Schlüssels aus der Kombination bestehender Attribute wird häufig ein *künstlicher Primärschlüssel (surrogate key)* als neues Attribut gewählt, der unabhängig von den realen Daten wie (Bezeichnung, Preis, Farbe) jeden Datensatz eindeutig identifiziert. Solch ein Feld ist zweckmäßigerweise eine eindeutige Nummer, oft kurz ID genannt. Das ist der Grund, warum Sie in jedem Verein eine Mitgliedsnummer, in der Hochschule eine Matrikelnummer oder bei einer Firma eine Kundennummer haben! SQL sieht sogar den eigenen Datentyp `SERIAL` vor, mit dem eine solche ID automatisch verwaltet wird, indem sie bei jedem neu einzufügenden Datensatz hochgezählt wird:

```
CREATE TABLE kanten (
  id SERIAL PRIMARY KEY,
  ...
);
```

oder

```
CREATE TABLE kanten (
  id SERIAL,
  ...
  PRIMARY KEY(id)
);
```

Die Anweisung bestimmt also das spezifizierte Attribut als ganzzahligen und automatisch verwalteten Primärschlüssel. Bei einigen RDBMS wie MS Access heißt dieser Datentyp auch `autoincrement`. Ein solcher künstlicher Primärschlüssel heißt entsprechend *seriell*. Wir dürfen beim Einfügen von Datensätzen in eine Tabelle mit einem seriellen Primärschlüssel dessen Wert jedoch nicht festlegen, sondern müssen ihn einfach weglassen.

Beispiel 5.5. Legen wir unsere Comicsammlung aus Beispiel 3.4 und 4.2 mit einem seriellen Primärschlüssel an:

```
CREATE TABLE alben (
  id serial primary key,
  titel varchar(50),
  reihe varchar(50),
  band  smallint,
  preis decimal(4,2),
  jahr  smallint
);
```

so können wir unsere Alben mit den Anweisungen

```
INSERT INTO alben (titel, reihe, band, preis, jahr) VALUES
  ('Asterix, der Gallier', 'Asterix', 1, 2.80, 1968),
  ('Asterix und Kleopatra', 'Asterix', 2, 2.80, 1968),
  ('Gespenster Geschichten', 'Gespenster Geschichten', 1, 1.20, 1974),
  ('Die Trabantenstadt', 'Asterix', 17, 3.80, 1974),
  ('Der große Graben', 'Asterix', 25, 5.00, 1980),
  ('Das Kriminalmuseum', 'Franka', 1, 8.80, 1985),
  ('Das Meisterwerk', 'Franka', 2, 8.80, 1986);
```

und

```
INSERT INTO alben (titel, reihe, jahr) VALUES ('Lucky Luke', 'Lucky Luke', 1976);
```

speichern. Insbesondere können wir also das Album Lucky Luke trotz unbekannter Bandnummer problemlos eintragen. Betrachten wir die gespeicherten Daten mit

```
SELECT * FROM alben;
```

so erhalten wir die Ergebnismenge

id	titel	reihe	band	preis	jahr
1	Asterix, der Gallier	Asterix	1	2.80	1968
2	Asterix und Kleopatra	Asterix	2	2.80	1968
3	Gespenster Geschichten	Gespenster Geschichten	1	1.20	1974
4	Die Trabantenstadt	Asterix	17	3.80	1974
5	Der große Graben	Asterix	25	5.00	1980
6	Das Kriminalmuseum	Franka	1	8.80	1985
7	Das Meisterwerk	Franka	2	8.80	1986
8	Lucky Luke	Lucky Luke	null	null	1976

Wir sehen, dass das RDBMS den seriellen Primärschlüssel automatisch hochgezählt hat. □

6

Daten analysieren und zusammenfassen (GROUP BY)

Kapitelübersicht

6.1 Aggregatfunktionen 38
6.2 Gruppieren nach Spalten mit GROUP BY 40
6.3 Gruppieren mit HAVING: Filtern mit Aggregatfunktionen 41
6.4 WHERE oder HAVING? 42
6.5 * Der OVER-PARTITION-Ausdruck 43

6.1 Aggregatfunktionen

In Abschnitt 2.6 auf Seite 16 haben wir bereits einige Funktionen von SQL kennengelernt. Sie rechnen mit Spaltenwerten der einzelnen Datensätze. Wollen wir jedoch mit Werten *verschiedener* Datensätze rechnen, kommen wir damit nicht weiter. Wir benötigen dafür sogenannte Aggregatfunktionen.

Eine *Aggregatfunktion*, auch *Gruppenfunktion* genannt, ist eine Funktion in SQL, die als Argument einen Spaltennamen erhält und die Spaltenwerte mehrerer Datensätze zu einem einzigen Wert zusammenführt. Tabelle 6.1 gibt einen Überblick über gängige Aggregatfunktionen. Sie umfassen also Funktionen zum Zählen oder Summieren von Einträgen einer Spalte oder

`COUNT(s)`, `COUNT(*)`	die Anzahl der ausgewählten Datensätze
`COUNT(DISTINCT s)`	Anzahl unterschiedlicher Werte für `s` der ausgewählten Datensätze
`MAX(x)`	der größte Eintrag in Spalte `x`
`MIN(x)`	der kleinste Eintrag in Spalte `x`
`SUM(x)`	die Summe aller Einträge von `x`
`AVG(x)`	Der Mittelwert aller Einträge von `x`
`STDDEV(x)`	die empirische Standardabweichung aller Einträge von `x` (D.h. die Einträge werden als Stichprobe einer Gesamtmenge mit unbekanntem Mittelwert betrachtet). Beachte: In MS Access lautet die Funktion `STDEV`!
`VARIANCE(x)`	die empirische Varianz aller Einträge von `x`

Tabelle 6.1: Gängige Aggregatfunktionen in SQL. Hier bezeichnet `s` einen beliebigen Spaltennamen, `x` den Namen einer Spalte von einem numerischen Datentyp

zur Bestimmung des größten oder kleinsten Eintrags. Ebenso kann man statistische Größen wie Mittelwert oder Standardabweichung bestimmen.[1]

Beispiel 6.1. *(Zählen von Datensätzen)* Um die Anzahl von Datensätzen zu bestimmen, können wir die Aggregatfunktion `COUNT` verwenden. Mit

```
SELECT COUNT(*) FROM alben WHERE reihe='Asterix';
```

ermitteln wir die Anzahl aller Datensätze der Reihe Asterix. Die Ergebnismenge lautet:

COUNT(*)
4

Statt dem Sternchen $*$ hätten wir in diesem Fall auch ein beliebiges Attribut der Tabelle `alben` einsetzen können, das Resultat bliebe dasselbe. Wichtig ist das Argument der COUNT-Funktion erst, wenn wir Fragen wie die folgende beantworten möchten: Wieviel *verschiedene* Reihen ungleich Asterix haben wir in unserem Albenbestand? Mit dem Argument `reihe` liefert die Abfrage

```
SELECT COUNT(reihe) FROM alben WHERE reihe <> 'Asterix';
```

dann:

COUNT(reihe)
3

Das ist jedoch die Anzahl aller *Alben*, die nicht Teil der Asterixreihe sind. Um die Anzahl der verschiedenen *Reihen* zu ermitteln, benötigen wir den zusätzlichen Parameter `DISTINCT`:

```
SELECT COUNT(DISTINCT reihe) FROM alben WHERE reihe <> 'Asterix';
```

ergibt

COUNT(DISTINCT reihe)
2

So erhalten wir die korrekte Antwort 2. □

Beispiel 6.2. *(Statistische Auswertungen)* Um sich statistische Größen eines Datenbestandes berechnen zu lassen, können wir einige der Aggregatfunktionen verwenden, zum Beispiel für den Durchschnitt und die Standardabweichung der Albenpreise sowie deren Minimum und Maximum:

```
SELECT ROUND(AVG(preis), 3) AS mittelwert,
       ROUND(STDDEV(preis), 3) AS standardabweichung,
       MIN(preis),
       MAX(preis)
FROM alben;
```

Hierbei wird für die berechneten Werte die Rundungsfunktion verwendet, um sie übersichtlich auszugeben:

mittelwert	standardabweichung	MIN(preis)	MAX(preis)
4.743	2.999	1.20	8.80

[1] In MariaDB und MySQL ist die Standardabweichung nicht die empirische Standardabweichung *(sample standard deviation)* einer Stichprobe von einer Gesamtheit von Daten, deren Mittelwert unbekannt ist, sondern die Standardabweichung im Sinne der Wahrscheinlichkeitstheorie *(population standard deviation)* einer vollständigen Population, deren Mittelwert bekannt ist. Vgl. `https://math.stackexchange.com/questions/15098/`. Will man dort die empirische Standardabweichung berechnen, so muss man `STDDEV_SAMP` verwenden, die übrigens auch in anderen Datenbanksystemen definiert ist.

□

Bemerkung 6.3. Aggregatfunktionen sind deutlich zu unterscheiden von den in Abschnitt 2.6 auf Seite 16 aufgeführten Funktionen, die auf einzelnen Datensätzen wirken, aber nicht auf Gruppen von Datensätzen. Die Abfrage

```
SELECT SQRT(preis) FROM alben
```

zeigt soviel Werte an wie es Datensätze gibt, während

```
SELECT AVG(preis) FROM alben
```

nur *einen* Wert anzeigt. □

Bemerkung 6.4. NULL-Werte werden von Aggregatfunktionen nicht berücksichtigt, also einfach ignoriert. Für den Fall, dass NULL-Werte stattdessen durch einen Standardwert ersetzt werden sollen, können wir die Funktion `ifnull` (bzw. `Nz` bei MS Access oder `NVL` bei Oracle) verwenden. □

Beispiel 6.5. *(Unterabfragen)* Um sich alle Titel anzuzeigen, deren Preis kleiner ist als der Durchschnittspreis aller Alben, muss man zunächst durch die Aggregatfunktion `AVG(preis)` den Durchschnitt berechnen und diesen als einzigen Wert der Ergebnismenge in einer WHERE-Klausel mit den Preisen der Alben filtern:

```
SELECT titel FROM alben WHERE preis < (SELECT AVG(preis) FROM alben);
```

Die in Klammern geschriebene Abfrage ist eine *Unterabfrage (Subquery)*, oft auch *verschachtelte Abfrage* genannt. Sie wird zuerst ausgeführt. □

6.2 Gruppieren nach Spalten mit `GROUP BY`

Mit `GROUP BY` wird in einem SELECT eine Gruppenbildung durchgeführt. Dabei werden jeweils Teilmengen von Ergebniszeilen zu einer Gruppe zusammengefasst, wenn sie in der pder den ausgewählten Spalten – der oder den *Gruppenspalten* – die gleichen Werte besitzen. Gruppenbildungen werden meist vorgenommen, um Aggregatfunktionen wie Summe, Maximum o.ä. für die ganze Gruppe zu berechnen und dann für jede Gruppe separat in einer Zeile auszugeben. Die Syntax lautet für n Gruppenspalten ($n \geqq 1$):

```
SELECT   spalten aus gruppenspalten, [aggegatfunktionen] FROM tabelle
GROUP BY gruppenspalte_1, ..., gruppenspalte_n
```

Hierbei wird nach den n Spalten `gruppenspalte_1`, ..., `gruppenspalte_n` gruppiert und je Gruppe ggf. die Aggregatfunktion(en) je Gruppe berechnet. Dabei dürfen neben Aggregatfunktionen nur Spalten angezeigt werden, die auch Gruppenspalten sind, also nach `GROUP BY` stehen.

Beispiel 6.6. Betrachten wir als typisches Beispiel für eine Gruppenbildung die Frage nach dem Durchschnittspreis der einzelnen Reihen unserer Comic-Alben aus Beispiel 3.4 auf Seite 20.

```
SELECT reihe, AVG(preis) FROM alben GROUP BY reihe;
```

reihe	AVG(preis)
Gespenster Geschichten	1.2
Asterix	3.6
Franka	8.8

Die Gruppenspalte ist hier `reihe` und sollte auch als einzige Spalte der Tabelle im SELECT angezeigt werden. Ansonsten brechen die meisten Datenbanksysteme mit einer Fehlermeldung ab. □

```
SELECT reihe,
       ROUND(AVG(preis), 3),
       ROUND(STDDEV(preis), 2) AS s,
       MIN(preis),
       MAX(preis)
FROM alben
GROUP BY reihe;
```

reihe	AVG(preis)	S	MIN(preis)	MAX(preis)
Gespenster Geschichten	1.2	null	1.20	1.20
Asterix	3.6	1.05	2.80	5.00
Franka	8.8	0.0	8.80	8.80

Regel 2. *Wenn in der Spaltenauswahl einer `SELECT`-Anweisung Spalten* und *Aggregatfunktionen auftreten, müssen alle Spalten in der `GROUP BY`-Komponente aufgelistet werden.*

6.3 Gruppieren mit `HAVING`: Filtern mit Aggregatfunktionen

In Datenbankanwendungen haben wir oft das Problem, Datensätze unter einer Bedingung zu filtern, die eine Aggregatfunktion enthält. Beispielsweise sind das Fragen wie: Welche Artikel wurden mehr als eine Million mal verkauft? Welche Prüflinge haben einen besseren Notendurchschnitt als eine 3? Welche Aktien sind sehr riskant, haben also Kurse mit einer Standardabweichung größer als die Hälfte ihres Mittelwerts?

Zur Lösung solcher Fragestellungen müssen wir auf jeden Fall die Datensätze mit `GROUP BY` geeignet zusammenfassen, also beispielsweise `GROUP BY artikel`, `pruefling` oder `aktie`. Allerdings wollen wir ja gar nicht alle Datensätze sehen, d.h. wir müssen filtern. Erster Ansatz: Verwenden wir doch die `WHERE`-Klausel! Leider wird das nicht funktionieren. Denn die Filterbedingung hängt hier jeweils von (mindestens) einer Aggregatfunktion ab, nämlich `COUNT(artikel)`, `AVG(note)` oder `STDDEV(kurs)`. In einer `WHERE`-Klausel dürfen aber nur Spaltennamen auftreten. Wir *müssen* daher in diesen Fällen die `HAVING`-Klausel verwenden:

```
... GROUP BY ... HAVING <Bedingung mit Aggregatfunktion> ...;
```

Nachdem mit dem Ausdruck `GROUP BY` die Ergebnisdatensätze einer SELECT-Anweisung gruppiert wurden, können mit `HAVING` in diesen Gruppen die Datensätze gefiltert werden, die der Bedingungen genügen. Auch hier ist bei der Auswertung bei NULL-Werten die dreiwertige Logik zu beachten: Sowohl falsche als auch unbekannte Wahrheitswerte fallen heraus.

Beispiel 6.7. Eine typische Fragestellung, die mit der `HAVING`-Klausel gelöst werden kann, ist für unser Comic-Datenbank die folgende: Welche Reihen haben als Durchschnittspreis ihrer Bände einen Wert kleiner 5 €?

```
SELECT reihe, AVG(preis) FROM alben GROUP BY reihe HAVING AVG(preis) <= 5;
```

reihe	AVG(preis)
Gespenster Geschichten	1.2
Asterix	3.6

□

Beispiel 6.8. Um herauszufinden, welche Reihen unserer Comics mehr als 3 Bände enthalten, können wir die folgende Anweisung verwenden:

```
SELECT reihe
FROM alben
GROUP BY reihe
HAVING count(band) > 3
```

Zur Überprüfung vergleichen wir die folgenden Anweisungen ohne und mit Filterbedingung:

```
SELECT reihe, COUNT(band) FROM alben
GROUP BY reihe
```

reihe	COUNT(band)
Gespenster Geschichten	1
Asterix	4
Franka	2

```
SELECT reihe, COUNT(band) FROM alben
GROUP BY reihe HAVING count(band) > 3
```

reihe	COUNT(band)
Asterix	4

Wir erkennen daraus, dass die Aggregatfunktion nicht in der Spaltenauswahl auftreten muss, wenn wir sie nicht angezeigt haben wollen. □

Unsere am Anfang dieses Unterkapitels genannten Beispielprobleme würden also entsprechend durch Abfragen wie folgt gelöst:

```
SELECT artikel FROM verkäufe
GROUP BY artikel
HAVING COUNT(artikel) > 1e6
```

oder

```
SELECT pruefling FROM pruefungen
GROUP BY pruefling
HAVING AVG(note) > 3
```

oder

```
SELECT aktie FROM portfolio
GROUP BY aktie
HAVING STDDEV(kurs) >= 0.5 * AVG(kurs)
```

6.4 `WHERE` oder `HAVING`?

Die HAVING-Klausel unterscheidet sich von der WHERE-Klausel, dass die erstere auf die einzelnen Guppen angewendet wird, letztere dagegen auf jeden einzelnen Datensatz. Auch ist die Abarbeitungsreihenfolge anders: Die WHERE-Klausel wird zunächst die jeden einzelnen Datensatz geprüft, bevor weitere Anweisungen der Abfrage ausgeführt werden. Bei HAVING werden *alle* Datensätze gruppiert, bevor für sie die Bedingung der HAVING-Klausel ausgewertet wird. Für große Datenmengen kann das zu längeren Laufzeiten führen. Auf der anderen Seite können in einer HAVING-Klausel aber auch Werte von Aggregatfunktionen berechnet werden. Das macht in einer WHERE-Bedingung keinen Sinn. Dagegen kann in einer HAVING-Klausel kein Alias verwenet werden.

Beispiel 6.9. Mit der folgenden Abfrage können wir uns die Reihen ungleich Asterix mit deren Anzahl Alben anzeigen lassen.

```
SELECT reihe, COUNT(*) FROM alben
GROUP BY reihe
HAVING reihe <> 'Asterix';
```

Sie liefert uns das Ergebnis:

reihe	COUNT(*)
Gespenster Geschichten	1
Franka	2

Die Abarbeitungsreihenfolge ist hier allerdings ungünstig, zuerst werden alle Datensätze gruppiert und erst danach diejenigen ungleich Asterix gefiltert. Dasselbe Resultat erhalten wir durch die folgende Abfrage:

```
SELECT reihe, COUNT(*) FROM alben
WHERE reihe <> 'Asterix'
GROUP BY reihe;
```

Hier wird zuerst gefiltert und dann erst gruppiert. Das ist effizienter. Hätten wir eine Datenbank mit sehr vielen Datensätzen, so wäre die erste Abfrage merklich langsamer. □

Beispiel 6.10. Wieviel Bände haben die Reihen in unserer Sammlung, die nur aus einem einzigen Wort bestehen? Zunächst ist hier die Bedingung („aus nur einem einzigen Wort“) zu überlegen. Aber mit dem Ansatz, dass eine Reihe aus mindestens zwei Wörtern ja mindestens ein Leerzeichen enthalten muss, kann mit Hilfe des `LIKE`-Operators und der Wildcard die Filterbedingung bilden:

```
reihe NOT LIKE '% %'
```

Da wir nach Reihen gruppieren möchten, können wir diese Bedingung als `HAVING`-Klausel verwenden:

```
SELECT reihe, COUNT(band) FROM alben GROUP BY reihe HAVING reihe NOT LIKE '% %'
```

Genauso gut wäre aber auch möglich, sie entsprechend in eine `WHERE`-Klausel zu packen: oder mit `WHERE`:

```
SELECT reihe, COUNT(band) FROM alben WHERE reihe NOT LIKE '% %' GROUP BY reihe
```

Beachten Sie auch hier die unterschiedliche Position von `HAVING` und `WHERE`. □

Man kann bei einem `GROUP BY` grundsätzlich immer da `HAVING` verwenden, wo auch ein `WHERE` möglich ist. Aber nicht umgekehrt:

Regel 3. *Filterbedingungen, die Aggregatfunktionen enthalten, müssen in die `HAVING`-Klausel. Eine `WHERE`-Klausel darf nur Spalten enthalten.*

6.5 * Der `OVER-PARTITION`-Ausdruck

Der `OVER-PARTITION`-Ausdruck ist eine Möglichkeit, in einer Abfrage Aggregatfunktionen auf einzelne Spalten anzuwenden, ohne dabei die Datensätze in der Ausgabe zusammenzufassen. Sie funktioniert praktisch wie ein „spaltenweises“ GROUP-BY, das die berechneten Ergebnisse allerdigs nicht gruppiert. Wir sprechen hierbei nicht von Gruppen, sondern von sogenannten *Partitionen* oder *Fenstern*. Bei einem `OVER`-Ausdruck handelt sich also um eine *Aggregation über Partitionen*. Die Syntax lautet:

```
SELECT [spaltenliste ...],
       aggregatfunktion (spalte) OVER (PARTITION BY spalte_x, ..., spalte_y)
       [...]
FROM tabelle ... ;
```

Die Funktion vor dem reservierten Wort `OVER` heißt *Fensterfunktion* und muss eine Aggregatfunktion sein. Der Ausdruck `OVER` definiert hier mit Hilfe der Anweisung `PARTITION BY` ein Fenster, d.h. eine benutzerdefinierte Gruppe von Datensätzen der Ergebnismenge, auf die die Fensterfunktion angewendet wird. Für ein Fenster mit `PARTITION BY` bestimmt dabei die Spaltenliste das Partitionierungskriterium. Lässt man die Klammer hinter `OVER` leer, so wird über den gesamten Datenbestand aggregiert. Eine Partition kann in der Klammer mit `ORDER BY` sortiert und so die logische Reihenfolge der Berechnungen durch die Fensterfunktion vorgegeben werden. Die Fensterfunktion wird dabei für jeden auszugebenden Datensatz separat ausgeführt, ganz im Gegensatz zu `GROUP BY`. Bei sehr großen Datenbeständen kann das zu erheblichen Laufzeiten führen.

An dieser Stelle sollten wir die auf den ersten Blick etwas komplizierte Syntax und Funktionsweise dieses Ausdrucks anhand eines Beispiels näher betrachten.

Beispiel 6.11. Wollen wir jeden einzelnen Titel mit der Anzahl der insgesamt in seiner Reihe erschienen Titel herausfinden, ohne sie dabei zu gruppieren, so können wir die folgende Anweisung anwenden:

```
SELECT titel, reihe, count(*) OVER (PARTITION BY reihe) FROM alben;
```

Damit erhalten wir die Ausgabe:

titel	reihe	COUNT(*) OVER (PARTITION BY reihe)
Der große Graben	Asterix	5
Asterix, der Gallier	Asterix	5
Asterix und Kleopatra	Asterix	5
Asterix als Legionär	Asterix	5
Die Trabantenstadt	Asterix	5
Das Kriminalmuseum	Franka	2
Das Meisterwerk	Franka	2
Gespenster Geschichten	Gespenster Geschichten	1

Möchten wir die Titel mit verschiedenen Aggregationsgrößen sehen, beispielsweise die Anzahl der im selben Jahr derselben Reihe erschienen Titel und den mittleren Preis pro Reihe insgesamt, so können wir das mit der folgenden Abfrage erreichen:

```
SELECT titel, reihe, jahr,
       count(*) OVER (PARTITION BY reihe, jahr) AS "Anzahl(reihe,jahr)",
       AVG(preis) OVER (PARTITION BY reihe) AS "mittlerer Preis(reihe)"
FROM alben;
```

Damit erhalten wir dann:

titel	reihe	jahr	Anzahl(reihe,jahr)	mittlerer Preis(reihe)
Asterix als Legionär	Asterix	NULL	1	3.440000
Asterix und Kleopatra	Asterix	1968	2	3.440000
Asterix, der Gallier	Asterix	1968	2	3.440000
Die Trabantenstadt	Asterix	1974	1	3.440000
Der große Graben	Asterix	1980	1	3.440000
Das Kriminalmuseum	Franka	1985	1	8.800000
Das Meisterwerk	Franka	1986	1	8.800000
Gespenster Geschichten	Gespenster Geschichten	1974	1	1.200000

Wollen wir abschließend uns für jeden Titel die Preisabweichung vom mittleren Preis der Reihe anzeigen lassen, so programmieren wir:

```
SELECT titel, preis,
       preis - AVG(preis) OVER (PARTITION BY reihe) AS "Preisdifferenz"
FROM alben;
```

Damit erhalten wir dann:

titel	preis	Preisdifferenz
Asterix und Kleopatra	2.80	-0.640000
Asterix, der Gallier	2.80	-0.640000
Der große Graben	5.00	1.560000
Die Trabantenstadt	3.80	0.360000
Asterix als Legionär	2.80	-0.640000
Das Kriminalmuseum	8.80	0.000000
Das Meisterwerk	8.80	0.000000
Gespenster Geschichten	1.20	0.000000

Eine solche Abfrage wäre mit anderen Ausdrücken wie `GROUP BY`, verschachtelten Abfragen oder Kombinationen daraus nicht möglich. Der `OVER-PARTITION`-Ausdruck ist damit eine echte Spracherweiterung von SQL, die den `WHILE`-Schleifen imperativer Programmiersprachen entspricht. Wenn die gewünschte Ergebnismenge dagegen auch mit einem `GROUP BY` bzw. verschachtelten Abfragen (Beispiel 7.5) erreicht werden kann, sollte `OVER-PARTITION` nicht verwendet werden, da ein `OVER-PARTITION`-Ausdruck sehr rechenaufwändig ist. □

7 Mengenoperationen

Kapitelübersicht

7.1 Mengenlehre: Mathematik der Mengen 46
7.2 Mengenoperationen in SQL 47
7.3 Verschachtelte `SELECT`-Anweisungen: `SELECT` in `SELECT` 49

Mengenoperationen in SQL sind ein Mittel, um die Ergebnismenge mehrerer Datenbankabfragen zu einer einzelnen Ergebnismenge zusammenzufassen. Grundlage dafür ist die Mengenlehre.

7.1 Mengenlehre: Mathematik der Mengen

Eine *Menge* (englisch: *set*) ist eine Zusammenfassung einzelner Elemente mit wohldefinierten Eigenschaften. Die *Mengenlehre* ist das Teilgebiet der Mathematik, das sich mit der Untersuchung von Mengen beschäftigt. Die gesamte moderne Mathematik ist in der Sprache der Mengenlehre formuliert und baut auf ihren Axiomen auf.

Der Begriff der Menge wurde 1895 von dem Mathematiker Georg Cantor formuliert. Zu Beginn des 20. Jahrhunderts wurden durch (heute so genannte) „naive“ Interpretationen des Mengenbegriffs logische Widersprüche entdeckt. Ein Beispiel dafür ist das folgende Paradox, das Russel 1902 in etwas anderer Form für Mengen formulierte und bereits in der Antike bekannt war.[1]

Beispiel 7.1. *(Das Lügnerparadox)* Wenn wir bei der Eingrenzung einer Mengenlehre nicht aufpassen, können wir sehr schnell innere Widersprüche erhalten. Betrachten wir dazu die Menge W aller wahren Aussagen und speziell die Aussage

$$a = \text{„Diese Aussage ist falsch“} \tag{7.1}$$

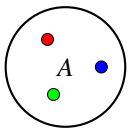
(Beachten Sie, dass die Aussage sich auf sich selbst bezieht.) Ist die Aussage wahr oder nicht? Oder formal: Gilt $a \in W$ oder $a \notin W$? Nehmen wir an, sie ist wahr, so ist ihre Aussage aber falsch: $a \in W \Rightarrow a \notin W$. Nehmen wir umgekehrt an, sie sei falsch, so ist ihre Aussage wahr: $a \notin W \Rightarrow a \in W$. ERROR! □

[1] „Ich sagte [...]: Alle Menschen sind Lügner.“ [Altes Testament, Psalm 116,11 (ca. 200 v. Chr.)]

Die Mengenlehre musste historisch (durch das sogenannte „Regularitätsaxiom“) so eingegrenzt werden, dass sie selbstbezügliche Elemente nicht erlaubt, also solche, die sich selbst enthalten wie in dem Beispiel. In wesentlichen Teilen gelang dies Ernst Zermelo 1907 und endgültig Abraham Fraenkel 1930, indem die Grundlagen der heute ZFC genannten Mengenlehre präzisiert wurden. Allerdings: Obwohl diese Mengenlehre das logische Fundament der modernen Mathematik bildet, kann man nicht beweisen, dass sie für unendliche Mengen in sich widerspruchsfrei ist. (Das folgt aus dem berühmten Zweiten Unvollständigkeitssatz des Mathematikers Kurt Gödel von 1931.)

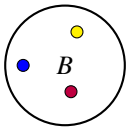
In der Informatik beschäftigen wir uns jedoch ausschließlich mit *endlichen* Mengen. Hier bewegen wir uns auf sicherem Boden, denn für diesen Fall ist die Widerspruchsfreiheit der Mengenlehre bewiesen.[2] Eine endliche Menge wird in der Mathematik üblicherweise durch Aufzählen seiner Elemente innerhalb geschweifter Klammern angegeben, also beispielsweise

$$A = \{\text{rot, grün, blau}\}. \qquad \text{(7.2)}$$

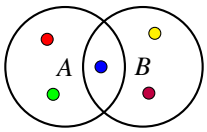


Betrachten wir eine zweite Menge

$$B = \{\text{blau, gelb, lila}\}, \qquad \text{(7.3)}$$

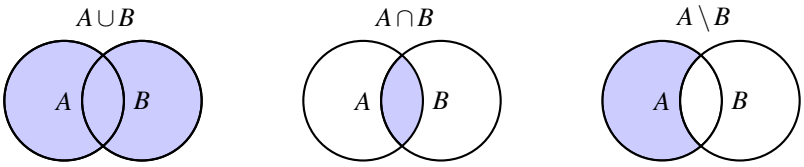


so können wir A und B durch ein Venn-Diagramm darstellen:



(7.4)

Entsprechend können wir die Mengenoperationen Vereinigung ($\cup$), Schnitt ($\cap$) und Differenz ($\setminus$) visualisieren:



Es gilt also $A \cup B = \{\text{rot, grün, blau, gelb, lila}\}$, $A \cap B = \{\text{blau}\}$ und $A \setminus B = \{\text{rot, grün}\}$.

7.2 Mengenoperationen in SQL

In SQL können Ergebnismengen von Datenbankabfragen mit den Mengenoperationen **`UNION`**, **`INTERSECT`**, **`EXCEPT`** zu einer einzigen Ergebnismenge zusammengefügt werden. Bei diesen Mengenoperationen gelten zwei Datensätze als gleich, wenn sie dieselben Spalteneinträge haben. Mit dieser Entsprechung ist **`UNION`** die Vereinigung zweier Ergebnismengen, **`INTERSECT`** deren Schnittmenge und **`EXCEPT`** deren Differenz. Eine weitere Mengenoperation ist **`IN`**:

```
SELECT ... FROM tabelle WHERE spalte IN (x, y, ...)
```

[2] https://en.wikipedia.org/wiki/General_set_theory

Sie prüft in einer `WHERE`-Klausel, ob ein ein Wert des Attributs `spalte` in der Menge $\{x, y, \dots\}$ enthalten ist.

Beispiel 7.2. Betrachten wir als erstes Beispiel für Mengenoperationen in SQL den Relationentyp `mengen(menge, element)` mit den folgenden Einträgen:

menge	element
A	rot
A	grün
A	blau

menge	element
B	blau
B	gelb
B	lila

Das ist genau eine Datenbankdarstellung unserer obigen mathematischen Mengen A und B. In SQL erzeugen wir sie wie folgt:

```
-- Tabellenstrukturen
CREATE TABLE mengen (
  menge   varchar(1),
  element varchar(4)
);
-- ------
-- Daten:
INSERT INTO mengen (menge, element) VALUES
  ('A', 'rot'),
  ('A', 'grün'),
  ('A', 'blau'),
  ('B', 'blau'),
  ('B', 'gelb'),
  ('B', 'lila');
```

Wenden wir nun die Mengenoperationen darauf an:

```
SELECT element AS "A ∪ B" FROM mengen WHERE menge = 'A'
UNION
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT element AS "A ∩ B" FROM mengen WHERE menge = 'A'
INTERSECT
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT element AS "A \ B" FROM mengen WHERE menge = 'A'
EXCEPT
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT menge, element FROM mengen WHERE element IN ('gold', 'lila', 'blau');
```

(Die Zeichen ∩ und ∪ haben die Unicodes 0x2229 und 0x222A). Dann erhalten wir jeweils die Ausgaben

$A \cup B$
lila
gelb
grün
blau
rot

$A \cap B$
blau

$A \setminus B$
grün
rot

menge	element
A	blau
B	lila
B	blau

Das entspricht genau unserem mathematischen Bild in Gleichung (7.4). □

7.3 Verschachtelte `SELECT`-Anweisungen: `SELECT` in `SELECT`

Eine verschachtelte `SELECT`-Anweisung ist eine Abfrage mit einer `WHERE`-Klausel auf eine andere SELECT-Anweisung, die ja auch eine – wenn auch nur temporäre – Tabelle ist. Hier wird also die Abfrage an eine Haupttabelle gestellt, wobei die WHERE-Bedingung eine Nebentabelle zum Vergleich heranzieht. Haupt- und Nebentabelle können dabei identisch sein, dürfen aber durchaus auch verschieden sein. Aus der Nebentabelle können ein oder mehrere Vergleichswerte entnommen werden. Eine solche Anweisung wird auch oft *Unterabfrage (Subquery)* oder *innere Abfrage* genannt.

```
SELECT ... FROM haupttabelle WHERE ... (
  SELECT ... FROM nebentabelle
);
```

Die Reihenfolge der Abarbeitung dieser Anweisung geht von innen nach außen (bzw. hier von unten nach oben): Zunächst wird die Unterabfrage ausgeführt, danach die Hauptabfrage. Verschachtelungen von Abfragen können nur deshalb funktionieren, da jede SELECT-Anweisung nach Bemerkung 3.2 auf Seite 19 als Ergebnismenge stets wieder eine Tabelle liefert.

