

HTL Hollabrunn



In Kooperation mit dem HERDT-Verlag stellen wir Ihnen eine PDF inkl. Zusatzmedien für Ihre persönliche Weiterbildung zur Verfügung. In Verbindung mit dem Programm HERDT|Campus ALL YOU CAN READ stehen diese PDFs nur Lehrkräften und Schüler*innen der oben genannten Lehranstalt zur Verfügung. Eine Nutzung oder Weitergabe für andere Zwecke ist ausdrücklich verboten und unterliegt dem Urheberrecht. Jeglicher Verstoß kann zivil- und strafrechtliche Konsequenzen nach sich ziehen.

SQL

**Grundlagen und
Datenbankdesign**

(Stand 2021)

Elmar Fuchs

1. Ausgabe, Juli 2021

ISBN 978-3-98569-009-1

SQL_2021



Bevor Sie beginnen ...	4	6.3 Bedingtes Einfügen / Aktualisieren von Daten	96
		6.4 Daten löschen	97
		6.5 Übung	98
1 Grundlagen zu Datenbanken	6		
1.1 Entwicklung der Datenbanken	6	7 Einfache Datenabfrage	100
1.2 Datenbankmodelle	8	7.1 Grundlagen zu einfachen Datenabfragen	100
1.3 Aufbau und Organisation einer Datenbank	15	7.2 Bedingungen definieren	106
1.4 Physische Datenbankarchitektur	20	7.3 Abfrageergebnisse gruppieren	111
1.5 Übung	25	7.4 Abfrageergebnisse sortieren	113
		7.5 Übung	114
2 Der Datenbankentwurf	26		
2.1 Einführung zum Datenbankentwurf	26	8 Schlüsselfelder und Indices	115
2.2 Der Datenbank-Lebenszyklus	26	8.1 Einführung zu Schlüsseln und Indizes	115
2.3 Datenbanken entwerfen	27	8.2 Schlüsselfelder festlegen und bearbeiten	117
2.4 Das Entity-Relationship-Modell	30	8.3 Indizes	125
2.5 Übung	43	8.4 Übung	128
3 Das relationale Datenmodell	44		
3.1 Begriffe aus dem Bereich relationaler Datenbanken	44	9 Funktionen in Abfragen	129
3.2 Transformation des ER-Modells in ein relationales Modell	48	9.1 Standard-Funktionen in SQL	129
3.3 Normalisierung des Datenbankschemas	51	9.2 Nicht standardisierte Funktionen	133
3.4 Theorie relationaler Sprachen	59	9.3 Übung	137
3.5 Übung	65		
4 Datenbanken	66	10 Datenabfragen für mehrere Tabellen	138
4.1 Die Datenbankabfragesprache SQL	66	10.1 Tabellen verknüpfen	138
4.2 Datenbank erstellen	68	10.2 Einfaches Verknüpfen von Tabellen	141
4.3 Datenbank anzeigen und auswählen	71	10.3 Tabellen verknüpfen mit JOIN	144
4.4 Datenbank löschen	73	10.4 Zwei Tabellen vereinigen	151
4.5 Übung	73	10.5 Schnitt- und Differenzmengen	152
		10.6 Unterabfragen	153
		10.7 Übung	155
5 Tabellen erstellen und verwalten	74		
5.1 Tabellen erstellen	74	11 Sichten	156
5.2 Datentypen festlegen	77	11.1 Vordefinierte Abfragen	156
5.3 Constraints in Tabellen verwenden	81	11.2 Sichten erstellen	157
5.4 Domänen verwenden	83	11.3 Sichten löschen	159
5.5 Vorhandene Tabellen anzeigen, ändern und löschen	86	11.4 Daten über Sichten einfügen, ändern und löschen	160
5.6 Übung	89	11.5 Übung	162
6 Daten einfügen, aktualisieren, löschen	90	12 Cursor	163
6.1 Daten einfügen	90	12.1 Sequenzielles Lesen von Datensätzen	163
6.2 Daten aktualisieren	94	12.2 Cursor erstellen	164
		12.3 Datenzugriff mit dem Cursor	165
		12.4 Cursor schließen	166

13 Zugriffsrechte und Benutzer verwalten	167	15 Stored Procedures	187
13.1 Sicherheitskonzepte	167	15.1 Programmabläufe speichern	187
13.2 Benutzerverwaltung unter PostgreSQL	168	15.2 Stored Procedures erstellen und bearbeiten	189
13.3 Benutzerverwaltung unter MariaDB	171	15.3 Beispielanwendung für Stored Procedures	193
13.4 Zugriffsrechte an Benutzer vergeben	173	15.4 Übung	197
13.5 Benutzern die Zugriffsrechte entziehen	176		
13.6 Übung	177		
14 Transaktionsverwaltung	178	16 Trigger	198
14.1 Konsistente Datenbestände und Transaktionen	178	16.1 Prozeduren automatisch ausführen	198
14.2 Transaktionen erstellen	180	16.2 Trigger erstellen	199
14.3 Transaktionen abschließen	184	16.3 Trigger bearbeiten und löschen	203
14.4 Transaktionen zurücksetzen	185	16.4 Übung	204
14.5 Übung	186		
		Stichwortverzeichnis	206

Bevor Sie beginnen ...

Voraussetzungen und Ziele

Zielgruppe

Dieses Buch richtet sich an alle, die die Datenbankabfragesprache SQL erlernen wollen:

- ✓ Personen, die eine IT-Grundlagenausbildung absolvieren,
- ✓ zukünftige Datenbankadministratoren, zu deren hauptsächlichen Aufgaben der Aufbau, die Erweiterung und die fortlaufende Administration eines Datenbanksystems auf der Basis von SQL gehört,
- ✓ (zukünftige) Programmierer und Webseitenentwickler, die große Datenmengen verwalten und auswerten müssen.

Empfohlene Vorkenntnisse

Folgende Kenntnisse erleichtern das Verständnis der Inhalte sowie die erfolgreiche Arbeit mit dem Buch:

- ✓ Grundkenntnisse im Umgang mit Konsolenbefehlen unter Windows oder Linux
- ✓ Grundkenntnisse in der Administration von Windows- oder Linux-Systemen

Hinweise zu Soft- und Hardware

Die Erläuterungen und Beispiele dieses Buches beziehen sich auf die Datenbanksysteme MariaDB (eine Abspaltung von MySQL) in der Version 10.5.10 und PostgreSQL in der Version 13.3. Diese beiden Systeme besitzen seit vielen Jahren einen konstant hohen Marktanteil (bei MariaDB in Form des Ursprungprodukts MySQL) und sind frei verfügbar. Dabei ist zu beachten,

- ✓ dass viele Anweisungen in anderen Datenbanksystemen (DBS) genauso angewendet werden können,
- ✓ dass sich Anweisungen auch zwischen MariaDB und PostgreSQL unterscheiden können bzw. einige Anweisungen nicht in beiden DBS existieren,
- ✓ dass die gezeigten SQL-Syntaxdefinitionen die wichtigsten Parameter enthalten, dass jedoch teilweise in den beiden Datenbanksystemen weitere systemspezifische Parameter existieren,
- ✓ dass DBS verschiedener Hersteller nicht den gesamten Umfang von SQL unterstützen und eigene Dialekte verwenden.

- Plus** **Ergänzende Lerninhalte: PostgreSQL und MariaDB installieren.pdf**
Hinweise zur Installation von PostgreSQL und MariaDB finden Sie im oben angegebenen BuchPlus-Dokument.
- Plus** **Ergänzende Lerninhalte: SQirreL installieren.pdf**
Im oben angegebenen BuchPlus-Dokument wird das Werkzeug SQuirreL beschrieben, das Sie alternativ zu der im Buch für die Screenshots verwendeten Kommandozeile für die Eingabe der SQL-Befehle verwenden können. Es erleichtert durch integrierte Hilfsmöglichkeiten wie beispielsweise die Syntaxunterstützung das Erlernen der Sprache SQL.

Lernziele

Die Kapitel 1 – 3 beinhalten eine Einführung in die Datenbanktheorie. In diesen Kapiteln erhalten Sie einen grundlegenden Überblick über die verschiedenen Datenbanktypen sowie den Aufbau und Einsatz von Datenbanksystemen. Darüber hinaus werden der Entwurf von Datenbanken und das relationale Datenmodell beschrieben.

In den anschließenden Kapiteln wird Ihnen das praktische Wissen über die Datenbanksprache SQL vermittelt, z. B. wie Sie Datenbanken und Tabellen mit SQL erstellen und bearbeiten und wie Sie Datenbestände auswerten können. Diese Inhalte können Sie prinzipiell auch ohne das Studium der Kapitel 1 – 3 verstehen. Für eine effektive Anwendung der Kenntnisse und speziell für einen gelungenen Entwurf zukünftig von Ihnen entwickelter Datenbanken spielen diese theoretischen Aspekte jedoch eine nicht zu unterschätzende Rolle.

- Plus** **Ergänzende Lerninhalte:** *SQL_XML.pdf*
Als Zusatz finden Sie im oben angegebenen BuchPlus-Dokument Informationen zu den Anwendungsmöglichkeiten von XML in SQL-Datenbanken.
- Plus** **Wissenstests:** *Datenbanktheorie und Sprache und Sprachelemente*
Zur Überprüfung Ihres erworbenen Wissens stehen Ihnen zwei Wissenstests unter den BuchPlus-Dokumenten zur Verfügung.

HERDT BuchPlus – unser Konzept:

Problemlos einsteigen – Effizient lernen – Zielgerichtet nachschlagen

Nutzen Sie dabei unsere maßgeschneiderten, im Internet frei verfügbaren Medien:



Wie Sie schnell auf diese BuchPlus-Medien zugreifen können, erfahren Sie unter www.herdt.com/BuchPlus.

1

Grundlagen zu Datenbanken

1.1 Entwicklung der Datenbanken

Datenbanken spielen beim Einsatz von Computern häufig eine zentrale Rolle. Wenn Arbeitsabläufe computerunterstützt abgewickelt werden, ist meistens die Speicherung großer Datens Mengen erforderlich. Typische Beispiele dafür sind Personalverwaltung, Lagerwirtschaft oder Bestell- und Rechnungswesen großer Unternehmen, wobei die Prozesse in großen Unternehmen mittels eines ERP-Systems (**Enterprise Ressource Planning**) realisiert werden. Die notwendigen Daten werden hier mit einem Datenbanksystem verwaltet. Weitere typische Einsatzgebiete für Datenbanken sind Webanwendungen wie E-Shops oder zentrale Nachschlage- und Informationsverzeichnisse, wie z. B. Wikipedia.

Beim Speichern der Daten in einfachen Dateisystemen werden z. B. folgende Probleme erkannt:

✓ **Redundanzen**

Viele Daten werden **mehrfach** gespeichert, was eine hohe Datenredundanz bedeutet. Dadurch werden Änderungen aufwendig, da die gleichen Daten mehrmals an verschiedenen Stellen geändert werden müssen.

✓ **Inkonsistenzen**

Bedingt durch die Redundanzen treten Inkonsistenzen auf, der Datenbestand ist fehleranfällig. Fehler entstehen, wenn Änderungen von **mehrfach** gespeicherten Daten nur an **einer** Stelle vorgenommen werden.

✓ **Eingeschränkter Mehrbenutzerbetrieb**

Die Datenspeicherung in Dateiform erfordert, die ganze Datei zu sperren, auch wenn nur ein Kundensatz verarbeitet wird. Eine gleichzeitige Bearbeitung anderer Daten in der Datei durch einen weiteren Benutzer ist in diesem Moment nicht möglich.

✓ **Datenschutzprobleme**

Ein Lesezugriff auf die gesamten Daten ist zum Teil uneingeschränkt möglich. Er kann jedoch, abhängig vom verwendeten Betriebssystem, durch Lesesperren oder Verschlüsselung unterbunden werden.

✓ **Fehlende Datenunabhängigkeit**

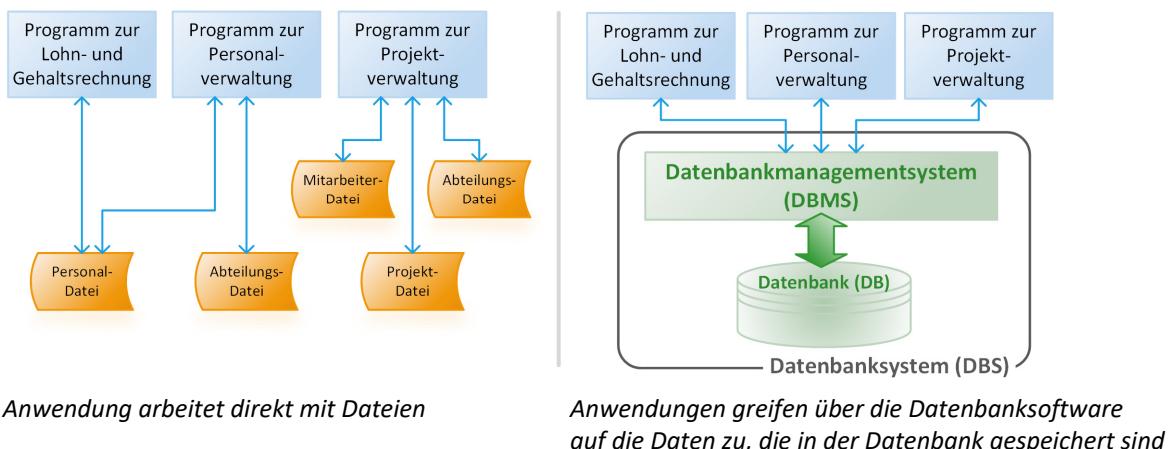
Eine komfortable Verwaltung der Daten ist nur durch die explizit dafür konzipierte Software möglich. Diese muss die Struktur der Daten kennen, um die Daten lesen zu können.

Jede Änderung der Datenstruktur erfordert so – neben einem Programm zur Umstrukturierung der Dateien – zwingend eine Änderung des verarbeitenden Programms.

Werden die gleichen Dateien in mehreren Anwendungen ausgewertet, muss jede eine eigene Datenverwaltung besitzen. Notwendige Änderungsarbeiten fallen damit mehrfach an.

Durch die Entwicklung von **Datenbanksystemen** (DBS) wurden diese Probleme nach und nach gelöst. In einem DBS werden die Daten in einer Datenbank zusammengefasst, die ausschließlich von dem **Datenbankmanagementsystem** (DBMS) der Datenbanksoftware verwaltet wird. Anwendungsprogramme greifen nun nicht mehr direkt auf die Daten zu, sondern stellen ihre Anforderungen nur noch an das Datenbankmanagementsystem. Die enge Verflechtung und Abhängigkeit von Daten und den damit arbeitenden Anwendungsprogrammen wird stark reduziert bzw. ganz aufgehoben.