Wir können grundsätzlich zwei Arten verschachtelter SELECT-Anweisungen unterscheiden: Solche, die auf einen Vergleichswert aus einer Untertabelle abfragen, und solche, die auf einen Wertebereich der Untertabelle abfragen. Erstere können einfach mit den bekannten Vergleichsoperatoren verwendet werden, für Letztere müssen wir spezielle Mengenoperatoren verwenden.

7.3.1 Unterabfragen auf einen einzigen Vergleichswert

Besteht eine Unterabfrage als Ergebnismenge aus genau einen Wert, beispielsweise ein Attributwert eines einzelnen Datensatzes oder dem Ergebnis einer Aggregatfunktion, so kann dieser mit dem Vergleichsoperator `=` in der `WHERE`-Klausel der Hauptabfrage bearbeitet werden.

```
SELECT spalte_1, ..., spalte_n FROM haupttabelle
WHERE spalte_x = (
  SELECT [spalte_y | aggregatfunktion] FROM nebentabelle ...
);
```

Handelt es sich bei dem Wert der Ergebnismenge um einen String, so kann statt des Vergleichsoperators entsprechend auch der `LIKE`-Operator verwendet werden. Handelt es sich bei dem Wert der Ergebnismenge dagegen um eine Zahl, so können auch die numerischen Vergleichsoperatoren `>`, `>=`, `<`, `<=` oder `<>` verwendet werden.

Beispiel 7.3. Wollen wir in unserer Comicsammlung wissen, welches die nach Erscheinungsjahr ältesten Alben sind, so können wir das nicht mit der Anweisung

```
SELECT titel, jahr FROM alben WHERE jahr = MIN(jahr);
```

denn dies widerspräche der Regel 3 auf Seite 43. Aber auch wenn wir `WHERE` durch `HAVING` ersetzen würden, kämen wir nur zu einer Fehlermeldung, da wir nicht nach Titel und Preis gruppieren. Wir müssen hier ein verschachteltes SELECT verwenden:

```
SELECT titel, jahr FROM alben WHERE jahr = (
  SELECT MIN(jahr) FROM alben
);
```

Hier ermittelt die Unterabfrage zunächst das minimale Jahr, bevor die äußere Abfrage alle Alben findet, die diesem Erscheinungsjahr gleichen. Man könnte schnell auf die Idee kommen, dass das die Abfrage auch □

Beispiel 7.4. Wollen wir in unserer Comicsammlung diejenigen Titel finden, deren Preis kleiner als der Durchschnittspreis aller Alben ist, so erreichen wir dies mit der Anweisung:

```
SELECT titel, preis FROM alben WHERE preis < (
  SELECT AVG(preis) FROM alben
);
```

Hier ermittelt die Unterabfrage zunächst den Durchschnittspreis (hier 4,74), mit dem die äußere Abfrage die billigeren Alben filtert. Damit erhalten wir

titel	preis
Asterix, der Gallier	2,80
Asterix und Kleopatra	2,80
Gespenster Geschichten	1,20
Die Trabantenstadt	3,80

als Ergebnismenge. □

Beispiel 7.5. Wollen wir – ähnlich wie in Beispiel 6.11 – die Titel einer speziellen Reihe mit ihren Preisdifferenzen zum Durchschnittspreis der Reihe sehen, so können wir das mit der folgenden verschachtelten Anweisung erreichen. Mit

```
SELECT titel, preis, preis - (
   SELECT AVG(preis) FROM alben WHERE reihe = 'Asterix'
) AS "Preisdifferenz"
FROM alben
WHERE reihe = 'Asterix'
```

erhalten wir beispielsweise die Titel der Asterixreihe mit ihren Differenzen zum Durchschnittspreis der Reihe:

titel	preis	Preisdifferenz
Asterix und Kleopatra	2.80	-0.640000
Asterix, der Gallier	2.80	-0.640000
Der große Graben	5.00	1.560000
Die Trabantenstadt	3.80	0.360000
Asterix als Legionär	2.80	-0.640000

Diese Anweisung ist von der Laufzeit her effizienter als die entsprechende Abfrage mit OVER-PARTITION in Beispiel 6.11. □

7.3.2 Unterabfragen auf mehrere Vergleichswerte

Besteht die Ergebnismenge der Unterabfrage aus Attributwerten mehrerer Datensätze, so können wir die Vergleichsoperatoren nicht mehr verwenden. Stattdessen müssen wir einen der *Mengenoperatoren* `EXISTS`, `IN`, `ANY` oder `ALL` verwenden. Ihre jeweilige Wirkung auf die Ergebnismenge `(...)` einer Unterabfrage ist wie folgt:

SQL Operator	Wirkung
`EXISTS (...)`	Prüft, ob die Unterabfrage überhaupt Daten liefert
x `IN (...)`	Prüft, ob der Wert für x in der Ergebnismenge der Unterabfrage vorkommt
$x \lessgtr$ `ANY (...)`	Prüft, ob $x \lessgtr$ irgendeinem Wert in der Ergebnismenge ist
$x \lessgtr$ `ALL (...)`	Prüft, ob $x \lessgtr$ jedem Wert in der Ergebnismenge ist

Hierbei bezeichnet $\lessgtr$ einen der Vergleichsoperatoren `<`, `<=`, `>`, `>=` oder `=`. Alle diese Ausdrücke müssen Teil einer `WHERE`- oder `HAVING`-Klausel sein. Die Operatoren `ANY` und `ALL` dienen dazu, Vergleiche mit dem Minimum oder dem Maximum einer Menge durchzuführen. Wollen wir bespielsweise diejenigen Werte einer Spalte `x` filtern, die kleiner gleich dem Minimum bzw. Maximum einer Ergebnismenge `(...)` sind, so erreichen wir dies durch

SQL:	`x <= ALL(...)`	`x <= ANY(...)`
Mathematisch:	$x \leqq \min(\ldots)$	$x \leqq \max(\ldots)$

Umgekehrt filtern wir mit

SQL:	`x >= ALL(...)`	`x >= ANY(...)`
Mathematisch:	$x \geqq \max(\ldots)$	$x \geqq \min(\ldots)$

diejenigen Werte, die größer gleich dem Maximum, bzw. dem Minimum, von `(...)` sind.

`NULL`-Werte Enthält die Unterabfrage einen `NULL`-Wert, so liefert der `ALL`-Operator *nie* den Wert `TRUE`! Der Grund ist, dass bei einem unbekannten Wert in der Menge ja auch unbekannt ist, ob das Vergleichskriterium erfüllt ist. Bei `ANY` dagegen wird ein `NULL`-Wert einfach ignoriert. Ferner sind strenggenommen die Operatoren `= ANY` bzw. `= ALL` in SQL überflüssig, denn einerseits ist `= ANY` völlig äquivalent zu `IN`, und andererseits liefert `= ALL` bei einer Ergebnismenge mit unterschiedlichen Werten nie `TRUE`.

Beispiel 7.6. Wollen wir in unserer Comic-Datenbank aus Beispiel 3.4 auf Seite 20 herausfinden, welche Titel insgesamt in einem Jahr veröffentlicht wurden, in denen ein Asterix-Album erschien, so können wir das mit der Anweisung erreichen:

```
SELECT jahr, titel FROM alben WHERE jahr IN (
   SELECT jahr FROM alben WHERE reihe='Asterix'
);
```

Das Ergebnis dieser Abfrage lautet:

jahr	titel
1968	Asterix, der Gallier
1968	Asterix und Kleopatra
1974	Gespenster Geschichten
1974	Die Trabantenstadt
1980	Der große Graben

Wollen wir stattdessen herausfinden, welche Alben aus anderen Reihen im selben Jahr wie ein Asterix-Album erschienen, sollten wir die Abfrage etwas modifizieren:

```
SELECT jahr, titel FROM alben WHERE reihe <> 'Asterix' AND jahr IN (
   SELECT jahr FROM alben WHERE reihe='Asterix'
);
```

Das liefert als Ergebnismenge nur die Gespenster Geschichten aus dem Jahr 1974. Wollen wir die Titel mit den günstigsten Preisen einer Reihe sehen, so können wir die folgende Abfrage senden:

```
SELECT titel, preis FROM alben WHERE preis IN (
  SELECT min(preis) FROM alben GROUP BY reihe
);
```

Sie liefert uns die Ergebnismenge:

titel	preis
Asterix, der Gallier	2.80
Asterix und Kleopatra	2.80
Gespenster Geschichten	1.20
Der große Graben	5.00
Das Kriminalmuseum	8.80
Das Meisterwerk	8.80
Asterix als Legionär	2.80

Wollen wir die Titel sortiert haben, so können wir `ORDER BY titel` anfügen.

Um schließlich herauszufinden, welche Titel einen Preis kleiner gleich den Durchschnittspreisen aller Reihen haben, müssen wir den `ALL`-Operator anwenden. Die Unterabfrage zur Ermittlung der Durchschnittspreise lautet

```
SELECT AVG(preis) FROM alben GROUP BY reihe HAVING AVG(preis) IS NOT NULL
```

und liefert die Werte

avg(preis)
1.20
3.60
8.80

Mit der `HAVING`-Klausel wird garantiert, dass kein `NULL`-Wert in der Ergebnismenge erscheint. Darauf können wir unsere Abfrage aufbauen:

```
SELECT titel, preis FROM alben WHERE preis <= ALL (
  SELECT AVG(preis) FROM alben GROUP BY reihe HAVING AVG(preis) IS NOT NULL
);
```

die als einzige Ergebniseintrag die Gespenster Geschichten mit dem Preis 1,20 liefert. □

8 Entity-Relationship Modelle

Kapitelübersicht

8.1 Probleme mit Daten in einer einzigen Tabelle 53
8.2 Datenmodellierung 55
8.3 ER-Diagramme 56

Wir haben in diesem Skript bis hierher erste Eindrücke von Datenbanken und der Programmierung mit SQL gewonnen. Wir wissen, wie wir per SQL Datenbanken und Tabellen erstellen und Daten damit speichern und sich anzeigen lassen kann. Aber vielleicht fragen Sie sich jetzt: Brauchen wir dazu wirklich Datenbanken? Etwas provozierend formuliert: Alles, was wir bis jetzt mit Datenbanken gemacht haben, hätten wir auch mit Excel hingekriegt! (Beziehungsweise jedem anderen Tabellenalkulationsprogramm wie z.B. LibreOffice Calc)

Wesentliche Ursache für diesen Eindruck ist zum Einen, dass wir mit unseren Datenbanken nur als Einzelnutzer gearbeitet haben. Die meisten Datenbanksysteme ermöglichen jedoch den gleichzeitigen Zugriff mehrerer Nutzer, eine Eigenschaft, die Excel so nicht bietet. Ein zweiter Grund liegt darin, dass wir bisher alle unsere Daten stets in einer einzigen Tabelle gespeichert haben. Die Konstruktion relationaler Datenbanksysteme ist aber gerade so angelegt, dass sie mehrere Tabellen mit logischen oder inhaltlichen Beziehungen untereinander speichern und manipulieren können. Damit gelingt es, den Datenhaushalt eines komplexen Systems wie zum Beispiel eines Unternehmens mit verschiedenen Beschäftigten, Produkten und Geschäftsprozessen strukturiert zu speichern, so dass zu jedem Zeitpunkt aktuelle Informationen von verschiedenen Nutzern verarbeitet werden können.

Den Entwurf und die Implementierung mehrerer Tabellen und deren Beziehungen untereinander werden wir für den Rest des Skripts behandeln. Betrachten wir aber zunächst einleitend, welche Probleme bei Datenspeicherung in einer einzigen Tabelle auftauchen können.

8.1 Probleme mit Daten in einer einzigen Tabelle

Wozu mehrere Tabellen? Das folgende Beispiel soll für einen einigermaßen realistischen Anwendungsfall kurz zusammenfassen, was wir mit einer einzigen Tabelle schaffen können, und was die Nachteile sind.

Beispiel 8.1. Ein produzierendes Unternehmen will eine Datenbank nutzen, um seine Umsätze zu speichern. Es setzt dafür eine Tabelle mit folgendem Relationentyp ein:

```
CREATE TABLE umsaetze (
  id          int SERIAL,
  datum       date,
  artikel     varchar(50),
  einzelpreis decimal(10,2),
  anzahl      int,
  kunde       varchar(50),
  wohnort     varchar(50),
  umsatz      decimal(10,2)
) DEFAULT CHARSET=UTF8;
```

Alles funktioniert wunderbar, das Unternehmen speichert die Umsätze für Oktober und November 2019:

ID	Datum	Artikel	Einzelpreis	Anzahl	Kunde	Wohnort	Umsatz
1	2023-11-06	Hammer	9.90	2	Anna	Hagen	19.80
2	2023-11-02	Säge	4.95	3	Bert	Meschede	14.85
3	2023-10-03	Hammer	9.90	1	Otto	Soest	9.90
4	2023-10-04	Mottek	12.95	1	Anna	Hagen	12.95
5	2023-11-06	Säge	4.95	2	Doro	Iserlohn	9.90
6	2023-10-06	Zange	3.90	4	Anna	Hagen	15.60
7	2023-11-06	Zange	3.90	3	Doro	Iserlohn	11.70
8	2023-10-06	Hammer	9.90	1	Otto	Soest	9.90
9	2023-11-05	Säge	4.95	4	Anna	Hagen	19.80
10	2023-11-06	Mottek	12.95	2	Anna	Hagen	25.90
11	2023-10-04	Mottek	12.95	3	Bert	Meschede	38.85
12	2023-10-06	Zange	3.90	1	Bert	Meschede	3.90

Die Geschäftsführung ist sehr zufrieden, sie kann alle ihr wichtigen Auswertungen durchführen lassen, zum Beispiel ...

- Welcher Kunde hat welchen Artikel gekauft?
  ```
  SELECT kunde, artikel FROM umsaetze ORDER BY kunde, artikel;
  ```
- Welcher Gesamtumsatz wurde im November 2023 erzielt?
  ```
  SELECT sum(umsatz) FROM umsaetze WHERE datum BETWEEN '2023-11-01' AND '2023-11-30'
  ```
- In welchen Ort wurde welcher Artikel wie oft verkauft?
  ```
  SELECT wohnort, artikel, sum(anzahl) FROM umsaetze GROUP BY wohnort, artikel
  ```
- Welcher Umsatz wurde je Artikel erzielt, absteigend sortiert nach Gesamtumsatz?
  ```
  SELECT artikel, sum(umsatz) AS gesamt FROM umsaetze GROUP BY artikel
  ORDER BY gesamt DESC;
  ```
- Welcher Artikel wurde im Oktober am meisten verkauft?
  ```
  SELECT artikel, sum(anzahl) AS gesamt FROM umsaetze
  WHERE datum BETWEEN '2023-10-01' AND '2023-10-31'
  GROUP BY artikel HAVING gesamt >= ALL (
    SELECT sum(anzahl) FROM umsaetze
    WHERE datum BETWEEN '2023-10-01' AND '2023-10-31'
    GROUP BY artikel
  );
  ```

  Alternativ würde hier übrigens auch eine Abfrage mit drei SELECTs funktionieren:

```
SELECT artikel, SUM(anzahl) FROM umsaetze
WHERE datum BETWEEN '2019-10-01' AND '2019-10-31'
GROUP BY artikel HAVING SUM(anzahl) = (
  SELECT MAX(gesamt) FROM (
    SELECT sum(anzahl) AS gesamt FROM umsaetze
    WHERE datum BETWEEN '2019-10-01' AND '2019-10-31'
    GROUP BY artikel
  )
);
```

Mit der innersten Abfrage wird wie oben eine Liste der Gesamtumsatzzahlen je Artikel ermittelt, für die aber nun in einem zweiten SELECT die Aggregatfunktion `MAX` zur Ermittlung des größten dieser Werte zwischengeschaltet wird, auf den dann in der äußersten Schleife in der `HAVING`-Klausel abgefragt wird. In der innersten Unterabfrage muss dabei allerdings ein Alias für Gesamtumsatzzahl benannt werden.

Aus Sicht der Geschäftsführung scheint also alles gut zu sein. Aber im Lauf der Zeit zeigen sich Probleme:

- Beim Einfügen eines neuen Umsatzes weiß ein Sachbearbeiter nicht den Wohnort des Kunden:

  ```
  INSERT INTO umsaetze (datum, artikel, einzelpreis, anzahl, kunde, umsatz)
  VALUES ('2019-10-21', 'Hammer', 9.90, 1, 'Otto', 9.90);
  ```

  Damit wird das Ergebnis unserer obigen Abfrage „In welchen Ort wurde welcher Artikel wie oft verkauft?“ fehlerhaft, obwohl die Datenbank den Ort ja „kennt“.

- Es wird unnötiger Speicherplatz verbraucht, denn dieselbe Information (Artikel-Einzelpreis, Kunde-Wohnort) wird bei *jedem einzelnen* Umsatz gespeichert.

Kann man diese Probleme beheben oder muss man halt damit leben? □

Gehen wir den in dem Beispiel auftauchenden Problemen auf den Grund. Können wir unsere Datenspeicherung anders strukturieren, so dass sie möglichst vermieden werden? Von einem abstrakten Standpunkt aus gesehen handelt es sich bei dem ersten Problem um einen Spezialfall von *Dateninkonsistenz*, also eines Zustands der Datenbank, in der Datensätze sich widersprechende Informationen enthalten. Bei dem zweiten Problem handelt es sich um *Speicherineffizienz*, ein Mangel, der sich bei sehr großen Datenbeständen wie bei einer Unternehmensdatenbank erheblich auswirken kann.

Was ist die tiefere Ursache dieser Mängel? Beide sind bedingt durch *Redundanz* von Daten, also die mehrfache und damit prinzipiell überflüssige Speicherung identischer Information. Wie kann man sie strukturell beschränken? Die Antwort der Theorie relationaler Datenbanken: Durch die Aufteilung zusammenhängender Informationen in mehrere Tabellen. In unserem obigen Beispiel würden wir also nicht alle Informationen eines Umsatzes in einer einzigen Tabelle speichern, sondern die Kundendaten in einer eigenen Tabelle für Kunden, die Artikeldaten in einer eigenen Artikeltabelle, und die spezifisch umsatzbezogenen Informationen in einer eigenen Umsatztabelle. Wie diese drei Tabellen aussehen könnten, werden wir am Ende dieses Kapitels sehen. Betrachten wir zunächst das Vorgehen, wie wir komplexe Informationgeflechte in realen Kontexten analysieren und damit besser verstehen kann, nämlich die Modellierung von Daten.

8.2 Datenmodellierung

Warum Datenmodellierung? Ein Softwaresystem deckt stets nur einen kleinen Teil der realen Welt ab, d.h. ein „abstrahiertes“ Modell. Zu diesem Modell gehören einerseits Prozesse und

Abläufe, andererseits Daten und Informationen. Wie wir bereits in Abbildung 1.1 auf Seite 7 sahen, sind die Daten in der Informatik hierarchisch organisiert, so dass Computer sie verarbeiten können. Um die Daten der realen Anforderungen in diese Hierarchie zu bringen, müssen wir sie modellieren, d.h. ein Modell der Daten entwerfen.

Datenmodellierung dient im Allgemeinen mindestens drei Zielen:

- Strukturierung der Anforderungen, um sie für die Entwickler programmierbar zu machen;
- Erleichterung der Kommunikation zwischen Anwendern und Entwicklern des Softwaresystems;
- Strukturierung der Daten, um sie effizient speichern und manipulieren zu können.

Die Datenmodellierung ist die Grundlage für den *logischen Datenbankentwurf*, also den Entwurf der Tabellen mit Attributen und Primärschlüsseln. Das ist die mittlere Ebene der ANSI-SPARC-Architektur in Abbildung 1.2 auf Seite 8. Als Methodik dazu verwendet man üblicherweise zur besseren Anschauung und Übersicht Diagramme, die Entity-Relationship-Diagramme. Zusammen mit einer Beschreibung der darin verwendeten Elemente bilden sie das *Entity-Relationship-Modell (ERM)*. Die grundlegende Idee ist, dass eine *Entity* das abstrahierte Modell einer Tabelle ist, so dass sich aus einem ERM die Tabellen der relationalen Datenbank in der Regel auf eindeutige Weise ableiten lassen.

8.3 ER-Diagramme

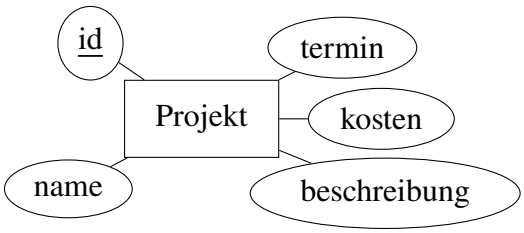
Für den logischen Datenbankentwurf werden die zu speichernden Daten meist mit *Entity-Relationship-Diagrammen (ER-Diagrammen)* grafisch modelliert. Dazu können verschiedene Notationen verwendet werden, die älteste und meistverbreitete ist die *Chen-Notation*.[1] Sie wird auch in diesem Skript verwendet. Es besteht aus drei Grundkomponenten: Dem Entitätstyp, der Beziehung und den Kardinalitäten.

8.3.1 Der Entitätstyp

Eine *Entität (entity)* im ERM ist ein individuelles Objekt der zu modellierenden Welt. Das kann ein realer Gegenstand sein wie ein Exemplar *Asterix der Gallier*, der Angestellte Otto Meier oder der Hammer mit der Artikelnummer 4711, aber auch ein immaterieller Vorgang, wie die Bestellung vom 1. April, oder ein Konstrukt wie der Liefervertrag Nr. 123 oder das Unternehmen ABC AG. Der *Entitätstyp (entity type)* ist der Oberbegriff oder die Kategorie einer oder mehrerer gleichartiger Entitäten, also *Buch*, *Angestellter*, *Artikel*, *Bestellung*, *Vertrag*, *Projekt* oder *Unternehmen*. (Die Entität ist daher eng verwandt mit dem Objektbegriff der Objektorientierung, der Entitätstyp entsprechend mit dem Begriff der Klasse.) Ein Entitätstyp wird im ER-Diagramm mit einem Rechteck dargestellt, in dem der Name des Typs steht:

Eine Entität hat in der Regel mehrere *Attribute*. Eines davon ist der Primärschlüssel der Entität, der sie unter allen anderen Entitäten stets eindeutig identifiziert. Häufig werden die Attribute als kleine Ellipsen mit dem Entitätstyp verbunden oder alternativ in dem Rechteck des Entitätstypen aufgelistet, also zum Beispiel:

[1] Chen (1976).



Projekt
- id
- name
- termin
- kosten
- beschreibung

Wir werden oft jedoch den Entitätstyp etwas kompakter in tabellarischer Form darstellen:

Entitätstyp	**Primärschlüssel**	**Attribute**
Projekt	id	name, termin, kosten, beschreibung

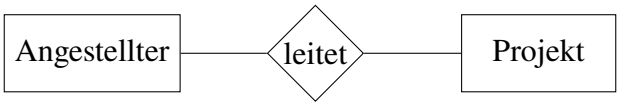
In dieser Form können auch größere und komplexere Datenmodelle übersichtlich dargestellt werden, indem jeder Entitätstyp seine eigene Zeile bekommt.

Bemerkung 8.2. Wir erkennen hier leicht, dass die Entitätstypen in einem ER-Diagramm die Tabellen einer relationalen Datenbank modellieren werden. Entsprechend sind die Attribute der Entitäten die Spalten der Tabelle, und eine Entität entspricht einem konkreten Datensatz, also einer Zeile der Tabelle. Bei der Bildung der Tabellen aus den Entitätstypen eines ER-Diagramms sind aber einige Regeln zu beachten. Damit werden wir uns im nächsten Kapitel näher beschäftigen. □

Bemerkung 8.3. Meist wird der Begriff Entitätstyp etwas unpräzise einfach auch mit „Entität“ bezeichnet. Wir werden das im Folgenden auch oft tun, wenn aus dem Zusammenhang ersichtlich bleibt, was gemeint ist. □

8.3.2 Die Beziehung

Eine *Beziehung (relationship)* im ERM ist der Zusammenhang zwischen zwei Entitäten. Sie wird in einem ER-Diagramm mit einem Verb in eine Raute zwischen zwei Entitätstypen dargestellt, z.B.:



8.3.3 Die Kardinalität

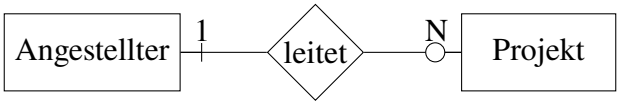
Eine *Kardinalität (cardinality)* drückt aus, wieviel Entitäten des einen Typs mit wieviel Entitäten des anderen Typs eine gegebene Beziehung maximal haben können und mindestens haben müssen. Dabei wird für die Maximalzahl eine der Möglichkeiten 1 oder N für „eins“ oder „mehrere“ über der Beziehung geschrieben, und ein Kreis oder ein Strich an der Beziehungslinie für die Mindestzahl. Damit gibt es vier mögliche Kardinalitäten:

Symbol	1 ┼	1 ○	N ┼	N ○
Modifizierte Chen-Notation	1	C	M	CM
Alternative Bezeichnung	1 „muss“	1 „kann“	N „muss“	N „kann“

(8.1)

Unter die Kardinalitäten sind die im Englischen üblichen Bezeichnungen 1, C (für “choice”), M (für “many”) und CM dargestellt, die sogenannte „Modifizierte Chen-Notation“. Im Deutschen

wird bei einem Strich auch oft von einer *Muss-Beziehung* gesprochen, bei einem Kreis von einer *Kann-Beziehung*. In einem ER-Diagramm wird für eine konkrete Entität des einen Entitätstyps die Kardinalität an dem jeweils gegenüberliegenden Entitätstypen notiert. Betrachten wir dazu das folgende Beispiel:



Ein*e Angestellte*r *kann* hier mehrere Projekte leiten, ein Projekt aber *muss* von genau einem oder einer Angestellten geleitet werden. Dies ist also eine 1-CM-Beziehung, oder auf deutsch etwas umständlicher eine Beziehung „1 zu N muss-kann“. Skizzieren wir die einzelnen Entitäten Angestellter_1, Angestellter_2, . . . , Angestellter_m und Projekt_1, Projekt_2, . . . , Projekt_n und ihre konkreten Beziehungen, so erhalten wir die Abbildung 8.1. Bei einer 1-CM-Beziehung

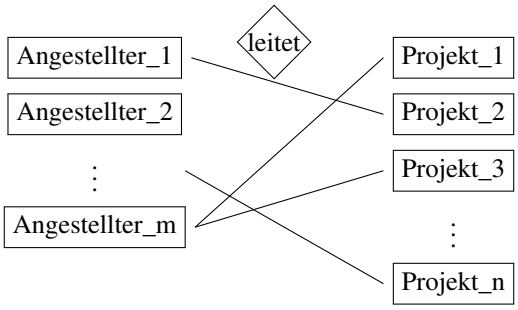
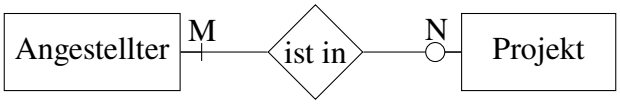


Abbildung 8.1: Beziehungen einzelner Entitäten zueinander bei einer 1-CM-Beziehung. Hier kann es auf der linken Seite Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite genau eine Beziehung haben muss.

kann es also auf der 1-Seite Entitäten geben, die keine Beziehung haben (z.B. Angestellter_2), während jede Entität der N-Seite genau eine Beziehung haben muss. Entsprechend wird für eine Beziehung „mehrere zu mehrere“ üblicherweise das Paar „M : N“ verwendet wie bei folgender M-CM-Beziehung:



Ein*e Angestellte*r kann hier in mehreren Projekten sein, ein Projekt muss mindestens eine oder einen Angestellten haben. (Man verwendet hier „M“ statt „N“ um zu verdeutlichen, dass beide Kardinalitäten unabhängig voneinander sind, also auf beide Seiten nicht die gleich viele Entitäten existieren müssen.) Bei einer M-CM-Beziehung kann es auf der linken Seite also Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite mindestens eine Beziehung haben muss.

Beispiel 8.4. *(ERM der Umsatzdatenbank aus Beispiel 8.1).* Wie können wir ein Datenmodell einer Umsatzdatenbank aus unserer ersten Version in Beispiel 8.1 entwerfen? Zunächst müssen wir dazu die Leitfrage beantworten: Welche der relevanten Daten gehören zusammen? Daraus nämlich ergeben sich die Entitätstypen und ihre Attribute. In dem Beispiel können wir leicht erkennen, dass wir drei Entitätstypen haben:

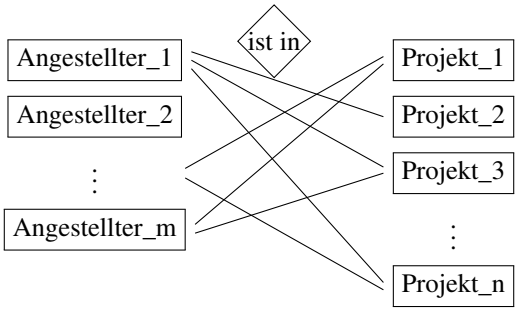
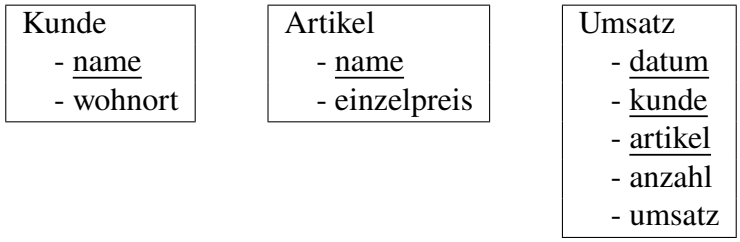


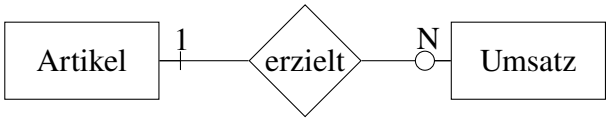
Abbildung 8.2: Beziehungen einzelner Entitäten zueinander bei einer M-CM-Beziehung. Hier kann es auf der linken Seite Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite mindestens eine Beziehung haben muss.



Oder etwas kompakter in tabellarischer Form:

Entitätstyp	**Primärschlüssel**	**Attribute**
Kunde	name	wohnort
Artikel	name	einzelpreis
Umsatz	datum, kunde, artikel	wohnort

Welche Beziehungen haben wir nun zwischen den Entitäten? Dazu müssen wir zunächst die Frage beantworten: Welche Entitätspaare gibt es? Kombinatorisch gesehen haben wir $\binom{3}{2} = 3$ Paare für drei Entitäten, nämlich Artikel – Umsatz, Kunde – Umsatz und Kunde – Artikel. Die erste Beziehung davon lässt sich darstellen als:



$\Rightarrow$ 1-CM (1:N „muss-kann“)

Mit einem gegebenen Artikel können also mehrere Umsätze erzielt werden, es kann aber auch Artikel geben, die gar keinen Umsatz erzielt. Umgekehrt muss zu einem gegebenen Umsatz genau ein Artikel gehören. Die zweite Beziehung lautet:

$\Rightarrow$ 1-CM (1:N „muss-kann“)

Diese Beziehung bedeutet, dass zu einer Entität „Kunde“ mehrere Entitäten „Umsätze“ gehören können, d.h. ein Kunde kann auch gespeichert sein, wenn er noch keinen Umsatz erzeugt hat. Ist das sinnvoll? Hier zeigt sich schon die gestalterische Wirkung der Datenmodellierung: Ein nur an Speichereffizienz interessierter Informatiker könnte argumentieren, dass für doch nur Kunden mit Umsatz sehen wollen, ein Vertriebler des Unternehmens dagegen würde eher den Aspekt sehen, dass der Kunde vielleicht für einen Artikel interessiert werden könnte und daher gespeichert werden sollte. Die dritte Beziehung ist wieder etwas leichter darzustellen:

$\Rightarrow$ CM-CM (M:N „kann-kann“)

Ein Kunde kann hier mehrere Artikel kaufen (kauft aber vielleicht auch keinen), und ein Artikel kann von mehreren Kunden gekauft werden (vielleicht aber auch gar nicht). Wie aber können wir nun die Beziehungen zwischen diesen Tabellen in SQL programmieren? Das werden wir im nächsten Kapitel sehen. □

9 Ableitung von Tabellen aus dem ERM

Kapitelübersicht

9.1 Fremdschlüssel . . . 61
9.2 Grundregeln der Implementierung von Beziehungen . . . 63
9.3 Die Hauptbeziehung 1:N . . . 64
9.4 Die Hauptbeziehung M:N . . . 66
9.5 Die Hauptbeziehung 1:1 . . . 69
9.6 Schlecht oder gar nicht implementierbare Beziehungen . . . 71
9.7 Spezielle Beziehungen . . . 72
9.8 SQL mit mehreren Tabellen . . . 77

Wie in Bemerkung 8.2 bereits angedeutet werden aus den Entitätstypen eines ER-Diagramms am Ende Tabellen einer relationalen Datenbank gebildet. Allerdings gibt es dabei einige Umsetzungsregeln zu beachten, die von den modellierten Kardinalitäten abhängen. Eine andere, ganz grundsätzliche Frage bleibt jedoch zunächst zu klären: Ein ER-Diagramm besteht neben den Entitätstypen aus Beziehungen und Kardinalitäten. Wenn also Entitätstypen durch Tabellen implementiert werden, wie können wir Beziehungen und Kardinalitäten in einer relationalen Datenbank realisieren? Das wesentliche technische Hilfsmittel dazu sind sogenannte Fremdschlüssel.

9.1 Fremdschlüssel

Definition 9.1. Ein *Fremdschlüssel* in einer Tabelle ist ein Schlüssel, der nur Werte enthält, die als Primärschlüssel von mindestens einer anderen Tabelle enthalten sind. □

In SQL gehört die Deklaration eines Fremdschlüssels zur Definition einer Tabelle und kann mit den reservierten Wörtern

```
FOREIGN KEY(...) REFERENCES ...
```

implementiert werden. In den Klammern hinter `FOREIGN KEY` steht das Tabellenattribut, das den Fremdschlüssel bildet, hinter `REFERENCES` der Name der Tabelle und in Klammern deren Primärschlüssel, auf den verwiesen wird.

```
CREATE TABLE tabelle_1 (
  primärschlüssel <datentyp>,
```

```
  ...,
  PRIMARY KEY(primärschlüssel)
);

CREATE TABLE tabelle_2 (
  ...,
  fremdschlüssel <datentyp>,
  FOREIGN KEY(fremdschlüssel) REFERENCES tabelle_1(primärschlüssel)
)
```

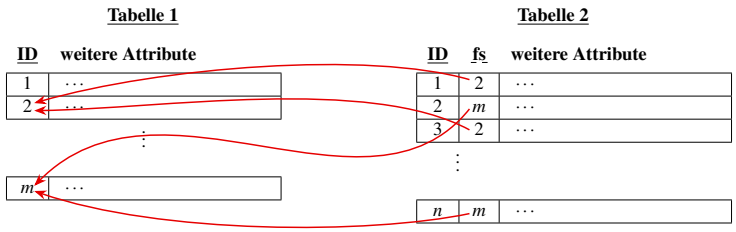


Abbildung 9.1: Die Fremdschlüssel (`fs`) der Datensätze von Tabelle 2 referenzieren auf die Primärschlüssel von Datensätzen der Tabelle 1.

Bildlich können wir uns vorstellen, dass der Fremdschlüssel jedes einzelnen Datensatzes der einen Tabelle auf die Primärschlüssel referenziert wie in Abbildung 9.1. Wir erkennen hier sofort, dass nicht unbedingt jeder Primärschlüssel referenziert wird, aber dass jeder Fremdschlüssel ungleich `NULL` auf einen Primärschlüssel referenzieren muss.

Eine Tabelle kann auch Fremdschlüssel aus mehreren anderen Tabellen enthalten, in SQL wird dann die `FOREIGN KEY`-Anweisung entsprechend wiederholt:

```
CREATE TABLE tabelle (
  ...,
  fremdschlüssel <datentyp>,
  FOREIGN KEY(fremdschlüssel_1) REFERENCES tabelle_1(primärschlüssel_1),
  ...,
  FOREIGN KEY(fremdschlüssel_n) REFERENCES tabelle_n(primärschlüssel_n)
)
```

Definition 9.2. *(Referenzielle Integrität)* Ein Fremdschlüssel einer Tabelle, der einen Wert ungleich `NULL` hat, muss auf den Primärschlüssel eines existierenden Datensatzes verweisen. □

Zur Erhaltung der referenziellen Integrität muss ein RDBMS verhindern, dass Datensätze mit referenzierten Primärschlüsseln einfach gelöscht werden. Um die referenzielle Integrität zu gewährleisten, kann man in SQL die Anweisung

```
ON DELETE RESTRICT
```

nach der Deklaration eines Fremdschlüssels geben, also:

```
CREATE TABLE tabelle_2 (
  ...,
  fs <datentyp>,
  FOREIGN KEY(fs) REFERENCES tabelle_1(primärschlüssel) ON DELETE RESTRICT
)
```

Die wichtigsten Anweisungen zur Erhaltung der referenziellen Integrität sind in Tabelle 9.1 aufgelistet. Sie decken alle möglichen Strategien ab, von strikter Löschverhinderung bis zur

Anweisung	Wirkung bei Löschung des referenzierten Datensatzes
`ON DELETE RESTRICT`	Datensatz wird nur gelöscht, wenn danach die referenzielle Integrität wiederhergestellt ist (Bei SQL Server: `ON DELETE NO ACTION`[1])
`ON DELETE SET DEFAULT`	Setzt den Fremdschlüssel auf seinen Standardwert
`ON DELETE SET NULL`	Fremdschlüssel wird auf `NULL` gesetzt
`ON DELETE CASCADE`	Löscht alle Datensätze mit, die auf den zu löschenden Datensatz referenzieren

Tabelle 9.1: SQL-Anweisungen zur Erhaltung der referenziellen Integrität von Fremdschlüsseln und die Wirkung bei Löschen des referenzierten Datensatzes.

kaskadierten Löschung aller betroffenen Datensätze.[2]

Im Allgemeinen werden Fremdschlüssel dazu verwendet, um Beziehungen in ER-Diagrammen zu implementieren. Auf welche Weise das genau geschieht, hängt von den Kardinalitäten der beteiligten Tabellen ab. Wir werden dies im Laufe des Kapitels noch behandeln.

9.2 Grundregeln der Implementierung von Beziehungen

Die erste Implementierungsregel ist eine eigentlich eine Konvention und betrifft die Tabellennamen.

Konvention. *Obwohl die Namen von Entitätstypen meist Substantive im Singular sind, sollten die Namen von Tabellen einer relationalen Datenbank im Plural benannt werden.*

Die weiteren Implementierungsregeln beziehen sich auf die einzelnen Kardinalitäten. Dazu beantworten wir zunächst aber die Frage: Wieviel verschiedene Implementierungsregeln benötigen wir eigentlich überhaupt?

Bemerkung 9.3. Da es mit (8.1) vier Kardinalitäten gibt, sind in einem ER-Diagramm theoretisch insgesamt

$$\binom{4}{2} + 4 = 10. \tag{9.1}$$

	kann-kann	muss-kann	kann-muss	muss-muss
1:N	□	□	□	□
M:N	□	□		□
1:1	□	□		□

Beziehungstypen zwischen zwei Entitätstypen möglich.[3] □

Regel 4. *Grundsätzlich wird jede der nach Bemerkung 9.3 möglichen zehn Entitätsbeziehungen durch geeignet angelegte Fremdschlüssel gemäß Abbildung 9.1 in SQL implementiert. Jeder referenzierte Datensatz wird in einem Fremdschlüssel des referenzierenden Datensatzes gespeichert.*

Genaugenommen werden wir allerdings nur für sechs der zehn möglichen Beziehungen konkrete Implementierungsregeln bestimmen können, wie wir im Folgenden sehen werden. Jede dieser Regeln legt dabei fest, in welche der Tabellen ein Fremdschlüssel einzufügen ist.

[1] `http://msdn.microsoft.com/en-us/library/ms186712.aspx`; vgl. `https://docs.microsoft.com/en-us/office/client-developer/access/desktop-database-reference/constraint-clause-microsoft-access-sql` für MS Access

[2] vgl. Piepmeyer (2011):S. 100ff.

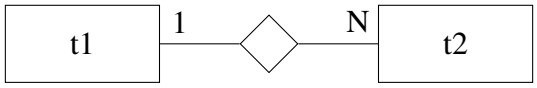
[3] Falls Ihnen im Rahmen der Statistik bereits die Kombinatorik vermittelt wurde, können Sie die Berechnung in Gleichung (9.1) schnell nachvollziehen: Die Anzahl aller möglichen Paarkombinationen bei vier Kardinalitäten ist $\binom{4}{2} = 6$, allerdings fehlen dabei die vier Beziehungstypen mit gleichen Kardinalitäten an beiden Enden, d.h. wir müssen sie zu $\binom{4}{2}$ addieren. Alternativ hätten wir aber im Übrigen auch argumentieren können: Die Anzahl aller Paarkombinationen mit Zurücklegen ist 4^2, dabei haben wir allerdings die $\binom{4}{2}$ mit verschiedenen Kardinalitäten doppelt gezählt, d.h. wir müssen sie davon abziehen: $4^2 - \binom{4}{2} = 16 - 6 = 10$.

Bemerkung 9.4. Die ersten Überlegungen in diesem Abschnitt zu einer Umsetzung eines Entity-Relationship-Modells in Datenbanktabellen beruhen im Wesentlichen zunächst nicht auf diesen 10 Fällen, sondern lediglich auf den drei *Hauptbeziehungen* 1:1, 1:N und M:N, als ohne die Mindestzahlen, und hier vor allem die Hauptbeziehungen 1:N und M:N.

Mit diesen beiden Hauptbeziehungen können wir am klarsten die Grundprinzipien der Implementierung von Beziehungen im Allgemeinen erkennen. Die Beziehung 1:1 ist dagegen etwas schwieriger und wird am Ende etwas ausführlicher betrachtet. □

9.3 Die Hauptbeziehung 1:N

Eine 1:N-Beziehung lässt sich wie folgt darstellen:



Hier werden einem einzelnen Datensatz in Tabelle t1 also mehrere Datensätze aus Tabelle t2 zugeordnet, und umgekehrt einem Datensatz in Tabelle t2 ein einziger Datensatz aus t1. Wollen wir nun Regel 4 anwenden, stellt sich sofort die Frage: In welche der Tabellen muss ein Fremdschlüssel eingefügt werden? In alle beide oder reicht ein einziger Fremdschlüssel in einer der Tabellen?

Mit einem kurzen Gedankenspiel können wir uns die Antwort herleiten: Würden wir gemäß Abbildung 9.1 den Fremdschlüssel eines rerenzierten Datensatzes aus t2 in Tabelle t1 speichern, bräuchten wir nach Regel 4 mehrere Attribute dafür. Wieviel aber genau, hängt vom konkreten Fall ab; vielleicht brauchen wir in einem Fall nur einen Fremdschlüssel, in einem anderen vielleicht zwei, in wieder einem anderen vielleicht 100. Selbst wenn wir genau wüssten, dass wir maximal 10 Fremdschlüssel bräuchten, wäre es ziemlich ineffizient, wenn wir in den meisten Fällen nur ein oder zwei verwenden müssen. Folgerung: Tabelle t1 ist kein guter Kandidat für die Speicherung von Fremdschlüsseln. Was ist umgekehrt mit Tabelle t2? Hier brauchen wir maximal einen Fremdschlüssel, egal wie groß N sein mag! Die Regel für die Hauptbeziehung 1:N lautet daher:

Regel 5. *Bei einer Beziehung 1:N wird nur ein Fremdschlüssel benötigt. Er wird in der N-Seite gespeichert, wie in Abbildung 9.2 dargestellt.*

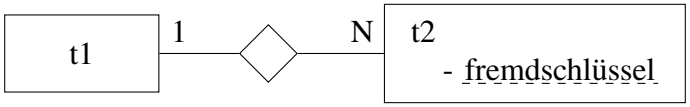
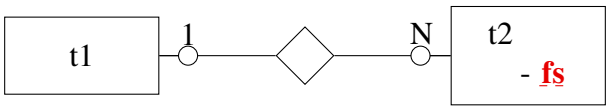


Abbildung 9.2: Implementierung einer Beziehung 1:N. Der Fremdschlüssel kommt auf die N-Seite

Es gibt drei Beziehungen der Kategorie 1:N in den Kombinationen „kann“ und „muss“, nämlich C-CM, 1-CM und C-M. Die ersten beiden werden wir jetzt näher untersuchen, C-M erst in Abschnitt 9.6 auf Seite 71.

9.3.1 C-CM-Beziehungen („1:N kann-kann“)

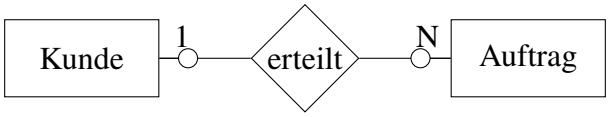


Regel 6. *Bei einer C-CM-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel kann* `NULL` *sein.*

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

```
CREATE TABLE t2 (
  ...,
  fs int,
  FOREIGN KEY(fs) REFERENCES t1(id) ON DELETE RESTRICT
);
```

Beispiel 9.5. Ein einfaches Beispiel für eine C-CM-Beziehung ist:



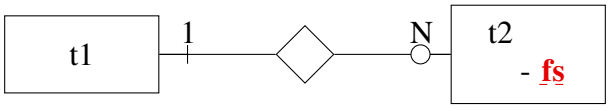
Ein Kunde kann hier mehrere Aufträge erteilen, ein Auftrag dagegen kann von höchstens einem Kunden erteilt werden. Es kann aber auch Aufträge geben, die nicht von einem Kunden erteilt werden, beispielsweise betriebsinterne Aufträge.

```
CREATE TABLE kunden (
  name varchar(50),
  wohnort varchar(50),
  PRIMARY KEY(name)
);
```

```
CREATE TABLE auftraege (
  datum date,
  kunde varchar(50),
  volumen decimal(10,2),
  FOREIGN KEY(kunde) REFERENCES kunden(name)
);
```

□

9.3.2 1-CM-Beziehungen („1:N muss-kann“)

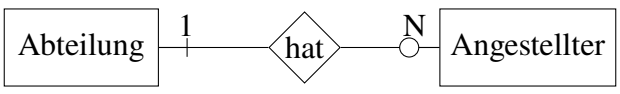


Regel 7. *Bei einer 1-CM-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel darf dabei nicht* `NULL` *sein.*

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

```
CREATE TABLE t2 (
  ...,
  fs int NOT NULL,
  FOREIGN KEY(fs) REFERENCES t1(id) ON DELETE RESTRICT
);
```

Beispiel 9.6. Ein Beispiel für eine 1-CM-Beziehung ist die folgende:



Hier kann eine Abteilung mehrere Angestellte haben, aber jede*r Angestellte muss zu einer Abteilung gehören. In SQL muss also der Fremdschlüssel in der Tabelle `angestellter` gespeichert werden, z.B. mit

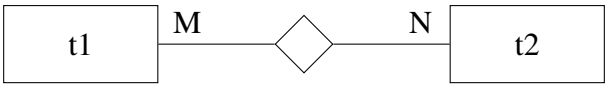
```sql
CREATE TABLE abteilung (
  name varchar(50),
  standort varchar(50),
  PRIMARY KEY(name)
);
```

```sql
CREATE TABLE angestellte (
  name varchar(50) PRIMARY KEY,
  abteilung varchar(50) NOT NULL,
  FOREIGN KEY(abteilung) REFERENCES abteilung(name)
);
```

Um die Tabelle `angestellter` anlegen zu können, muss zuvor die Tabelle `abteilung` bereits angelegt sein. □

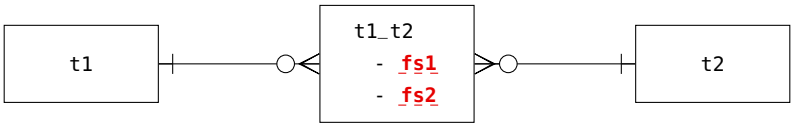
9.4 Die Hauptbeziehung M:N

Zur Umsetzung einer M:N-Beziehung müssen wir uns zunächst ein paar Gedanken machen: Mit unseren bisherigen Regeln, die Beziehung mit Fremdschlüsseln zu implementieren, kommen wir hier nicht weiter.



Denn wieviel Fremdschlüssel müssten wir für `t1` verwenden und wieviel für `t2`? Um beliebig viele Datensätze vom Typ `t2` zu referenzieren, bräuchten wir für `t1` unbegrenzt viele Fremdschlüssel, und umgekehrt – das allein ist ja schon rein logisch nicht realisierbar. Und selbst wenn es uns durch Zauberei (oder einen Trick) gelänge, hätten wir das Problem der referenziellen Integrität aus Definition 9.2 auf Seite 62 noch nicht gelöst, wonach alle diese Fremdschlüssel auf einen Datensatz verweisen müssen. Ist eine Umsetzung einer solchen Beziehung in einer relationalen Datenbank also gar nicht möglich?

Die Lösung besteht darin, die Beziehung mit Hilfe einer dritten „künstlichen“ Tabelle auzulösen, mit einer sogenannten *Beziehungstabelle*. Jeder Datensatz dieser Tabelle speichert gewissermaßen jedes Datensatzpaar der beiden Grundtabellen, das eine Beziehung bildet. Auf diese Weise erhält man statt der einen M-N-Beziehung zwei 1-N-Beziehungen:



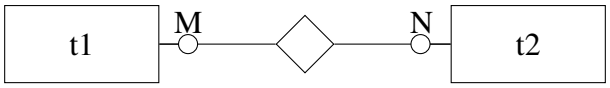
Da die Beziehungstabelle keine Entität nach dem ERM ist, sondern eigentlich ja ein technisches Artefakt zur Implementierung, ist dieses Diagramm der aufgelösten Entitätsbeziehung ein „Tabellendiagramm“. Um Tabellendiagramme von ER-Diagrammen zu unterscheiden, benutzen wir hierfür die neben der Chen-Notation auch gebräuchliche *Krähenfußnotation (crow's foot notation)*. Hier werden die Kardinalitäten ohne hochstehende Ziffern oder Buchstaben dargestellt, sondern mit Kreisen, Strichen und „Krähenfüßen“ am Ende der Beziehungen:

(9.2)

Vgl. Piepmeyer (2011:S. 120) und – im Vergleich – die Chen-Notation (8.1) auf Seite 57.

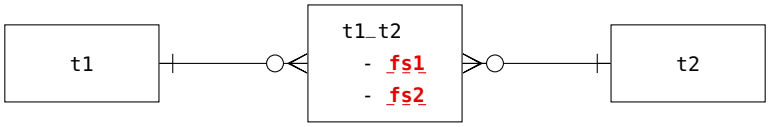
9.4.1 Die Beziehung CM-CM („M:N kann-kann“)

Mit den Überlegungen des vorherigen Abschnitts betrachten wir nun eine CM-CM-Beziehung:



Wir können eine CM-CM-Beziehung durch zwei 1-CM-Beziehungen auflösen, die wir jeweils mit Regel 7 implementieren:

Regel 8. *Eine CM-CM-Beziehung wird zu zwei 1-CM-Beziehungen gemäß Regel 7 mit Hilfe einer Beziehungstabelle mit zwei Fremdschlüsseln aufgelöst, die die Primärschlüssel der beiden Grundtabellen sind.*




```sql
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

```sql
CREATE TABLE t1_t2 (
  id SERIAL,
  fs_1 int NOT NULL,
  fs_2 int NOT NULL,
  ...,
  PRIMARY KEY(id);
  FOREIGN KEY(fs_1) REFERENCES t1(id)
     ON DELETE RESTRICT,
  FOREIGN KEY(fs_2) REFERENCES t2(id)
     ON DELETE RESTRICT
);
```

```sql
CREATE TABLE t2 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

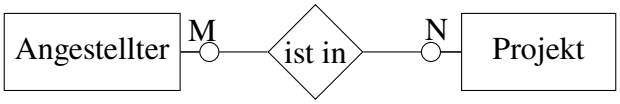
Ist zusätzlich durch den konkreten Anwendungsfall garantiert, dass zwei konkrete Entitäten (Datensätze) höchstens *eine Beziehung zueinander haben können, so können die beiden Fremdschlüssel den Primärschlüssel der Beziehungstabelle bilden:*

```sql
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

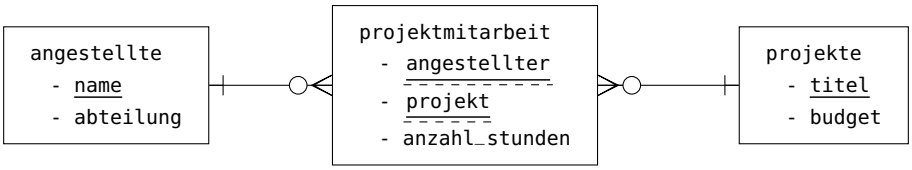
```sql
CREATE TABLE t1_t2 (
  fs_1 int,
  fs_2 int,
  ...,
  PRIMARY KEY(fs_1, fs_2),
  FOREIGN KEY(fs_1) REFERENCES t1(id)
     ON DELETE RESTRICT,
  FOREIGN KEY(fs_2) REFERENCES t2(id)
     ON DELETE RESTRICT
);
```

```sql
CREATE TABLE t2 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

Beispiel 9.7. Ein Beispiel für eine CM-CM-Beziehung ist:



Ein*e Angestellte kann also in mehreren Projekten sein, und in einem Projekt können mehrere Angestellte arbeiten. Diese Beziehung muss durch eine Beziehungstabelle aufgelöst werden:



Sie kann neben den beiden Fremdschlüsseln, wie hier, weitere spezielle Attribute enthalten. Dieses Tabellenmodell wird wie folgt in SQL implementiert:

```
CREATE TABLE angestellte (
  name varchar(50),
  abteilung varchar(50),
  PRIMARY KEY (name)
);
```

```
CREATE TABLE projektmitarbeit (
  angestellter varchar(50),
  projekt varchar(50),
  anzahl_stunden int,
  PRIMARY KEY(projekt, angestellter),
  FOREIGN KEY(projekt)
     REFERENCES projekte(titel),
  FOREIGN KEY(angestellter)
     REFERENCES angestellte(name)
);
```

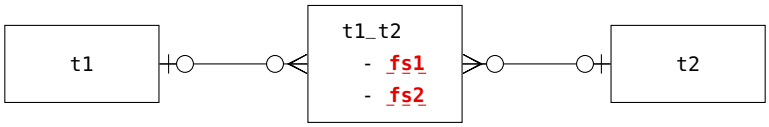
```
CREATE TABLE projekte (
  titel varchar(50),
  budget decimal(10,2),
  PRIMARY KEY(titel)
);
```

Die Fremdschlüssel bilden hier den Primärschlüssel, können also nicht `NULL` sein. Daher können die Restriktionen `NOT NULL` weggelassen werden. □

Mit den „Muss-“Kardinalitäten wird durch die Datenbank gesichert, dass eine CM-CM-Beziehung auch nur dann als Datensatz der Beziehungstabelle gespeichert werden kann, wenn die Datensätze der beiden beteiligten Entitäten bereits existieren. In seltenen Anwendungsfällen muss eine CM-CM-Beziehung jedoch auch in zwei C-CM-Beziehungen aufgelöst werden. Das ist der Fall für sogenannte *gerichtete* Beziehungen, die in dem Sinne „unsymmetrisch“ sind, dass ein Datensatz der einen Tabelle zwar den Datensatz der anderen „kennt“, aber nicht umgekehrt. (Zu beachten ist dabei, dass eine solche Beziehung im Allgemeinen in einem ER-Diagramm nicht dargestellt werden kann, sondern sich aus dem Kontext des jeweiligen Anwendungsfalls ergibt.)

Spezialfall *gerichtete* Beziehung

Regel 9. *In seltenen Fällen ist eine CM-CM-Beziehung gerichtet und muss in zwei C-CM-Beziehungen aufgelöst werden:*



Damit wird sie mit Regel 6 implementiert durch:

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

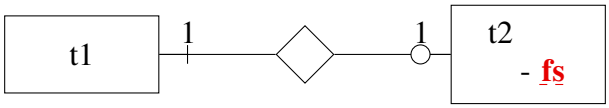
```
CREATE TABLE t1_t2 (
  id SERIAL,
  fs_1 int,
  fs_2 int,
  ...,
  PRIMARY KEY(id);
  FOREIGN KEY(fs_1) REFERENCES t1(id)
     ON DELETE RESTRICT,
  FOREIGN KEY(fs_2) REFERENCES t2(id)
     ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

Ein Beispiel dafür werden wir weiter unten mit dem Datenmodell für rekursive Beziehungen zur Darstellung gerichteter Graphen in Beispiel 9.14 kennen lernen.

9.5 Die Hauptbeziehung 1:1

9.5.1 1-C-Beziehungen („1:1 muss-kann“)



Bei der 1-C-Beziehung muss der Fremdschlüssel auf die C-Seite und darf nicht `NULL` sein, denn es muss ja stets mindestens einen Datensatz der Tabelle `t1` geben. Diese Bedingung ist in SQL einfach mit der Integritätsregel `NOT NULL` implementierbar. Da der Fremdschlüssel in Tabelle t2 ist, kann auch nur *höchstens* ein Datensatz aus t1 zu einem Datensatz aus t2 gespeichert werden. Wie aber legt man in SQL fest, dass auch zu einem Datensatz aus t1 stets höchstens ein Datensatz aus t2 gespeichert werden kann? Hier hilft das reservierte Wort `UNIQUE`, das verhindert, dass in einer Tabelle zwei Datensätze mit dem gleichen Attributwert gespeichert werden, das also keine Dubletten zulässt. Damit erhalten wir die folgende Ableitungsregel:

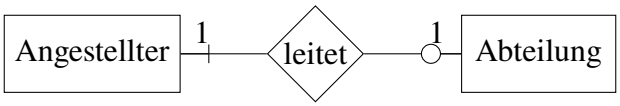
Regel 10. *Bei einer 1-C-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel muss dabei `NOT NULL` und `UNIQUE` sein.*

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

```
CREATE TABLE t2 (
  ...,
  fs int NOT NULL UNIQUE,
  FOREIGN KEY(fs) REFERENCES t1(id) ON DELETE RESTRICT
);
```

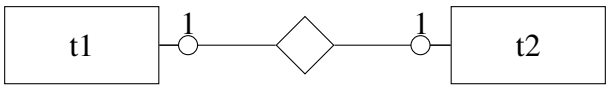
Das reservierte Wort `UNIQUE` verhindert, dass in einer Tabelle zwei Datensätze mit dem gleichen Attributwert gespeichert werden. Damit wird die 1 rechts implementiert.

Beispiel 9.8. Eine einfache 1-C-Beziehung ist die folgende:

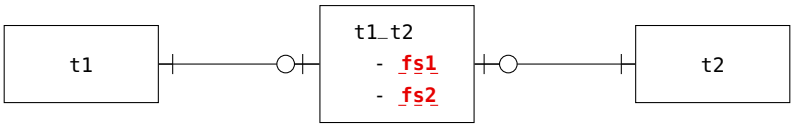


Hier kann ein*e Angestellter höchstens eine Abteilung leiten, aber eine Abteilung muss genau einen Leiter haben. □

9.5.2 C-C-Beziehungen („1:1 kann-kann“)



Auf den ersten Blick scheint es nicht besonders schwer, diese Beziehung ähnlich wie die 1-C-Beziehung in Regel 10 zu implementieren. Beispielsweise könnte man einfach den Fremdschlüssel wie dort bei t2 vorsehen und ihn lediglich mit `UNIQUE` einschränken, also `NOT NULL` weglassen und so den Wert `NULL` erlauben; Dubletten wären dann trotzdem nicht zugelassen. Leider erfordert bei vielen SQL-Dialekten `UNIQUE` jedoch die Integritätsregel `NOT NULL`. Falls zudem in einem gegebenen Anwendungsfall das tatsächliche Vorkommen der Beziehung eher selten ist, so würde es bei großen Datenbeständen zu ineffizientem Speicherplatzbedarf führen, da die Fremdschlüsselattribute fast überall NULL wären. Wir sollten daher auch in diesem Fall mit einer Beziehungstabelle arbeiten:



Damit wird eine C-C-Beziehung also in zwei 1-C-Beziehungen aufgelöst.

Regel 11. *Einer C-C-Beziehung wird durch eine Beziehungstabelle aufgelöst, die die Primärschlüssel jeweils als Fremdschlüssel erhält. Beide Fremdschlüssel müssen dabei* `NOT NULL` *und* `UNIQUE` *sein und bilden den Primärschlüssel der Tabelle.*

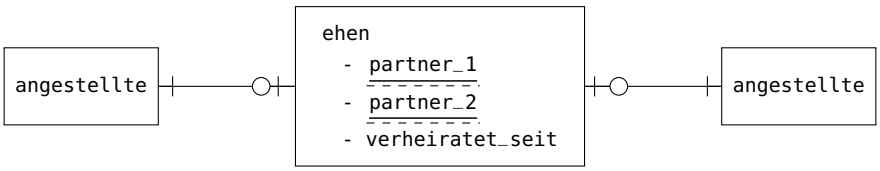
```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

```
CREATE TABLE t1_t2 (
  fs_1 int NOT NULL UNIQUE,
  fs_2 int NOT NULL UNIQUE,
  PRIMARY KEY(fs_1, fs_2);
  FOREIGN KEY(fs_1) REFERENCES t1(id)
     ON DELETE RESTRICT,
  FOREIGN KEY(fs_2) REFERENCES t2(id)
     ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
  id int,
  ...,
  PRIMARY KEY(id)
);
```

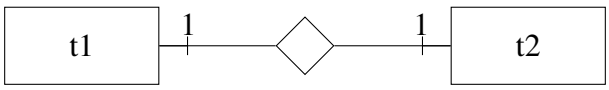
Beispiel 9.9. Eine C-C-Beziehung ist die folgende:

Sie wird aufgelöst durch eine Beziehungstabelle `ehen`:



In die Beziehungstabelle werden nur Ehen eingetragen, die tatsächlich vorhanden sind. Damit werden diese relativ selten auftretenden Fälle sehr effizient gespeichert. □

9.5.3 1-1-Beziehungen („1:1 muss-muss“)



Eine 1-1-Beziehung ist eine ganz spezielle Beziehung, die in zwei oder mehr Tabellen gar nicht zu implementieren ist. Zu jeder Entität des Typs t1 müsste es nämlich stets genau eine Entität des Typs t2 geben und umgekehrt, insbesondere müsste es also in Tabelle `t1` stets genauso viele Datensätze geben wie in Tabelle `t2`. Die beiden Teilnehmer einer solchen Beziehung müssten also *gleichzeitig* entstehen, was technisch nicht möglich ist.[4] Da die beiden beteiligten Entitätstypen ihre Unabhängigkeit vollständig eingebüßt haben, werden sie üblicherweise in einer einzigen Tabelle implementiert:

[4] Piepmeyer (2011):S. 137.

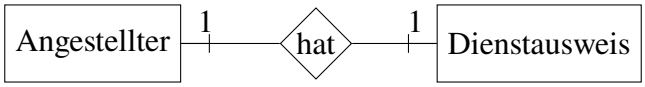
Regel 12. *Zwei Entitätstypen mit einer 1-1-Beziehung („1:1 muss-muss“) werden zu einer einzelnen Tabelle zusammengefasst.*

```
CREATE TABLE t (
  attribute_von_t1 <Typ> NOT NULL UNIQUE,
  ...,
  attribute_von_t2 <Typ> NOT NULL UNIQUE,
  ...
)
```

Falls nur einer der Entitätstypen in dem ER-Diagramm weitere Beziehungen hat, kann man diesen als Tabellennamen verwenden, ansonsten sollte ein neuer Name gesucht werden.