Ein DBS besteht aus einer Anzahl von Datenbanken und dem DBMS. Die Datenbank ist die Sammlung von logisch zusammengehörigen Daten zu einem Sachgebiet. Das DBMS stellt die Schnittstelle zwischen der Datenbank und deren Benutzern (z. B. den Anwendungsprogrammen) her. Es gewährt einen effizienten und adäquaten (adäquat = angemessen) Zugriff auf die Daten und sorgt dabei für eine zentrale Steuerung und Kontrolle. Außerdem wird durch das DBMS ein Schutz gegen Hard- und Softwarefehler gewährleistet, sodass beispielsweise bei Programm- oder Systemabstürzen die Daten nicht verloren gehen bzw. wiederhergestellt werden können.



Eine Datenbank weist folgende Eigenschaften auf:

- ✓ In Datenbanken sind Daten entsprechend ihren natürlichen Zusammenhängen gespeichert. Dabei ist es nicht entscheidend, in welcher Form die Daten in Anwendungen benötigt werden. Die Daten der Datenbank bilden einen Ausschnitt aus der realen Welt ab.
- ✓ Auf die Daten einer Datenbank können viele Benutzer gleichzeitig zugreifen. Das Datenbankmanagementsystem verwaltet sowohl die Daten als auch die Zugriffe darauf und sorgt dafür, dass dieselben Daten nicht gleichzeitig von mehreren Benutzern bearbeitet werden können. Auch wenn mehrere Anwendungen mit einer gemeinsamen Datenbasis arbeiten, so wird in den Anwendungsprogrammen meist nur auf einen Teil davon zugegriffen (auf eine Sicht auf die Datenbank).

Das Datenbankmanagementsystem ist die Software, welche die Datenbanken verwaltet.

Es ermöglicht ...

- ✓ das Anlegen von Datenbanken,
- ✓ die Speicherung, Änderung und Löschung der Daten,
- ✓ das Abfragen der Datenbank,
- ✓ die Verwaltung von Benutzern, Zugriffen und Zugriffsrechten.

Um von Anwendungsprogrammen bzw. Benutzern auf die Daten einer Datenbank zugreifen zu können, stellt das DBS ein Sprachkonzept zur Verfügung. Bei **relationalen DBS** ist das meist die **Datenbanksprache SQL** (Structured Query Language – strukturierte Abfragesprache).

1.2 Datenbankmodelle

Der Übergang von der Nutzung des Dateisystems zur Datenbank vollzog sich schrittweise.

- ✓ Zunächst wurden die Dateien auf Magnetbändern gespeichert (50er Jahre).
- ✓ Diese wurden durch den Einsatz der schnelleren Magnetplatten abgelöst. Sie brachten den Vorteil des Direkt- und Mehrfachzugriffs (60er Jahre).
- ✓ In den ersten Datenbanksystemen (1. Generation) wurde das hierarchische Datenmodell verwendet (70er Jahre).
- ✓ Die Weiterentwicklung brachte das Netzwerkmodell hervor.
- ✓ Die nächste Datenbank-Generation wurde mit dem relationalen Modell eingeleitet. Dieses stammt bereits aus den 70er Jahren. Im folgenden Jahrzehnt wurde die maßgebliche praktische Umsetzung realisiert.
- ✓ Parallel dazu entwickelten sich alternative Ansätze in Form der spalten- bzw. der dokumentorientierten Datenbanken.
- ✓ Dem Paradigma objektorientierter Sprachen folgend wurden objektorientierte Datenbanken entwickelt (Anfang der 90er Jahre).
- ✓ Der nachfolgenden Datenbank-Generation wurde das objektrelationale Datenmodell zugrunde gelegt, das die Vorteile der objektorientierten und der relationalen Datenmodelle vereinigt (Mitte der 90er Jahre).
- ✓ Seit Mitte des ersten Jahrzehnts des neuen Jahrhunderts ist ein verstärkter Trend zu den sogenannten NoSQL-Datenbanken zu verzeichnen, welcher sich in den letzten Jahren weiter verstärkt hat.

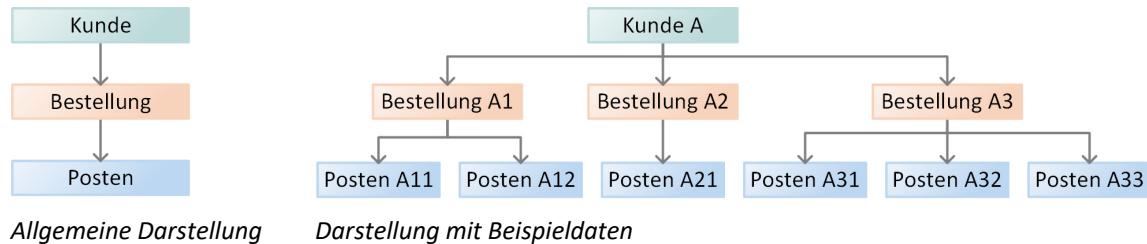
Hierarchische Datenbanken

Das hierarchische Datenmodell wurde entwickelt, um unterschiedlich lange Datensätze (zusammengehörige Informationen, eine bestimmte Sache betreffend) zu verarbeiten. Die Datensätze werden so aufgeteilt, dass gleichartige Daten zu kleineren Datengruppen zusammengefasst werden. Diese Gruppen bilden die Knoten der Hierarchie. So entsteht eine **baumartige Struktur**, die streng hierarchisch geordnet ist.

Jeder untergeordnete Knoten ist von seinem übergeordneten Knoten abhängig. Die Struktur entspricht einer **Vater-Sohn-Beziehung**. Ein Vater kann mehrere Söhne haben, ein Sohn aber nur einen Vater. Die Struktur kann nicht ohne den Wurzelknoten existieren.

Beispiel

Die Abbildung zeigt die hierarchische Struktur einer Kundenverwaltung. Jeder Kunde kann eine unterschiedliche Anzahl von Bestellungen mit einer bestimmten Anzahl von Posten aufgeben. So ist nicht vorhersehbar, wie lang ein bestimmter Datensatz ist. Die Daten der Kunden, Bestellungen und Posten bilden im hierarchischen Datenmodell die Knoten des Hierarchiebaums. Für jeden Knoten wird in der Datenbank ein Datensatz angelegt.



Eine bekannte Datenbank mit hierarchischer Struktur ist IMS/DB von IBM.

Nachdem die hierarchischen Datenbanken bis auf einzelne Nischen aus der praktischen Anwendung verdrängt wurden, finden heute mit der starken Verbreitung der erweiterbaren Auszeichnungssprache XML (Extensible Markup Language) ihre theoretischen Ansätze wieder eine Anwendung.

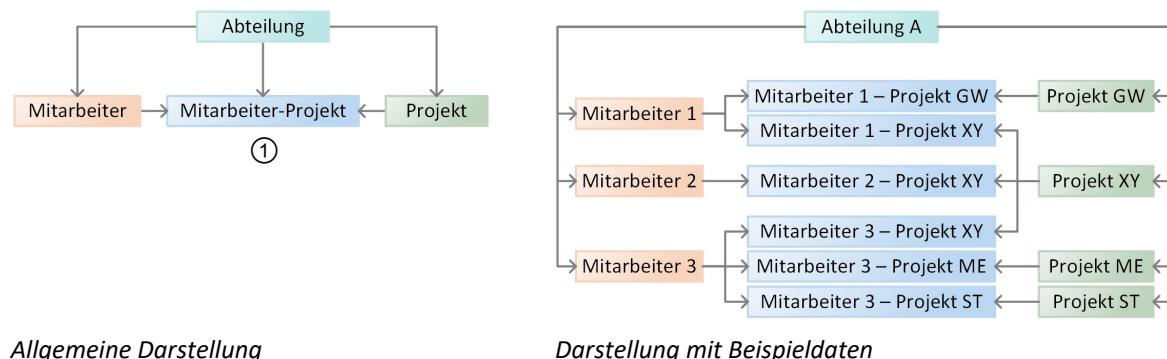
Netzwerkdatenbanken

Beim Netzwerkmodell werden gleichartige Daten in **Recordsets** gespeichert, die miteinander in Beziehung stehen. Einem Record eines Recordsets können dabei mehrere Records eines anderen Recordsets zugeordnet werden, was als Pfeil in der grafischen Darstellung erscheint. Durch diese Beziehungen zwischen den Recordsets entsteht ein gerichteter Graph, der auch als Netzwerk bezeichnet wird. Die Beziehungen werden hier als **Sets** (Mengen) bezeichnet. Die Sets sind in der Datenbank fest definiert.

Beispiel

Es wird die Projektverwaltung einer Abteilung betrachtet. Zu einer Abteilung gehören mehrere Mitarbeiter.

In jeder Abteilung wird an mehreren Projekten gearbeitet. An jedem Projekt arbeiten mehrere Mitarbeiter der Abteilung mit, jeder Mitarbeiter kann aber auch an mehreren Projekten mitarbeiten. Diese Beziehungen sind in der Abbildung durch Pfeile dargestellt.



Beim Netzwerkmodell darf eine Beziehung zwischen zwei Recordsets immer nur in eine Richtung zeigen. Zwischen den Recordsets *Mitarbeiter* und *Projekt* wäre eine beidseitige Beziehung notwendig (mehrere Mitarbeiter arbeiten an mehreren Projekten mit). Das Problem wird durch das Einfügen eines zusätzlichen Recordsets *Mitarbeiter-Projekt* gelöst ①.

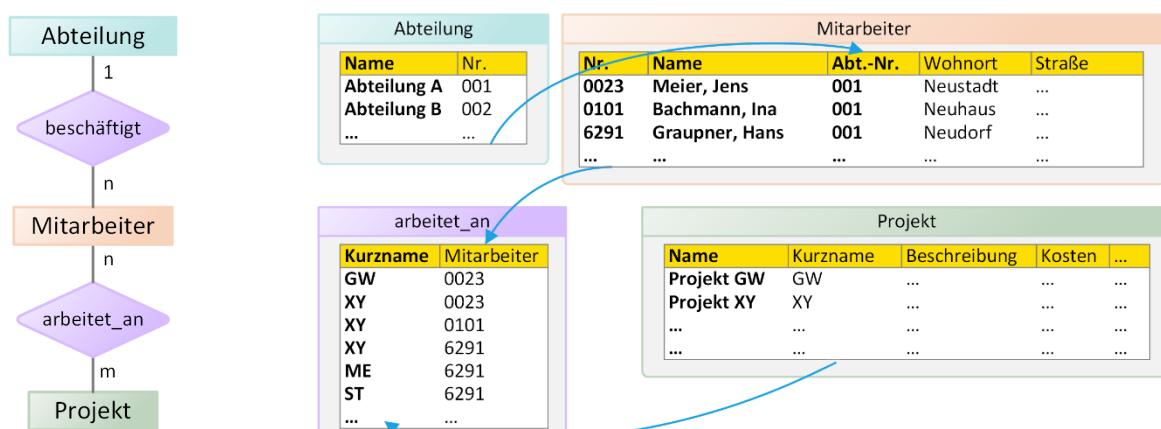
Eine verbreitet zum Einsatz gekommene Netzwerkdatenbank ist u. a. die UDS (Universal Datenbank System) von Siemens.

Relationale Datenbanken

Relationale Datenbanken sind am weitesten verbreitet. Die Daten werden in Tabellenform gespeichert, in sogenannten **Relationen** (mathematische Ausdrucksweise). Zwischen den Relationen (Tabellen) können **Beziehungen** definiert werden. Es sind verschiedene Beziehungsarten möglich, die sich durch die Anzahl der miteinander in Beziehung stehenden Datensätze (Tabellenzeilen, Tupel) unterscheiden (einer mit einem 1:1, einer mit mehreren 1:n, mehrere mit mehreren n:m). Die grafische Darstellung der Relationen und der zugehörigen Beziehungen erfolgt meist im Entity-Relationship-Modell (ERM). Die Beziehungen werden dort durch Linien dargestellt, die mit den Kardinalitäten (1, n, m) versehen sind. Über Abfragen ist es möglich, für die bestehenden Datenbanken unterschiedliche Auswertungen durchzuführen. Für die Abfrage und Auswertung der Daten hat sich die Abfragesprache **SQL** durchgesetzt.

Beispiel

Die Projektverwaltung lässt sich im relationalen Modell wie folgt darstellen: Einer Abteilung gehören **mehrere** Mitarbeiter an, jeder Mitarbeiter ist aber nur in **einer** Abteilung beschäftigt. Zwischen den Relationen *Abteilung* und *Mitarbeiter* besteht eine 1:n-Beziehung. Jeder Mitarbeiter kann an **mehreren** Projekten mitarbeiten, ein Projekt kann wiederum von **mehreren** Mitarbeitern bearbeitet werden. Zwischen den Relationen *Mitarbeiter* und *Projekt* besteht eine n:m-Beziehung.



Darstellung im ER-Modell

Darstellung mit Beispieldaten

Die Daten der Tabellen werden physisch zeilenweise auf einem externen Speichermedium gesichert. Für die häufige Veränderung und Bearbeitung von Daten, die bei Geschäftsvorfällen notwendig ist, das **Online Transaction Processing** (OLTP), eignet sich diese Art der Speicherung sehr gut.

Es gibt eine Vielzahl relationaler Datenbanken, neben anderen Oracle, Microsoft SQL, DB2 von IBM, MySQL, MariaDB und PostgreSQL.

Spaltenorientierte Datenbanken

Spaltenorientierte Datenbanken basieren ebenfalls auf dem architektonischen Ansatz der Relationen bzw. Tabellen. Im Gegensatz zu den relationalen Datenbanken erfolgt die physische Speicherung hier jedoch in einer anderen Form. Die Werte der Tabellen werden pro Spalte – also über ein Attribut aller Tupel hinweg – auf das externe Speichermedium geschrieben.

Anhand der externen Speicherung der Relation *arbeitet_an* des Beispiels der Projektverwaltung zeigt sich der Unterschied folgendermaßen:

Relationale Datenbank:

GW, 0023; XY, 0023; XY, 0101; XY, 6291; ME, 6291; ST, 6291

Spaltenorientierte Datenbank:

GW, XY, XY, XY, ME, ST; 0023, 0023, 0101, 6291, 6291, 6291

arbeitet_an	
Kurzname	Mitarbeiter
GW	0023
XY	0023
XY	0101
XY	6291
ME	6291
ST	6291
...	...

Diese Speicherform bietet bei der reinen Auswertung von Daten, dem **Online Analytical Processing (OLAP)**, deutliche Geschwindigkeitsvorteile, da hier häufig nur die Werte einzelner Spalten (Attribute) von Interesse sind. Der Leseprozess kann auf diese Spalten beschränkt werden. Damit entfällt das Lesen einer Vielzahl nicht benötigter Daten der einzelnen Tupel.

Bei der Bearbeitung von Daten kann diese Speicherform ebenfalls Vorteile besitzen. Dies ist immer dann der Fall, wenn nur ein Attribut über viele Tupel hinweg geändert werden muss.

Einer der ersten Vertreter der spaltenorientierten Datenbanken war Sybase IQ. Mittlerweile gibt es eine Vielzahl kommerzieller (z. B. Oracle Retail Predicative Application Serve [RPAS]) und nicht kommerzieller (z. B. Apache Cassandra) Implementierungen. Auch in freien Datenbanksystemen wie MariaDB und PostgreSQL wird die spaltenorientierte Speicherung von Daten optional unterstützt.