Im Allgemeinen sollte schon beim Modellieren einer 1-1-Beziehung auf jeden Fall hinterfragt werden, ob sie tatsächlich überhaupt so vorliegt. In der Realität kommt sie eher selten vor.

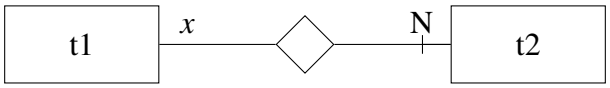
Beispiel 9.10. Ein Beispiel für eine 1-1-Beziehung ist die folgende:



Hier ist es sinnvoll, die Attribute des Dienstausweises einfach in die Tabelle `angestellter` aufzunehmen. □

9.6 Schlecht oder gar nicht implementierbare Beziehungen

Von den bisher aufgeführten sechs Beziehungstypen fehlen nach Bemerkung 9.3 auf Seite 63 noch vier. Das sind die Beziehungen x-M, bei denen mindestens ein Teilnehmer mindestens einmal auftreten *muss* und mehrfach auftreten kann,



($x = 1$ oder N), also:

- C-M („1:N kann-muss“),
- 1-M („1:N muss-muss“),
- CM-M („M:N kann-muss“) und
- M-M („M:N muss-muss“).

Zwar könnten wir diese Beziehungen grundsätzlich wie die x-CM-Beziehungen mit einer Beziehungstabelle auflösen, allerdings kann SQL dazu keine ausreichenden Integritätsregeln garantieren.[5]

	kann-kann	muss-kann	kann-muss	muss-muss
1:N	☑	☑	×	×
M:N	☑	×		×
1:1	☑	☑		☑

Das relationale Modell stößt hier an seine logischen Grenzen.

[5] Piepmeyer (2011):S. 139ff.

9.7 Spezielle Beziehungen

9.7.1 IS-A-Beziehungen

Eine *IS-A*-Beziehung ist eine Beziehung zwischen zwei Entitäten, in der ein Entitätstyp den anderen spezialisiert. Was ist damit gemeint? Betrachten wir dazu zwei Tabellen, nennen wir die eine tgrund wie „Grundtabelle“ und die andere t_spez wie „spezialisierende Tabelle“. Sie bilden eine IS-A-Beziehung, wenn dann die Tabelle tspez automatisch alle Attribute der anderen Tabelle tgrund übernimmt und noch weitere enthält.

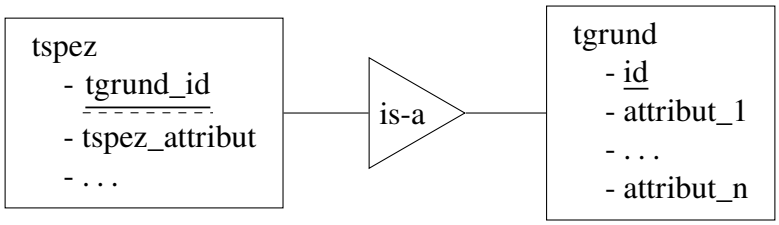


Tabelle tspez erweitert also die Attribute der Tabelle tgrund. Entsprechend heißt tspez hier die spezialisierende (oder „erbende“) Tabelle, und tgrund die generalisierende (oder „vererbende“) Tabelle. Der Primärschlüssel der Tabelle tspez ist hierbei gleichzeitig Fremdschlüssel auf Tabelle tgrund. Um eindeutig erkennbar zu sein, sollte er den Namen des Primärschlüssels mit dem Namen der vererbenden Tabelle als Präfix haben, also hier: `tgrund_id`.

Regel 13. *Eine* IS-A-*Beziehung wird implementiert, indem der Primärschlüssel der spezialisierenden Tabelle gleichzeitig der Fremdschlüssel auf den Primärschlüssel der vererbenden Tabelle ist.*

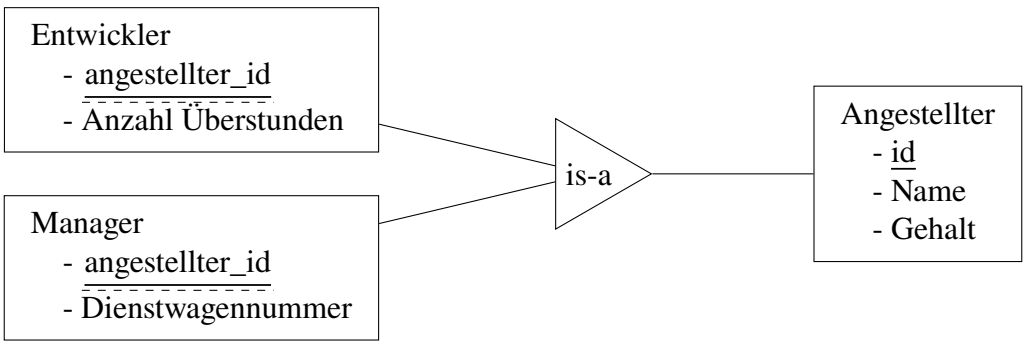
```
CREATE TABLE t1 (
  t2_id <typ>,
  t1_attribut <typ_t1>,
  ...
  PRIMARY KEY(t2_id),
  FOREIGN KEY(t2_id) REFERENCES t2(id) ON DELETE CASCADE
);
```

```
CREATE TABLE t2 (
  id <Typ>,
  attribut_1 <typ_1>,
  ...,
  attribut_n <typ_n>,
  PRIMARY KEY(id)
);
```

Der Primärschlüssel der erbenden Tabelle tspez sollte den Namen des Primärschlüssels mit dem Namen der vererbenden Tabelle als Präfix haben, also hier: `tgrund_id`.

De facto handelt es sich bei einer IS-A-Beziehung um eine spezielle 1-C-Beziehung (1:1 kann-muss), vgl. Regel 10 auf Seite 69. Die generalisierende Tabelle muss hier also vor der spezialisierenden Tabelle erzeugt worden sein. Wird ein Datensatz aus der Grundtabelle gelöscht, so wird wegen `ON DELETE CASCADE` der referenzierende Datensatz aus der spezialisierenden Tabelle automatisch mit gelöscht.

Beispiel 9.11. Betrachten wir als ein Beispiel die speziellen Rollen von Angestellten in einem Unternehmen.



Hier sind also Entwickler und Manager spezielle Angestellte. Als Angestellte haben sie gemeinsame Attribute (Name, Gehalt), in ihren Rollen jedoch auch unterschiedliche Attribute. Für die Implementierung in SQL erstellen wir zunächst die Grundtabelle `angestellte`:

```
CREATE TABLE angestellte (
  id int,
  name varchar(30),
  gehalt decimal(10,2),
  PRIMARY KEY(id)
);
```

Mit Regel 13 lauten dann die spezialisierenden Tabellen

```
CREATE TABLE entwickler (
  angestellten_id int,
  überstunden int,
  PRIMARY KEY(angestellten_id),
  FOREIGN KEY(angestellten_id) REFERENCES angestellte(id) ON DELETE CASCADE
);
```

und

```
CREATE TABLE manager (
  angestellten_id int,
  dienstwagen int,
  PRIMARY KEY(angestellten_id),
  FOREIGN KEY(angestellten_id) REFERENCES angestellte(id) ON DELETE CASCADE
);
```

Um nun jeweils einen Datensatz für `ebtwickler` und `manager` zu speichern, muss dabei stets vorher der Datensatz mit demselben Primärschlüssel in `angestellte` eingefügt werden:

```
INSERT INTO angestellte (id, name, gehalt) VALUES (1, 'Anna', 4567.89);
INSERT INTO manager (angestellten_id, dienstwagen) VALUES (1, 4711);
--
INSERT INTO angestellte (id, name, gehalt) VALUES (2, 'Otto', 3456.78);
INSERT INTO entwickler (angestellten_id, überstunden) VALUES (2, 24);
```

Dann haben die Tabellen die folgenden Datensätze gespeichert:

`SELECT * FROM entwickler;`

angestellten_id	überstunden
2	24

`SELECT * FROM manager;`

angestellten_id	dienstwagen
1	4711

`SELECT * FROM angestellte;`

id	name	gehalt
1	Anna	4567.89
2	Otto	3456.78

Löscht man nun einen Datensatz aus der Grundtabelle `angestellte`, so wird der referenzierende Datensatz aus der spezialisierenden Tabelle wegen `ON DELETE CASCADE` automatisch auch gelöscht. Nach

```
DELETE FROM angestellte WHERE id = 2;
```

beispielsweise sind nur noch die folgenden Datensätze gespeichert:

`SELECT * FROM entwickler;`

angestellten_id	überstunden

`SELECT * FROM manager;`

angestellten_id	dienstwagen
1	4711

`SELECT * FROM angestellte;`

id	name	gehalt
1	Anna	4567.89

Beim Einfügen eines Datensatzes einer IS-A-Beziehung müssen also tatsächlich *zwei* Datensätze eingefügt werden. Ebenso muss für die Übereinstimmung von Primärschlüssel und Fremdschlüssel manuell gesorgt werden. Beim Löschen eines Datensatzes aus der Grundabelle (angestellte) wird automatisch auch der entsprechende Datensatz aus der anderen gelöscht. Löscht man aber umgekehrt einen Datensatz aus einer spezialisierenden Tabelle, so bleibt der vererbende Datensatz bestehen – was aber auch vernünftig ist, schließlich könnte in unserem Beispiel ja ein*e Entwickler*in irgendwann auch Manager*in werden. Also statt des obigen Löschens aus der Angestelltentabelle:

```
DELETE FROM entwickler WHERE angestellten_id = 2;
INSERT INTO manager (angestellten_id, dienstwagen) VALUES (2, 4712);
```

□

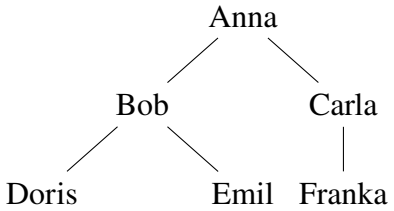
Bemerkung 9.12. Eine IS-A-Beziehung entspricht einer Vererbungshierarchie in der Objektorientierung. Im Gegensatz zu objektorientierten Programmiersprachen kann ein relationales Datenbanksystem eine IS-A-Beziehung nicht komplett automatisch verwalten. □

9.7.2 Rekursive Beziehungen

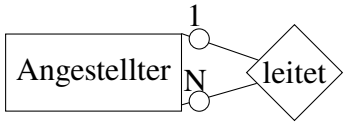
Manchmal gibt es Anwendungsfälle, in denen Entitäten vom selben Entitätstyp Beziehungen miteinander haben. Solche Beziehungen heißen *rekursiv*. Beispiele sind die Knoten von Graphen oder Netzwerken, aber auch die Einheiten einer Hierarchie wie das Organigramm eines Unternehmens. Es gibt zwei wesentliche rekursive Beziehungen, eine C-CM- und eine CM-CM-Beziehung:

Die rekursive C-CM-Beziehung kann bei Hierarchien und baumartigen Strukturen modelliert werden, die rekursive CM-CM-Beziehung bei Graphen und Netzwerken. Entsprechend können sie mit der Regel 6 bzw. Regel 8 aufgelöst werden.

Beispiel 9.13. Gegeben sei die folgende Vorgesetztenstruktur eines Unternehmens:



Um diese Hierarchie zu speichern, kann man die folgende rekursive C-CM-Beziehung modellieren:



D.h. ein Angestellter kann mehrere Angestellte leiten und jeder Angestellte hat höchstens einen Chef. Mit Regel 6 lässt sich diese C-CM-Beziehung mit dem Fremdschlüssel `chef` (oder `leiter`) wie folgt implementieren:

```
CREATE TABLE angestellte (
  name VARCHAR(10),
  chef VARCHAR(10),
  PRIMARY KEY(name),
  FOREIGN KEY (chef) REFERENCES angestellte(name) ON DELETE SET NULL
);
```

Mit dem folgenden Einfügebefehl können wir dann die obige Vorgesetztenstruktur speichern:

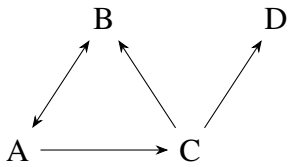
```
INSERT INTO angestellte(name, chef) VALUES
  ('Anna', NULL), ('Bob','Anna'), ('Carla','Anna'),
  ('Doris','Bob'), ('Emil', 'Bob'), ('Franka', 'Carla');
```

Dann ergibt ein SELECT auf diese Tabelle:

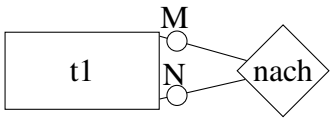
name	chef
Anna	NULL
Bob	Anna
Carla	Anna
Doris	Bob
Emil	Bob
Franka	Carla

□

Beispiel 9.14. *(Digraph)* Ein *Digraph (directed graph)* ist ein Graph, dessen Knoten durch gerichtete Kanten („Pfeile“) verbunden sind, beispielsweise



für die vier Knoten A, B, C, D. Ein Knoten kann nun auf mehrere Knoten weisen, aber es können auch mehrere Knoten auf ihn weisen. Es handelt sich also um eine rekursiven CM-CM Beziehung (M:N „kann-kann“):



Nach Regel 9 für CM-CM-Beziehungen aufgelöst ergibt sich daraus das Tabellendiagramm

und die Implemtierung lautet:

```
CREATE TABLE knoten (
  name varchar(1),
  PRIMARY KEY (name)
);
CREATE TABLE kanten (
  id SERIAL,
```

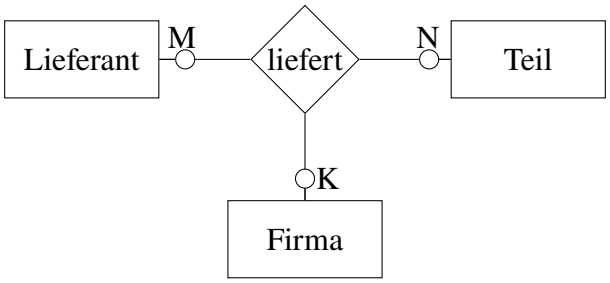
```
  von  varchar(1),
  nach varchar(1),
  PRIMARY KEY (id),
  FOREIGN KEY (von)  REFERENCES knoten(name) ON DELETE CASCADE,
  FOREIGN KEY (nach) REFERENCES knoten(name) ON DELETE CASCADE
);

-- Knoten einfügen:
INSERT INTO knoten(name) VALUES ('A'), ('B'), ('C'), ('D');
-- Kanten A <-> B, A -> C, C -> B, C -> D einfügen:
INSERT INTO kanten (von, nach)
VALUES ('A','B'), ('B','A'), ('A','C'), ('C', 'B'), ('C','D');
```

□

9.7.3 Mehrwertige Beziehungen

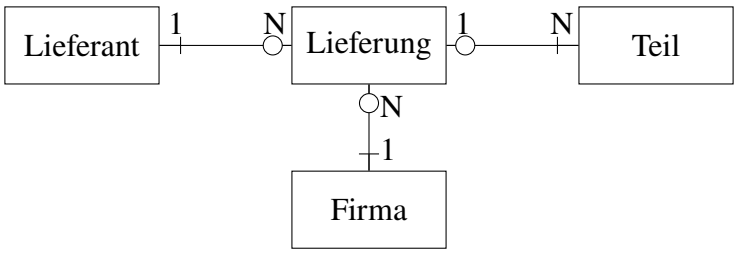
Neben den bisher behandelten 2-wertigen Beziehungen gibt es manchmal auch höherwertige Beziehungen. So ist beispielweise eine dreiwertige Beziehung durch die Entitäten *Lieferant*, *Teil* und *Firma* gegeben:



Diese Beziehung wird durch eine zusätzliche Beziehungstabelle aufgelöst, die jeden Primärschlüssel der Grundtabellen als Fremdschlüssel enthält:

Lieferung (lieferant_id, teil_id, firmen_id, anzahl)

In der Regel können die drei Fremdschlüssel den Primärschlüssel der Beziehungstabelle bilden. So entstehen aus der dreiwertigen CM-CM-CM-Beziehung drei zweiwertige C-CM-Beziehungen: Sie wird durch eine Beziehungstabelle in zweiwertige 1:N-Beziehungen aufgelöst:



Entsprechend können wir mit höherwertigen Beziehungen verfahren, so dass wir am Ende immer ein Tabellendiagramm erhalten können, das ausschließlich aus zweiwertigen Beziehungen besteht.

Allerdings bleibt zu bemerken, dass höherwertige Beziehungen in der Praxis eher selten auftreten.[6] Da sie auch eine sehr enge Abhängigkeit der beteiligten Tabellen bewirken, die die Komplexität bei SQL-Anweisungen erhöht, sollte man im Allgemeinen versuchen, sie gleich als zweiwertige Entitäten zu modellieren.

[6] Piepmeyer (2011):S. 130ff.

9.8 SQL mit mehreren Tabellen

Wie wir im Zusammenhang mit der Modellierung der Entity-Relationships schon gesehen haben, ist die Information in relationalen Datenbanken für Anwendungen aus der Praxis über mehrere Tabellen verteilt. SQL muss also Mechanismen bereitstellen, Abfragen über mehrere Tabellen in der Ergebnismenge zusammenzuführen. Dabei ist ganz allgemein folgende grundsätzliche Regel zu beachten.

Regel 14. *Ein Spaltenname muss in einer SELECT-Abfrage eindeutig sein. Falls ein Spaltenname* `s` *in mehreren Tabelle vorkommt, muss er nach einem Punkt an den gewünschten Tabellennamen* `tabelle` *angefügt werden, also*

```
SELECT ..., tabelle.s, ... FROM ..., tabelle, ... ;
```

Mit dem reservierten Wort **`AS`** *kann jeder Spalte ein neuer Name (Alias)gegeben werden, der für die Abfrage gültig ist, vgl. Abschnitt 3.2 auf Seite 21. Entsprechend kann auch jeder Tabelle ein Alias vergeben werden.*

9.8.1 Kartenspiel

Als Anwendungsbeispiel betrachten wir ein Kartenspiel mit 32 Karten, das die vier Freunde Daniel, Donald, Daisy und Daniel (gleicher Name wie der erste) spielen. Wir möchten die folgenden Blätter in einer Datenbank speichern:

- Daniel hat Herz Ass,
- Daisy hat Pik 7 und Pik Bube,
- Daniel hat Karo Ass.

Unsere Datenbank soll hier generell nur die Blätter für genau ein Spiel speichern können.

Bevor wir die Tabellen programmieren, entwerfen wir zunächst das Datenmodell. Offensichtlich gibt es zwei Entitäten, die Spieler und die Karten. Da es sich nur um ein einziges Spiel handelt, kann jede Karte höchstens einem Spieler gehören, aber jeder Spieler kann mehrere Karten haben. Es handelt sich also um eine C-CM-Beziehung (Abbildung 9.3). Sie braucht

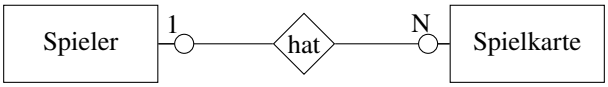
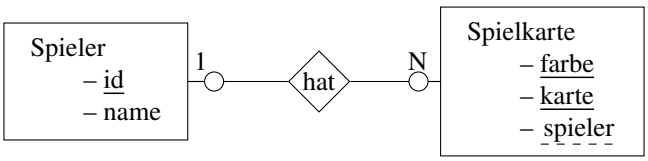


Abbildung 9.3: ER-Modell eines Kartenspiels

nicht weiter aufgelöst zu werden, wir können sie direkt als Tabellen implementieren, die die Relationentypen *spieler(id, name)* und *spielkarten(farbe, karte, spieler)* darstellen:



oder etwas kompakter in tabellarischer Form:

Tabelle	Primärschlüssel	Attribute
Spieler	id	name
Spielkarten	(farbe, karte)	spieler

Hierbei ist der Primärschlüssel `id` der Spieler künstlich, da der Name mehrfach auftritt. Entsprechend ist der Fremdschlüssel `spieler` in dem Relationentyp *spielkarten* der Primärschlüssel desjenigen Spielers, der die Karte hat. Diese Relationentypen können wir in unserer Datenbank wie folgt als Tabellen implementieren:

```
CREATE TABLE spieler (
  id SERIAL PRIMARY KEY,
  name varchar(20),
) DEFAULT CHARSET=utf8;
--
CREATE TABLE spielkarten (
  farbe varchar(5) NOT NULL,
  karte varchar(5) NOT NULL,
  spieler int,
  PRIMARY KEY(farbe, karte),
  FOREIGN KEY(spieler) REFERENCES spieler(id) ON DELETE RESTRICT
) DEFAULT CHARSET=utf8;
```

Damit können wir die drei obigen Blätter wie in Tabelle 9.2 speichern. Die folgenden SQL-

spieler

id	name
1	Daniel
2	Donald
3	Daisy
4	Daniel

spielkarten

farbe	karte	spieler
Herz	Ass	1
Pik	7	3
Pik	Bube	3
Karo	Ass	4

Tabelle 9.2: Daten der Tabellen der Kartenspieldatenbank

Anweisungen speichern die einzelnen Blätter entsprechend in unserer Datenbank:

```
INSERT INTO spieler (name) VALUES
('Daniel'), ('Donald'), ('Daisy'), ('Daniel');
INSERT INTO spielkarten (farbe, karte) VALUES
('Herz', 'Ass'), ('Pik', '7'), ('Pik', 'Bube'), ('Karo', 'Ass');
```

Um sich nun die Karten von Daisy anzusehen, müssen wir die folgende Abfrage an die Datenbank schicken:

```
SELECT farbe, karte FROM spielkarten WHERE spieler = 2;
```

Das Ergebnis lautet dann:

farbe	karte
Pik	7
Pik	Bube

Wie können wir aber das Blatt von Daisy anzeigen lassen, wenn wir ihren Primärschlüssel gar nicht kennen? Dafür müssen wir erst in der Tabelle `spieler` herausfinden, welche ID sie hat, um dann eine äußere SELECT-Anweisung auf sie auszuführen:

```
SELECT farbe, karte FROM spielkarten WHERE spieler = (
   SELECT id FROM spieler WHERE name='Daisy'
);
```

Diese Abfrage funktioniert allerdings nur, solange der Name Daisy höchstens einmal in der Tabelle `spieler` vorkommt. Wollen wir entsprechend herausfinden, welche Karten die Spieler namens Daniel haben, so müssen wir auf mehrere Einträge in der Ergebnismenge der Unterabrage selektieren:

```
SELECT farbe, karte FROM spielkarten WHERE spieler IN (
   SELECT id FROM spieler WHERE name='Daniel'
);
```

also `IN` statt = verwenden.

9.8.2 Mehrere Kartenspiele

Betrachten wir nun eine etwas andere Situation, die unser obiges Datenmodell über den Haufen wirft. Nehmen wir an, wir wollen die Kartenverteilungen eines gesamten Spieleabends speichern, also mehrere Spiele. Das heißt, im Gegensatz zu unserem Modell in Abbildung 9.3 kann eine Spielkarte von mehreren Spielern aufgenommen werden. Wir haben also eine CM-CM-

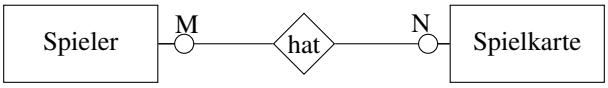


Abbildung 9.4: ER-Modell mehrerer Kartenspiele

Beziehung vorliegen. Mit Regel 8 wird sie aufgelöst durch eine Beziehungstabelle und zwei 1-1-Beziehungen (Abbildung 9.5).

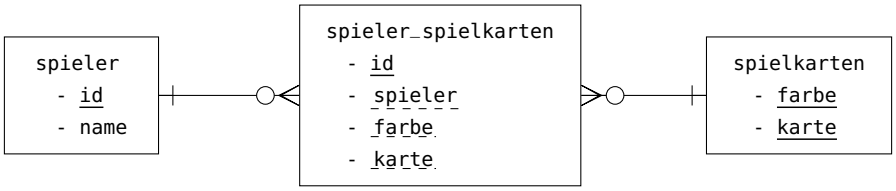


Abbildung 9.5: Tabellenmodell zum ER-Modell in Abbildung 9.4

In tabellarischer Form lautet dieses Modell etwas kompakter:

Tabelle	Primärschlüssel	Attribute
Spieler	id	name
Spielkarten	(farbe, karte)	spieler
Spieler_Spielkarten	id	spieler, farbe, karte

Mit den folgenden SQL-Anweisungen werden die vier Spieler*innen, ein vollständiges Skatspiel (32 Karten) und die Beziehungstabelle angelegt:

```
CREATE TABLE spieler (
  id SERIAL PRIMARY KEY,
  name varchar(20) NOT NULL
);
--
CREATE TABLE spielkarten (
  farbe varchar(5),
  karte varchar(5) NOT NULL,
  PRIMARY KEY (farbe, karte),
);
```

```
--
CREATE TABLE spieler_spielkarten (
  id SERIAL PRIMARY KEY,
  spieler int NOT NULL,
  farbe varchar(5) NOT NULL,
  karte varchar(5) NOT NULL,
  FOREIGN KEY (spieler) REFERENCES spieler(id) ON DELETE RESTRICT,
  FOREIGN KEY (farbe, karte) REFERENCES spielkarten(farbe, karte) ON DELETE RESTRICT
);
```

Das Einfügen der fünf Karten für Daniel, Daisy und Daniel geschieht durch:

```
INSERT INTO spieler (name) VALUES
 ('Daniel'), ('Donald'), ('Daisy'), ('Daniel');
INSERT INTO spielkarten (farbe, karte) VALUES
 ('Kreuz', '7'),    ('Kreuz', '8'),    ('Kreuz', '9'),     ('Kreuz', '10'),
 ('Kreuz', 'Bube'), ('Kreuz', 'Dame'), ('Kreuz', 'König'), ('Kreuz', 'Ass'),
 ('Pik',   '7'),    ('Pik',   '8'),    ('Pik',   '9'),     ('Pik',   '10'),
 ('Pik',   'Bube'), ('Pik',   'Dame'), ('Pik',   'König'), ('Pik',   'Ass'),
 ('Herz',  '7'),    ('Herz',  '8'),    ('Herz',  '9'),     ('Herz',  '10'),
 ('Herz',  'Bube'), ('Herz',  'Dame'), ('Herz',  'König'), ('Herz',  'Ass'),
 ('Karo',  '7'),    ('Karo',  '8'),    ('Karo',  '9'),     ('Karo',  '10'),
 ('Karo',  'Bube'), ('Karo',  'Dame'), ('Karo',  'König'), ('Karo',  'Ass');
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES
 (1, 'Herz', 'Ass'),
 (3, 'Pik',  '7'),
 (3, 'Pik',  'Bube'),
 (4, 'Karo', 'Ass'),
```

Weitere Karten könnten an die drei veerteilt werden mit mehr INSERTs:

```
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (1, 'Herz', 'Ass');
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (2, 'Kreuz', 'Dame');
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (4, 'Pik', '7');
```

Um die Karten anzeigen zu lassen, die Daisy im Laufe des Spieleabends auf der Hand hatte, müssen wir nun aus der Beziehungstabelle selektieren:

```
SELECT farbe, karte FROM spieler_spielkarten WHERE spieler = (
  SELECT id FROM spieler WHERE name='Daisy'
);
```

Entsprechend erhält man die Karten, die die Spieler mit Namen Daniel hatten, mit der Anweisung:

```
SELECT farbe, karte FROM spieler_spielkarten WHERE spieler IN (
  SELECT id FROM spieler WHERE name='Daniel'
);
```

„Nach außen“ unterscheiden sich die Versionen 1-CM und CM-CM also nicht, das „interne“ Tabellenmodell allerdings ist komplexer geworden. Sollten Sie im Ihrer späteren Laufbahn an einem Projekt beteiligt sein, in dem Daten zu modellieren sind — sei es als Anwender oder als Entwickler —, so wissen Sie, worauf Sie zu achten haben: Fragen Sie lieber einmal zu viel nach, ob es sich bei einer konkreten Datenkonstellation wirklich um eine 1:N oder um eine M:N Beziehung handelt. Allein der kleine Austausch von 1 durch M kann im schlimmsten Fall eine ganze Datenbank für den praktischen Einsatz unbrauchbar machen!

10 Normalisierung

Kapitelübersicht

10.1 Anomalien 81
10.2 Die ersten vier Normalformen 84
10.3 Anwendungsfall: Die Comic-Alben 86
10.4 Zusammenfassung 88

Wir haben Datenmodelle bisher vorwiegend auf der Ebene von Tabellen behandelt, angefangen von der Modellierung der Entitätstypen und ihrer Beziehungen untereinander bis zur Ableitung des konkreten Tabellenmodells daraus nach bestimmten Ableitungsregeln. Die Normalisierung nun ist ein standardisiertes Verfahren, mit dem sich auf *Ebene der Attribute* der Tabellen Redundanzen (d.h. die mehrfache Speicherung gleicher Daten) und Anomalien vermeiden lassen. Anomalien sind fatal für Datenbanken, da sie beim Einfügen, Ändern oder Löschen von Datensätzen zu inkostenten oder lückenhaften Dateninhalten führen. Redundanzen auf der anderen Seite können zu Anomalien führen, verhindern aber auf jeden Fall eine effiziente Datenspeicherung, was insbesondere bei großen Datenbanken zu ernsten Problemen führen kann.

Bei der Normalisierung unterscheidet man in der Literatur bis zu sechs sogenannte *Normalformen*, wovon in der Praxis jedoch nur vier berücksichtigt werden. Diese vier Normalformen werden wir in diesem Kapitel behandeln.

10.1 Anomalien

Eine *Anomalie* bezeichnet in der Informatik ein Fehlverhalten der Datenverwaltung, die zu Inkonsistenzen der Daten führen. Es gibt mehrere Arten von Anomalien. Im Allgemeinen für ein Mehrbenutzersystem wichtig sind Anomalien durch mangelhafte Synchronisation der Daten. Da sie vorwiegend durch das Managementsystem einer Datenbank gelöst werden und vor allem bei paralleler Datenspeicherung auftreten, werden wir uns im Weiteren aber damit nicht beschäftigen.[1] Daneben gibt es nämlich noch auch drei für den Einbenutzerbetrieb wichtige Anomalien, die das Einfügen, Ändern und Löschen betreffen. Wir verwenden dazu das Datenbeispiel aus Piepmeyer (2011:S. 143ff). Die Tabelle erscheint auf den ersten Blick recht plausibel. Jeder Datensatz speichert die für ein Album relevanten Daten — also alles klar! Gehen wir nun davon

[1] Siehe dazu auch `https://de.wikipedia.org/wiki/Anomalie_(Informatik)`

alben

reihe	band	titel	verlag	jahr
Asterix	1	Asterix, der Gallier	Ehapa	1968
Asterix	17	Die Trabantenstadt	Ehapa	1974
Asterix	25	Der große Graben	Ehapa	1980
Tim und Struppi	1	Der geheimnisvolle Stern	Carlsen	1972
Franka	1	Das Kriminalmuseum	Epsilon	1985
Franka	2	Das Meisterwerk	Epsilon	1986

Tabelle 10.1: Beispieldaten der Tabelle `alben` zur Illustration der Anomalien.

aus, dass Alben einer gegebenen Reihe *immer* im gleichen Verlag erscheinen, so können wir dies nicht mit Integritätsregeln automatisch sicherstellen. Anders sieht es für Wertebereiche von Attributen aus: Wir können mit dem reservierten Wort `CHECK` bei den Attributdeklarationen die Integritätsregeln einfügen, dass die Nummer des Bandes positiv sein muss und das Erscheinungsjahr 1937 oder später sein soll,[2] d.h. wir können das folgende Tabellenschema erstellen:

```
CREATE TABLE alben(
  reihe  varchar(30),
  band   int CHECK(band > 0),
  titel  varchar(30) NOT NULL,
  verlag varchar(30) NOT NULL,
  jahr   int CHECK(jahr >= 1937),
  PRIMARY KEY(reihe, band)
);
```

Mit der folgenden SQL-Anweisung können wir dann die folgenden Beispieldaten speichern:

```
INSERT INTO alben(reihe, band, titel, verlag, jahr) VALUES
  ('Asterix', 1, 'Asterix der Gallier', 'Ehapa', 1968),
  ('Asterix', 17, 'Die Trabantenstadt', 'Ehapa', 1974),
  ('Asterix', 25, 'Der große Graben', 'Ehapa', 1980),
  ('Tim und Struppi', 1, 'Der geheimnisvolle Stern', 'Carlsen', 1972),
  ('Franka', 1, 'Das Kriminalmuseum', 'Epsilon', 1985),
  ('Franka', 2, 'Das Meisterwerk', 'Epsilon', 1986);
```

Die nichtimplementierbare Regel „jede Reihe immer im selben Verlag“ können wir bestenfalls als „Programmierrichtlinie“ festlegen, aber ein Anwender kann möglicherweise nichts davon wissen oder sich einfach vertippen. So kann es zu den folgenden Anomalien kommen.

10.1.1 Die Einfüge-Anomalie

Eine *Einfüge-Anomalie* liegt vor, wenn ein Datensatz nicht in eine Datenbank eingefügt werden kann, da nicht alle Informationen zum Primärschlüssel vorliegen. Für unsere Beispieldaten würde eine Anomalie etwa durch den folgenden Befehl bewirkt:

```
INSERT INTO alben(reihe, band, titel, verlag, jahr) VALUES
  ('Asterix', 2, 'Asterix und Kleopatra', 'Ehara', 1968);
```

Durch einen blöden Tippfehler (oder wider besseren Wissens) haben wir gegen die obige Regel verstoßen und die Reihe Asterix in einem neuen Verlag erscheinen lassen. Unser Datenbestand ist gemäß dieser Regel nun inkonsistent.

[2] Im Jahr 1937 erschien das erste Album *(comic book)* der Comicreihe *Detective Comics*; in derselben Reihe erschien im Mai 1939 übrigens erstmals Batman, vgl. `https://de.wikipedia.org/wiki/Comic#Heft-_und_Buchformate`.

10.1.2 Die Änderungsanomalie

Eine *Änderungsanomalie* liegt vor, wenn bei Änderung eines Attributwertes nicht alle betroffenen Datensätze geändert werden. Ähnlich wie bei der Einfüge-Anomalie ergäbe für unsere Beispieldaten der folgenden Änderungsbefehl eine Anomalie:

```
UPDATE alben SET verlag='Springer' WHERE reihe='Asterix' AND band=1;
```

Auch hier liegt dann also eine Dateninkonsistenz gemäß unserer Regel vor.

10.1.3 Die Löschanomalie

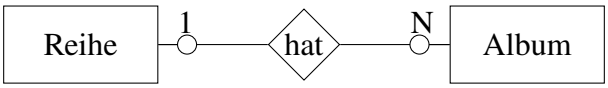
Eine *Löschanomalie* liegt vor, wenn durch das Löschen eines Datensatzes mehr Informationen als erforderlich verloren gehen. Löschen wir beispielsweise aus unserem Beispieldaten das Album *Tim und Struppi*:

```
DELETE FROM alben WHERE reihe='Tim und Struppi' AND band=1;
```

Eigentlich wollten wir doch nur ein einzelnes Album aus unserer Datenbank löschen, z.B. weil wir es aktuell nicht mehr haben. Verloren haben wir jedoch zusätzlich jegliche Information darüber, in welchem Verlag diese Reihe erscheint, obwohl das gar nicht unser Ziel . Die Löschanomalie hat also eine etwas andere Qualität als die anderen beiden Anomalien: Zwar führt sie nicht zu Dateninkonsistenzen gemäß unserer Regel, aber vernichtet mehr Information als gewollt.

10.1.4 Auflösung der Anomalien durch Normalisierung

Anomalien sind für Datenbanken ein ernstes Problem. Sie führen zu Dateninkonsistenzen und zu überflüssigen Informationsverlusten. Nicht alle lassen sich durch Integritätsregeln in SQL verhindern. Müssen wir uns also damit abfinden? Betrachten wir dazu die wesentliche Ursache von Anomalien: Wir haben in einem Datensatz *zu viel* Information gespeichert. Stattdessen wäre es vorteilhaft, die Information auf mehrere Datensätze zu verteilen. So muss ein Album ja nur „wissen“, zu welcher Reihe es gehört: gemäß unserer Regel „jede Reihe immer im selben Verlag“ muss die Information über den Verlag in einem eigenen Datensatz gespeichert werden. Wir benötigen also *zwei* Tabellen:



Das ergibt die Tabellenschemata

```
CREATE TABLE reihen(
  reihe  varchar(30),
  verlag  varchar(30) NOT NULL,
  PRIMARY KEY(reihe)
);
CREATE TABLE alben(
  reihe  varchar(30),
  band   int CHECK(band > 0),
  titel  varchar(30) NOT NULL,
  jahr   int CHECK(jahr >= 1937),
  PRIMARY KEY(reihe, band),
  FOREIGN KEY(reihe) REFERENCES reihen(reihe)
);
```

und die Datenspeicherungen

```
INSERT INTO reihen(reihe, verlag) VALUES
  ('Asterix', 'Ehapa'),
  ('Tim und Struppi', 'Carlsen'),
  ('Franka','Epsilon');

INSERT INTO alben(reihe, band, titel, jahr) VALUES
  ('Asterix', 1, 'Asterix, der Gallier', 1968),
  ('Asterix', 17, 'Die Trabantenstadt', 1974),
  ('Asterix', 25, 'Der große Graben', 1980),
  ('Tim und Struppi', 1, 'Der geheimnisvolle Stern', 1972),
  ('Franka', 1, 'Das Kriminalmuseum', 1985),
  ('Franka', 2, 'Das Meisterwerk', 1986);
```

Damit können die obigen Anomalien nicht mehr passieren! Wir haben also mit einem konsequenten Redesign unseres Tabellenmodells die Informationen so verteilt, dass sie unseren expliziten Regeln entsprechen. Was aber ist hier eigentlich geschehen? In einem tieferen Sinn haben wir aus unserem ursprünglichen Tabellenentwurf Redundanzen entfernt. Um dies systematisch durchzuführen, wird in der Informatik das Verfahren der *Normalisierung* eines Datenmodells durchgeführt. Es liefert mit einer schrittweisen Redundanzsbefreiung in Form von sogenannten Normalformen eine Art Qualitätsverbesserung eines Datenbanktentwurfs. Dazu betrachtet es die Beziehungen der Attribute einer Tabelle untereinander. Wie dies geschieht, werden wir im Folgenden detailliert betrachten.

10.2 Die ersten vier Normalformen

Die Normalisiserung mit den Normalformen ist ein schrittweiser Prozess, in dem im Grunde Qualitätskriterien an einen einmal entworfenes Datenmodell angewendet werden, um Datenredundanz und Anomalien zu verhindern. Um die Normalformen zu erläutern, verwenden wir das folgende Anwendungsbeispiel meines Kollegen Hermann Johannes, der es in seinem Lehrbrief zu Datenbanken und in seinen Vorlesungen benutzte. Nehmen wir an, in einem Projekt zur Erstellung einer Datenbank für die Auftragsabwicklung seien durch die Datenmodellierung die folgenden Tabellen ermittelt worden:

Tabelle	**Primärschlüssel**	**Attribute**
Kunden	Kundennr	Firma, Ort, AuftragsNr
Aufträge	AuftragsNr	Auftragsdatum, Lieferdatum
Artikel	ArtikelNr	Bezeichnung, LagerNr, Lagerort
Positionen	ArtikelNr, AuftragsNr	Menge, Nettopreis
Rechnungen	RechnungsNr	Datum, AuftragsNr, Nettopreis, Umsatzsteursatz, Bruttopreis

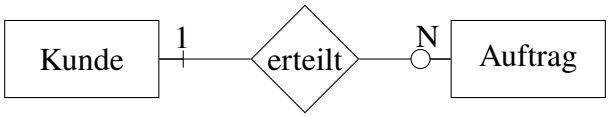
(10.1)

10.2.1 Erste Normalform

Eine Tabelle ist in der *ersten Normalform* (1NF), wenn ihre Attribute nur einfache Werte besitzen und nicht wieder Tabellen sind: Es gibt keine Tabellen in einer Tabelle. Beispielsweise widerspricht die folgende Tabelle *kunden* der 1NF:

kunden(KundenNr, Firma, Ort, AuftragsNr)

Denn falls der Kunde mehrere Aufträge erteilt, enthält die Spalte Auftragsnummer eine Liste von mehreren Werten, was einer Tabelle entspricht. Lösung: Da es sich hier um eine C-CM-Beziehung handelt,



ist sie falsch aufgelöst. Stattdessen muss das Attribut Auftragsnummer aus der Tabelle Kunde gestrichen werden und die Tabelle Aufträge die Kundennummer als Fremdschlüssel erhalten.

Mit anderen Worten: Halten wir uns an die Implementierungsregeln aus Abschnitt 9.2, und insbesondere an Regel 6 und Regel 7, so ist die erste Normalform automatisch erfüllt.

10.2.2 Zweite Normalform

Eine Tabelle ist in der *zweiten Normalform*, wenn die folgenden Bedingungen erfüllt sind.

1. Sie befindet sich in der ersten Normalform.
2. Attribute, die nicht zum Primärschlüssel gehören, sind von *allen* Schlüsselattributen abhängig.

(„Eine Tabelle mit natürlichem Primärschlüssel bildet eine logische Einheit von Nichtschlüsselattributen.“) Die folgende Tabelle ist zwar in der ersten Normalform, aber nicht in der zweiten:

Positionen(ArtikelNr, AuftragsNr, Menge, Preis),

denn der Preis ist nicht abhängig von der AuftragsNr. Lösung: Das Attribut Preis muss in eine (neue) Tabelle Artikel ausgelagert werden.

10.2.3 Dritte Normalform

Eine Tabelle ist in der *dritten Normalform*, wenn die folgenden Bedingungen erfüllt sind.

1. Sie befindet sich in der zweiten Normalform.
2. Es gibt keine indirekten (transitiven) Abhängigkeiten zwischen zwei Tabellenattributen.

(„Nichtschlüsselattribute sind unabhängig voneinander.“) Die folgende Tabelle ist in der zweiten Normalform, aber nicht in der dritten:

Artikel (ArtikelNr, Preis, LagerNr, Lagerort),

denn der Lagerort hängt von der Lagernummer und daher transitiv von der Artikelnummer ab:

$$\boxed{\text{ArtikelNr} \Rightarrow \text{LagerNr} \quad \text{und} \quad \text{LagerNr} \Rightarrow \text{LagerOrt},} \quad \text{also} \quad \boxed{\text{ArtikelNr} \Rightarrow \text{LagerOrt}.}$$

Lösung: Das Attribut Lagerort muss in eine Tabelle *Lager* mit Primärschlüssel *LagerNr* ausgelagert und dieses Attributes in *Artikel* als Fremdschlüssel aufgenommen werden:

Artikel(ArtikelNr, LagerNr, Preis)

10.2.4 Vierte Normalform

Eine Tabelle ist in der *vierten Normalform*, wenn sie in 3NF ist und keine aus der Datenbank ableitbaren Attribute enthält. („Es gibt keine berechenbaren Kennzahlen.“)) Die folgende Tabelle ist in 3NF, aber nicht in 4NF:

Rechnung(RechnungsNr, Datum, AuftragsNr, Nettopreis, Umsatzsteuersatz, Bruttopreis),

denn der Bruttopreis ergibt sich aus dem Nettopreis und dem Umsatzsteuersatz. Lösung: Das Attribut Bruttopreis muss gestrichen werden.

10.2.5 Daumenregeln zur Überprüfung, ob Normalformen verletzt sind

Als Daumenregel zur Überprüfung, ob eine der Normalformen verletzt ist, kann folgende Liste der notwendigen Bedingungen für die *Verletzung* der jeweiligen Normalform dienen.

Normalform	Kann nur verletzt sein, wenn . . .
1NF	. . . bei einer 1:N-Beziehung der Fremdschlüssel bei 1 ist
2NF	. . . ein natürlicher Primärschlüssel aus Fremdschlüsseln besteht
3NF	. . . ein Fremdschlüssel eines der Attribute ist
4NF	. . . ein Attribut aus Daten der Datenbank ableitbar / berechenbar ist

Beachten Sie, dass die genannten Bedingungen nicht alle hinreichend sind. Zum Beispiel muss die zweite Normalform ja nicht verletzt sein, wenn der Primärschlüssel natürlich ist. Andererseits ist die genannte notwendige Bedingung für 4NF auch schon hinreichend.

10.2.6 Beispiel: Tabellen vor und nach der Normalisierung

Zusammenfassend bewirkt die Normalisierung des ursprünglichen Ausgangsmodells (10.1) unserer Datenmodellierung die folgenden Veränderungen.

Tabelle	Primärschlüssel	Attribute
Kunden	Kundennr	Firma, AuftragsNr, Ort
Aufträge	AuftragsNr	Auftragsdatum, Lieferdatum
Artikel	ArtikelNr	Bezeichnung, LagerNr, Lagerort
Positionen	ArtikelNr, AuftragsNr	Menge, Nettopreis
Rechnungen	RechnungsNr	Datum, AuftragsNr, Nettopreis, Umsatzsteursatz, Bruttopreis

Tabelle	Primärschlüssel	Attribute
Kunden	Kundennr	Firma, Ort
Aufträge	AuftragsNr	KundenNr, Auftragsdatum, Lieferdatum
Artikel	ArtikelNr	Bezeichnung, LagerNr, Nettopreis
Positionen	ArtikelNr, AuftragsNr	Menge
Rechnungen	RechnungsNr	Datum, AuftragsNr, Umsatzsteursatz
Lager	LagerNr	Ort

10.3 Anwendungsfall: Die Comic-Alben

Beispiel 10.1. [3] Wider besseres Wissen haben wir in Beispiel 3.4 auf Seite 20 unsere Comicalben in einer einzigen Tabelle abgelegt.

Tabelle	Primärschlüssel	Attribute
Album	Reihe, Band	Titel, Preis, Verlag, Jahr

[3] nach Piepmeyer (2011):S. 207ff.

Sie widerspricht aber insbesondere der zweiten Normalform, denn die Reihe eines Albums ist nicht abhängig von seinem Titel, muss also in eine eigene Tabelle ausgelagert werden:

Tabelle	Primärschlüssel	Attribute
Reihe	Reihe	Verlag
Album	Reihe, Band	Titel, Preis, Jahr

Nach der Normalisierung sieht das ER-Modell also wie folgt aus:

Es ist also eine 1-CM-Beziehung zwischen zwei Entitäten. Da jetzt alle Normalformen erfüllt sind, wie man schnell nachprüft, ist das Datenmodell damit normalisiert. Die Datenbank können wir es daher wie folgt implementieren:

```sql
CREATE TABLE reihen (
  reihe  varchar(30),
  verlag  varchar(30) NOT NULL,
  PRIMARY KEY(reihe)
);
```

```sql
CREATE TABLE alben (
  reihe  varchar(30),
  band   int CHECK(band > 0),
  titel  varchar(30) NOT NULL,
  preis  decimal(10,2),
  jahr   int CHECK(jahr >= 1937),
  PRIMARY KEY(reihe, band),
  FOREIGN KEY(reihe) REFERENCES reihen(reihe)
);
```

Angereichert mit den Daten:

```sql
INSERT INTO reihen (reihe, verlag) VALUES
  ('Asterix', 'Ehapa'),
  ('Tim und Struppi', 'Carlsen'),
  ('Franka','Epsilon'),
  ('Lucky Luke', 'Egmont'),
  ('Gespenster Geschichten', 'Bastei'),
  ('Prinz Eisenherz', 'Carlsen');
```

```sql
INSERT INTO alben (titel, reihe, band, preis, jahr) VALUES
  ('Asterix, der Gallier', 'Asterix', 1, 2.80, 1968),
  ('Asterix und Kleopatra', 'Asterix', 2, 2.80, 1968),
  ('Asterix als Legionär', 'Asterix', 10, 3.00, NULL),
  ('Die Trabantenstadt', 'Asterix', 17, 3.80, 1974),
  ('Zarter Schmelz', 'Lucky Luke', 1, 16.00, 2021),
  ('Der große Graben', 'Asterix', 25, 5.00, 1980),
  ('Das Kriminalmuseum', 'Franka', 1, 8.80, 1985),
  ('Das Meisterwerk', 'Franka', 2, 8.80, 1986),
  ('Der geheimnisvolle Stern', 'Tim und Struppi', 1, NULL, 1972),
  ('Tim und der Haifischsee', 'Tim und Struppi', 23, NULL, 1973);
```

ergibt das die Datenbestände:

reihen

reihe	verlag
Asterix	Ehapa
Tim und Struppi	Carlsen
Lucky Luke	Egmont
Franka	Epsilon
Prinz Eisenherz	Carlsen

alben

titel	reihe	band	preis	jahr
Asterix, der Gallier	Asterix	1	2.80	1968
Asterix und Kleopatra	Asterix	2	2.80	1968
Asterix als Legionär	Asterix	10	3.00	NULL
Die Trabantenstadt	Asterix	17	3.80	1974
Zarter Schmelz	Lucky Luke	1	5.00	2021
Der große Graben	Asterix	25	5.00	1980
Das Kriminalmuseum	Franka	1	8.80	1985
Das Meisterwerk	Franka	2	8.80	1986
Der geheimnisvolle Stern	Tim und Struppi	1	NULL	1972
Tim und der Haifischsee	Tim und Struppi	23	NULL	1973

□

10.4 Zusammenfassung

Die Normalisierung ist ein standardisiertes Verfahren, um Redundanzen und Anomalien durch ein falsches Tabellenschema zu vermeiden. Es wird nach der Entity-Relationship-Modellierung und der Ableitung des Tabellenmodells durchgeführt. Während diese beiden Modellierungsschritte die wesentlichen Tabellen des Datenmodells liefern, werden bei der Normalisierung die Attribute der einzelnen Tabellen betrachtet.

Schrittweise werden die sogenannten Normalformen 1NF bis 4NF untersucht. 1NF überprüft im Wesentlichen, ob die Regeln 6 und 7 bei der Tabellenableitung aus dem ER-Diagramm eingehalten wurden. 2NF betrachtet das Verhältnis der Attribute zum Primärschlüssel und 3NF das Verhältnis von Nicht-Schlüsselattributen untereinander. 4NF schließlich entfernt berechenbare Attribute.

11 Joins

Kapitelübersicht

11.1 Inner Joins 89
11.2 Left und Right Joins 91
11.3 Joins mit mehr als zwei Tabellen 96
11.4 Self Joins 98

Manchmal begegnet man dem Problem, sich Daten anzeigen zu lassen, für deren Auswahl Informationen aus mehreren Tabellen benötigt werden. In SQL können auf verschiedene Weise Abfragen auf mehrere Tabellen durchgeführt werden. Die allgemeine Variante ist, die zu betrachtenden Tabellen hinter `FROM` einfach mit Komma getrennt aufzulisten und gewünschte Verknüpfungen der Tabellen in der WHERE-Klausel zu schreiben, neben weiteren Filterbedingungen. Diese Variante haben wir in Kapitel 9.8 behandelt. Viel mächtiger und für meist eleganter und verständlicher ist jedoch ein „Join“, der zwischen Verknüpfungs- und Filterbedingungen sauber trennt. Es gibt zwei Typen von Joins, den „Inner Join“, auch „Equi-Join“ genannt, und den „Outer Join“. Die wichtigsten Varianten beider Join-Typen werden wir im Folgenden behandeln.

11.1 Inner Joins

Um zwei Tabellen zu verknüpfen, kann die JOIN-Klausel verwendet werden, hinter dem ON-Parameter steht die Verknüpfungsbedingung. Eine zusätzliche Filterbedingung kann, wie gewöhnlich, in einer optionalen WHERE-Klausel festgelegt werden. Die Syntax dafür lautet:

```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
INNER JOIN rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung;
```

Die Verknüpfungsbedingung wird oft *Join-Bedingung* oder `ON`-Klausel genannt. Der Join-Typ `INNER` vor `JOIN` kann nach SQL-Standard weglassen werden, in MS Access allerdings muss er verwendet werden. Sehr oft besteht die Verknüpfungsbedingung darin, dass der Fremdschlüssel der einen Tabelle mit dem Primärschlüssel `id` der anderen Tabelle übereinstimmen muss. Die ON-Klausel lautet daher typischerweise:

```
... ON linke_tabelle.fremdschlüssel = rechte_tabelle.id ...
```
(11.1)

Die Ergebnis eines Joins ist eine Tabelle, die die Spalten beider Tabellen enthält und die verknüpften Datensätze zu einem Datensatz zusammen fügt.

Beispiel 11.1. Betrachten wir die normalisierte Datenbank von Comic-Alben aus Beispiel 10.1:

alben

titel	reihe	band	preis	jahr
Gespenster Geschichten	1	1	1.20	1974
Asterix, der Gallier	2	1	2.80	1968
Asterix und Kleopatra	2	2	2.80	1968
Asterix als Legionär	2	10	3.00	NULL
Die Trabantenstadt	2	17	3.80	1974
Lucky Luke	NULL	1	5.00	1976
Der große Graben	2	25	5.00	1980
Der geheimnisvolle Stern	3	1	NULL	1972
Tim und der Haifischsee	3	23	NULL	1973
Das Kriminalmuseum	4	1	8.80	1985
Das Meisterwerk	4	2	8.80	1986

reihen

id	name
1	Gespenster Geschichten
2	Asterix
3	Tim und Struppi
4	Franka
5	Prinz Eisenherz

Hierbei ist `alben` die linke Tabelle, die mit der rechten Tabelle `reihen` über einen Fremdschlüssel verknüpft ist, d.h. der ON-Parameter lautet entsprechend Gleichung (11.1):

```
...  ON alben.reihe = reihen.id  ...
```

Mit dem Befehl

```
SELECT * FROM alben INNER JOIN reihen ON alben.reihe = reihen.id
```

erhalten wir damit die Ergebnistabelle:

titel	reihe	band	preis	jahr	id	name
Gespenster Geschichten	1	1	1.20	1974	1	Gespenster Geschichten
Asterix als Legionär	2	10	3.00	*null*	2	Asterix
Asterix und Kleopatra	2	2	2.80	1968	2	Asterix
Asterix, der Gallier	2	1	2.80	1968	2	Asterix
Der große Graben	2	25	5.00	1980	2	Asterix
Die Trabantenstadt	2	17	3.80	1974	2	Asterix
Der geheimnisvolle Stern	3	1	*null*	1972	3	Tim und Struppi
Tim und der Haifischsee	3	23	*null*	1973	3	Tim und Struppi
Das Kriminalmuseum	4	1	8.80	1985	4	Franka
Das Meisterwerk	4	2	8.80	1986	4	Franka

Die Gesamtheit der Spalten besteht also aus allen Spalten beider Tabellen. Verwenden wir nun einen Join, um uns nur den Namen der Reihe und den Titel des Albums anzeigen zu lassen, allerdings mit der Bezeichnung „reihe“ für den Reihennamen. Dazu wählen wir nur zwei Spalten aus:

```
SELECT alben.titel, reihen.name AS reihe
FROM alben
INNER JOIN reihen ON alben.reihe = reihen.id;
```

und erhalten so die Ergebnistabelle:

titel	reihe
Asterix als Legionär	Asterix
Asterix und Kleopatra	Asterix
Asterix, der Gallier	Asterix
Das Kriminalmuseum	Franka
Das Meisterwerk	Franka
Der geheimnisvolle Stern	Tim und Struppi
Der große Graben	Asterix
Die Trabantenstadt	Asterix
Gespenster Geschichten	Gespenster Geschichten
Tim und der Haifischsee	Tim und Struppi

Mit dem Ausdruck `AS reihe` erreichen wir, dass in der Ergebnistabelle statt des Attributnamen `name` eben `reihe` steht.

Wollen wir uns nun nur die Alben aus der Asterix-Reihe anzeigen lassen, so fügen wir einfach eine geeignete WHERE-Klausel an:

```
SELECT reihen.name AS reihe, alben.titel FROM alben
INNER JOIN reihen ON alben.reihe = reihen.id
WHERE reihen.name='Asterix';
```

reihe	titel
Asterix	Asterix, der Gallier
Asterix	Asterix und Kleopatra
Asterix	Asterix als Legionär
Asterix	Die Trabantenstadt
Asterix	Der große Graben

Wir sehen mit diesem Beispiel, das bei einem Inner Join beide Tabellen passende Einträge haben müssen. Die Reihe „Prinz Eisenherz“ taucht hier nirgendwo auf, ebenso das Album „Lucky Luke“. Daraus leiten wir das folgende allgemeine Merkmal eines Inner Joins ab. □

Regel 15. (Merkmal eines Inner Joins) *Durch einen Inner Join können nur Datensätze angezeigt werden, die in den beiden Tabellen miteinander verknüpft sind. Beide Tabellen sind dabei gleichberechtigt. Lediglich bei bei* `SELECT * FROM ...` *wird die Reihenfolge der angezeigten Spalten der Ergebnismenge vertauscht, da die Spalten der linken Tabelle hier stets zuerst erscheinen.*

11.2 Left und Right Joins

Im Gegensatz zu einem Inner Join werden bei einem *Outer Join* auch Datensätze ohne Verknüpfung zur verknüpften Tabelle angezeigt. Eine der Tabellen ist dabei jedoch führend in dem Sinne, dass von ihr auch nichtverknüpfte Datensätze angezeigt werden. Je nach Typ des Outer Joins ist das die linke oder die rechte Tabelle. Entsprechend gibt es einen `LEFT JOIN` mit der Syntax

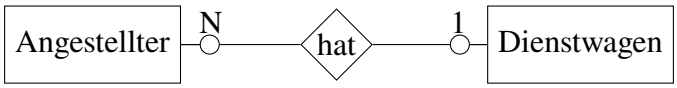
```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
LEFT JOIN rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung;
```

oder einen `RIGHT JOIN` mit der Syntax

```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
RIGHT JOIN rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung;
```

(Statt `JOIN` darf man bei den meisten RDBMS jeweils auch `OUTER JOIN` schreiben. Da aber mit einem der Wörter `LEFT` oder `RIGHT` der Typ des Joins schon eindeutig festgelegt ist, werden wir `OUTER` im Folgenden weglassen.)

Beispiel 11.2. Betrachten wir als Anwendungsfall eine kleine Datenbank, in der die Angestellten einer Firma und die Dienstwagen gespeichert sind. Grundsätzlich kann jede* Angestellte nur einen Dienstwagen gestellt bekommen, aber mehrere Angestellte können sich einen Dienstwagen teilen.



Gegeben seien dazu die beiden folgenden Tabellen.

```
-- Tabellenstrukturen:
CREATE TABLE angestellte (
   id int PRIMARY KEY,
   name varchar(10),
   dienstwagen int,
   FOREIGN KEY (dienstwagen) REFERENCES dienstwagen(id) ON DELETE RESTRICT
);
CREATE TABLE dienstwagen (
   id int PRIMARY KEY,
   kennzeichen varchar(10)
);
-- Daten:
INSERT INTO angestellte (id, name, dienstwagen) VALUES
  (1, 'Anna', 2), (2, 'Otto', null), (3, 'Alice', 1), (4, 'Bob', 1);
INSERT INTO dienstwagen (id, kennzeichen) VALUES
  (1, 'HA-FH 1234'), (2, 'HA-FH 2345'), (3, 'HA-FH 3456');
```

also

angestellte

id	name	dienstwagen
1	Anna	2
2	Otto	*null*
3	Alice	1
4	Bob	1

dienstwagen

id	kennzeichen
1	HA-FH 1234
2	HA-FH 2345
3	HA-FH 3456

Wie können wir nun alle Angestellten mit ihren Dienstwagen anzeigen lassen, wobei für Angestellte ohne Dienstwagen der Wert `NULL` erscheinen soll?

Ein Left Join über alle Spalten ergibt

```
SELECT * FROM angestellte LEFT JOIN dienstwagen
  ON angestellte.dienstwagen = dienstwagen.id;
```

id	name	dienstwagen	id	kennzeichen
1	Anna	2	2	HA-FH 2345
2	Otto	*null*	*null*	*null*
3	Alice	1	1	HA-FH 1234
4	Bob	1	1	HA-FH 1234

Wir sehen, alle Datensätze der linken Tabelle `angestellte` sind in der Ergebnismenge enthalten sowie die verknüpften Datensatze aus der rechten Tabelle `dienstwagen`. Alle nicht eingeteilten Dienstwagen werden nicht angezeigt. Ein Right Join über alle Spalten dagegen hat das Resultat:

```
SELECT * FROM angestellte RIGHT JOIN dienstwagen
  ON angestellte.dienstwagen = dienstwagen.id;
```

id	name	dienstwagen	id	kennzeichen
1	Anna	2	2	HA-FH 2345
3	Alice	1	1	HA-FH 1234
4	Bob	1	1	HA-FH 1234
null	*null*	*null*	3	HA-FH 3456

Hier sind alle Datensätze der rechten Tabelle `dienstwagen` enthalten, sowie die mit ihnen verknüpften aus der linken Tabelle `angestellte`. Entsprechend werden alle Angestellten ohne Dienstwagen, hier also Otto, nicht angezeigt. □

Bemerkung 11.3. Die Wirkungsweise der verschiedenen Joins ist in Abbildung 11.1 illustriert. Hierbei sind A und B zwei Tabellen, deren Datensätze als Mengen dargestellt sind. Die Schnittmenge repräsentiert dabei jeweils die durch die ON-Bedingung verknüpften Datensätze. Im

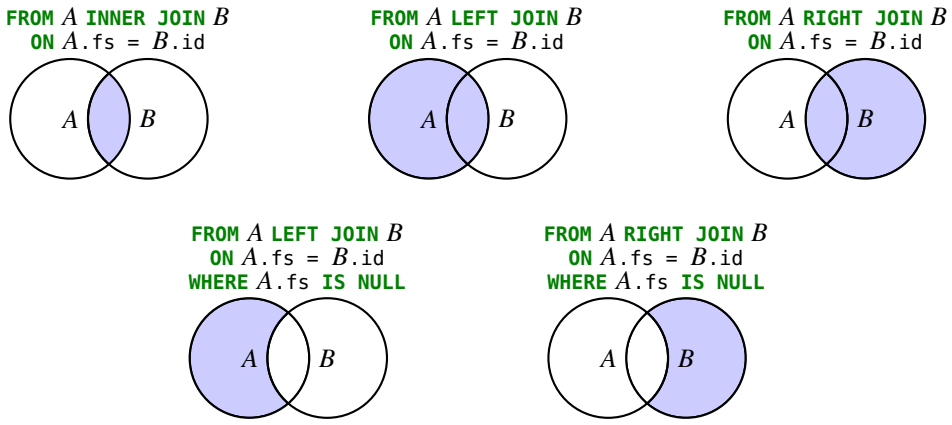
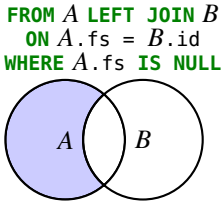


Abbildung 11.1: Wirkung von Joins auf zwei Tabellen A und B. Die Schnittmenge repräsentiert die durch die Join-Bedingung verknüpften Datensätze. (A`.fs` ist der Fremdschlüssel von A, B`.id` der Primärschlüssel von B.) Vgl. [SQL1, „Mehr zu JOIN“]

Umkehrschluss sind die Teile der Mengen außerhalb der Schnittmenge diejenigen Datensätze, die keine Verknüpfung zu der jeweils anderen Tabelle haben. Ein Inner Join hat als Ergebnismenge also nur die miteinander verknüpften Datensätze beider Tabellen, ein Left Join alle Datensätze der Tabelle A, und ein Right Join alle der Tabelle B. Wollen wir, wie in der zweiten Zeile, die nichtverknüpften Datensätze explizit anzeigen, so muss eine WHERE-Klausel hinzugefügt werden, um die Datensätze mit dem Wert `NULL` ihres Verknüpfungsschlüssels zu filtern, in der Regel der Fremdschlüssel. □

Bemerkung 11.4. Tatsächlich hätten wir für den vorletzten Fall in Abbildung 11.1,



keinen `LEFT JOIN` gebraucht, da für die Filterbedingung in der `WHERE`-Klausel keine Information aus Tabelle *B* benötigt wird. Dasselbe Ergebnis würde hier also auch durch eine Anweisung nur mit *A* erreicht, d.h. durch

```
FROM A
WHERE A.fs IS NULL
```

statt

```
FROM A LEFT JOIN B
 ON A.fs = B.id
WHERE A.fs IS NULL
```

Wir werden im folgenden Beispiel aber Verknüpfungsbedingungen betrachten, die nicht auf einem Fremdschlüssel `A.fs` basieren, sondern auf anderen Informationen aus *B*. □