Dokumentorientierte Datenbanken

Nicht alle Informationen lassen sich gut in zweidimensionalen Relationen ablegen. Mengen strukturierter Informationen, die sich in den Ausprägungen der Einzelinformationen über die vorhandenen Attribute hinweg oder in der Menge der Werte der einzelnen Attribute unterscheiden, sind nicht effektiv in einem starren Relationenschema abbildbar.

Dieser Fakt führte zum Entwurf der dokumentenorientierten Datenbanken. Eine zusammengehörige Informationsmenge wird in einem Dokument gespeichert. Dieser Begriff steht dabei für eine strukturierte Information und hat nichts mit dem gleichen Begriff, wie er im Kontext der Textverarbeitung bekannt ist, zu tun. Die Daten innerhalb eines Dokuments werden in Feldname-Wert-Paaren abgespeichert. Dabei muss nicht jedes Feld in jedem Dokument vorhanden sein und einzelne Felder können in unterschiedlichen Dokumenten eine unterschiedliche Anzahl von Werten besitzen.

In Analogie zu dem Projektbeispiel könnte ein Ausschnitt aus einer dokumentenorientierten Datenbank beispielsweise folgendermaßen aussehen:

```
{
  (Vorname: Anton, Nachname: Schulze, AktProjekte: P1; P2; P3);
  (Vorname: Maria, Nachname: Meier, Funktion: Projektleiter, AktProjekte: P2);
  (Vorname: Matthias, Nachname: Schneider, Funktion: Mitarbeiter, AktProjekte: P4);
}
```

Ein typischer Vertreter dokumentenorientierter Datenbanken ist Lotus Notes Domino. Analog zu den spaltenorientierten Datenbanken bieten freie Datenbanksysteme wie MariaDB und PostgreSQL optionale Erweiterungen für die dokumentenbasierte Ablage von Daten an.

Objektorientierte Datenbanken

Objektorientierte Datenbanken (OODB) sind nach dem Paradigma der objektorientierten Programmiersprachen entwickelt worden. Das Ziel der Entwicklung der objektorientierten DBS war es, ein DBS zu schaffen, in welchem Objekte unserer Umwelt mit ihren Eigenschaften und ihrem Verhalten nachgebildet (analog zu objektorientierten Programmiersprachen) und ohne großen Aufwand in einer Datenbank gespeichert und verwaltet werden können.

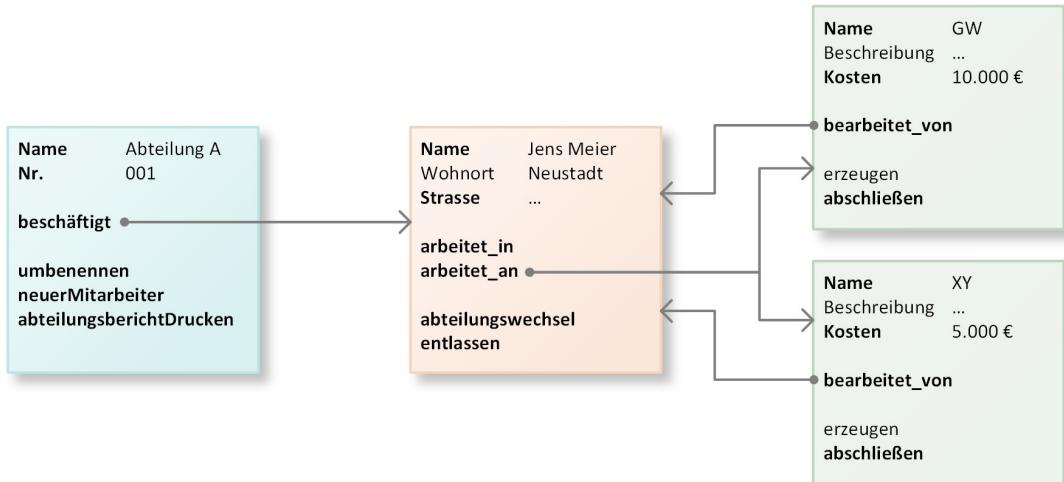
Jedes Objekt der Datenbank enthält Dateninformationen (Attribute), Verweise auf andere Objekte und Operationen (Methoden), die das Verhalten des Objekts widerspiegeln. Die Zusammenfassung von Daten und Operationen über diese Daten wird als Kapselung bezeichnet. Die Definition der Objekte (Daten, Verweise, Methoden) erfolgt über sogenannte Klassen. Durch die freie Beschreibung von Objekt-Klassen lassen sich selbst komplexeste Datenstrukturen in einer Datenbank verwalten, was in den flachen Tabellen relationaler DBS nur über mehrere in Beziehung stehende Tabellen möglich ist. Neben der Kapselung von Daten und Operationen sind weitere objektorientierte Konzepte in OODBS implementiert, wie z. B. die Vererbung, die Überladung von Methoden und dynamische Bindungen. Zusätzlich stehen in OODBS mehr Datentypen zur Verfügung als in den zuvor beschriebenen Datenbankmodellen (z. B. Arrays oder Klassentypen).

Beispiel

Für die Projektverwaltung werden drei verschiedene Objekt-Klassen (Abteilung, Mitarbeiter und Projekte) definiert. Bestandteil der Klassendefinition sind auch die Verweise auf andere Objekte sowie die verschiedenen Operationen (Methoden). So hat ein Objekt der Klasse Abteilung einen Verweis auf eine unbestimmte Menge von Objekten der Klasse Mitarbeiter. Jeder Mitarbeiter besitzt einen eindeutigen Verweis auf seine Abteilung und einen Verweis auf eine unbestimmte Anzahl von Projekten. Jedes Projekt hat einen Verweis auf eine unbestimmte Anzahl von Mitarbeitern. Die Methoden der Klasse Abteilung könnten beispielsweise *neuerMitarbeiter*, *umbenennen* oder *abteilungsberichtDrucken* sein.



Darstellung im objektorientierten Datenbankschema



Darstellung mit Beispieldaten

Objektorientierte Datenbanken sind nach wie vor nur wenig verbreitet. Eine der seltenen Umsetzungen ist db4o, deren weitere Entwicklung 2011 jedoch eingestellt wurde.

Objektrelationale Datenbanken

Objektrelationale Datenbanksysteme (ORDBS) wurden entwickelt, um die Nachteile relationaler DBS zu beseitigen. Solche Nachteile sind z. B. das „magere“ Typsystem von relationalen DBS (umfangreiche Texte, Excel-Tabellen, XML-Dokumente, Audiodaten usw. können nur mit erhöhtem Aufwand oder gar nicht in der DB gespeichert werden) und Probleme bei der Abbildung der Objekte der realen Welt auf das relationale Modell, was als „impedance mismatch“ bezeichnet wird. Das Ziel der Entwicklung war es, die Vorteile der Speicherung komplexer Objekte wie im OODBS zu nutzen und dabei die einheitliche Abfragesprache SQL beizubehalten. Es sollten die Vorteile des relationalen und des objektorientierten Modells in dem neuen DB-Modell vereinigt werden. Dazu musste nicht alles von Grund auf neu entwickelt werden. Es wurde das vorhandene relationale Modell genutzt, das um die objektorientierten Konzepte erweitert wurde. Diese Erweiterungen werden verwendet, um die Objekte zu interpretieren und zu verwalten. Verschiedene Erweiterungen werden mit dem Datenbanksystem ausgeliefert, es existieren aber auch zusätzliche Erweiterungen von Fremdfirmen und Sie können auch eigene erstellen. Die Erweiterungen werden beispielsweise als DataBlades (Informix) oder Cartridges (Oracle) bezeichnet.

Anwendungsprogramme können über die erweiterte SQL-Sprache (Standard SQL:1999, früher als SQL-3 bezeichnet) auf objektrelationale Datenbanken zugreifen. Dieser Standard wurde von der International Standardization Organization (ISO) und dem American National Standards Institute (ANSI) 1999 verabschiedet.

Eine am Markt weitverbreitete objektrelationale Datenbank ist PostgreSQL.

NoSQL-Datenbanken

Mit den im Zuge der Entwicklung des Internets sprunghaft ansteigenden Volumen von Daten in einzelnen Datenbanken wurden die Bemühungen zur Entwicklung alternativer Architekturformen von Datenbanken wieder verstärkt. Ursache dafür waren die Probleme, die relationale Datenbanken bei Anwendungen mit sehr großen Datenmengen haben. Diese treten dann auf, wenn es zu einer extrem großen Anzahl gleichzeitiger Schreib- und Lesezugriffe oder Datenänderungen kommt. Ein weiterer Punkt ist die starke Abhängigkeit der relationalen Datenbanken von einem Schema. Schemaänderungen großer Datenbanken sind sehr zeitaufwendig und im Hinblick auf einen möglichen Datenverlust nicht ungefährlich. NoSQL-Datenbanken besitzen diese starke Abhängigkeit von einem Schema nicht. Typisch für NoSQL-Datenbanken ist die Unterstützung von verteilten Datenbanken auf mehreren Servern, bei der durchaus eine redundante Datenhaltung erfolgen kann.

Vorreiter der Bewegung der NoSQL-Datenbanken waren beispielsweise Google, Amazon, eBay und Facebook.

Die aktuelle Bedeutung der NoSQL-Datenbanken zeigt sich auch an der Entwicklung der beiden im Buch verwendeten Datenbanksysteme. Sowohl MariaDB als auch PostgreSQL bieten eine Anzahl von NoSQL-Funktionen.

Der Begriff NoSQL – im Sinne von „nicht SQL“ – trat zuerst als Name einer kleinen Datenbank Ende der 90er Jahre des letzten Jahrhunderts auf. Später wurde er als Sammelbegriff der Bewegung von Datenbanken, die ein anderes als das relationale Konzept verfolgen, im Sinne von „nicht nur (not only) SQL“ verwendet. Dabei greift die Bewegung durchaus Konzepte auf, die bereits zu einem früheren Zeitpunkt betrachtet wurden. Exemplarisch hierfür stehen die dokumentenorientierten Datenbanken sowie die spaltenorientierten Datenbanken.

NoSQL-Datenbanken können in drei Gruppen unterteilt werden:

- ✓ **Dokumentenorientierte Datenbanken**
- ✓ **Graphen-Datenbanken**

Dieser Typ spielt seine Stärken vor allem bei Anwendungen aus, die eine Vielzahl von Querverbindungen zwischen Daten verwalten müssen. Typisch ist dies zum Beispiel bei Webanwendungen wie Twitter (Wer folgt wem?). Architektonisch können Graphen-Datenbanken als Nachfolger der Netzwerkdatenbanken betrachtet werden. Die Daten werden als Knoten dargestellt und die Beziehungen zwischen ihnen durch die Verbindungen zwischen diesen Knoten.

- ✓ **Key/Value-Datenbanken** (zum Teil in Kombination mit spaltenorientierten Datenbanken)
Key/Value-Datenbanken besitzen ein sehr einfaches Schema. Einem Key (Schlüssel), dargestellt durch eine willkürliche Zeichenkette, ist jeweils ein Value (Wert, Einzelwert, Liste oder Set) zugeordnet. Eine wichtige Unterteilung der Key/Value-Datenbank stellt der Ort der Speicherung der Daten dar – im RAM oder auf einem externen Medium.

Eine moderne dokumentenorientierte Datenbank ist neben dem noch immer aktuellen Lotus Notes die CouchDB. Eine bekannte Graphen-Datenbanken ist Neo4J. Typische Vertreter des Key/Value-Datenbanktyps sind Big Table von Google, Cassandra und Redis.

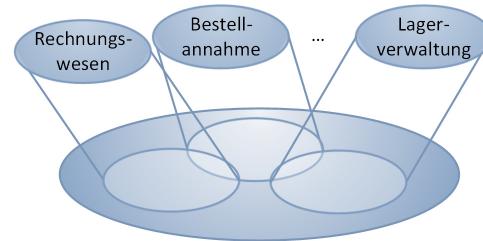
1.3 Aufbau und Organisation einer Datenbank

Eines der wichtigsten Ziele, welches ein DBS realisieren muss, ist die **Datenunabhängigkeit**. Diese wird durch die Trennung der physischen Speicherung der Daten und deren Verwaltung von den Anwendungsprogrammen erreicht. Zum einen sollte eine physische Datenunabhängigkeit bestehen, d. h., für Programme und Benutzer sollte die physische Organisation der Daten transparent sein. So kann die Struktur der gespeicherten Daten geändert werden, ohne dass die Anwendungsprogramme geändert werden müssen. Zum anderen ist auch die logische Datenunabhängigkeit eine wichtige Anforderung an ein Datenbanksystem. Damit kann zwischen einer logischen Gesamtstruktur der Datenbank und den anwenderspezifischen Sichten auf die Daten unterschieden werden. So können weitere Anwendungen und Sichten auf eine bestehende Datenbank erstellt werden, ohne dass bereits existierende Anwendungen dadurch beeinflusst werden.

Eine **Sicht** ist ein Ausschnitt einer Datenbank, der für eine Anwendung bzw. ein Problem relevanten Daten enthält.

Beispiel

Ein Unternehmen des Großhandels besitzt eine große Datenbank, die die Artikeldaten, die Daten der Lagerverwaltung, die Kundendaten und die Daten der Bestellannahme und Rechnungslegung enthält. In den verschiedenen Abteilungen werden nur Teile der Datenbank, bestimmte Sichten, benötigt. So werden bei der Bestellannahme nur die Kunden- und Artikeldaten sowie die im Lager vorhandenen Artikel benötigt.

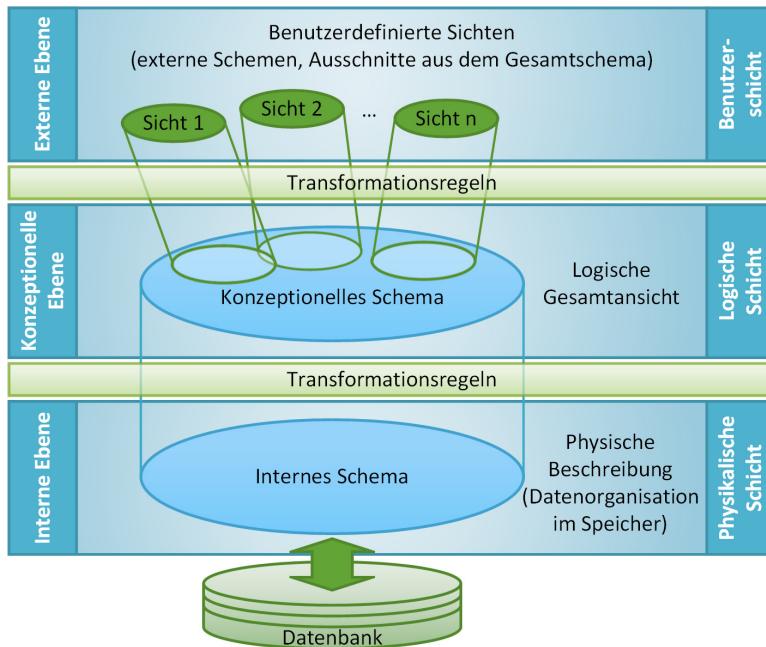


Logische Gesamtansicht auf die DB des Unternehmens

Für die Verwaltung der Artikel im Lager wird nur auf die Artikel- und Lagerdaten zugegriffen. Das Rechnungswesen verwendet die Kundendaten sowie die Artikel- und Bestelldaten. Jede Abteilung verwendet nur die Daten, die sie auch benötigt. Für die Rechnungslegung wird beispielsweise keine Information zum Lagerbestand benötigt.