Beispiel 11.5. *(Wann ist ein Join notwendig?)* Gegeben seien zwei Tabellen, von denen wir annehmen, dass sie unabhängig voneinander sind, also keine direkte Beziehung miteinander haben:

freunde

name	wohnort
Anna	Hagen
Bert	Iserlohn
Cindy	Soest
Dora	Meschede
Ernie	Lüdenscheid

kollegen

name	abteilung
Anna	Controlling
Cindy	IT
Ernie	Vertrieb
Gerd	Marketing

Wie lautet eine SQL-Abfrage, die uns die Namen unserer Freunde anzeigt, die nicht auch Kollegen sind? Gesucht ist also der Fall  aus Abbildung 11.1, d.h. ein Left Join: Wähle alle Datensätze aus `freunde` aus, deren `name` nicht in `kollegen` ist,

```
SELECT freunde.name FROM freunde
LEFT JOIN kollegen ON freunde.name = kollegen.name
WHERE kollegen.name IS NULL;
```

Da in der Tabelle `freunde` kein Fremdschlüssel oder sonst eine Information aus der Tabelle `kollegen` vorhanden ist, *müssen* wir hier einen Outer Join verwenden! □

Regel 16. *Ein Join ist immer notwendig, wenn für eine Abfrage Information aus mehreren Tabellen benötigt wird.*

Beispiel 11.6. *(Left und Right Joins)* Gegeben seien zwei Tabellen, von denen wir annehmen, dass sie unabhängig voneinander sind, also keine direkte Beziehung miteinander haben:

freunde

name	wohnort
Anna	Hagen
Bert	Iserlohn
Cindy	Soest
Dora	Meschede
Ernie	Lüdenscheid

kollegen

name	abteilung
Anna	Controlling
Cindy	IT
Ernie	Vertrieb
Gerd	Marketing

Wie lautet eine SQL-Abfrage, die uns die Namen derjenigen anzeigt, die nicht gleichzeitig Freunde und Kollegen sind? Überlegen wir uns dazu die Lösung schrittweise:

1. Wähle alle Datensätze aus `freunde` aus, deren `name` nicht in `kollegen` ist. 

2. Wähle alle Datensätze aus `kollegen` aus, deren `name` nicht in `feunde` ist. 

3. Vereinige beide Ergebnistabellen. 

1. Hier wird ein Left Join benötigt (siehe Beispiel 11.5)

```
SELECT freunde.name FROM freunde
LEFT JOIN kollegen ON freunde.name = kollegen.name
WHERE kollegen.name IS NULL;
```

name
Bert
Dora

(11.2)

2. Hier verwenden wir analog einen Right Join:

```
SELECT kollegen.name FROM freunde
RIGHT JOIN kollegen ON freunde.name = kollegen.name
WHERE freunde.name IS NULL;
```

name
Gerd

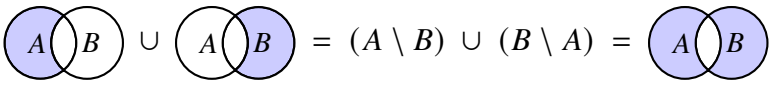
(11.3)

3. Vereinigen wir zum Schluss die beiden Abfragen (11.2) und (11.3) mit `UNION`, so erhalten wir unsere Lösung:

```
SELECT freunde.name FROM freunde
  LEFT JOIN kollegen ON freunde.name = kollegen.name
  WHERE kollegen.name IS NULL
UNION
SELECT kollegen.name FROM freunde
  RIGHT JOIN kollegen ON freunde.name = kollegen.name
  WHERE freunde.name IS NULL
```

name
Bert
Dora
Gerd

Tor! Übrigens hätten wir das Problem auch mengentheoretisch lösen können, indem wir jeweils alle Datensätze der einen Tabelle selektieren und davon alle Datensätze der anderen als Menge subtrahieren, dasselbe mit vertauschten Tabellen selektieren; und schließlich beides vereinen:



(11.4)

also in SQL:

```
( (SELECT name FROM freunde) EXCEPT (SELECT name FROM kollegen) )
UNION
( (SELECT name FROM kollegen) EXCEPT (SELECT name FROM freunde) );
```

Diese Lösung ist nach meinem Empfinden noch eleganter, allerdings funktioniert sie nicht für alle Datenbanksysteme (insbesondere nicht Access). □

11.3 Joins mit mehr als zwei Tabellen

Für die Anzahl der Tabellen eines Joins gibt es keine Obergrenze. Joins bieten sich daher an, bei Beziehungstabellen statt der Fremdschlüssel aussagekräftigere Bezeichnungen der referenzierten Tabelle anzuzeigen. Betrachten wir dazu als Anwendungsbeispiel das Problem, unsere Comic-Datenbank um die Möglichkeit zu erweitern, auch die Autoren der Alben zu speichern.[1] Ein Autor kann dabei mehrere Alben schreiben, umgekehrt kann ein Album mehrere Autoren haben. Da jedes Album mindestens einen Autor haben muss, erhalten wir damit das das Entity-Relationship-Diagramm in Abbildung 11.2. Da zwischen Album und Autor eine CM-CM-Bezie-

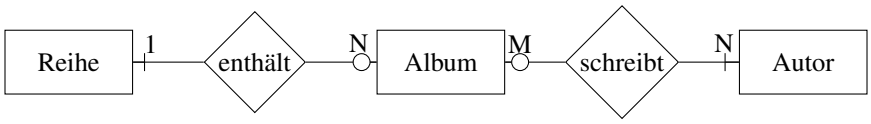


Abbildung 11.2: ER-Diagramm der Comics mit Autoren

hung vorliegt, müssen wir zwischen ihnen eine Beziehungstabelle einführen. Die Attribute eines Autors sollen nur aus einer ID und seinem Namen bestehen, d.h. die Autoren sind vom Relationentyp *autoren(id, name)*, die Beziehungstabelle vom Typ *albenautoren(autor, titel)*. Die vollständige Tabellenstruktur ist damit wie folgt:

Tabelle	Primärschlüssel	Attribute
reihen	id	name
alben	titel	reihe, band, preis, jahr
autoren	id	name
albenautoren	autor, titel	

In SQL erzeugen wir damit eine Tabelle `autoren`, und entsprechend eine Tabelle `albenautoren`:

```
CREATE TABLE autoren (
  id int NOT NULL,
  name varchar(20) NOT NULL,
  PRIMARY KEY(id)
);
```

```
CREATE TABLE albenautoren (
  autor int NOT NULL,
  titel varchar(50) NOT NULL,
  PRIMARY KEY(autor,titel),
  FOREIGN KEY(autor) REFERENCES autoren(id) ON DELETE RESTRICT,
  FOREIGN KEY(titel) REFERENCES alben(titel) ON DELETE RESTRICT
);
```

Speichern wir zum Schluss noch Daten in die Tabellen:

```
INSERT INTO autoren (id, name) VALUES
  (1, 'Uderzo'),
  (2, 'Goscinny'),
  (3, 'Hergé'),
  (4, 'Kuijpers'),
  (5, 'Franquin'),
  (6, 'Morris');
```

```
INSERT INTO albenautoren (autor, titel) VALUES
  (1, 'Asterix und Kleopatra'),
  (1, 'Asterix, der Gallier'),
  (1, 'Der große Graben'),
  (1, 'Die Trabantenstadt'),
  (2, 'Asterix und Kleopatra'),
  (2, 'Asterix, der Gallier'),
  (2, 'Die Trabantenstadt'),
  (3, 'Der geheimnisvolle Stern'),
  (3, 'Tim und der Haifischsee'),
  (4, 'Das Kriminalmuseum'),
  (4, 'Das Meisterwerk'),
  (6, 'Lucky Luke');
```

[1] Piepmeyer (2011):S. 215ff.

Zusammen mit den Tabellen und Daten aus Beispiel 10.1 haben wir die Datenbank von Comic-Alben vervollständigt.

Wie können wir uns nun eine Liste der Autoren und den Reihen, an denen sie mitgewirkt haben, jeweils mit Namen anzeigen? Anhand des Tabelldiagramms in Abbildung 11.2 erkennen wir, dass wir die Namen in den Tabellen `autoren` und `reihen` nur über zwei weitere Tabellen verknüpfen können. Wir können das mit einer einfachen Aneinanderreihung von **`INNER JOINS`** programmieren:

```
SELECT DISTINCT autoren.name AS autor, reihen.name AS reihe
FROM       autoren
INNER JOIN albenautoren ON autoren.id = albenautoren.autor
INNER JOIN alben        ON albenautoren.titel = alben.titel
INNER JOIN reihen       ON reihen.id = alben.reihe
```

Hinweis: Verwenden Sie bei einem Join über mehr als zwei Tabellen zur Unterstützung ein ER-Diagramm!

Gibt es in unserer Comic-Datenbank Autoren, für die kein Titel gespeichert sind, und anders herum Titel ohne Autor? Das können wir mit zwei Left Joins feststellen:

```
SELECT alben.titel AS "Titel ohne Autor"
FROM alben
LEFT JOIN albenautoren ON alben.titel = albenautoren.titel
WHERE albenautoren.titel IS NULL;
```

und

```
SELECT autoren.name AS "Autoren ohne Titel"
FROM autoren
LEFT JOIN albenautoren ON autoren.id = albenautoren.autor
WHERE albenautoren.titel IS NULL;
```

Diese beiden Abfragen liefern die folgenden Ergebnismengen:

Titel ohne Autor
Gespenster Geschichten
Asterix als Legionär

Autoren ohne Titel

Autoren ohne Titel
Franquin

Wollen wir entsprechend die Autoren anzeigen, denen kein Titel zugeordnet werden kann, so gelingt dies mit einem mehrfachen Left Join:

```
SELECT DISTINCT autoren.name AS autor, reihen.name AS reihe
FROM       autoren
LEFT JOIN albenautoren ON albenautoren.autor = autoren.id
LEFT JOIN alben        ON albenautoren.titel = alben.titel
LEFT JOIN reihen       ON reihen.id = alben.reihe
WHERE reihe IS NULL;
```

So erhalten wir die Ergebnismenge

Autor	Reihe
Franquin	*null*
Morris	*null*

Die Ursache ist für beide Autoren verschieden: Für Franquin ist kein Titel gespeichert, für Morris konnte zwar ein Titel zugeordnet werden, diesem aber keine Reihe. Wollen wir nur die Autoren sehen, denen keine Titel zugeordnet werden können, müssen wir die `WHERE`-Klausel durch

```
WHERE alben.titel IS NULL
```

ersetzen.

11.4 Self Joins

Ein Self Join ist ein Join, der eine Tabelle mit sich selber verknüpft. In der Regel ist ein Self Join ein `INNER JOIN`. Solche Joins sind immer dann nötig, wenn Werte einer Spalte aus verschiedenen Datensätzen verknüpft werden sollen. Da der Join in diesem Fall auf beiden Seiten dieselbe Tabelle verknüpft, müssen die beiden „Instanzen“ (also Ergebnistabellen) durch einen Alias unterschieden werden:

```
SELECT spalte_1, ..., spalte_n
FROM tabelle AS t1
INNER JOIN tabelle AS t2 ON t1.spalte_k = t2.spalte_k
[WHERE filterbedingung]
```

(Oft wird `AS` auch weggelassen.)

Beispiel 11.7. Betrachten wir wieder unsere normalisierte Datenbank von Comic-Alben aus Beispiel 10.1. Eine typische Fragestellung für einen Self Join ist die folgende: In welchen Jahren ist es in einer Reihe jeweils zu einer Preiserhöhung gekommen? Oder etwas anders ausgedrückt: Was sind die Erscheinungsjahre aller Alben, zu denen es in der gleichen Reihe ein günstigeres Vorgängeralbum gab? Ein erster Lösungsansatz ist die folgende Anweisung, mit der wir die Datensätze einer Reihe miteinander verknüpfen und uns alle Spalten der Ergebnistabelle anzeigen lassen, nachdem diejenigen Reihen gefiltert sind, bei denen das Jahr echt größer ist:

```
SELECT *
FROM alben AS a1
INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.jahr < a2.jahr;
```

Sie ergibt die Ergebnismenge:

titel	reihe	band	preis	jahr	titel	reihe	band	preis	jahr
Asterix, der Gallier	2	1	2.80	1968	Der große Graben	2	25	5.00	1980
Asterix, der Gallier	2	1	2.80	1968	Die Trabantenstadt	2	17	3.80	1974
Asterix und Kleopatra	2	2	2.80	1968	Der große Graben	2	25	5.00	1980
Asterix und Kleopatra	2	2	2.80	1968	Die Trabantenstadt	2	17	3.80	1974
Die Trabantenstadt	2	17	3.80	1974	Der große Graben	2	25	5.00	1980
Das Kriminalmuseum	4	1	8.80	1985	Das Meisterwerk	4	2	8.80	1986
Der geheimnisvolle Stern	3	1	null	1972	Tim und der Haifischsee	3	23	null	1973

Links sehen wir die Instanz `a1` unserer Albentabelle, rechts die Instanz `a2`. In `a2` sind die Einträge, die uns interessieren, d. h. `a2` ist unsere Ergebnistabelle. Grenzen wir nun unsere anzuzeigende Spaltenauswahl auf Werte aus `a2` ein und filtern daraus diejenigen Alben, bei denen der Preis gestiegen ist:

```
SELECT a2.jahr, a2.titel, a2.reihe, a2.preis
FROM alben AS a1
INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.jahr < a2.jahr AND a1.preis < a2.preis;
```

Damit erhalten wir:

jahr	titel	reihe	preis
1980	Der große Graben	2	5.00
1974	Die Trabantenstadt	2	3.80
1980	Der große Graben	2	5.00
1974	Die Trabantenstadt	2	3.80
1980	Der große Graben	2	5.00

Die Jahre von Preiserhöhungen innerhalb einer Reihe erhalten wir damit schließlich durch die Anzeige von Reihe, Jahr und Preis sowie die Eliminierung von Dubletten durch `DISTINCT` und geordnet nach Jahren:

```
SELECT DISTINCT a2.reihe, a2.jahr, a2.preis
FROM alben AS a1
INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.jahr < a2.jahr AND a1.preis < a2.preis
ORDER BY a2.jahr;
```

also:

reihe	jahr	preis
2	1974	3.80
2	1980	5.00

Definitiv gab es damit bei der Reihe 2 (Asterix) Preiserhöhungen 1974 und 1980. Wenn wir davon ausgehen, dass die Preise im Laufe der Zeit nie kleiner werden, gab es irgendwann vor 1974 (und nach 1968) eine erste Preiserhöhung, und danach eine (irgendwann vor) 1980. □

Zwischenfrage 11.8. Wie werden im Ergebnis der letzten Abfrage in obigem Beispiel 11.7 statt der Reihen-IDs die Namen der Reihen angezeigt? [2]

[2] Die letzte Abfrage des Beispiels wird um einen weiteren Inner Join mit der Tabelle reihen erweitert:

```
SELECT DISTINCT reihen.name, a2.jahr, a2.preis
FROM alben AS a1
INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
INNER JOIN reihen ON a2.reihe = reihen.id
WHERE a1.jahr < a2.jahr AND a1.preis < a2.preis
ORDER BY a2.jahr
```

12 Views

Kapitelübersicht

12.1 Was sind Views? 100
12.2 Warum Views? 101
12.3 Beispiele für Views 101
12.4 Änderungen von Daten über Views 103

12.1 Was sind Views?

Wie wir wissen, erzeugt ein `SELECT` als Ergebnismenge stets wieder eine Tabelle. Um diese Tabelle allerdings nicht nur temporär auf dem Bildschirm zu sehen, können wir sie auch als eine *View* (oft auch auf Deutsch: *Ansicht*) speichern:

```
CREATE VIEW ansicht AS (
  SELECT ...
);
```

Die Klammern um die SELECT-Anweisung können zwar auch weggelassen werden, jedoch helfen sie dabei, die Struktur zu erkennen. Es ist daher sehr zu empfehlen, sie zu verwenden. Eine View `ansicht` kann mit dem Befehl

```
DROP VIEW IF EXISTS ansicht;
```

auch wieder aus der Datenbank gelöscht werden.

Beispiel 12.1. Um in der Datenbank aus Beispiel 10.1 auf Seite 86 die aktuell gespeicherten Titel der Reihe Asterix zu ermitteln, können wir eine View programmieren:

```
CREATE VIEW asterix AS (
  SELECT band, titel FROM alben
  WHERE reihe = 2
  ORDER BY band
);
```

Mit

```
SELECT * FROM asterix;
```

können wir diese View aufrufen und sie gibt uns die entsprechende Ergebnismenge zurück. □

Intern speichert das RDBMS in ihrem Systemkatalog eine View tatsächlich als einen Block von Anweisungen, also als ein SQL-Unterprogramm. D.h. jeder Aufruf einer View ist eigentlich ein SELECT-Aufruf. Eine View ist daher eine *virtuelle* Tabelle, die Daten werden in der Datenbank nicht extra gespeichert. Views sind also redundanzfrei. Insbesondere sieht man zum Zeitpunkt eines Aufrufs stets eine Ergebnistabelle mit aktuellem Inhalt.

12.2 Warum Views?

Reale Datenbanksysteme sind häufig sehr komplex, was sowohl den Datenbestand angeht, als auch das Datenmodell mit oft sehr vielen verknüpften Tabellen. Für bestimmte Auswertungen sind jedoch meist gar nicht alle Daten und Tabellen interessant, vielmehr werden nur Teile davon oder aggregierte Informationen benötigt. Beispielsweise möchte ein Vertriebsleiter nicht jeden einzelnen Umsatz des Unternehmens sehen, sondern mit einem Blick erkennen, welche Kunden besonders viel, welche eher wenig Umsatz bewirken, oder welche Artikel für jeden Betreuer gut oder schlecht laufen.

In der Regel werden für solche Auswertungen mehr oder weniger komplizierte SQL-Anweisungen benötigt, die die gewünschten Informationen über vielleicht mehrere Joins bereitstellen. Diese können wir als eine View speichern und so den Nutzern der Datenbank zur Verfügung stellen. Aus Sicht der Programmierung handelt es sich hier um eine *Programmierschnittstelle (application programming interface, API).*

12.3 Beispiele für Views

Betrachten wir nun einige Beispiele für Views.

Beispiel 12.2. Verwenden wir wieder unsere normalisierte Datenbank aus Abschnitt 11.3, Seite 96. Um uns Band, Titel, Jahr, Autoren und Reihen in Klartext anzeigen zu lassen, können wir einen Join über die notwendigen Tabellen erzeugen und als View speichern:

```
CREATE VIEW asterix AS (
  SELECT
    alben.band AS band, alben.titel AS titel, alben.jahr AS jahr,
    autoren.name AS autor, reihen.name AS reihe
  FROM       autoren
  INNER JOIN albenautoren ON autoren.id         = albenautoren.autor
  RIGHT JOIN alben        ON albenautoren.titel = alben.titel
  INNER JOIN reihen       ON reihen.id          = alben.reihe
  WHERE reihen.name = 'Asterix'
  ORDER BY band
);
```

Der Right Join bewirkt, dass auch Alben angezeigt werden, denen kein Autor zugeordnet werden kann. Wir können nun mit

```
SELECT * FROM asterix WHERE jahr < 1980
```

diese View wie eine Tabelle abfragen, um uns alle ihre Inhalte mit einem Jahreseintrag kleiner 1980 anzeigen zu lassen:

band	titel	jahr	autor	reihe
1	Asterix, der Gallier	1968	Goscinny	Asterix
1	Asterix, der Gallier	1968	Uderzo	Asterix
2	Asterix und Kleopatra	1968	Goscinny	Asterix
2	Asterix und Kleopatra	1968	Uderzo	Asterix
17	Die Trabantenstadt	1974	Goscinny	Asterix
17	Die Trabantenstadt	1974	Uderzo	Asterix

Ändern wir die zugrunde liegenden Daten, so zeigt der View beim nächsten Aufruf den aktualisierten Stand. Beispielsweise erscheint mit den Anweisungen

```
UPDATE alben SET jahr=1976 WHERE titel='Asterix als Legionär';
SELECT * FROM asterix WHERE jahr < 1980;
```

in der Liste zusätzlich

band	titel	jahr	autor	reihe
	⋮			
10	Asterix als Legionär	1976	*null*	Asterix
	⋮			

da nun das Erscheinungsjahr nicht mehr `NULL` ist. Entsprechend erscheint dann nach den Anweisungen

```
INSERT INTO albenautoren (autor, titel) VALUES (1, 'Asterix als Legionär');
SELECT * FROM asterix WHERE jahr < 1980;
```

als Autor für *Asterix als Legionär* der Autor Uderzo. □

Views sind auch geeignet, um aggregierte Informationen anzeigen zu lassen, wie wir an folgendem Beispiel erkennen können.

Beispiel 12.3. Sei wieder die Datenbank aus Abschnitt 11.3 gegeben. Eine View, die Namen der Reihen, die Anzahl ihrer Titel und deren Durchschnittspreis anzeigt, können wir wie folgt bereitstellen:

```
CREATE VIEW reiheninfo AS (
  SELECT reihen.name AS reihe,
         COUNT(titel) AS alben,
         ROUND(AVG(preis),2) AS "Durchschnittspreis"
  FROM reihen INNER JOIN alben ON reihen.id = alben.reihe
  GROUP BY reihen.name
);
```

Sie ergibt mit

```
SELECT * FROM reiheninfo;
```

die Ergebnismenge

reihe	alben	durchschnittspreis
Asterix	5	3.48
Franka	2	8.80
Gespenster Geschichten	1	1.20
Tim und Struppi	2	NULL

Eine View kann also auch mit Aggregatfunktionen Informationen über die zugrunde liegenden Daten liefern. □

Das Bereitstellen geeigneter Views erlaubt also einen einfachen Zugriff, ohne Kenntnis des darunter liegenden, möglicherweise komplizierten Datenmodells, aber auch ohne Aufweichung der Normalisierung. Ein weiterer Vorteil von Views ist aus technischer Sicht, dass das RDBMS keinen zusätzlichen Aufwand zur Vorbereitung der Abfrage benötigt, denn die View-Abfrage wurde bereits bei der Erstellung vom Precompiler analysiert und effizient optimiert.

Ein Nachteil von Views kann sein, dass bei großen Datenbeständen die durchzuführenden Anweisungen sehr komplex sein können und sehr lange Laufzeiten benötigen. Historisch führten übrigens derartige Probleme dazu, dass Unternehmen ab Mitte der 1990er Jahre für bestimmte hochaggregierte Managementberichte sogenannte OLAP-Systeme (für *Online Analytical Processing*) einführten, die bewusst Datenredundanz in Kauf nahmen, um gerade bei großen Datenbeständen Performanz zu gewinnen.[1]

12.4 Änderungen von Daten über Views

Grundsätzlich sieht der SQL-Standard vor, dass Views mit **INSERT**, **UPDATE** und **DELETE** verändert werden können, so wie jede Tabelle auch. Allerdings können sich dabei verschiedene Probleme ergeben. Denn eine Änderung von Daten einer View ändert gar nicht die View selber, sondern die zugrunde liegenden Tabellen. Beispielsweise würde in unserem obigen Beispiel 12.1 der Befehl

```
INSERT INTO asterix (band, titel) VALUES (101, 'Supermann');
```

ohne Fehlermeldung ausgeführt. Aber eine Abfrage auf die View zeigt, dass dieser Eintrag gar nicht in der View auftaucht. Was ist passiert? In Wirklichkeit hat der Befehl die Tabelle `alben` verändert, und zwar wurde ein neuer Datensatz mit Titel Supermann und Band Nummer 101 eingefügt, alle fehlenden Attributwerte wurden mit **NULL** belegt. Das wird durch den SQL-Befehl aber überhaupt nicht klar!

Zwar verlangte Codd mit seiner 6. Regel, dass eine „einfache“ View stets auch veränderbar sein soll. Es kann jedoch mathematisch bewiesen werden, dass ein für diese Überprüfung notwendiger Algorithmus gar nicht existiert.[2] Kommen Aggregatfunktionen ins Spiel, wie beispielsweise die Albenanzahl in unserer View `reiheninfo` in Beispiel 12.3, so wird diese Tatsache offensichtlich: Die Anweisung

```
UPDATE reiheninfo SET alben = alben - 1
```

würde in einer normalen Tabelle bewirken, dass die Werte des Attributs `alben` um eins verringert werden. Was soll hier aber bei einer virtuellen Tabelle geschehen? In der Reihe Asterix haben wir beispielsweise fünf Alben, d.h. es müsste aus der Tabelle `alben` ein Asterix-Album gelöscht werden. Wie sollte der Algorithmus vorgehen, um den passenden Datensatz auszuwählen? Noch absurder ist die Vorstellung, wie der Algorithmus die folgende Änderung des mittleren Preises der Reihen entscheiden sollte:

```
UPDATE reiheninfo SET "Durchschnittspreis" = "Durhschnittspreis" + 0.5
```

(In vielen RDBMS sind solche Anweisungen deshalb auch gar nicht erst erlaubt!) Es sind diese grundsätzlichen und auch ganz praktischen Probleme, die dazu anraten, Änderungen an Views nicht vorzunehmen, auch wenn das verwendete RDBMS es per SQL erlauben sollte. Views sind für SELECT-Anweisungen dagegen ausgesprochen nützlich und dafür uneingeschränkt zu empfehlen.

[1] `https://de.wikipedia.org/wiki/Online_Analytical_Processing`
[2] Piepmeyer (2011):S. 238f.

13
* Rekursionen

Kapitelübersicht

13.1 Die Grundlage: Common Table Expressions mit `WITH` 104
13.2 Was sind Rekursionen? 105
13.3 Rekursionen mit SQL 106
13.4 Anwendung auf Graphen und Netzwerke 109
13.5 * Die Turing-Vollständigkeit von SQL 110

In der Programmierung ist es üblich, Anweisungen in kleine, leicht verständliche Einheiten zusammenzufassen, also in Subroutinen oder Unterprogramme mit einem eigenen Namen. Je nach Programmiersprache werden sie Methoden, Funktionen, Prozeduren oder „Subs“ genannt. Damit werden ganze Blöcke von Anweisungsfolgen als Algorithmen leicht wiederverwendbar, und zudem die Funktionalitäten eines Softwaresystems strukturiert.

Zwar hat SQL auch Funktionen und Prozeduren, wie wir in den Abschnitten 2.6 und 6.1 gesehen haben. Allerdings ermöglicht SQL im Allgemeinen keine Deklaration von Subroutinen (auch wenn einige Dialekte wie MariaDB oder Azure SQL dies ermöglichen). Der Grundbaustein von SQL sind eben Abfragen, nicht Instruktionen.

Um Abfragen wiederverwendbar zu machen gibt es seit SQL-92 die Views. Sobald ein View mit create angelegt wurde, hat sie einen Namen im Datenbankschema und kann in Abfragen wie eine Tabelle genutzt werden. In SQL:1999 wurde dieses Konzept mit der with-Klausel erweitert, die „Common Table Expressions“.

13.1 Die Grundlage: Common Table Expressions mit `WITH`

Eine *Common Table Expression (CTE)*, auch *Subquery Factoring* genannt, ist auf eine einzelne Anweisung begrenzte View und bekommt mit dem reservierten Wort `WITH` einen Namen. Die Syntax für eine CTE lautet:

```
WITH subroutinenname(x_1, ..., x_n) AS (
     SELECT s_1, ..., s_n FROM ...
)
SELECT x_1, ..., x_n FROM subroutinenname ...;
```

Die Anweisung in den Klammern hinter `AS` ist die eigentliche Definition der Abfrage und bekommt den nach nach `WITH` angegebenden Namen. Die in dieser Abfrage definierten Spalten

`s_1`, . . . , `s_n` können dann unter den im ersten Klammerpaar definierten Spaltennamen `x_1`, . . . , `x_n` in der anschließenden `SELECT`-Abfrage als Spaltenauswahl verwendet werden. Hinter keiner der geschlossenen Klammern darf hier ein Semikolon stehen, da sie als eine einzige SQL-Anweisung verarbeitet wird. Die Parameterliste hinter dem Subroutinennamen muss außerdem genau so viele Variablennamen `x_i` haben wie Spalten `s_i` in der Abfragedefinition. Die Parameterliste darf weggelassen werden, dann werden die Namen (bzw. Aliasnamen) der Spalten der inneren Abfrage verwendet.

Beispiel 13.1. Wir können die View aus Beispiel 12.3 auch als CTE definieren:

```
WITH reiheninfo(reihe, alben, mittlerer_Preis) AS (
  SELECT reihen.name,
         COUNT(titel),
         ROUND(AVG(preis),2)
  FROM reihen INNER JOIN alben ON reihen.id = alben.reihe
  GROUP BY reihen.name
)
SELECT * FROM reiheninfo
```

Allerdings können wir hier für die Parameterliste der CTE keinen beliebigen String als Variablennamen wählen, sondern müssen Variablen mit einem zusammenhängenden Wort (möglichst ohne Sonderzeichen) verwenden. □

Im Gegensatz zu `CREATE VIEW` ist `WITH` also keine eigenständige Anweisung, es muss unmittelbar ein `SELECT` folgen. Es ist also nur eine temporäre View, die nur während der Laufzeit gilt. Entsprechend gibt es grundsätzlich daher keine Probleme einer Änderung oder Löschung der Daten wie bei Views (die allerdings nur von PostgreSQL unterstützt werden). Mehr Informationen zu CTEs siehe

`https://modern-sql.com/de/feature/with`

oder

`https://www.essentialsql.com/introduction-common-table-expressions-ctes/`

13.2 Was sind Rekursionen?

Was geschieht eigentlich, wenn eine Funktion sich selber aufruft? Ist eine derartige Konstruktion überhaupt möglich? In der Tat sind solche Selbstaufrufe in der Informatik durchaus üblich. Sie heißen dort *Rekursionen* und bilden ein mächtiges und sehr wichtiges Konstruktionsprinzip für Algorithmen. Allerdings muss man ihre Funktionsweise beherrschen und bei ihrem Einsatz einige Regeln beachten. Rekursionen entsprechen in den Ingenieurwissenschaften den Rückkopplungen technischer Systeme, zum Beipiel bei einem Bildschirm, der die Aufnahmen der Kamera wiedergibt, die ihn gerade filmt, oder bei einem Mikrofon, das seinen gerade selbst aufgenommenen und per Lautsprecher ausgegebenen Schall wieder aufnimmt. Rekursionen gibt es auch in der Kunst und in der Literatur, dort als *Mise en abyme* (französisch: „in den Abgrund gestellt“) bezeichnet. Hier ist zum Beispiel der Ersteller eines Textes selber Teil dieses Textes, („als mich der Dichter erfand, hatte er ganz was anderes mit mir im Sinn“ [1]), oder ein Bild ist Teil eines Motivs in diesem Bild (Abbildung 13.1). (Zur Mathematik des Mise en abyme in der Kunst siehe auch die Internetquellen [LdS, Le])

Ähnlich wie solche Rückkopplungen können auch Rekursionen zu einem „infiniten Regress“ führen, d.h. zu einer unendlich oft wiederholten Prozessabfolge, also einer „Endlosschleife“. Was sind die Kriterien für eine solche endliche Rekursion?