3-Ebenen-Modell

Am 3-Ebenen-Modell nach ANSI-SPARC 1978 (American National Standards Institute/Standards Planning and Requirements Committee) werden die unterschiedlichen Sichtweisen auf einen Datenbestand dargestellt. Das Prinzip der physischen und logischen Datenunabhängigkeit wird hier deutlich. Darunter versteht man die Möglichkeit des Datenzugriffs durch Programme oder Anwender, ohne dass diese über Kenntnisse der tatsächlichen technischen Umsetzung der physischen Speicherung und der Manipulationsprozesse der Daten verfügen.



Auf der **externen Ebene** erfolgt die Darstellung der Daten, wie sie in den einzelnen Anwendungen benötigt werden. In den Benutzersichten (kurz: Sichten) werden Teile der logischen Gesamtansicht so wiedergegeben, dass dem Benutzer nur die Daten zugänglich sind, mit denen er arbeiten darf. Die Typen der Daten und deren Beziehungen in den Sichten können dabei anders aufgebaut sein als im konzeptionellen Schema. So sind häufig nur Teile der Daten eines Datenobjekts für eine Anwendung relevant. Beispielsweise wird bei der Rechnungslegung der Lagerbestand der Artikel nicht benötigt, der zum Datenobjekt *Artikel* gehört. Die Benutzer sind mithilfe der Datenabfrage- und Datenmanipulationssprache (DQL – **Data Query Language**/DML – **Data Manipulation Language**) in der Lage, auf die Daten zuzugreifen (zu selektieren und zu lesen) und sie zu verändern (bzw. zu löschen und neue hinzuzufügen). Das Einrichten der Datenbanken einschließlich der Sichten ist in größeren Unternehmen Aufgabe des Datenbankadministrators.

In der **konzeptionellen Ebene** werden alle Daten eines Anwendungsbereichs (z. B. Gesamtheit der Daten eines Unternehmens) zusammengefasst, die in der Datenbank gespeichert werden sollen. Auch die logischen Zusammenhänge sowie Änderungsvorschriften für die Daten müssen beschrieben werden. So entsteht eine logische Gesamtansicht, die auch als „Ausschnitt aus der realen Welt“ bezeichnet wird. Die Beschreibung der Daten und ihrer Zusammenhänge erfolgt so, wie diese in der Realität vorkommen, und ist nicht auf die Belange einzelner Anwendungen zugeschnitten. Beispielsweise werden alle Daten eines Unternehmens in der Datenbank abgelegt, die in den einzelnen Abteilungen ausgewertet bzw. verwaltet werden müssen. Das konzeptionelle Schema (konzeptionelle Modell) wird mithilfe einer geeigneten Datendefinitionssprache (DDL – **Data Definition Language**) beschrieben. Diese Aufgaben erledigt in der Regel der Datenbankadministrator.

Die **interne Ebene** beschreibt die Organisation der Daten auf den Speichermedien sowie die Zugriffsmöglichkeiten auf diese Daten. Vom Datenbankadministrator wird, vom konzeptionellen Modell ausgehend, eine physische Datenorganisation entwickelt, die für alle Benutzer einen optimalen Datenzugriff sicherstellt. Das interne Modell wird direkt in die Datenbank übertragen.

Zwischen den 3 Ebenen erfolgt eine Transformation der Schemen ineinander. Dafür besitzt das DBMS **Transformationsregeln**.

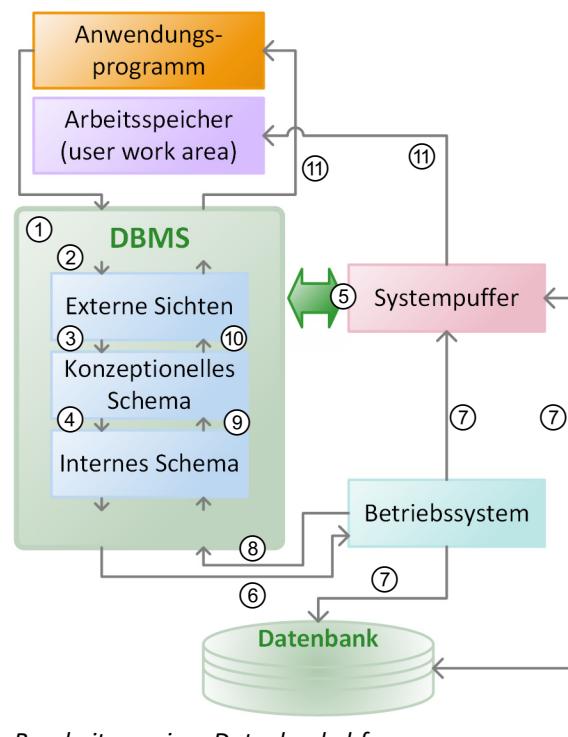
Datenbankmanagementsystem (DBMS)

Das DBMS ist ein Softwarepaket, welches die Verwaltung der Datenbank übernimmt und alle Zugriffe darauf regelt. Die wichtigste und am häufigsten ausgeführte Aufgabe des DBMS ist die **Bereitstellung der Daten**. Als Blackbox betrachtet, nimmt das DBMS dafür die Benutzeranfragen entgegen, ermittelt die angefragten Daten aus der Datenbank und liefert sie dem Benutzer bzw. dem Anwendungsprogramm zurück. Dabei vollzieht das DBMS folgende Arbeitsschritte:

- ✓ Das DBMS empfängt die Anfragen, in denen Daten einer bestimmten externen Sicht angefordert werden.
- ✓ Es liest die Definition der angeforderten Sicht und überprüft die Syntax der Anfrage.
- ✓ Nun überprüft es, ob der Benutzer die Rechte besitzt, auf die Daten zuzugreifen.
- ✓ Mithilfe der Transformationsregeln, die zwischen der externen Sicht und dem konzeptionellen Schema gelten, werden die benötigten Datenobjekte ermittelt.
- ✓ Über die Transformationsregeln, die zwischen dem konzeptionellen Schema und internen Schema angewandt werden, ermittelt das DBMS die physischen Datenobjekte und die Zugriffspfade.
- ✓ Das DBMS beauftragt das Betriebssystem zum Lesen der ermittelten Speicherbereiche.
- ✓ Das Betriebssystem legt diese gelesenen Blöcke im Systempuffer des DBMS ab.
- ✓ Die gelesenen Daten werden über die Anwendung der Transformationsregeln in entgegengesetzter Richtung umgewandelt. Dabei wird die gewünschte Auswahl der Daten zusammengestellt.
- ✓ Das DBMS stellt sicher, dass die übergebenen Daten für andere Benutzer so lange gesperrt sind, bis die Bearbeitung der Daten beendet wird.
- ✓ Die gesuchten Daten (im externen Format) werden an das Anwendungsprogramm bzw. den Benutzer übergeben.

Diese Schrittfolge wird in der folgenden Grafik dargestellt. Dabei werden auch technische Aspekte betrachtet.

- ① Eine Anfrage wird an das DBMS gestellt.
- ② Der Befehl wird analysiert und die zugehörige externe Sicht wird ermittelt.
- ③ Über die Transformationsregeln wird das konzeptionelle Schema ermittelt.
- ④ Über Transformationsregeln wird das interne Schema ermittelt.
- ⑤ Ein Teil der Daten wird im Systempuffer gehalten. Das DBMS prüft, ob sich die angeforderten Daten im Systempuffer befinden.
- ⑥ Sind die Daten nicht im Systempuffer, müssen sie über das Betriebssystem dorthin geladen werden.



- ⑦ Das Betriebssystem tauscht die vorhandenen Daten (Datenseiten – Pages) durch die angeforderten Daten aus und speichert gegebenenfalls geänderte Daten in der Datenbank.
- ⑧ Das Betriebssystem informiert das DBMS über die Bereitschaft der angeforderten Daten.
- ⑨–⑩ Die gewünschten Daten werden über die Transformationsregeln in das Format der betreffenden Sicht umgewandelt.
- ⑪ Das DBMS übergibt die angeforderten Daten und die Statusinformationen an die Anwendung.

Weitere Aufgaben des DBMS

✓ Sicherung der Integrität (in sich richtige und widerspruchsfreie Daten)

Durch die Anwendung der im konzeptionellen Schema vorgegebenen Integritätsbedingungen kann die logische Richtigkeit (Konsistenz) der Daten (entsprechend den Zusammenhängen in der Praxis) gewahrt werden.

Beispiel: In einem System werden Kunden und Rechnungen gespeichert. Ein Kunde darf nur dann gelöscht werden können, wenn auch alle ihm zugeordneten Rechnungen gelöscht sind. Würde der Kunden trotz vorhandener Rechnung gelöscht, fehlen in den Rechnungen entsprechende Daten wie Name, Vorname und Rechnungsadresse.

✓ Datensicherung (Recovery)

Das DBMS ist in der Lage, nach einem Systemabsturz, einem Absturz der Anwendung oder anderen Fehlern die Datenbank wieder in einen konsistenten Zustand zu überführen.

Zu diesem Zweck verfügt das DBMS meist über ein internes Logbuch.

✓ Synchronisation

Meist arbeiten mehrere Benutzer gleichzeitig mit einer Datenbank. Das DBMS hat hier die Aufgabe, parallel ablaufende Transaktionen (Folge von Lese- und Schreib-Operationen) der Benutzer zu synchronisieren, d. h. die Zugriffe so zu verwalten, dass die Integrität der Datenbank gewahrt bleibt. Diese Aufgabe wird vom integrierten Transaktions-Manager übernommen.

Beispiel: Solange das Anwendungsprogramm des Personalbüros die Daten von Frau Maier bearbeitet, kann kein anderer Benutzer mit diesem Datensatz arbeiten. Erst wenn der Datensatz gespeichert wurde, können andere wieder darauf zugreifen. Es gibt aber Einstellungen für das DBS, die diesen Schutzmechanismus ausschalten bzw. dessen Reaktionen steuern.

✓ Zugriffssteuerung

Einige Daten, wie beispielsweise die Gehälter der Angestellten, dürfen nur für bestimmte Personenkreise zugänglich sein. Das DBMS bietet die Mittel dafür, dass der Datenbankadministrator entsprechende Zugriffsrechte für jeden Benutzer bzw. für Benutzergruppen festlegen kann.

Weitere Komponenten des DBMS

Meist besitzen Datenbanksysteme noch weitere Komponenten:

- ✓ **Data Dictionary/Repositories**

Das Data Dictionary (Datenlexikon, -wörterbuch; auch als Meta-Datenbank oder Katalog bezeichnet) dient der Speicherung von Informationen über die Daten der Datenbank und deren Verwaltung. Es werden darin beispielsweise das Datenbankschema, die Sichten und die Zugriffsrechte auf die Datenbank abgelegt. Der Anwender kann über das Dictionary Informationen über die Datenbank erhalten und Leistungsanalysen durchführen lassen.

In großen DBS werden Repositories verwendet, welche umfangreicher sind als Data Dictionaries. Sie enthalten zusätzlich zu den Informationen des Dictionarys noch Informationen über die Benutzer und die Anwendungsprogramme.

Der Inhalt des Data Dictionarys bzw. des Repository ist stark vom DBS-Hersteller abhängig.

- ✓ **Logbuch**

Datenbanksysteme verfügen über ein Logbuch, in welchem Informationen über die Transaktionsvorgänge verzeichnet sind, wie beispielsweise der Beginn und das Ende der Transaktion und der Zustand der Daten zu Beginn der Transaktion. Treten Systemfehler auf, werden die Informationen des Logbuchs zum Wiederherstellen der Datenbank verwendet.

Größere Datenbanksysteme bieten meist noch zusätzliche Komponenten, die den Anwender bzw. den Anwendungsprogrammierer bei seiner Arbeit unterstützen:

Entwurfswerzeuge zum Datenbankentwurf	Entwurfswerzeuge zum Datenbankentwurf unterstützen den Anwender beim Entwurf der Datenbank, sodass er nicht auf die Anwendung der Datendefinitionssprache (DDL) angewiesen ist.
Abfrage-Generatoren	Abfrage-Generatoren ermöglichen dem Anwender das Erzeugen von Datenbankabfragen auch ohne Kenntnisse der Datenbank-Abfragesprache.
Report-Generatoren	Report-Generatoren erzeugen Berichte über Datenbankinhalte in den verschiedensten Formen (z. B. Tabellen mit Kopf- und Fußzeilen und Zwischensummen).
Tools zur Erstellung von Business-Grafiken	Tools zur Erstellung von Business-Grafiken ermöglichen die grafische Darstellung von Daten der Datenbank in Diagramm-Form.
CASE-Werkzeuge	CASE-Werkzeuge (Computer Aided Software Engineering – computergestützter Softwareentwurf) dienen dem Entwurf von Datenbankanwendungen, wobei der Quellcode der Anwendung automatisch generiert wird.
Utilities zur Fehleranalyse	Utilities zur Fehleranalyse helfen dem Anwender, Fehler in der Datenbank-Struktur aufzufinden und zu beseitigen.
Funktionen zur Komprimierung und Reorganisation der Datenbank	Funktionen zur Komprimierung und Reorganisation der Datenbank sind notwendig, wenn häufig Daten gelöscht und geändert wurden, da nicht mehr benötigter Speicherplatz nicht automatisch freigegeben wird. Bei der Ausführung der Funktionen wird die Datenbank reorganisiert und nicht benötigter Speicher freigegeben.
Archivierungsfunktionen	Archivierungsfunktionen werden für das Kopieren und Archivieren von Datenbeständen der Datenbank eingesetzt.

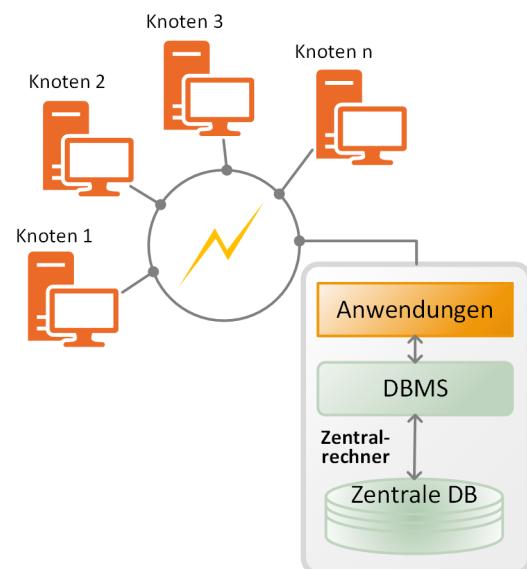
1.4 Physische Datenbankarchitektur

Die physischen Konzepte der Architektur von DBS ergeben sich aus dem logischen Konzept der Datenbank-Entwicklung (3-Ebenen-Architektur) in Verbindung mit der Rechnerumgebung. Dabei werden verschiedene Betrachtungsweisen angewendet. Zum einen werden **zentralisierte DBS** und **verteilte DBS** unterschieden, bei denen die lokale Anordnung der Datenbanken und der Rechner ausschlaggebend sind. Zum anderen führt die zentrale Speicherung der Daten auf einem Rechner (dem Server), von dem aus die Arbeitsplatzrechner (die Clients) auf die Daten zugreifen können, zum **Client/Server-Konzept**. Ein anderer Ansatz ist der Einsatz von Parallelrechnern und Multiprozessorsystemen, der auch von DBS genutzt werden kann (**parallele DBS**).

Zentralisierte DBS

In einem zentralisierten DBS werden das gesamte DBMS und die Anwendungen auf einem Rechner abgelegt, der als zentraler Verwaltungsrechner bzw. Zentralrechner (auch Host oder Mainframe) bezeichnet wird. An den anderen Standorten befinden sich „dumme“ Terminals, die nur der Ein- und Ausgabe dienen (wenig eigene Funktionalität). Von diesen Terminals aus haben alle Benutzer die gleichen Sichten auf die Datenbank, die von den auf dem Zentralrechner laufenden Anwendungen erzeugt wird.

Die Datenbank eines zentralisierten DBS ist im Vergleich zu verteilten Datenbanken relativ einfach zu administrieren. Bezuglich Antwortzeiten und Ausfallsicherheit kann es aber auf dem Zentralrechner zu Problemen kommen.



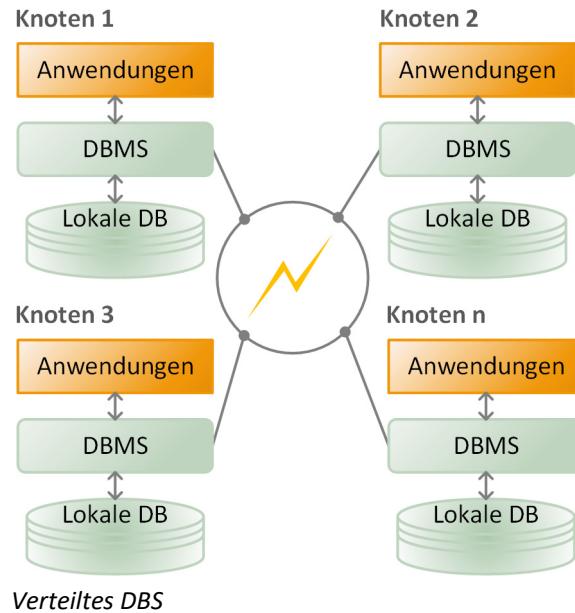
Netz mit Terminals und zentralisierter Datenbank

In modernen Rechnernetzen werden als Endgeräte „intelligente“ Arbeitsplatzrechner eingesetzt. Die Datenbankanwendungen und die Client-Software der DBS können zusammen auf diesem Rechner laufen. Damit kann dem zentralisierten DBS ein Teil der Arbeit abgenommen werden, indem dort z. B. die Syntaxprüfung und die Optimierung der Abfragen durchgeführt werden.

Verteilte DBS

Um die Vorteile vernetzter Rechnersysteme (z. B. schneller Datenaustausch ohne externe Datenträger), die geografisch weit voneinander entfernt sein können, auch auf dem Gebiet der Datenbanken effektiv ausnutzen zu können, wurden und werden neue Methoden gesucht.

Verteilte Datenbanken sind eine Menge von mehreren logisch zusammengehörigen (Teil-) Datenbanken, die in einem Netz auf mehreren lokal getrennten Computern (z. B. in verschiedenen Städten) gespeichert sind. Ein **verteiltes DBMS** besitzt Mechanismen zur Zusammenführung und Abfrage der verteilten Datenbanken. Durch das DBMS wird die Verteilung der Daten vor dem Anwender verborgen. Dem Benutzer erscheint es wie ein zentralisiertes DBS, da er nur auf der externen Ebene arbeitet.



Beispielsweise besitzen die Filialen einer Bank jeweils die Kundendaten ihrer Kunden, die Zentrale kann aber auf alle Kundendaten zugreifen.

Verteilte DBS haben im Vergleich zu zentralisierten DBS entscheidende Vorteile:

Lokale Autonomie	Lokale Autonomie ermöglicht effektivere Anfragen, da die Daten dort gespeichert sind, wo sie gebraucht werden (besonders bedeutsam bei Unternehmen mit dezentraler Unternehmensstruktur).
Zuverlässigkeit und Verfügbarkeit	Die Verfügbarkeit wird verbessert, da der Ausfall eines Knotens nicht zum Ausfall des gesamten Systems führt. Gezielte Redundanz erhöht die Zuverlässigkeit.
Leistung	Die Leistung wird durch Parallelarbeit an verschiedenen Orten erhöht. Zugriffe können gleichzeitig durchgeführt und die Zugriffsposition genauer festgelegt werden, da die lokalen Datenbanken kleiner sind.
Erweiterbarkeit	Eine Erweiterbarkeit des Systems, wie z. B. das Hinzufügen eines neuen Knotens, wird auf relativ einfachem Wege ermöglicht.

Die Anwendung verteilter DBS bringt aber auch Nachteile mit sich:

Komplexität	Die Komplexität der Aufgaben (Synchronisation, Bearbeitung von Anfragen usw.) ist fast immer sehr hoch.
Dezentrale Verwaltung	Eine dezentrale Verwaltung bringt zusätzlichen Aufwand für die Verwaltung und die Synchronisation mit sich.
Sicherheit	Die Sicherheit ist zu gewährleisten, d. h. sowohl die Datensicherheit der lokalen Datenbanken als auch die Sicherheit im Netz (z. B. bei Datenübertragungen, Zugriffen auf Daten usw.).

Kosten	Es entstehen Kosten vor allem für die Software und die Kommunikation.
Übergang von zentralisierten auf verteilte DBS	Der Übergang von zentralisierten auf verteilte DBS verursacht Kosten durch den Umstieg auf neue Software. Auch im Bereich der Hardware können Kosten entstehen, z. B. für eine neue Kommunikationsinfrastruktur. Häufig ist neues Personal bzw. die Schulung des vorhandenen Personals erforderlich. Es werden gegebenenfalls Werkzeuge zur Überführung der vorhandenen Datenbanken in das neue System benötigt.

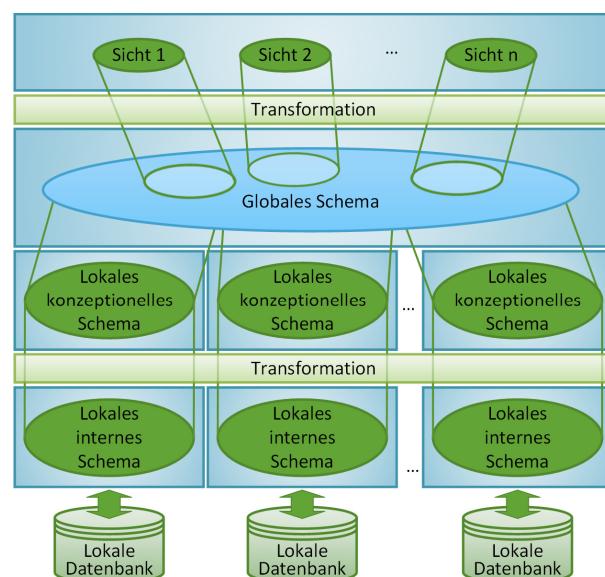
Die Verteilung eines Datenbanksystems kann unter verschiedenen Aspekten erfolgen:

- ✓ Ein DBS kann **homogen** verteilt sein, d. h., logisch zusammengehörige Datenbanken werden auf verschiedene Orte verteilt. Sie werden von derselben Datenbanksoftware verwaltet.
- ✓ Sollen unterschiedliche Datenbanken, z. B. aus verschiedenen Unternehmen, Unternehmensteilen oder Abteilungen, zusammen verwaltet und soll auf die Daten mehrerer unterschiedlicher Datenbanken gleichzeitig zugegriffen werden, ist das DBS **heterogen** verteilt. Die Datenbanken können auch verschiedene Datenmodelle besitzen. Heterogen verteilte DBS werden auch als Multidatenbanksysteme (MDBS) bezeichnet.
- ✓ Ein weiterer Punkt, nach dem eine Unterscheidung von verteilten DBS möglich ist, ist die Verwendung replizierter Daten (dieselben Daten befinden sich an verschiedenen Stellen und werden laufend abgeglichen).

Das 3-Ebenen-Modell muss für verteilte DBS erweitert werden. Die Aufteilung der Datenbanken auf die einzelnen Rechner muss darin deutlich werden, ebenso wie der Unterschied von homogen oder heterogen verteilten Datenbanken. Die Änderungen betreffen vor allem die konzeptionelle Ebene. Auf externer Ebene ist kein Unterschied zu zentralisierten DBS zu erkennen.

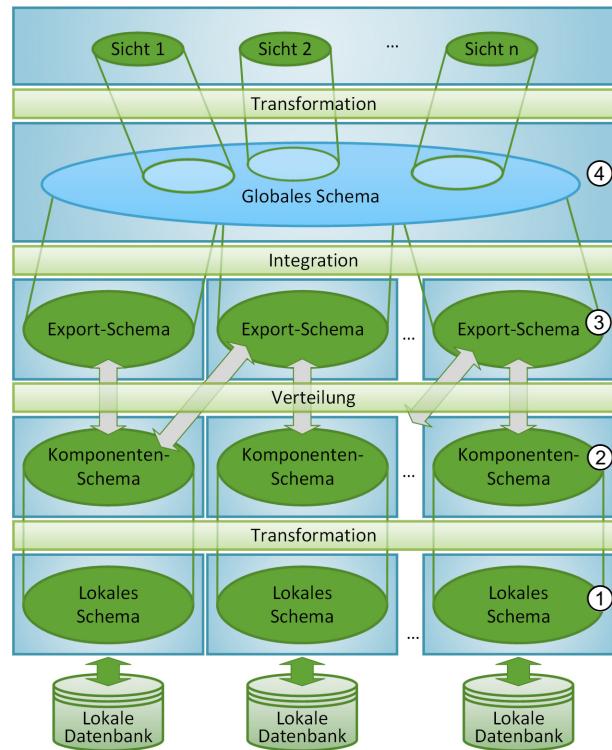
✓ Homogene Verteilung

In der konzeptionellen Ebene findet eine Aufteilung des konzeptionellen Schemas statt. Es wird ein gemeinsames konzeptionelles Schema für die gesamte Datenbank erstellt, auf dem die externen Sichten beruhen. Das Gesamtschema wird in lokale konzeptionelle Schemen unterteilt, für jede lokale Datenbank ein Schema. Für jedes lokale konzeptionelle Schema existiert dann ein lokales internes Schema.



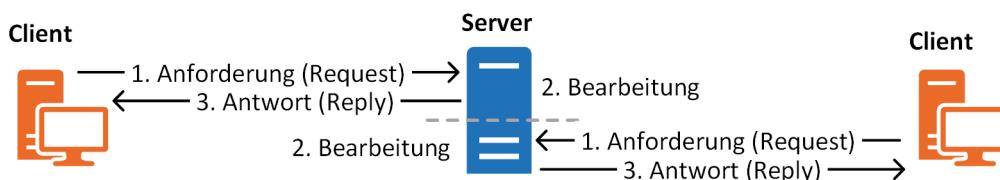
✓ **Heterogene Verteilung**

Die Zusammenführung mehrerer unterschiedlicher Datenbanken bringt einen höheren Aufwand mit sich. Die Daten aus den lokalen Schemen ① müssen zunächst in ein gemeinsames Datenbankschema transformiert werden. Ein transformiertes lokales Schema wird im Komponenten-Schema ② gespeichert. Die Daten aus den Komponenten-Schemen werden aufgeteilt und in den Exportschemen ③ abgelegt. Ein Exportschema kann sich aus Teilen eines oder mehrerer Komponenten-Schemen zusammensetzen. Aus einem Komponenten-Schema können aber auch mehrere Exportschemen erzeugt werden. Die Exportschemen werden wiederum im gemeinsamen globalen Schema ④ abgebildet.



Client-Server DBS

Die meisten heutigen DBS arbeiten nach dem Client-Server-Konzept. Es realisiert ein funktional verteiltes System, in dem zwei unabhängige Prozesse über eine definierte Schnittstelle miteinander kommunizieren (als Prozess wird hier ein eigenständig laufendes Programm verstanden) – der Server und der Client. Die Kommunikation erfolgt über einen Anforderung-Antwort-Dialog. Der Client stellt eine Anforderung an den Server, der Server bearbeitet die Anforderung und gibt die gewünschte Antwort an den Client zurück. Der Server stellt die Dienstleistungen zur Verfügung und der Client nimmt sie in Anspruch. Häufig werden die Dienste eines Servers von mehreren Clients – auch gleichzeitig – genutzt.



Die Initiative geht hierbei immer vom Client aus, der Server verharrt so lange in Warteposition, bis ein Client seine Anforderungen sendet.

- ✓ Client und Server können sich physisch sowohl auf dem gleichen Rechner befinden als auch auf verschiedenen Rechnern, die über ein Netzwerk verbunden sind.
- ✓ Ein Rechner (bzw. ein Programm) kann sowohl als Client als auch als Server arbeiten. Es ist von der momentanen Tätigkeit abhängig, ob gerade ein Dienst in Anspruch genommen oder bereitgestellt wird.

Bezogen auf ein DBS ist die einfachste Form eines Client-Server-Systems die, dass ein Datenbank-Anwendungsprogramm die Dienste eines Datenbank-Servers in Anspruch nimmt, z. B. durch eine Datenbankanfrage. Der Server führt die Anfrage über die Datenbank aus und sendet die selektierte Datenmenge an den Client, also das Anwendungsprogramm, zurück.

Bei relationalen DBS werden diese Abfragen meist in der Datenbank-Abfragesprache SQL formuliert. Der Server wird entsprechend als SQL-Server bezeichnet. Der SQL-Server (entspricht dem DBMS oder einem Teil davon) verwaltet die Datenbank und bearbeitet die SQL-Anfragen der Clients. Der Client ist beispielsweise eine Datenbankanwendung, welche die Anzeige und Bearbeitung der Daten ermöglicht, die ihm der SQL-Server geliefert hat.

Parallele DBS

Parallele Datenbanksysteme laufen auf Multiprozessorsystemen oder Parallelrechnern und nutzen gleichzeitig die Leistung mehrerer Prozessoren. Damit werden eine Leistungssteigerung und eine Verkürzung der Bearbeitungszeit bei Datenbankanfragen und Transaktionen erreicht. Bei großen Datenbanken mit vielen Benutzern verursacht die sequenzielle Verarbeitung von Abfragen zum Teil inakzeptable Antwortzeiten. Besonders beim Einsatz von objektorientierten Datenbanksystemen, die häufig mit komplex strukturierten Objekten arbeiten, ist ein erhöhter Aufwand an Rechenzeit notwendig, der Parallelverarbeitung erforderlich macht.