[1] E.T.A. Hoffmann: *Prinzessin Brambella*, `https://books.google.de/books?id=bACoDAAAQBAJ&pg=PG129`

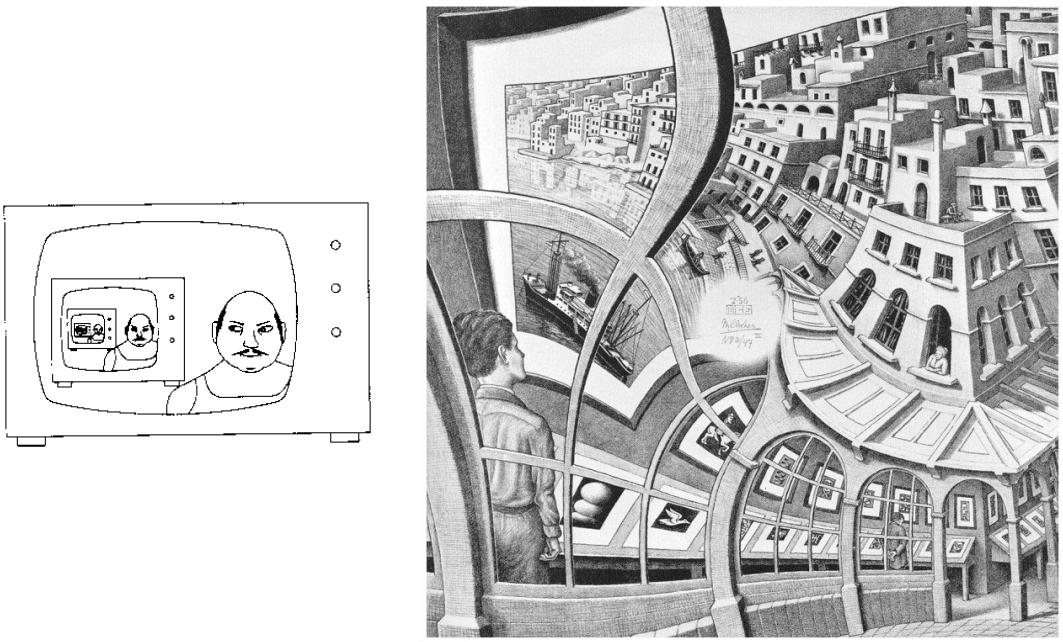


Abbildung 13.1: Rekursionen: Rückkopplung und M. C. Eschers „Kunstdruckgalerie“ (*Prentententoonstelling*) von 1956 als *Mise en abyme*. Quellen: Wirth (1999), Wikimedia.org

Architektur einer terminierenden Rekursion

Die Rekursion ist ein fundamentales Konzept sowohl der Mathematik als auch der Informatik. Sie hat allerdings nur dann einen Sinn, wenn sie als Algorithmus auch „terminiert“, also keine Endlosschleife ausführt. Ähnlich wie bei einer Schleife, also einer Iteration, muss entweder eine Abbruchbedingung des Aufrufprozesses oder umgekehrt eine Rekursionsbedingung zum erneuten Aufruf der Funktion implementiert werden, die dazu führt, dass die Rekursion endet. In jedem Falle dürfen bei einem Funktionsaufruf nicht dieselben Parameterwerte wie für den eigenen Aufruf verwendet werden, d.h. mindestens ein Parameterwert muss sich ändern, so dass die Abbruchbedingung nach endlich vielen Aufrufen auch erreicht wird. Eine solche Bedingung heißt bei Rekursionen *Basisfall*.

13.3 Rekursionen mit SQL

Betrachten wir zunächst zur Erläuterung ein erstes einfaches Beispiel, bevor wir uns der Syntax einer Rekursion in SQL widmen.

Beispiel 13.2. Definieren wir in SQL eine Funktion, die rekursiv von 1 bis 10 hochzählt. Der Basisfall lautet hier einfach „gib 1 aus“, die Rekursionsbedingung „solange $n \leqq 10$“ und der Rekursionsaufruf $n \to n+1$. Also:

```
WITH RECURSIVE hochzaehlen(n) AS (
  SELECT 1                    -- initiale Unterabfrage (Basisfall)
  UNION ALL
  SELECT n+1 FROM hochzaehlen -- Rekursionsaufruf
    WHERE n < 10              -- Rekursionsbedingung
)
SELECT n FROM hochzaehlen;
```

Das Ergebnis ist die Ausgabe 1, 2, . . . , 10. □

Die Syntax einer Rekursion in SQL lautet allgemein:

```
WITH RECURSIVE subroutine(x_1, ..., x_n) AS (
  SELECT startwert_1, ..., startwert_n    -- initiale Unterabfrage (Basisfall)
  UNION ALL
  SELECT V(x_1,..., x_n) FROM subroutine  -- Rekursionsaufruf
    WHERE <bedingung>                     -- Rekursionsbedingung
)
SELECT x_1, ...,x_n FROM subroutine;
```

(In MS Access und Azure SQL muss `RECURSIVE` weggelassen werden.) Hier ist $V(x_1, \ldots, x_n)$ eine Operation mit den n Argumenten der Parameter der rekursiven Subroutine, die wieder n Werte als Ergebnis liefert.[2] Der Basisfall gibt die Startwerte der n Parameter vor. Er wird mit dem Rekursionsschritt durch `UNION ALL` verknüpft, die Rekursionsbedingung ist in der `WHERE`-Klausel festgelegt. (Statt `UNION ALL` kann auch einfach `UNION` verwendet werden, allerdings sollte bei komplexeren Rekursionen aus Laufzeitgründen `ALL` hinzugefügt werden.) Damit die Rekursion *terminiert*, muss die Operation V die Werte so verändern, dass sie nach endlich vielen Aufrufen auch endet.

Eine Rekursion in SQL ist also insgesamt eine CTE, die sich selbst aufruft.

Ein weiteres in der Informatik oft verwendetes Beispiel stammt aus der Mathematik: die Fakultät.

Beispiel 13.3. *(Fakultät $n!$)* Die Fakultätsfunktion einer nichtnegativen ganzen Zahl $n \in \mathbb{N}_0$ wird mit $n!$ bezeichnet (gesprochen: „n Fakultät“) und ist definiert als das Produkt

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1.$$

Beispielweise ist $1! = 1$, $3! = 3 \cdot 2 \cdot 1 = 6$ und $5! = 120$. Per Definition ist $0! = 1$. Eine rekursive Implementierung dieser Funktion in SQL ist das folgende Programm:

```
-- Fakultät rekursiv berechnet
-- nach: https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL
WITH RECURSIVE f(n, aggregat) AS (
  SELECT 0, 1                         -- initiale Unterabfrage (Basisfall)
  UNION ALL
  SELECT n+1, (n+1)*aggregat FROM f   -- rekursive Unterabfrage
    WHERE n < 12                      -- Rekursionsbedingung
)
SELECT n, aggregat AS "n!" FROM f
```

Beispielsweise ergibt ein Aufruf mit der Rekursionsbedingung $n < 3$ den in Abbildung 13.2 dargestellten Aufrufablauf. Man erkennt hierbei, dass die Aufrufe $f(n, \texttt{aggregat})$, ausgehend vom Basisfall ($n = 0$), jeweils im Rekursionsschritt den gerade berechneten Wert für `aggregat` mit n multiplizieren und als neuen Aggregatwert ausgeben. Am Ende wird also tatsächlich der Wert 6 als Endwert ausgegeben. Der für den Datentyp `int` größtmögliche Wert ist $12! = 479\,001\,600$. □

Schließlich ein weiteres „beliebtes“ mathematisches Standardbeispiel, der Euklid’sche Algorithmus zur Bestimmung des größten gemeinsamen Teilers („ggT“) zweier natürlicher Zahlen.

Beispiel 13.4. *(Euklid’scher Algorithmus)* Der Euklid’sche Algorithmus bestimmt den größten gemeinsamen Teiler ggT (m, n) zweier natürlicher Zahlen m, n.[3] Zum Beispiel gilt für die Zahlen

[2] Mathematisch ist V also entweder numerisch eine *vektorwertige* Funktion $V : \mathbb{R}^n \to \mathbb{R}^n$, d.h. ein „Vektorfeld“, oder – im Falle nichtnumerischer Spalten – ein n-Tupel.

[3] vgl. de Vries (2022).

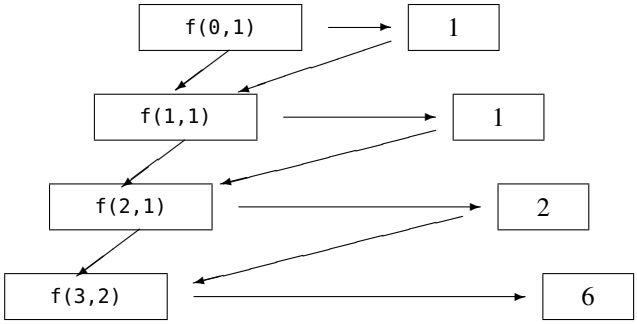


Abbildung 13.2: Aufrufablauf der Fakultätsfunktion. Das erste Argument von f ist hier in der Reihenfolge $n = 1, 2, 3$, das zweite Argument der gerade berechnete Funktionswert `aggregat` $= 1, 1, 2, 6$.

$m = 60$ und $n = 33$, dass $\mathrm{ggT}(60, 33) = 3$. Eine rekursive Implementierung des Euklid'schen Algorithmus in SQL ist das folgende Programm:

```
WITH RECURSIVE euklid (m,n) AS (
  SELECT 60, 33                              -- Basisfall
  UNION ALL
  SELECT n, m % n FROM euklid WHERE n > 0    -- Rekursionsschritt
)
SELECT m AS ggT FROM euklid;
```

Man erkennt hierbei, dass die Aufrufe `euklid`, ausgehend vom Basisfall ($m = 60$, $n = 33$), jeweils im Rekursionsschritt die gerade zuvor berechneten Werte für *vertauschen* und für den neuen Wert des zweiten Arguments n den Term $m\%n$ berechnen (sprich „m modulo n“), also den Rest der Division von n durch m. Das ergibt die Ausgabe:

m
60
33
27
6
3

Am Ende wird also tatsächlich der Wert 3 ausgegeben. Mit dem Programm

```
WITH RECURSIVE euklid (m,n) AS (
  SELECT 60, 33                              -- Basisfall
  UNION ALL
  SELECT n, m % n FROM euklid WHERE n > 0    -- Rekursionsschritt
)
SELECT min(m) AS ggT FROM euklid;
```

wird nur der ggT ausgegeben. □

Insgesamt können wir daraus die folgenden allgemeinen Beobachtungen ableiten. Es muss einen Basisfall geben, für den die Lösung bekannt ist; in Beipiel 13.2 ist $n = 1$, für Beispiel 13.3 ist es $0! = 1$ und für Beispiel 13.4 ist es $(m, n) = (60, 33)$. Für jeden einzelnen Rekursionsschritt muss nach dem inneren `SELECT` definiert sein, wie mit dem jeweils zurückgegebenen Ergebnis verfahren werden soll; in Beisspiel 13.2 ist es die Addition `n+1`, in Beispiel 13.3 die Multiplikation `n*aggregat`, also „funktional“ ausgedrückt $n \cdot f(n-1)$, und für 13.4 ist es das Eintauschen von m durch den alten Wert von n und des neuen Werts für n durch den Term $m\%n$ der alten Werte.

Basisfall und Rekursionsschritt werden stets mit `UNION ALL` verknüpft. Weitere Informationen zu Rekursionen mit SQL:

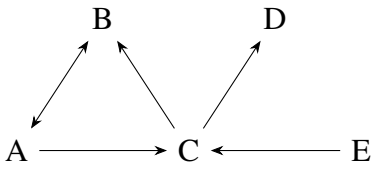
`https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL`

`https://mariadb.com/kb/en/library/recursive-common-table-expressions-overview/`

13.4 Anwendung auf Graphen und Netzwerke

13.4.1 Tiefensuche

Ein wichtiges Anwendungsfeld von Rekursionen sind Graphen und Netzwerke. Mit Beispiel 9.14 auf Seite 75 haben wir im Zusammenhang mit rekursiven Beziehungen Graphen bereits kennengelernt. Ein wichtiger Algorithmus zum systematischen Auffinden aller Knoten von einem bestimmten Startknoten ist die *Tiefensuche (depth first search)*. Gegeben seien die vier Städte New York, Boston, Washington, und Raleigh mit den folgenden Verkehrsverbindungen:



Eine Implementierung der Tiefensuche mit nur einer Tabelle lautet wie folgt:

```
-- Tabellenschemata: ---
CREATE TABLE knoten (
  name varchar(50),
  PRIMARY KEY (name)
);
CREATE TABLE kanten (
  id SERIAL,
  von  varchar(50),
  nach varchar(50),
  PRIMARY KEY (id),
  FOREIGN KEY (von)  REFERENCES knoten(name) ON DELETE CASCADE,
  FOREIGN KEY (nach) REFERENCES knoten(name) ON DELETE CASCADE
);
-- Knoten einfügen:
INSERT INTO knoten(name) VALUES ('A'), ('B'), ('C'), ('D'), ('E');
-- Kanten A <-> B, A -> C, C -> B, C -> D, E -> C einfügen:
INSERT INTO kanten (von, nach) VALUES
  ('A','B'), ('B','A'), ('A', 'C'), ('C', 'B'), ('C','D'), ('E', 'C');
```

mit der Rekursion:

```
-- Tiefensuche (Depth First Search), nach:
-- https://mariadb.com/kb/en/library/recursive-common-table-expressions-overview/
WITH RECURSIVE suche (ziel, weg) AS (
  SELECT DISTINCT von, von FROM kanten WHERE von = 'A' -- Basisfall: weg = ziel
  UNION ALL
  SELECT kanten.nach, CONCAT(suche.weg, ' → ', kanten.nach)
    FROM kanten INNER JOIN suche ON suche.ziel = kanten.von
```

```
    WHERE POSITION(kanten.nach IN suche.weg) = 0 -- Rekursionsbedingung:
                                                 -- ziel ist noch nicht in Weg
)
SELECT weg FROM suche;
```

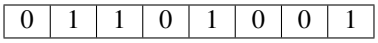
Die Ergebnismenge lautet dann:

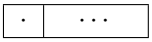
weg
A
A → B
A → C
A → C → B
A → C → D

Da der Datentyp der beiden Spalten `ziel` und `weg` der CTE `suche` anhand des zuerst auftretenden Basisfalls durch die Spalte `von` der zugrundeliegenden Tabelle `kanten` abgeleitet wird, in der Spalte `weg` aber ein längerer String gespeichert werden muss, muss der String ausreichend groß eingerichtet werden.

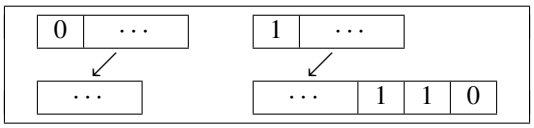
13.5 * Die Turing-Vollständigkeit von SQL

Grundsätzlich ist SQL Turing-vollständig, d.h. jede überhaupt programmierbare Funktion kann auch in SQL implementiert werden.[4] Allerdings scheint zur Zeit (Ende 2025) die Datenbank PostgreSQL die einzige zu sein, die dies ermöglicht. Der zum Entwicklerteam von PostgreSQL gehörende Informatiker Andrew Gierth[5] bewies dies 2009 mit einem SQL-Programm, das einen speziellen endlichen Automaten simuliert, ein sogenanntes *Cyclic Tag System*,[6] von dem bekannt ist, dass es Turing-vollständig ist.[7] Ein solcher Automat besteht aus „Zuständen“ und einer endlichen Liste von „Produktionsregeln“. Ein Zustand wird durch einen String von Symbolen, beispielsweise von ‘0’ und ‘1’ dargestellt,



und bildet ein sogenanntes „Wort“. Eine Produktionsregel legt hier bei einem gegebenen Zustand des Automaten fest, welcher String im nächsten Iterationsschritt an das aktuelle Wort angehängt wird, wenn das erste Bit des Wortes eine ‘1’ ist; bei einer ‘0’ wird nichts angehängt. In beiden Fällen wird das erste Bit dann gelöscht. Für die Produktionregel ‘110’ beispielsweise wird entsprechend der String ‘110’ an das aktuelle Wort 

angehängt:



Nachdem die Produktionsregel angewandt wurde, wird für den neuen Zustand die nächste Produktionsregel aus der Liste verwendet; ist das Ende der Liste erreicht, wird wieder die erste Produktionregel verwendet, usw. Ein *Cyclic Tag System* ist beendet („terminiert“), wenn das Wort leer ist, also kein Symbol mehr enthält.

Die Produktionsregeln eines solchen Systems können wir als Tabelle

[4] Zum Begriff der Turing-Vollständigkeit siehe z.B. https://de.wikipedia.org/wiki/Turing-Vollständigkeit

[5] vgl. `https://www.postgresql.org/community/contributors/`

[6] A. Gierth, *http://assets.en.oreilly.com/1/event/27/High Performance SQL with PostgreSQL Presentation.pdf*, vgl. *https://wiki.postgresql.org/wiki/Cyclic_Tag_System*, *https://esolangs.org/wiki/Cyclic_tag_system* oder *https://www.wolframscience.com/nks/p95/*

[7] Wolfram (2002):S. 93ff, 677ff.

```
p(num, pos, tag)
```

mit drei Spalten speichern, deren Bedeutung wie folgt ist: `num` ist die Nummer der Produktionsregel, `pos` ist der Index des im Automaten gerade registrierten Bits, `tag` ist das zu setzende Bit. Gierth verwendete die drei Produktionsregeln

```
CREATE TABLE p (num smallint, pos smallint, tag text);
INSERT INTO p (num, pos, tag) VALUES
  (0,0,'1'),(0,1,'1'),(0,2,'0'),          -- 1. Produktionsregel: 110
  (1,0,'0'),(1,1,'1'),                    -- 2. Produktionsregel: 01
  (2,0,'0'),(2,1,'0'),(2,2,'0'),(2,3,'0'); -- 3. Produktionsregel: 0000
```

In jedem Iterationsschritt wird Bit 0 entfernt, die übrigen Bits um eine Stelle nach links verschoben und genau dann, wenn Bit 0 den Wert 1 hatte, wird der Inhalt der aktuellen Produktionsregel an das Ende des Strings gesetzt. Für den nächsten Iterationsschritt wird dann die zyklisch nächste Regel verwendet, d.h. nach dem Zyklus

$$\texttt{num} = 0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 0 \to \dots.$$

Die Zustände des Automaten werden durch das folgende SQL-Programm implementiert:

```
WITH RECURSIVE w(iter,pos,tag) AS (        -- Tabelle der Wörter
  SELECT 0,0,'1'
  UNION ALL
  SELECT w.iter + 1,
    CASE
      WHEN w.pos = 0
      THEN p.pos + max(w.pos) OVER (PARTITION BY w.iter)
      ELSE w.pos - 1
    END,
    CASE
      WHEN w.pos = 0
      THEN p.tag
      ELSE w.tag
  END
  FROM w LEFT JOIN p ON (w.pos = 0 AND w.tag = '1' AND p.num = w.iter % 3)
  WHERE (w.pos > 0 OR p.num IS NOT NULL)
)
SELECT iter, pos, tag FROM w ORDER BY iter, pos;
```

Die Ausgabe in PostgreSQL lautet dann, ergänzt um das gesamte Bitwort der Tabelle `w`, wie in Tabelle 13.1 dargestellt. An den in der Spalte *Wort* zusammengefassten Bitwörtern erkennen wir, dass wir tatsächlich ein *Cyclic Tag System* programmiert haben. Zusammengefasst erhalten wir damit das folgende theoretische Resultat.

Satz 13.5. *Mit der rekursiven* `WITH`*-Klausel und Aggregationen über Partitionen ist SQL eine Turing-vollständige Programmiersprache.*

Beispiel 13.6. Stephen Wolfram gibt in einem Beispiel[8] das Cyclic Tag System mit den zwei Produktionsregeln

$$11, \quad 10$$

an. In SQL umgesetzt lautet das System mit dem Start-Tag 1:

[8] Wolfram (2002):S. 95.

iter	pos	tag	Wort
0	0	1	1
1	0	1	
1	1	1	
1	2	0	110
2	0	1	
2	1	0	
2	2	0	
2	3	1	1001
3	0	0	
3	1	0	
3	2	1	
3	3	0	
3	4	0	
3	5	0	
3	6	0	0010000
4	0	0	
4	1	1	
4	2	0	
4	3	0	
4	4	0	
4	5	0	010000

iter	pos	tag	Wort
5	0	1	
5	1	0	
5	2	0	
5	3	0	
5	4	0	1000
6	0	0	
6	1	0	
6	2	0	
6	3	0	
6	4	0	
6	5	0	
6	6	0	
6	7	0	00000000
7	0	0	
7	1	0	
7	2	0	
7	3	0	
7	4	0	
7	5	0	
7	6	0	0000000

iter	pos	tag	Wort
8	0	0	
8	1	0	
8	2	0	
8	3	0	
8	4	0	
8	5	0	000000
9	0	0	
9	1	0	
9	2	0	
9	3	0	
9	4	0	00000
10	0	0	
10	1	0	
10	2	0	
10	3	0	0000
11	0	0	
11	1	0	
11	2	0	000
12	0	0	
12	1	0	00
13	0	0	0

Tabelle 13.1: Gierths Produktionsregeln

```
CREATE TABLE p (num smallint, pos smallint, tag text);
INSERT INTO p (num, pos, tag) VALUES
  (0,0,'1'),(0,1,'1'),              -- 1. Produktionsregel: 11
  (1,0,'1'),(1,1,'0');              -- 2. Produktionsregel: 10
--
WITH RECURSIVE w(iter,pos,tag) AS (  -- Tabelle der Wörter
  SELECT 0,0,'1'
  UNION ALL
  SELECT w.iter + 1,
    CASE
      WHEN w.pos = 0
      THEN p.pos + max(w.pos) OVER (PARTITION BY w.iter)
      ELSE w.pos - 1
    END,
    CASE
      WHEN w.pos = 0
      THEN p.tag
      ELSE w.tag
  END
  FROM w LEFT JOIN p ON (w.pos = 0 AND w.tag = '1' AND p.num = w.iter % 2)
  WHERE (w.pos > 0 OR p.num IS NOT NULL)
  AND w.iter < 20 -- harter Abbruch bei Iteration 20 ...
)
SELECT STRING_AGG(tag,'' ORDER BY pos) AS word FROM w GROUP BY iter;
```

Hierbei wird die Aggregatfunktion `STRING_AGG` verwendet, die die Tags wegen `GROUP BY iter` über die Iterationen gruppiert und nach Positionsreihenfolge sortiert zu jeweils einem String konkateniert. Beachten wir außerdem die letzte Bedingung der `ON`-Klausel

```
p.num = w.iter % 2
```

Da wir nun nämlich nur zwei Produktionsregeln haben, muss hier entsprechend modulo 2 gerechnet werden, nicht modulo 3 wie im obigen Beispiel mit drei Produktionen. Am Ende des `WITH`-Ausdrucks wird die Iteration bei Nummer 20 abgebrochen; ansonsten würde die Rekursion in diesem Falle endlos weiterlaufen. Die Ausgabe ergibt dann die Wortfolge:

```
1
11
110
1011
01110
1110
11010
101011
0101110
101110
0111010
111010
11011010
10101011
010101110
10101110
010111010
10111010
011101010
11101010
110101010
```

Das entspricht der grafischen Ausgabe in Wolfram (2002:S. 95), siehe auch `https://www.wolframscience.com/nks/p95/`. □

Satz 13.5 impliziert, wie oben bereits erwähnt, dass jede überhaupt programmierbare Funktion auch in SQL implementiert werden kann. Diese theoretische Implementierbarkeit sagt natürlich noch nichts darüber aus, wie einfach oder wie effizient eine konkrete Implementierung aussieht. So ist mir beispielsweise nicht bekannt, wie eine sogenannte μ-rekursive Funktion wie die Ackermann-Funktion in SQL aussehen könnte. Deren übliche Definition verwendet eine verschachtelte Rekursion, die in imperativen Programmiersprachen leicht umsetzbar ist, in SQL als deklarativer Programmiersprache mir jedoch bisher nicht gelang. Vielleicht haben Sie, verehrte Leserin, verehrter Leser, ja eine Idee. Lassen Sie es mich gerne wissen!

A Anhang

Kapitelübersicht

A.1 Übersicht der SQL-Anweisungen 114
A.2 Die 12 Codd'schen Regeln 116
A.3 Relationale Vollständigkeit 120

A.1 Übersicht der SQL-Anweisungen

Zwei Anweisungen werden in SQL mit einem Semikolon getrennt, nach der letzte auszuführende Anweisung darf es weggelassen werden. Der Befehlssatz von SQL ist in drei Teilbereiche gegliedert, die DDL, die DCL und die DML.

Data Definition Language (DDL) Mit der Data Definition Language können die Datenbank und die Struktur ihrer Tabellen angelegt und verwaltet werden. Konkret können Datenbanken, Tabellen und User angelegt, verändert und gelöscht werden. Dazu stehen die Ausdrücke `CREATE`, `ALTER` und `DROP` zur Verfügung, die kombiniert werden können mit den Ausdrücken `DATABASE`, `TABLE` und `USER`.

$$\left\{\begin{array}{c}\texttt{CREATE}\\ \texttt{ALTER}\\ \texttt{DROP}\end{array}\right\} + \left\{\begin{array}{c}\texttt{DATABASE}\\ \texttt{TABLE}\\ \texttt{USER}\end{array}\right\} + \textit{Name} + \textit{[Zusatzoptionen]};\qquad \text{(A.1)}$$

Beispielsweise wird mit dem Befehl

```
CREATE DATABASE xyz;
```

eine Datenbank mit dem Namen `xyz` erzeugt. Grundsätzlich können dann mit `ALTER DATABASE xyz` Änderungen der Datenbank vorgenommen werden, und mit `DROP DATABASE xyz` wird die Datenbank vollständig gelöscht. Allerdings sind diese Befehle in den verschiedenen Datenbanksystemen nicht einheitlich – oder manchmal gar nicht – definiert. In MS Access, OpenOffice oder LibreOffice Base beispielsweise ist das Wort `Database` noch nicht einmal reserviert, d.h. diese Befehle sind dort nicht verfügbar. Eine Datenbank wird in diesen Systemen über die graphische Benutzeroberfläche verwaltet.

Der Relationentyp wird in SQL definiert, indem die Spaltennamen mit ihrem Datentyp aufgeführt werden:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
   spalte_2 datentyp_2,
     ...
   spalte_n datentyp_n,
   PRIMARY KEY(spalte_x, ..., spalte_y)
);
```

Hier bestimmt die in den Klammern nach `PRIMARY KEY` aufgeführte Spaltenauswahl diejenigen Spalten der Tabelle, die ihren sogenannten Primärschlüssel bilden. Die Zeile kann auch weggelassen werden, dann hat die Tabelle keinen Primärschlüssel. Auf diese Weise wird also der Relationentyp

```
tabelle (datentyp_1, ..., datentyp_n)
```

indirekt definiert. In den meisten Datenbanksystemen kann man den Zeichensatz festlegen, in dem die Werte in Spalten mit Textformaten gespeichert werden, am besten mit UTF-8:

```
CREATE TABLE tabelle (
   spalte_1 datentyp_1,
     ...
) DEFAULT CHARSET=utf8;
```

Die Struktur einer Tabelle wird durch `ALTER` geändert. Zum Beispiel wird mit der Anweisung

```
ALTER TABLE tabelle ADD neue_spalte datentyp;
```

eine neue Spalte in die Tabelle `tabelle` eingefügt. Mit

```
ALTER TABLE tabelle DROP spalte;
```

Die gesamte Tabelle wird mit dem Befehl

```
DROP TABLE tabelle;
```

gelöscht. Für weitere Informationen zur DDL sei auf die Webseite [SQL1 → DDL] verwiesen.

Data Manipulation Language (DML) Mit der *Data Manipulation Language (DML)* werden Anweisungen für die Verwaltung der Daten selbst bereitgestellt, also Datensätze anlegen, lesen, aktualisieren und löschen. Diese vier zentralen Funktionen auf Daten werden mit dem Kürzel CRUD (für *create*, *read*, *update* und *delete*) zusammengefasst.

- Mit dem Ausdruck `INSERT` werden Datensätze angelegt und in eine Tabelle eingefügt. Die Syntax in SQL lautet:

  ```
  INSERT INTO tabelle (spalte_1, ..., spalte_n) VALUES
    (wert_11,  ...,  wert_1n),
        ...
    (wert_m1,  ...,  wert_mn);
  ```

  Damit werden m Datensätze mit ihren Werten für die n Spalten eingefügt. Wichtig ist, dass die im ersten Klammerpaar angegebene Reihenfolge der Spalten der Reihenfolge der Werte entsprechen muss. Bei manchen RDBMS kann man pro INSERT nur einen Datensatz einfügen, z.B. bei Oracle. Will man alle Spalten der Tabelle füllen, kann man die Spaltennamen in den Klammern vor `VALUES` auch weglassen. Dazu muss natürlich die genaue Anzahl und Anordnung der Spalten in dem Tabellenschema bekannt sein.

- Daten können in SQL mit `SELECT` gelesen werden. Die Syntax lautet:

```
SELECT [DISTINCT] spaltenliste
FROM    tabellenliste
[WHERE    bedingungsliste]
[GROUP BY spaltenliste   ]
[HAVING   bedingungsliste]
[UNION    weiterer SELECT-Ausdruck]
[ORDER BY spaltenliste   ]
```

  Die Ausdrücke in eckigen Klammern sind optional und können daher entfallen. Statt der Spaltenliste kann einfach ein Sternchen $*$ stehen. Die Ausdrücke in den Zeilen nach dem `FROM`-Ausdruck heißen *Klauseln (clauses)*. Die Reihenfolge der Klauseln ist fest im SQL-Standard vorgegeben. Die wichtigsten Teile werden in den folgenden Kapiteln erläutert. Das Resultat einer Selektion ist eine Menge von Datensätzen mit den Attributen der Spaltenliste, also wieder eine Tabelle. Sie wird *Ergebnismenge (result set)* genannt. Vgl. dazu auch Bemerkung 3.2 auf Seite 19.

- Einträge eines bestehenden Datensatzes werden mit dem Befehl `UPDATE` geändert. Mit der Syntax

```
UPDATE tabelle SET
  spalte_1 = wert_1,
    ...
  spalte_n = wert_n
[WHERE Bedingungsliste]
```

  werden in der Tabelle `tabelle` die Werte der Spalten 1 bis n aller Datensätze geändert, die der WHERE-Klausel genügen. Die WHERE-Klausel kann auch weggelassen werden, allerdings werden dann die Werte *aller* Datensätze aktualisiert – das ist wahrscheinlich in den seltensten Fällen so gewollt.

- Der SQL-Befehl `DELETE` ist für das Löschen ganzer Datensätze zuständig. Die Syntax lautet:

```
DELETE FROM tabelle [WHERE <Bedingungsliste>];
```

  Die Bedingungsliste gibt an, welche Voraussetzungen ein Datensatz der Tabelle `tabelle` zu erfüllen hat, um gelöscht zu werden. Hierbei kann zwar die WHERE-Klausel weggelassen werden, allerdings werden dann entsprechend *alle* Datensätze der Tabelle gelöscht. (Eine Wiederherstellung einmal gelöschter Datensätze ist Datenbank-intern nicht möglich!)

Data Control Language (DCL) Die *Data Control Language (DCL)* ist der Sprachteil von SQL, der für die Verwaltung von Zugriffsrechten auf die Datenbank zuständig ist. Sie besteht im Wesentlichen aus den zwei Befehlen `GRANT` für die Vergabe von Zugriffsrechten und `REVOKE` für deren Entzug. Für weitere Details siehe [SQL1 → DCL].

A.2 Die 12 Codd'schen Regeln

E.F. Codd und seine Mitarbeiter benötigten von ihren ersten theoretischen Ansätzen bis zur Fromulierung der inzwischen berühmt gewordenen zwölf Regeln mehr als 15 Jahre. Im Oktober 1985 veröffentlichte E.F. Codd zum ersten Mal in der „Computerworld“ seine Regeln, die sein

Mitarbeiter C.J. Date knapp ein Jahr später um weitere zwölf Regeln für virtuelle RDBMS *(relational data base management system)* erweiterte.

Die seither entstandenen DBMS *(data base management systems)* nähern sich dem relationalen Modell jedoch nur sehr langsam. So bieten nahezu alle Hersteller als Abfragesprache SQL mit zahlreichen Abweichungen untereinander. Es wird fast immer nur ein Teil der relationalen Algebra unterstützt.

Um sich hier richtig zu orientieren und einschätzen zu können, in welchem Grad ein DBMS relational ist oder nicht, sind die zwölf Regeln ein sehr nützliches und auch einfaches Hilfsmittel. Deutlich aufwendiger wäre es, zu unterscheiden, inwieweit die einzelnen DBMS das vollständige relationale Modell (das hier auch nur angerissen werden konnte) unterstützen.

Streng genommen sind es sogar dreizehn Regeln, doch E.F. Codd trennte die erste ab, weil diese fundamental ist und alle anderen auf diesem Äxiom“ aufbauen.

Axiom: Jedes relationale Datenbanksystem (RDBMS) muss in der Lage sein, die gesamte Datenbank mit seinen relationalen Fähigkeiten, wie dies im relationalen Modell spezifiziert ist, selbst zu verwalten. Dies muss auch dann möglich sein, wenn das DBMS zusätzlich nichtrelationale Fähigkeiten unterstützt. Außerdem muss die Sprache, mit der auf die Daten zugegriffen wird, auf relationalem Niveau sein, d.h. jedes Select, Update oder Delete (eben SQL als Zufriffssprache) muss den Zugriff auf mehrere Datensätze (Tupel) gleichzeitig unterstützen. Die Auswahl eines oder keines Datensatzes ist in diesem Zusammenhang als Spezialfall der Umschreibung „mehrerer Datensätze“ zu betrachten.

Regel 1: *In einer relationalen Datenbank werden alle Informationen ausschließlich auf einer logischen Ebene und nur auf genau eine Art und Weise durch Werte in Relationen (Tabellen) dargestellt.* Damit spielt es keine Rolle, wie das RDBMS die Daten physikalisch auf Server und Festplatten verteilt. Es wird ausdrücklich untersagt, auf die logischen, dem Benutzer zugänglichen Ebene Mechanismen wie Pointer, hardwarenahe Adressen, Sektornummern u.ä. zu benutzen.

Neben den Anwenderdaten, den eigentlichen Nutzinformationen, müssen auch die Namen der Relationen, Spalten und Domänen einer Datenbank in Form von Zeichenketten in Relationen dargestellt werden. Soche Relationen sind normalerweise im sogenannten Systemkatalog abgelegt. Der Systemkatalog wird somit praktisch ein Teil der Datenbank, er ist dynamisch veränderbar und während jeder Datenbanksitzung aktiv.

Diese erste Regel wird oft auch als *Informationsregel* bezeichnet, denn mit ihr ist die Datenbank-Administration (DBA) in der Lage, die Integrität der Datenbank zu erhalten, ja sie vereinfacht seine Arbeit erheblich. Mit Hilfe des Systemkatalogs ist es jederzeit möglich, Zustände der Datenbank abzufragen, Aussagen über den Umfang der Tabellen, deren Indizierung und Struktur zu treffen.

Regel 2: *Jedes einzelne Element in einer relationalen Datenbank ist immer durch eine logische Kombination aus dem Namen der Relation, einem Primärschlüsselwert und dem Spaltennamen erreichbar.*

Das Grundkonzept des relationalen Modells ist mengenorientiert. Befehle wie Skip (xBase) zum Positionieren auf den nächsten Satz existieren nicht. Daher ist es wichtig, dass das RDBMS mindestens einen garantierten Zugriffsmechanismus auf einen einzelnen Wert der Datenbank implementiert. Auch hier gilt: Der Zugriffsmechanismus findet auf der logischen Ebene statt, physikalische Addressierungen sind nicht erlaubt. Fundamentale Bedeutung hat auch die entsprechende Umsetzung des relationalen Modells, nach der jede Relation einen eindeutigen Primärschlüssel ohne NULL-Werte besitzen muss.

Regel 3: *Das relationale System muss unabhängig von Datentyp Indikatoren unterstützen, die auf der logischen Ebene fehlende Informationen ersetzen.* Sie müssen sich außerdem von solchen Konstrukten wie leere Zeichenketten für Stringfelder, der Ziffer Null für numerische Felder oder des Datums 01.01.0000 unterscheiden. Das relationale Datenbanksystem hat darüberhinaus Funktionen bereitzustellen, die die Manipulation solcher Indikatoren ermöglicht. Auch diese Funktionen wiederum sind unabhängig von Datentyp.

In der Praxis wird dieser Forderung durch die Unterstützung des sogenannten NULL-Wertes entsprochen. Die ursprünglich und in zahlreichen Datenbanksystemen heute noch üblichen speziellen Werte sind im relationalen Modell ungeeignet, da der Benutzer für jede Spalte oder Domäne eine dem Datentyp angepasste Technik anwenden müsste.

Ein kurzes Beispiel soll dies verdeutlichen. In einem Formular wird ein Antragsteller nach der Zugehörigkeit zu einer bestimmten Versicherungsorganisation gefragt. Falls er dort mit „ja“ quittiert, muss er in einem weiteren Feld das Beitrittsdatum angeben. Doch was macht die Erfassungsroutine, falls der Antragsteller mit „nein“ quittiert und es demzufolge kein Beitrittsdatum gibt? Wird an dieser Stelle der eben erwähnte spezielle Wert 01.01.0000 gespeichert, müssen alle Operationen der Datenbank, die diese Spalte benutzen, diesen Wert speziell abarbeiten. Dies kann bei großen oder verteilten Systemen, bei zahlreichen Anwendern und Anwendungsprogrammen schwierig sein; deshalb wird der NULL-Wert gespeichert.

Um die Integrität der Datenbank zu gewährleisten, muss es allerdings möglich sein, Felder zwingend als „NOT NULL“ zu definieren. Primärschlüsselfelder müssen z.B. immer einen Wert enthalten.

Regel 4: *Da die Beschreibung der Datenbank (der Systemkatalog) auf einer logischen Ebene erfolgt, und zwar in genau der gleichen Art und Weise wie die Darstellung der Nutzdaten, muss es für autorisierte Benutzer möglich sein, mit der gleichen Zugriffssprache auf diese Systemdaten zuzugreifen wie auf die normalen Daten.*

Dies ist einMerkmal, das nicht-relationale Datenbanken normalerweise nicht bieten. Jeder Anwender, egal ob End-User, Programmierer oder Administrator, muss nur eine Sprache lernen. Autorisierte Benutzer können den Katalog leicht erweitern. Mit dieser Regel wird die konzequente Umsetzung eines einheitlichen Datenmodells fortgesetzt. Datenbanken mit Netzstruktur nach den CODASYL-Vorschlägen, aber auch hierarchische Datenbanken wie IMS[1], die zur Bauteilverwaltung der Saturn V Rakete im Rahmen des Apolloprogramms der NASA in den 1960er Jahren eingesetzt wurde, setzen die Kenntnis von zwei verschiedenen Modellen voraus.

Regel 5: *Um auf die gespeicherten Datenbestände zugreifen zu können, muss das DBMS entsprechende Programmierschnittstellen unterstützen, damit eine geeignete Programmiersprache die Daten ansprechen kann. und folgende Bedingungen erfüllt: (i) Ihre einzelnen Statements müssen aus Zeichenketten mit einer wohldefinierten Syntax bestehen. (ii) Die Sprache muss umfassend sein und Kommandos zur Daten- und Viewdefinition, zur Manipulation der Daten, zur Autorisierung des Zugriffs, zur Sicherung der Integrität und zum Verpacken in Pakete (Transaktionen) enthalten.*

Weder die netzförmig strukturierten Systeme noch die hierarchischen Systeme kennen eine solche umfassende Srache. Alle Zugriffe erfolgen über 3GL-Sprachen wie Cobol. SQL dagegen ist eine umfassende Sprache, die all diese Bedingungen erfüllt.

Regel 6: *Das RDBMS enthält einen Algorithmus, mit dem zum Definitionszeitpunkt einer „einfachen“ View festgelegt werden kann, ob in dieser View Datensätze eingefügt oder ge-*

[1] Piepmeyer (2011):S. 17ff.

löscht und welche Spalten verändert werden dürfen. Die Definition dieser Eigenschaft muss im Systemkatalog gespeichert werden.

Diese Regel ist jedoch nur schwer umzusetzen, im allgemeinen Fall ist ein solcher Algorithmus sogar unmöglich[2]. Die Datenbank muss dafür Mechanismen bereitstellen, die dafür sorgen, dass die in der View erfolgten Änderungen an den entsprechenden Basis-Tabellen vollzogen werden müssen. Das Ganze muss außerdem noch tabellenunabhängig sein, d.h. die Änderungen sind einzig und allein von der View-Definition abhängig. Sie können somit zum Zeitpunkt ihrer Definition auch entschieden werden. Die Änderungen sind nicht von der Definition und den Daten der Basis-Tabellen abhängig.

Codd empfiehlt hier View-Definitionen, die ausschließlich durch Basis-Tabellen ausgedrückt werden oder deren Definition man solange erweitert, bis sie durch solche Basisrelationen ausgedrückt werden können. Außerdem sollten die Views „einfach“ sein, d.h. sie besitzen:

- im Verhältnis zu den im Select oder Update genannten Tabellen doppelt soviele Projektoren.
- im Verhältnis zu den im Select oder Update genannten Tabellen doppelt soviele algebraische Selects,
- maximal vier Operatoren der Klasse Union, Outer-Union, Differenz und Intersektion und maximal vier Operatoren der Klasse Join und Relationale Division.

Diese Regel bezieht sich ausdrücklich auf diese „einfachen“ Views, da ansonsten auf der logischen Ebene nicht entschieden werden kann, ob Änderungen in der View möglich sind. Das RDBMS sollte den Benutzer bei der Definition von Views auf solche einfachen und damit änderbaren Views einschränken.

Regel 7: *Eine Basis- oder Ergebnistabelle kann man in einem RDBMS wie einen Operanden benutzen. Dies ist nicht nur beim Selektieren von Daten möglich, sondern gilt auch für die Kommandos Insert, Update und Delete.* Ziel dieser Regel ist es, dem System einen größeren Spielraum bei der Optimierung seiner Laufzeitaktionen zu geben. Beispielsweise kann es den Zugriffspfad auf die Daten selbst festlegen.

Regel 8: *Anwendungsprogramme und deren Oberfläche bleiben für den Benutzer logisch unverändert, auch wenn Veränderungen an der Speicherstruktur oder der Zugriffsmethode vorgenommen werden.* Das bedeutet nichts anderes, als dass das RDBMS die hardwarenahen Momente (wie Speicherausstattung, CPU, schnelles Netzwerk usw.) ganz klar vom logischen Aufbau der Datenbank abkoppelt. Dies hat zur Folge, dass beispielsweise ein Tuning der Datenbank vorgenommen werden kann oder die Datenbank auf ein größeres leistungsfähigeres System portiert wird, ohne dass ein Anwendungsprogramm verändert werden muss. Damit ist auch eine strikte Trennung der Aufgaben des Server-Systems von denen des Client-Systems möglich. Ein Anwendungsprogramm darf auf die Datenbank niemals mit expliziten Indizes zugreifen. Die Benutzung und Verwaltung der Indizes ist Sache des RDBMS.

Regel 9: Manche Änderungen an der Struktur der Datenbank können vor den Anwedungsprogrammen verborgen werden, indem einfach eine View definiert wird, die der alten Struktur entspricht. Die 9. Regel definiert dies so:

Anwendungsprogramme bleiben logisch unbeeinträchtigt von informationserhaltenden Veränderungen an den Basisrelationen, wenn es theoretisch möglich ist, diese Unabhängigkeit zu gewährleisten. So kann es beispielsweise notwendig werden, dass eine sehr große Tabelle physisch

[2] Piepmeyer (2011):S. 238.

auf zwei oder gar mehr Festplatten verteilt werden muss. Das kann entweder zeilenweise über den Zeileninhalt oder spaltenweise über den Spaltennamen vorgenommen werden. Damit bleiben die Primärschlüssel in beiden Teilrelationen erhalten. Eine entsprechende View-Definition kann diese Veränderung verbergen. Auch das Gegenteil, die Zusammenführung zweier Relationen in eine Tabelle ist denkbar. Dieser Vorgang wird auch *verlustfreier Join* genannt.

Mit der 9. Regel kann das logische Datenbankdesign regelmäßig verändert werden, um etwa das Performance-Verhalten zu verbessern, ohne dass dies Auswirkungen auf ein Anwenderprogramm hat. Die 8. Und 9. Regel zusammen lassen auch ein gewisses Maß an Fehlertoleranz beim Datenbankdesign zu.

Regel 10: *Alle Integritätsbedingungen, die spezifisch für eine Datenbank gelten, müssen mit Hilfe der relationalen Datenbankbeschreibungssprache definierbar sein. Außerdem müssen sie im Systemkatalog abgelegt werden. Ihre Definition in den Anwendungprogrammen ist unzulässig.*

Damit ist eine Verwaltung der Integritätsregeln an zentraler Stelle möglich, Informationen über nicht ausreichend zu identifizierende Objekte werden nie in relationalen Datenbanken abgelegt. Dafür sind die beiden Regeln Entity und referentielle Integrität verantwortlich. Die in der Praxis häufig erforderlichen zusätzlichen Integritätsregeln sollen durch die Zugriffssprache definiert und im Systemkatalog abgespeichert werden.

Regel 11: *Ein RDBMS mit einer Zugriffssprache bleibt logisch unbeeinträchtigt, wenn entweder die verteilte Datenhaltung eingeführt oder umgekehrt verteilte Daten auf einem System zusammengeführt werden.* Codd formulierte diese Regel sogar noch lapidarer. Dadurch kann sogar ein nicht-verteiltes DBMS die 11. Regel voll unterstützen.

Regel 12: *Es soll keine implementierten Möglichkeiten geben, die Regeln für relationale System zu unterwandern.* Falls also ein relationales System über eine sogenannte Low-Level-Sprache verfügt, so dürfen damit nicht die mit der High-Level-Sprache ausgedrückten Integritätsregeln und Zwangsbedingungen verletzt oder umgangen werden. Unter Low-Level-Sprache wird eine Sprache verstanden, die einen satzweisen Zugriff auf die Datenbestände erlaubt, während eine High-Level-Sprache mengenorientiert arbeitet.

An dieser Regel scheitern die meisten nicht-relationalen Systeme, da sie häufig eine Sprache oder Programmierschnittstelle unterhalb der relationalen Sprache besitzen. Auch wenn mit Zugangsregeln geprüft wird, wer die Low-Level-Sprache benutzen darf, bleibt die Regel nach Codd verletzt.

Fazit: Auch wenn es hin und wieder gar nicht einfach zu bestimmen ist, welche Systeme den Regeln gemäß relational oder nicht-relational sind, für den größten Teil der existierenden Datenbanksysteme lassen sich die relationalen Regeln durchweg anwenden.

A.3 Relationale Vollständigkeit

In Abschnitt 1.3 ab Seite 9 wurden Relationen betrachtet. Sie bilden die mathematische Grundlage für das Konzept der relationalen Datenbanken und von SQL. Da Relationen Mengen sind, können sie miteinander verknüpft werden. Die grundlegenden fünf Operationen sind die Projektion, das kartesische Produkt, die Vereinigung, die Differenz und die Selektion. Welchen praktischen Nutzen aber hat die Relationenalgebra eigentlich? Wo sind ihre Grenzen?

So wie die Turing-Maschine das mathematische Referenzmodell für universelle („Turing-vollständige“) Programmiersprachen ist, so ist die Relationenalgebra ein Referenzmodell für Abfragesprachen. Es ist sehr genau bekannt, welche Ausdrücke sich mit der Algebra formulieren

lassen. Wenn wir eine Abfragesprache haben, die zur Relationenalgebra äquivalent ist, dann kennen wir auch ihre Möglichkeiten und Grenzen.

Die fünf zugrunde liegenden Operationen der Relationenalgebra können alle effizient implementiert werden. Selbst wenn die Operationen mehrfach kombiniert werden, können ihre Laufzeiten nicht exponentiell wachsen. Die Relationenalgebra ist den universellen Programmiersprachen daher zwar hinsichtlich ihrer Ausdrucksfähigkeit unterlegen. Doch haben diese universellen Sprachen den Nachteil, dass wir bei Programmen, die in diesen Sprachen entwickelt wurden, mit exponentiellen Laufzeiten rechnen müssen. Bei Sprachen, die ausschließlich auf der Relationenalgebra fußen, kann es so etwas nicht geben.[3]

Jedes RDBMS enthält eine Komponente, die als Optimierer bezeichnet wird[4] und dafür verantwortlich ist, dass SQL-Anweisungen effizient ausgeführt werden. Der Optimierer erreicht dieses Ziel, indem er Ausdrücke der Relationenalgebra geschickt umformt.

Die Relationenalgebra hat also durchaus praxisrelevante Bedeutung. Programmiersprachen, die die Operationen der Relationenalgebra implementieren, werden *relational vollständig* genannt. Die Abfragen, die sich mit relational vollständigen Sprachen formulieren lassen, die *nur* die Relationenalgebra umsetzen, können also anders als in Sprachen wie C oder Java keine exponentiellen Laufzeiten haben. In den allermeisten Fällen reichen die Mittel dieser Algebra völlig aus, um Abfragen zu formulieren, die für praktische Zwecke von Belang sind. Die Relationenalgebra stellt somit einen guten Kompromiss zwischen möglichst kurzen Rechenzeiten und möglichst starker Ausdrucksfähigkeit dar.

Satz A.1. *SQL ist relational vollständig.*

Beweis. Seien $m, n, k \in \mathbb{N}$, mit $m \geqq 1$, $n \geqq 1$ und $0 \leqq k \leqq n$. Seien ferner die Relationen $R(A_1, \ldots, A_n)$, $S(A_1, \ldots, A_n)$ und $T(A_{i_1}, \ldots, A_{i_k}, B_{k+1}, \ldots, B_m)$ gegeben, wobei die Indizes $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ paarweise ungleich seien. Die fünf elementaren Operationen der Relationenalgebra können in SQL wie folgt umgesetzt werden:

1. Die Projektion auf $(A_1, \ldots, A_i)$ für ein $i \leqq n$: `SELECT DISTINCT` $A_1, \ldots, A_i$ `FROM` R; [5]

2. Das Produkt $S \bowtie T$ von S und T. Für $k = 0$, d.h. $S \cap T = \emptyset$, durch:

   `SELECT DISTINCT` $A_1, \ldots, A_n, B_1, \ldots, B_m$ `FROM` S, T;

   Für $k > 0$, d.h. $S \cap T \neq \emptyset$, durch:

   `SELECT DISTINCT` $A_1, \ldots, A_n, B_{k+1}, \ldots, B_m$ `FROM` S `INNER JOIN` T
   `ON` $S.A_{i_1} = T.A_{i_1}, \ldots, S.A_{i_k} = T.A_{i_k}$;

3. Die Vereinigung $R \cup S$ von R und S:

   (`SELECT` $A_1, \ldots, A_n$ `FROM` R) `UNION` (`SELECT` $A_1, \ldots, A_n$ `FROM` S);

4. Die Differenz $R \setminus S$ von R und S:

   (`SELECT` $A_1, \ldots, A_n$ `FROM` R) `EXCEPT` (`SELECT` $A_1, \ldots, A_n$ `FROM` S);

5. Die Selektion $\sigma[P](R)$ von R unter der Bedingung P:

[3] Piepmeyer (2011):S. 67.

[4] Piepmeyer (2011):Kapitel 20.6.

[5] Die triviale Projektion auf die leere Menge, d.h. $i = 0$ (Select auf null Spalten), ist in SQL allerdings nicht möglich.

SELECT DISTINCT $A_1, \ldots, A_n$ **FROM** R **WHERE** P;

Weitere Details finden sich in der Literatur[6].

□

Da SQL jedoch die Relationenalgebra als echte Teilmenge in seinem Sprachumfang umfasst, ist es auch weit mächtiger als eine reine relational vollständige Programmiersprache. Zum Beispiel wird die relationale Vollständigkeit bereits durch **NULL**-Werte und die durch sie bewirkte dreiwertige Logik (Abschnitt 4), durch die Aggregatfunktionen sowie durch die Operationen **GROUP BY** und **ORDER BY** übertroffen. Umso mehr heben rekursiv definierte Vereinigungen von Relationen wie die CTEs und Aggregationen über Partitionen mit **OVER** SQL über die Mächtigkeit einer rein relational vollständigen Sprache, wie wir mit Satz 13.5 auf Seite 111 gesehen haben.

[6] Piepmeyer (2011:Kapitel 4), `https://www.relationaldbdesign.com/database-design/module2/sql-relationallyComplete.php`, `https://www.dbis.informatik.uni-goettingen.de/Teaching/DB-WS1819/DBBlatt3ML.pdf`.

Literatur

Bartol Jr., T. M. et al. (2015). „Nanoconnectomic upper bound on the variability of synaptic plasticity“. In: *eLife* 4(e10778). DOI: 10.7554/eLife.10778.

Chen, P. P.-S. (März 1976). „The Entity-Relationship Model: Toward a Unified View of Data“. In: *ACM Transactions on Database Systems* 1(1). DOI: 10.1145/320434.320440.

Codd, E. F. (1970). „A relational model of data for large shared data banks“. In: *Communications of the ACM* 13(6), S. 387. DOI: 10.1145/357980.358007.

de Vries, A. (2007). *Algebraic hierarchy of logics unifying fuzzy logic and quantum logic*. URL: https://arxiv.org/abs/0707.2161.

— (2022). *Algorithmik*. Vorlesungsskript. Hagen. URL: https://www.fh-swf.de/media/de/neu_np/fb_tbw_1/dozentinnen_2/professorinnen_5/devries_1/algorithmik.pdf.

Heuer, A. et al. (2020). *Datenbanken Kompaktkurs*. 1. Aufl. mitp: Frechen.

Krcmar, H. (2015). *Informationsmanagement*. 6. Aufl. Springer Gabler: Berlin Heidelberg. DOI: 10.1007/978-3-662-45863-1.

Nørretranders, T. (1997). *Spüre die Welt. Die Wissenschaft des Bewusstseins*. Rowohlt Verlag: Reinbek bei Hamburg.

Piepmeyer, L. (2011). *Grundkurs Datenbanksysteme*. Carl Hanser Verlag: München Wien.

Wirth, N. (1999). *Algorithmen und Datenstrukturen*. B.G. Teubner: Stuttgart Leipzig.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media: Champaign.

Zimmermann, M. (1993). „Das Nervensystem – nachrichtentechnisch gesehen“. In: *Physiologie des Menschen*. Hrsg. von R. F. Schmidt und G. Thews. Springer Verlag: Berlin Heidelberg, S. 176–183.

Internetquellen

[Base] https://wiki.documentfoundation.org/images/0/02/Base_tutorial.pdf – SQL Tutorial zu OpenOffice.org Base (englisch)

[Fiddle] https://www.dbfiddle.uk/ – DB Fiddle, Online-Tool zum Testen von SQL-Abfragen auf verschiedenen Datenbanksystemen

[EDB] http://wikis.gm.fh-koeln.de/ – edb (e-Learning Datenbank Portal), Wiki zu Datenbanken der FH Köln

[LdS] H. Lenstra & B. de Smit: *Escher and the Droste effect*. http://escherdroste.math.leidenuniv.nl/ – Mathematisch begründete Vervollständigungen von Eschers Kunstdruckgalerie

[Le] Jos Ley: *The mathematics behind the Droste effect.* http://www.josleys.com/article_show.php?id=82 – Mathematik zur Rekursion in der bildenden Kunst

[Libre] https://wiki.documentfoundation.org/images/f/f6/Base_Gesamtband_einseitig_V60.pdf – Handbuch zu LibreOffice Base

[Libre-2] https://wiki.documentfoundation.org/Faq/Base – FAQs zu LibreOffice Base

[SQL1] https://de.wikibooks.org/wiki/Einf%C3%BChrung_in_SQL – Einführung SQL, WikiBook

[W3S] https://www.w3schools.com/sql/ – SQL Tutorial von w3schools

[Wen] https://www.essentialsql.com/ – Kris Wenzel: *Essential SQL.* SQL Tutorial (Englisch)

[Win] https://modern-sql.com/de – M. Winand: *Modern SQL.* Informationen rund um SQL

Index

μ-rekursiv, 113
ABS, 16
ALL, 50, 107
ALTER, 15, 115
AND, 23
ANY, 50
ASC, 26
AS, 21, 77, 104
AVG, 38
CASCADE, 72, 73
CHECK, 15, 82
CONCAT, 16
COUNT, 38
CREATE, 14, 34, 114
DELETE, 15, 116
DESC, 26
DISTINCT, 19, 39, 99
DROP, 15, 115
EXCEPT, 47
EXISTS, 50
FOREIGN KEY, 61
GRANT, 116
GROUP BY, 40, 122
IFNULL, 32
INSERT, 15, 115
INTERSECT, 47
IN, 47, 50
IS, 31
JOIN, 89
KEY, 61
LEFT JOIN, 91
LIKE, 24, 49
MAX, 38
MIN, 38
NOT LIKE, 25
NOT NULL, 29
NOT, 23, 31
NULL, 28, 51, 122
NVL, 32
Nz, 32
ON DELETE CASCADE, 62, 72
ON DELETE NO ACTION, 62
ON DELETE RESTRICT, 62
ON DELETE SET DEFAULT, 62
ON DELETE SET NULL, 62, 75
ON, 89
ORDER BY, 25, 44, 122
OR, 23
OUTER, 92
OVER, 44
PARTITION BY, 44
POSITION, 16
POWER, 16
PRIMARY KEY, 34, 115
RECURSIVE, 107
REFERENCES, 61
REPLACE, 16
RESTRICT, 62
REVOKE, 116
RIGHT JOIN, 91
ROUND, 16
SELECT, 15, 18, 100, 116
SERIAL, 36
SQRT, 16
STDDEV, 38
STRING_AGG, 112
SUBSTRING, 16
SUM, 38
TRIM, 16
UNION, 47, 107, 109
UNIQUE, 69
UPDATE, 15, 116
VALUES, 15, 115
VARIANCE, 38
VIEW, 100
WHERE, 22
WITH, 104
%, 24
_, 24
!=, 22
*, 18
<=, 22, 49
<>, 22, 49
<, 22, 49
=, 22, 49
>=, 22, 49
>, 22, 49
int, 107
STDDEV_SAMP, 39

abgeschlossene Operation, 20
ABS, 16
Access, 13, 22, 24, 32, 89
Ackermann-Funktion, 113
Aenderungsanomalie, 83
Aggregatfunktion, 38, 42, 44, 49, 112, 122
Aggregation über Partitionen, 43, 122
Algorithmus, 103, 104
Alias, 21, 42, 77
ALL, 50, 107
ALTER, 15, 115
AND, 23

Anomalie, 81
ANSI, 11
ANSI-SPARC-Architektur, 8
Ansicht (View), 100
ANY, 50
API, 7
Apolloprogramm, 118
Array, 7
AS, 21, 77, 104
ASC, 26
ASCII, 7
Attribut, 10, 56
Aussage, logische –, 23
autoincrement, 36
Automat, 110
AVG, 38

Basisfall, 106
Beziehung, 57
Beziehungstabelle, 66, 69
Binärwort, 7
Bit, 6
Boole'sche Algebra, 23

CASCADE, 72, 73
case sensitive, 12
CHECK, 15, 82
Chen-Notation, 56, 66
Codd, 116
Codd'sch 6. Regel, 103
Common Table Expression, 104, 122
CONCAT, 16
COUNT, 38
CREATE, 14, 34, 114
CRUD, 15, 115
CTE, 104, 122
Cyclic Tag System, 110

D, 21
Dataphor, 21
Datenbank, 8, 9
Datenbankentwurf, 56
Datenbanksystem, 9
Datenmenge, 6
Datenmodellierung, 56
Datensatz, 10
Datenstruktur, 7
Datentyp, 7, 13
DBMS, 9
DCL, 14, 116
DDL, 14, 114
deklarative Programmiersprache, 11, 113
DELETE, 15, 116
DESC, 26
Differenz, 121
Digraph, 75
DISTINCT, 19, 39, 99
DML, 15, 115
dreiwertige Logik, 30, 122
DROP, 15, 115
Dublette, 19, 69

Einfüge-Anomalie, 82
endlicher Automat, 110
Entität, 56, 66
Entitätsintegrität, 35
Equi-Join, 89
Ergebnismenge, 18, 116
ERM, 56, 66
Euklid'scher Algorithmus, 107
Excel, 53
EXCEPT, 47
EXISTS, 50

Feld, 10
Fenster, 44
FOREIGN KEY, 61
Fremdschlüssel, 61
Funktion, 21, 38

gerichtete Beziehung, 68
ggT, 107
Gleitkommazahl, 7
GRANT, 116
Graph, 74, 109
GROUP BY, 40, 122
Gruppenfunktion, 38, 42
Gruppenspalte, 40
Gültigkeitsregel, 15

Hauptbeziehung, 64
Hierarchie, 74

IFNULL, 32
imperative Programmiersprache, 11, 113
IMS, 118
IN, 47, 50
ineffiziente Speicherung, 55
Information, 28, 83, 101
Informationsgehalt, 6
Informationsregel, 117
Inkonsistenz von Daten, 55
Inner Join, 89
innere Abfrage, 49
INSERT, 15, 115
Instanz, 98
int, 107
Integrität
    Entitäts-, 35
    referenzielle –, 62
Integritätsregel, 21, 29, 35, 62, 69, 82
Interpreter, 11
INTERSECT, 47
IS, 31
IS-A-Beziehung, 72
Iteration, 106

JOIN, 89
Join, 91
Join-Bedingung, 89

Kann-Beziehung, 58
Kardinalität, 57
kartesisches Produkt, 9

KEY, 61
Klausel, 22, 116
Kollation, 26
Kommentar, 12
Konkatenation, 17, 112
Krähenfußnotation, 66

leerer String, 25
LEFT JOIN, 91
LIKE, 24, 49
Literal, 12
Loeschanomalie, 83
Logik, dreiwertige –, 30, 122
logische Aussage, 23
logischer Operator, 23
Lukasiewicz, Jan (1878–1956), 31

Managementberichte, 103
MAX, 38
Menge, 46
Mengenlehre, 46
Mengenoperationen, 95
Mengenoperationen in SQL, 47
Mengenoperator, 50
MIN, 38
modulo, 113
MS Access, 13
Muss-Beziehung, 58
mü-rekursiv, 113

NASA, 118
natürlicher Primärschlüssel, 34
Netzwerk, 74, 109
Normalisierung, 81, 84
NOT, 23, 31
NOT LIKE, 25
NOT NULL, 29
NULL, 28, 40, 51, 118, 122
NVL, 32
Nz, 32

Objektorientierung, 10, 56, 74
OLAP-System, 103
ON, 89
ON DELETE CASCADE, 62, 72
ON DELETE NO ACTION, 62
ON DELETE RESTRICT, 62
ON DELETE SET DEFAULT, 62
ON DELETE SET NULL, 62, 75
ON-Klausel, 89
Optimierer, 121
OR, 23
ORDER BY, 25, 44, 122
Organigramm, 74
OUTER, 92
Outer Join, 91
OVER, 44

parallele Datenspeicherung, 81
PARTITION BY, 44
Partitionen, 43
persistente Datenspeicherung, 8
POSITION, 16
PostgreSQL, 110
POWER, 16
PRIMARY KEY, 34, 115
Primärschlüssel, 34, 56, 115, 117
Produkt, 9, 121
Produktionsregel, 110
Programmierschnittstelle, 101
Projektion, 19, 121
Prädikat, 22

Query Execution Plan, 11

RDBMS, 10, 117
RECURSIVE, 107
Redundanz, 55, 81, 84, 101
REFERENCES, 61
referenzielle Integrität, 62
Rekursion, 105, 106, 113
rekursive Beziehung, 74
Rel, 21
Relation, 9, 19
relational vollständige Programmiersprache, 121
relationale Datenbank, 10
Relationentyp, 10, 14
REPLACE, 16
reservierte Wörter, 12
RESTRICT, 62
REVOKE, 116
RIGHT JOIN, 91
ROUND, 16

Saturn V, 118
Schema, 10, 14
Schlüssel, 33
schreibungsinsensitiv, 24
schreibungssensitiv, 12
SELECT, 15, 18, 100, 116
Selektion, 18, 121
Semikolon, 14, 114
SERIAL, 36
serieller Primärschlüssel, 36
Single-Row-Funktion, 16
SQL, 11
SQRT, 16
Standardabweichung, 39
STDDEV, 38
STDDEV_SAMP, 39
String, 12
STRING_AGG, 112
Struktur einer Tabelle, 10, 14
Subquery, 40, 49
Subquery Factoring, 104
Subroutine, 104
SUBSTRING, 16
SUM, 38
Synchronisation, 81
Syntax, 12
Syntaxbaum, 11

Tabellendiagramm, 66

Tabellenstruktur, 10, 14
terminierende Rekursion, 107
Tiefensuche, 109
TRIM, 16
Tupel, 10
Turing-vollständig, 110, 120

Unicode, 7, 15
UNION, 47, 107, 109
UNIQUE, 69
Unterabfrage, 40, 49
Unterprogramm, 104
unvollständige Information, 28
UPDATE, 15, 116
UTF-8, 7, 14, 115

VALUES, 15, 115
VARIANCE, 38
Venn-Diagramm, 47
Vereinigung, 121
Vergleichsoperator, 22, 49
Vergleichsoperator =, 49
Verknüpfungsbedingung, 89
verlustfreier Join, 120
verschachtelte Abfrage, 40
verschaltete SELECT-Anweisung, 49
View, 100, 104
virtuelle Tabelle, 101

WHERE, 22
WHERE versus HAVING, 42
WHERE-Klausel, 89
WHILE, 45
WITH, 104
Wort (Automatentheorie), 110

zeitlich stabil, 35
zusammengesetzter Primärschlüssel, 34