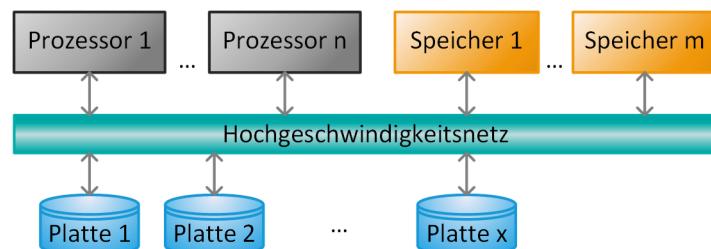
Unter einem parallelen DBS ist nicht zu verstehen, dass mehrere Benutzer gleichzeitig Anfragen an ein DBMS richten können und diese Anfragen dann zeitlich versetzt, aber gewissermaßen parallel abgearbeitet werden. Diese Arbeitsweise ist auch bei sequenziell arbeitenden DBS üblich. Eine echte Parallelarbeit wird durch den Einsatz von Parallelrechnern oder durch die Anwendung paralleler Algorithmen erreicht, die gleichzeitig auf mehreren Prozessoren ausgeführt werden.

In einem parallelen System sind mehrere Prozessoren, Platten- und Hauptspeicher über eine sehr schnelle Leitung (Hochgeschwindigkeitsnetz) miteinander verbunden. Die Arbeitsweise paralleler Datenbanksysteme hängt von der konkreten Rechnerarchitektur ab, die grundsätzliche Arbeitsweise ist aber die gleiche:

- ✓ Die Daten werden auf die verfügbaren Platten verteilt.
- ✓ Datenbankabfragen und Transaktionen werden so zerlegt, dass sie auf mehreren Prozessoren gleichzeitig abgearbeitet werden können.

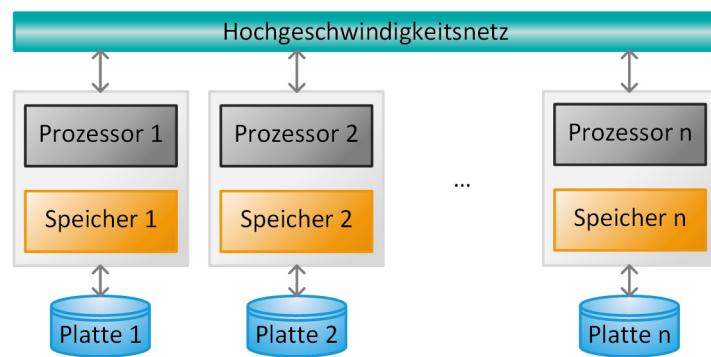
Prinzipiell gibt es drei verschiedene Architekturentypen für parallele Systeme:

- ✓ **Shared-Memory-Architektur** (Shared-Everything-Architektur)
Alle Prozessoren des Systems können auf den gemeinsamen Speicher zugreifen und über diesen kommunizieren.
Die für die Ausführung einer Datenbankoperation benötigten Daten werden von den ausführenden Prozessoren über das Netzwerk angefordert und im Speicher bereitgestellt.



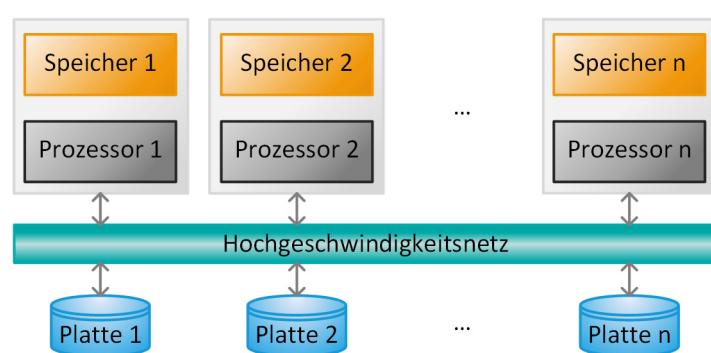
✓ **Shared-Nothing-Architektur**

Jedem Prozessor sind eigene Speichermedien (Haupt- und Plattspeicher) zugeordnet, auf die er exklusiv zugreift. Bei diesem System verfügt jeder Prozessor über eine Kopie des DBMS. Eine wichtige Arbeitsweise dieser Architekturform ist das Weitergeben von Funktionen an einen anderen nicht ausgelasteten Prozessor.



✓ **Shared-Disk-Architektur**

Die Hauptspeicher sind den Prozessoren lokal zugeordnet, die Plattspeicher werden aber gemeinsam genutzt. Die Arbeitsweise ist ähnlich wie bei der Shared-Nothing-Architektur. Die Daten werden aber wie bei der Shared-Memory-Architektur über das Netzwerk angefordert.



1.5 Übung

Fragen zur Datenbanktheorie

Übungsdatei: --

Ergebnisdatei: [Antworten.pdf](#)

1. Nennen Sie wichtige Gründe, die zur Entwicklung von Datenbanksystemen führten.
2. Welche Datenbankmodelle kennen Sie? Wodurch sind sie gekennzeichnet?
3. Nennen Sie die Namen der 3 Ebenen des 3-Ebenen-Modells und geben Sie an, was in jeder Ebene dargestellt wird.
4. Was ist ein Datenbankmanagementsystem? Welche Aufgaben hat es?
5. Was ist ein Data Dictionary und wozu wird es benötigt?
6. Welche physischen Datenbankarchitekturen kennen Sie? Erläutern Sie jeweils kurz den Aufbau.

2

Der Datenbankentwurf

2.1 Einführung zum Datenbankentwurf

Wird bei der Konzeption von Anwendungsprogrammen festgestellt, dass für die Verwaltung der Daten eine Datenbank benötigt wird, beginnt deren Planungsprozess, der sogenannte Datenbankentwurf. Dabei handelt es sich um einen Prozess, bei dem genau festgestellt wird, welche Daten von der Anwendung benötigt werden und ob andere Anwendungen ebenfalls mit diesen und eventuell anderen Daten aus dem Bereich arbeiten sollen. Dabei werden folgende Aspekte betrachtet:

- ✓ Welche **logische Struktur** soll die Datenbank haben, d. h., welche Sichten werden auf die Datenbank benötigt und wie können diese in einem gemeinsamen Schema zusammengefasst werden?
- ✓ Wie soll die **physische Struktur** der Datenbank aussehen, d. h., in welcher Form werden die Daten gespeichert und wie soll darauf zugegriffen werden?
- ✓ Welche **zusätzlichen Bedingungen** müssen eingehalten werden, d. h., gibt es vonseiten der Anwendungen Bedingungen, Einschränkungen usw.?

Der Entwurf erfolgt nach dem 3-Ebenen-Modell. Das heißt, es müssen die Schemata für die externe, die konzeptionelle und die interne Ebene aufgestellt werden. Für den Entwurfsprozess kommen spezielle Techniken zum Einsatz. Die grafische Darstellung der aufgestellten Modelle erfolgt mit dem Entity-Relationship-Modell, das sich für den Datenbankentwurf durchgesetzt hat.

2.2 Der Datenbank-Lebenszyklus

Bei der Entwicklung und dem Einsatz von Software werden die verschiedenen Phasen wie z. B. Analyse, Planung, Entwicklung, Testen und Anwendung von Software unterschieden und unter dem Begriff des Software-Lebenszyklus zusammengefasst. Diese Einteilung in Entwicklungsphasen kann ebenfalls auf dem Gebiet der Datenbanken angewendet werden.

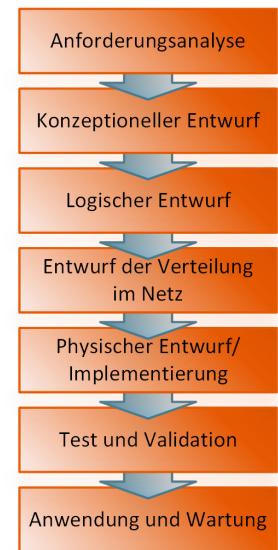
Die **Analyse der Anforderungen** grenzt die Inhalte der neuen Datenbank ein und dient der Festlegung der Benutzergruppen und Anwendungen. Dabei werden die Datenobjekte, deren Eigenschaften und Beziehungen sowie mögliche Vorgänge (Aktualisierungen, Abfragen) und Randbedingungen ermittelt. Das Resultat der Anforderungsanalyse ist die Anforderungsspezifikation.

Der **konzeptionelle Entwurf** umfasst die Modellierung der Sichten und die Integration der Sichten in ein Gesamtschema. Dafür werden meist Entity-Relationship-Diagramme erstellt. Auf diesem Gebiet werden aber auch UML-Diagramme eingesetzt (UML – **Unified Modeling Language**).

Diese grafischen Darstellungen werden in der Phase des **logischen Entwurfs** in das Datenmodell des Ziel-DBS (z. B. in das relationale Datenmodell) transformiert und die gesamte Datenbank wird so aufbereitet, dass eine effektive Speicherung möglich ist (das Datenbankschema wird normalisiert).

Bei verteilten Datenbanken ist ein **Entwurf für die Verteilung** der Datenbanken im Netz erforderlich.

Nun kann die Datenbank mithilfe der Sprachmittel des DBMS erstellt und die benötigten Anfragen können formuliert werden, was als **physischer Entwurf** bzw. **Implementierung** bezeichnet wird. Für relationale DBS geschieht dies in der Abfragesprache SQL. Im Falle einer Übernahme von Daten aus alten Datenbanken oder von Dateiinhalten erfolgt an dieser Stelle eine Konvertierung dieser Daten in das neue Format.



Der Datenbank-Lebenszyklus

Die Datenbank und die erstellten Abfragen werden nun **getestet** und die Ergebnisse werden auf ihre Gültigkeit bezüglich der Anforderungen geprüft (**validiert**). So wird die Sicherung der Datenbank-Qualität gewährleistet.

In der Phase der **Anwendung** muss die Datenbank ständig **gewartet** werden. Im Laufe dieser Zeit können sich Änderungen des Datenbankschemas ergeben. In diesem Fall ist eine **Reorganisation** der Datenbank notwendig.

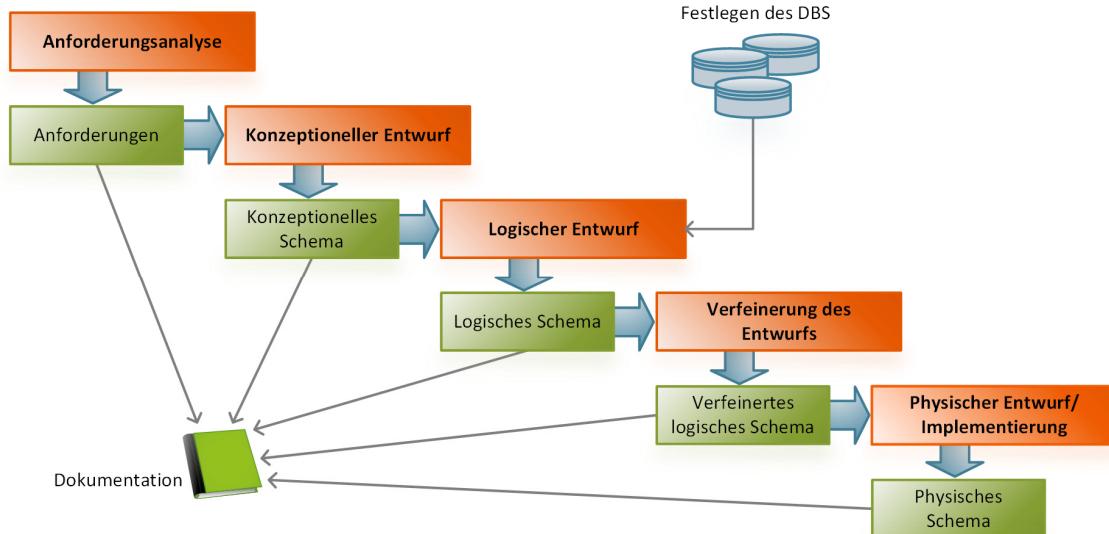
2.3 Datenbanken entwerfen

Der Entwurf der Datenbank hat einen großen Anteil an der Qualität der Datenbank. Der konzeptionelle Entwurf ist mit besonderer Sorgfalt zu erstellen. Darin werden die Sichten definiert und im konzeptionellen Schema zusammengeführt. Bei der Modellierung des Ausschnitts aus der realen Welt (des konzeptionellen Schemas) ist besonders auf **Vollständigkeit** und **Korrektheit**, **Minimalität** und **Modifizierbarkeit** zu achten.

Die Entwurfsphasen

Der Entwurf der Datenbank beginnt bei der Analyse der Anforderungen und ist mit dem physischen Entwurf der Datenbank abgeschlossen, wie es in der Abbildung des Datenbank-Lebenszyklus dargestellt ist.

Anforderungsanalyse	In der Anforderungsanalyse werden die Anforderungen aller Benutzer an die neue Datenbank zusammengetragen. Diese Anforderungen werden meist nach bestimmten Kriterien klassifiziert, z. B. nach Abteilungen bzw. Benutzergruppen. Wichtig ist, dass festgelegt wird, welche Daten gespeichert werden sollen (was zu speichern ist) und wie die Daten zu bearbeiten sind.
Konzeptioneller Entwurf	Am Ende dieser Phase liegen die Sichten und das konzeptionelle Gesamtschema (meist als Entity-Relationship-Diagramm) vor. Beim Entwurf können Sie verschiedene Vorgehensweisen verwenden. Entweder Sie entwerfen zuerst die Sichten und fügen diese dann zu einem konzeptionellen Schema zusammen (Top-down-Methode – Methode der schrittweisen Verfeinerung) oder umgekehrt (Bottom-up-Methode – Methode der schrittweisen Verallgemeinerung). In den Ergebnisdiagrammen ist genau definiert, welche Daten(-objekte) mit welchen Eigenschaften in der Datenbank abgebildet werden sollen, welche Beziehungen zwischen den Daten(-objekten) bestehen, ob es Abhängigkeiten oder/und Integritätsbedingungen gibt usw. Bevor der logische Entwurf erstellt werden kann, muss festgelegt werden, für welches DBS die Datenbank aufgebaut werden soll.
Logischer Entwurf	Nun erfolgt die Umsetzung des konzeptionellen Schemas in das Datenmodell des Datenbanksystems. Dafür stehen meist entsprechende Transformationsregeln zur Verfügung (diese sind nicht zu verwechseln mit denen des DBMS zur Transformation der Schemen der 3 Ebenen). Anschließend wird das Datenbankschema normalisiert, wodurch z. B. Redundanzen beseitigt werden.
Verfeinerung des logischen Entwurfs	Nun kann der logische Entwurf z. B. in Hinblick auf häufige bzw. bevorzugte Abfragen, die in den Anforderungen formuliert wurden, optimiert werden. Dabei werden Erweiterungen und gegebenenfalls Änderungen am relationalen Schema durchgeführt (z. B. durch das Einfügen von Indizes).
Physischer Entwurf/Implementierung	In der letzten Entwurfsphase erfolgt die Definition des internen Schemas. Es werden geeignete Speicherstrukturen und Zugriffsmechanismen darauf festgelegt. Ein wichtiger Aspekt ist auch das Laufzeitverhalten des DBS, welches durch einen effizienten Zugriff auf die relevanten Daten verbessert werden kann. In der Datendefinitionssprache (DDL) des DBS werden nun das interne, das konzeptionelle und das externe Schema implementiert. Bei relationalen Datenbanksystemen werden auch die Relationen und Views (Sichten) definiert. Die Festlegung der Zugriffsrechte erfolgt ebenfalls in dieser Phase.



Datenbank-Entwurfsphasen (Wasserfallmodell)

Abstraktionskonzepte

Bei der Erstellung eines Datenmodells werden die Objekte und deren Eigenschaften untersucht. Es werden zuerst alle Daten (Objekte) gesammelt. In einem Prozess der Abstraktion werden dann gleichartige Mengen von Objekten zusammengefasst und auf relevante Eigenschaften untersucht. In der Informatik existieren bestimmte Konzepte, nach denen dieser Abstraktionsprozess erfolgt.

In den folgenden Definitionen werden die Begriffe Objekt und Klassen verwendet. Mit einer Klasse ist eine Menge gleichartiger Objekte gemeint. Ein Objekt ist ein bestimmtes Element der Klasse. Beispielsweise werden in der Klasse *Autos* alle in Deutschland zugelassenen Autos zusammengefasst. Ein ganz spezielles Auto, z. B. Ihr Auto, ist ein Objekt der Klasse *Autos*.

✓ Klassifikation

Gleichartige Dinge (Objekte) mit gemeinsamen Eigenschaften werden zu Klassen zusammengefasst.

✓ Aggregation

Eine neue Klasse wird aus anderen, bereits existierenden Klassen zusammengesetzt bzw. besteht zum Teil aus Objekten anderer Klassen.

✓ Generalisierung (Verallgemeinerung)

Zwischen bestimmten Klassen wird eine Teilmengenbeziehung hergestellt. Dabei stellt eine Klasse eine Verallgemeinerung der anderen Klasse dar. Zum Beispiel ist die Klasse *Tier* eine Verallgemeinerung der Klassen *Vögel*, *Reptilien* und *Säugetiere*. Die Eigenschaften der verallgemeinerten Klasse werden den Klassen vererbt, die Teilmengen dieser Klasse sind.

✓ Assoziation

Objekte bzw. Klassen können miteinander in Beziehung gesetzt (assoziiert) werden. Diese Beziehung kann zwischen zwei oder mehr Objekten aufgebaut werden.

✓ Identifikation

Eigenschaftswerte bzw. Kombinationen von Eigenschaftswerten der Objekte werden als Schlüssel definiert und dienen der eindeutigen Identifizierung des Objekts. Über diese Schlüssel werden die Objekte assoziiert.

2.4 Das Entity-Relationship-Modell

Das Entity-Relationship-Modell (kurz: ER-Modell oder ERM) ist das bekannteste und meistverwendete grafische Hilfsmittel für den Datenbankentwurf, wird aber auch in anderen Bereichen der Informatik eingesetzt, in denen Ausschnitte der realen Welt modelliert werden. Es ist unabhängig von einem bestimmten Datenmodell und unterliegt nicht den Einschränkungen der Datenmodelle, die sich durch deren Implementierung ergeben. Eingeführt wurde das ERM und die mit ihm verbundene Notation im Jahr 1976 durch Peter Chen. Das ER-Modell ermöglicht es, die konzeptionellen Entwürfe einer Datenbank auf leicht verständliche Art grafisch darzustellen und die Abstraktionskonzepte anzuwenden. Eine spezielle Form der Darstellung wird häufig nach ihrem Erfinder auch als Chen-Notation bezeichnet.

Die zwei Grundbausteine des ER-Modells sind die Entities (Entitäten) und die Relationships (Beziehungen). Entities und Relationships haben Attribute. Es können beliebig viele Objekte einer Entitätsmenge und einer Beziehungsmenge existieren.

Zur Darstellung erweiterter Beziehungsformen, welche besonders im Kontext der objekt-relationalen Datenbanken eine wichtige Rolle spielen, gibt es das ERM in einer erweiterten Form als EERM (Erweitertes/Extended ERM). Dieses beinhaltet neben den Standardarten mit den Aggregationen (Part-of-/Teil-von-Beziehungen) und der Generalisierung/Spezialisierung (Is-a-/Ist-ein-Beziehungen) weitere Formen der Beziehungen. Diese ermöglichen es, eine umfangreiche semantische Datenmodellierung durchzuführen.

Elemente und grafische Darstellung des ER-Modells

Entität (Entity), Entitätsmenge (Entity-Set), Entity-Typ

Als **Entität** (Entity) werden unterscheidbare (identifizierbare) Dinge aus der realen Welt bezeichnet. Dies können Personen, Gegenstände, Firmen oder Ähnliches sein. Entitäten unterscheiden sich von einander durch ihre jeweiligen Eigenschaften bzw. Eigenschaftswerte. Eine Entität wird synonym als Objekt bezeichnet.

Der **Entity-Typ** kategorisiert gleichartige Entitäten. Zu einem Entity-Typ gehören Entitäten, die sich durch die gleichen Eigenschaften (Attribute) beschreiben lassen. Für eine Entität e des Entity-Typs E wird die Schreibweise $e \in E$ aus der Mengenlehre verwendet. Bei der Modellierung werden nicht einzelne Entitäten betrachtet, sondern der Entity-Typ. Die nebenstehende Abbildung zeigt die grafische Darstellung von Entity-Typen in Form von Rechtecken.

Eine **Entitätsmenge** (Entity-Set) ist eine Sammlung von gleichartigen Entitäten, d. h. von Entitäten mit gleichen Eigenschaften, aber unterschiedlichen Eigenschaftswerten, zu einem bestimmten Zeitpunkt. Entitäten einer Entitätsmenge gehören zu einem bestimmten Entity-Typ. Entitätsmengen sind zeitlich veränderlich.

<u>Entitäten:</u>	Abteilung Forschung Mitarbeiter Schmidt Projekt 1009
<u>Entitätsmengen:</u>	Alle Abteilungen Alle Mitarbeiter Alle Projekte
<u>Entity-Typen:</u>	
	Abteilung Mitarbeiter Projekte

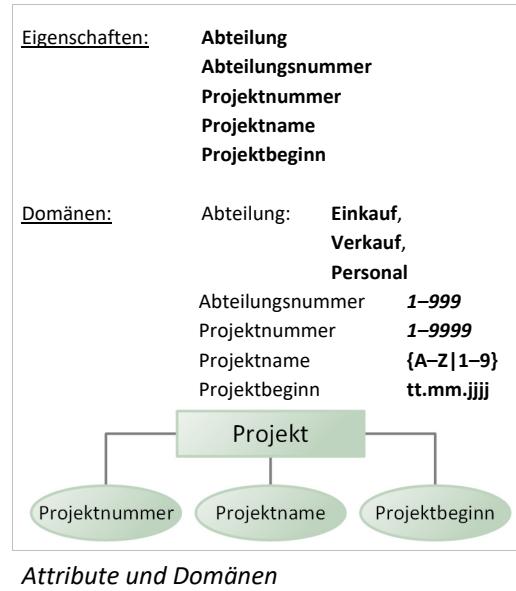
Entität, Entitätsmenge und Entity-Typ

Attribute (Eigenschaften), Domänen

Attribute bzw. Eigenschaften charakterisieren eine Entität, einen Entity-Typ, eine Beziehung bzw. einen Beziehungstyp. Die Attribute besitzen einen Namen und einen Wert (Value).

Eine **Domäne** beschreibt den zulässigen Wertebereich einer Eigenschaft. Das können fest vorgegebene Werte sein (z. B. Januar, Februar ...), Bereiche (z. B. von 0 bis 999, von A bis G) oder Mengenangaben bzw. Datentypangaben (z. B. natürliche Zahl, reelle Zahl, Datum).

In der grafischen Darstellung werden die Eigenschaften als Ellipsen oder Kreise dargestellt. Diese sind über ungerichtete Kanten mit dem Entity-Typ verbunden.



Schreibweisen

Der Entity-Typ *Projekte* kann in Mengenschreibweise dargestellt werden:

$$\text{Projekte} = \{\text{Projektname}, \text{Projektnummer}, \text{Projektbeginn}\}$$

Die Entität e_1 ist ein Projekt und besitzt die Attribute

$$\begin{aligned} \text{Projektname} &= \text{Gehaltsrechnung}, \\ \text{Projektnummer} &= 012 \text{ und} \\ \text{Projektbeginn} &= 03.03.2021. \end{aligned}$$

In der Mengenschreibweise lässt sich das wie folgt darstellen:

$$e_1 = \{\text{Gehaltsrechnung}, 012, 03.03.2021\}$$

Bei den Attributen wird zwischen beschreibenden Attributen (den anwendungsspezifischen Eigenschaften) und identifizierenden Attributen (den Schlüsseln zur eindeutigen Identifikation einer Entität in der Entitätsmenge) unterschieden.

In einigen Datenbanksystemen wird der Begriff Domäne auch für eine Gruppe von Datensätzen in einer Tabelle oder Abfrage verwendet, z. B. in Microsoft Access.

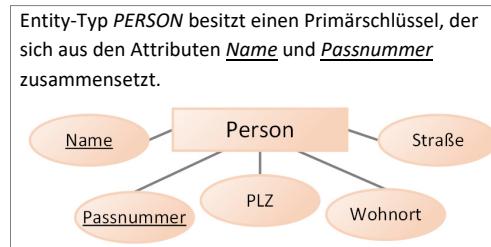
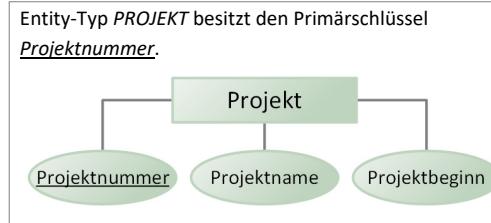
Schlüssel und Primärschlüssel

Ein Schlüssel setzt sich aus einem oder mehreren Attributen zusammen. Schlüssel sollten so kurz wie möglich (Minimalitätsanforderung), aber so lang wie nötig sein. Das ist wichtig, da sie beim späteren Einsatz der Datenbank für die Suche in der Datenbank eingesetzt werden und so jedes zusätzliche Attribut im Schlüssel einen Zeitverlust und mehr Speicherbedarf für den Index bedeutet. Ist unter den vorhandenen Attributen und Attributkombinationen keine, die als Schlüssel eingesetzt werden kann, wird ein künstliches Attribut (z. B. ein Zählfeld) hinzugefügt. Dieses bekommt für jede Entität einen anderen Wert und wird als Schlüssel eingesetzt.

Der **Primärschlüssel** ermöglicht die eindeutige Identifizierung einer Entität einer Entitätsmenge dadurch, dass sein Wert in einer Entitätsmenge nur ein einziges Mal vorkommt. Das Attribut, welches eine Entität eindeutig beschreibt, wird als identifizierendes Attribut bezeichnet. Ist dies über ein Attribut nicht möglich, kann der Primärschlüssel aus mehreren identifizierenden Attributen bestehen.

Ein Entity-Typ kann mehrere Schlüssel besitzen, die für bestimmte Abfragen oder Sortierungen benötigt werden, aber nur **einen** Primärschlüssel.

In der grafischen Darstellung des ER-Modells werden die den Primärschlüssel bildenden Attribute unterstrichen dargestellt.



Der Primärschlüssel

Beziehung (Relationship), Beziehungsmenge und -typ

Durch **Beziehungen** werden die Wechselwirkungen oder Abhängigkeiten von Entitäten ausgedrückt. Beziehungen können Eigenschaften besitzen.

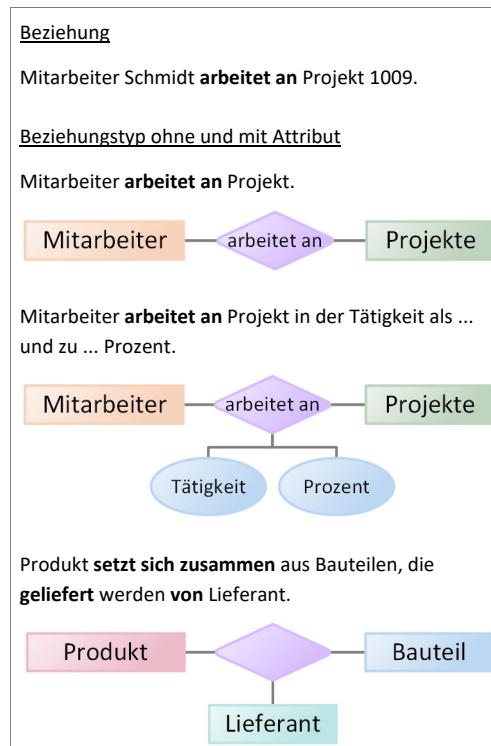
Eine **Beziehungsmenge** (Assoziation) ist eine Sammlung von Beziehungen gleicher Art zur Verknüpfung von Entitätsmengen (Tabellen).

Ein **Beziehungstyp** ist, analog zum Entity-Typ, die Abstraktion gleichartiger Beziehungen.

Ein Beziehungstyp wird grafisch durch eine Raute dargestellt, die durch zwei Kanten mit den Entity-Typen verbunden ist, die assoziiert werden sollen. In der Raute kann der Name des Beziehungstyps stehen.

Beziehungen können durch **Attribute** beschrieben werden, beispielsweise in welcher Tätigkeit ein Mitarbeiter an einem Projekt mitarbeitet und zu wie viel Prozent.

In der Regel stehen zwei Entity-Typen in Beziehung, was durch das Tupel $B(E_1, E_2)$ ausgedrückt wird. Es können aber auch mehrere Entity-Typen assoziiert werden. In dem Tupel $B(E_1, E_2, \dots, E_k)$ werden alle an der Beziehung beteiligten Entity-Typen angegeben. Die Anzahl der an einer Beziehung beteiligten Entitäten wird als **Grad der Beziehung** bezeichnet. Wird beispielsweise ein Produkt aus mehreren Bauteilen verschiedener Lieferanten zusammengesetzt, hat das Tupel die Form $B(Produkt, Bauteil, Lieferant)$. Der Grad der Beziehung ist dann 3 (ternär).



Beziehung und Beziehungstyp

Folgende Beziehungsgrade sind möglich:

- ✓ Binär, wenn genau zwei Entities miteinander verbunden sind
- ✓ Ternär, wenn genau drei Entities miteinander verbunden sind
- ✓ n-är, wenn mehrere, d. h. genau n Entities miteinander verbunden sind
- ✓ Rekursiv unär, wenn ein Entity mit sich selbst in Beziehung steht

Kardinalität (Komplexität)

Über die **Kardinalität** wird festgelegt, wie viele Entitäten einer Entitätsmenge mit Entitäten einer anderen Entitätsmenge in Beziehung stehen können, z. B. wie viele Mitarbeiter an einem Projekt mitarbeiten. In der Regel erfolgt die Kennzeichnung von Kardinalitäten durch folgende Angaben:

- ✓ 1 – genau eine Zuordnung
- ✓ n, m – eine oder mehrere Zuordnungen

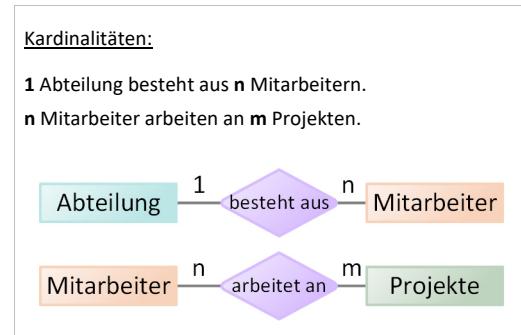
Auf mögliche Erweiterungsformen der Kardinalität geht der Abschnitt „Kardinalität im ER-Modell“ ein.

Daraus ergeben sich folgende Möglichkeiten für die Darstellung der Beziehungen (ohne Erweiterung):

1:1 – Eins-zu-eins-Beziehung	Jede Entität einer Entitätsmenge ist genau einer Entität einer anderen Entitätsmenge zugeordnet.
1:n – Eins-zu-n-Beziehung	Jede Entität einer Entitätsmenge ist einer oder mehreren Entitäten einer anderen Entitätsmenge zugeordnet.
n:m – n-zu-m-Beziehung	Eine oder mehrere Entitäten einer Entitätsmenge können einer oder mehreren Entitäten einer anderen Entitätsmenge zugeordnet werden.

Für die Kardinalität kann auch eine Zahl festgelegt werden, z. B. 2, wenn **immer genau 2** Entitäten einer Entitätsmenge in Beziehung stehen.

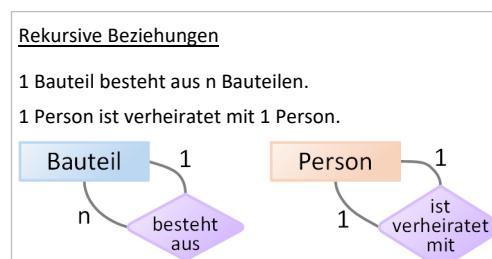
Besteht die Möglichkeit, dass es für einige Entitäten keine Zuordnung in Beziehungen gibt, kann den Angaben 1, n oder m eine 0 hinzugefügt werden. Zum Beispiel bedeutet 0, n, dass 0, 1, 2 ... n Entitäten infrage kommen.



Darstellung von Beziehungen

Spezielle Beziehungsformen

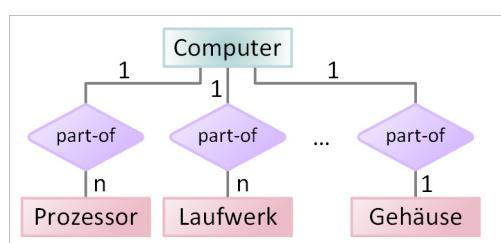
Rekursive Beziehungen entstehen, wenn ein Entity-Typ eine Assoziation auf sich selbst besitzt (rekursiv unäre Beziehung). Beispielsweise kann eine rekursive Beziehung zwischen Bauteilen bestehen, die wiederum aus anderen Bauteilen zusammengesetzt werden, oder zwischen Personen, die mit einer anderen Person verheiratet sind.



Darstellung rekursiver Beziehungen

Zur Darstellung erweiterter Beziehungsformen wurde das Entity-Relationship-Modell um die Aggregation und die Generalisierung/Spezialisierung erweitert.

Das Konzept der **Aggregation** wird durch eine **Part-of-Beziehung** (Ist-Teil-von-Beziehung) ausgedrückt (n-äre Beziehung). Beispielsweise besteht ein Computer aus einem oder mehreren Prozessoren, Laufwerken, einem Gehäuse usw.



Darstellung einer Part-of-Beziehung

Is-a-Beziehungen drücken eine **Generalisierung** (Verallgemeinerung) oder **Spezialisierung** aus, d. h., sie stellen eine Teilmengenbeziehung dar. Beispielsweise können die Entity-Typen *PKW*, *MOTORRAD* und *LKW* in einem Entity-Typ *FAHRZEUG* zusammengefasst (generalisiert) werden. Andererseits sind *PKW*, *MOTORRAD* und *LKW* Spezialisierungen des Entity-Typs *FAHRZEUG*. In der Mengenschreibweise wird das wie folgt ausgedrückt:

PKW ⊆ *FAHRZEUG*
MOTORRAD ⊆ *FAHRZEUG*
LKW ⊆ *FAHRZEUG*

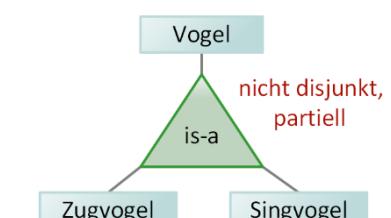
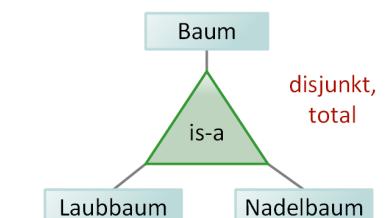
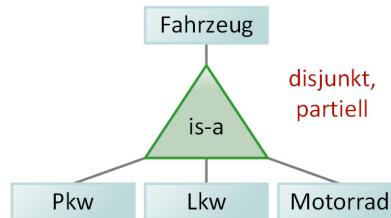
Eine Is-a-Beziehung kann folgende Eigenschaften besitzen:

- ✓ Sie ist entweder disjunkt oder nicht disjunkt.

Disjunkt	Alle Teilmengen sind echte Teilmengen, kein Element der einen Teilmenge kommt in einer anderen Teilmenge vor.
Nicht disjunkt	Die Teilmengen können gemeinsame Elemente enthalten.

- ✓ Sie ist entweder total oder partiell.

Total	Es gibt keine weitere Teilmenge zu dieser Spezialisierung.
Partiell	Es gibt weitere Teilmengen, die aber nicht aufgeführt sind.



Darstellung von Is-a-Beziehungen

Die verwendete Form der Grafik für die Is-a-Beziehung basiert auf dem in der UML verwendeten Pfeil von einer UnterkLASSE (Spezialisierung) zu einer OberKLASSE (Generalisierung). Alternativ kann auch die für allgemeine Beziehungen verwendete Raute genutzt werden.

Überblick über die grafischen Konstrukte des ER-Modells (Chen-Notation)

Entity-Typ (Klassifizierung)	
Schwacher Entity-Typ (durch eigene Attribute nicht eindeutig identifizierbar – immer über Relationship an weiteren Entity-Typ gebunden)	
Attribut (Eigenschaft)	
Primärschlüssel (Identifizierendes Attribut)	
Beziehung (Relation, Relationship, Assoziation)	
Part-of-Beziehung (Ist-Teil-von-Beziehung, Aggregation) (Erweiterte Darstellungsform)	
Is-a-Beziehung (Ist-ein-Beziehung, Spezialisierung oder Generalisierung, Teilmengenbeziehung) (Erweiterte Darstellungsform)	

Kardinalität im ER-Modell

Ein Problem der Kardinalitätsangabe in der Ausgangsform des ER-Modells nach Chen besteht darin, dass nicht ersichtlich wird, ob sich eine konkrete Entität in einer Beziehung befinden muss, ob es sich also um eine Kann- oder eine Muss-Beziehung handelt. Zur Darstellung von Muss-/Kann-Beziehungen existieren alternative Darstellungsformen:

- ✓ die Min-Max-Notation
- ✓ die Martin-Notation (sogenannte Krähenfuss-Notation)
- ✓ UML (Unified Modelling Language)

Die **Min-Max-Notation** verfeinert die Angabe der Kardinalität einer Beziehung durch das Vorgeben einer unteren und einer oberen Grenze für die Anzahl der Beziehungen, an denen eine Entität beteiligt ist.

<p>So kann in einer Beziehung „Person zu bisherigen Urlaubsorten“ die Untergrenze 0 (wenn die Person noch nie im Urlaub war) und die Obergrenze eine beliebig große Zahl sein. Es handelt sich um eine Kann-Beziehung. Umgekehrt können in einem Ort keine, eine oder beliebig viele Personen ihren Urlaub verbracht haben.</p>	<pre> graph LR Person[Person] -- "(0..M)" --> Diamond{machte Urlaub in} Diamond -- "(0..M)" --> Ort[Ort] </pre>
<p>Im Gegensatz dazu besitzt in einer Beziehung „Person zu bisherigen Aufenthaltsorten“ die Untergrenze den Wert 1, da eine existierende Person mindestens einen Aufenthaltsort in ihrem bisherigen Leben hatte. Es handelt sich also um eine Muss-Beziehung. Aus Sicht der Entität 'Ort' ändert sich in dem Beispiel nichts.</p>	<pre> graph LR Person[Person] -- "(1..M)" --> Diamond{hielt sich auf in} Diamond -- "(0..M)" --> Ort[Ort] </pre>

Die Min-Max-Notation ist im Laufe der Jahre in die Chen-Notation übernommen worden (Modifizierte Chen-Notation, MC-Notation). In der erweiterten Form sind die folgenden zusätzlichen Angaben zulässig:

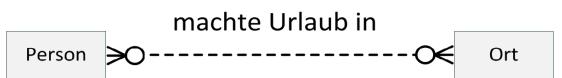
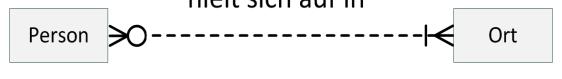
- ✓ c – keine oder eine Zuordnung (can, Kann-Beziehung)
- ✓ m – eine oder mehrere Zuordnungen (must, Muss-Beziehung)

Entsprechend existieren hier neben den bereits vorgestellten Beziehungen 1:1, 1:n und n:m weitere:

1:c – Eins-zu-höchstens-eins-Beziehung	<p>Jede Entität einer ersten Entitätsmenge kann keiner oder genau einer Entität einer zweiten Entitätsmenge zugeordnet werden.</p>
1:m – Eins-zu-mindestens-eins-Beziehung	<p>Jede Entität einer ersten Entitätsmenge ist mindestens einer Entität einer zweiten Entitätsmenge zugeordnet.</p>
1:mc – Eins-zu-beliebig-viele-Beziehungen	<p>Eine Entität einer ersten Entitätsmenge kann keiner, einer oder mehreren Entitäten einer zweiten Entitätsmenge zugeordnet werden.</p>

Die nach ihrem Erfinder James Martin genannte **Martin-Notation** wird auf Grund ihrer Darstellung der Mehrfachbeziehungen 1:n auch als **Krähenfuß-Notation** bezeichnet. Grafisch werden die Kardinalitäten durch die Angabe von 0 für keine, den senkrechten Strich | für eine und den Krähenfuß < für beliebig viele dargestellt.

Wie in der Chen-Notation werden die Entitäten als Rechtecke abgebildet. Die Beziehungen werden jedoch nicht durch Rauten sondern durch beschriftete Beziehungslinien abgebildet, an deren Ende die Symbole für die Kardinalität stehen. Attribute werden im Rechteck der Entität aufgelistet, der Primärschlüssel wird dabei über die Kennzeichnung PK markiert.

Die Beziehung Person/bisherige Urlaubsorte stellt sich in der Krähenfuß-Notation wie folgt dar:	
Hinsichtlich der Beziehung Person/bisherige Aufenthaltsorte ändert sich die rechte Seite der Beziehungsdarstellung entsprechend:	

UML (Unified Modeling Language) ist eine einheitliche grafische Modellierungssprache. Sie kommt heute bei fast allen Softwareentwicklungsprozessen zum Einsatz, da sie stark auf die Belange der aktuell vorwiegend zum Einsatz kommenden objektorientierten Programmierung zugeschnitten ist. Das HERDT Buch *Objektorientierter Softwareentwurf mit UML* führt ausführlich in die Thematik ein.

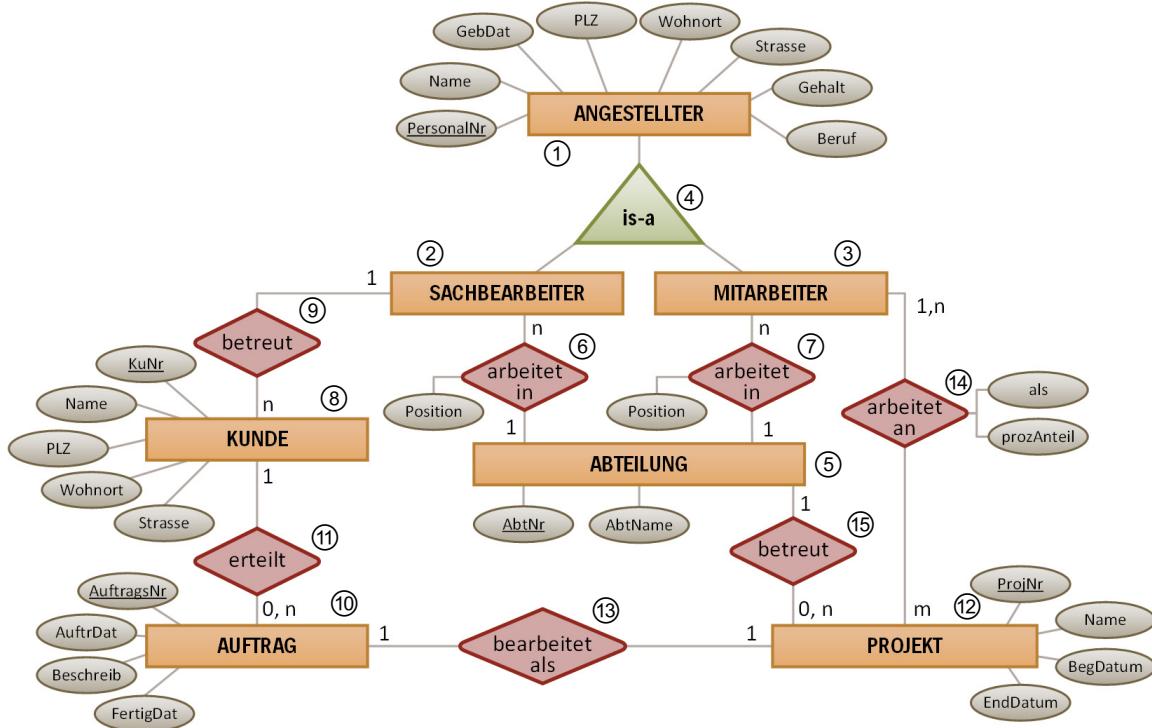
Datenbankentwurf mit dem ER-Modell

Ist die Anforderungsanalyse abgeschlossen, können Sie das konzeptionelle Schema der Datenbank mit dem ER-Modell entwerfen. Dabei kann nach der Top-down- oder nach der Bottom-up-Methode vorgegangen werden. Bei der Top-down-Methode wird das grob entworfene Modell schrittweise verfeinert. Diese Verfeinerung wird auf die Entitäten, die Attribute und Beziehungen angewendet. Dabei werden beispielsweise Entity-Typen und Beziehungstypen zerlegt bzw. zusammengefügt, Spezialisierungen bzw. Generalisierungen und Aggregationen aufgebaut und Attribute sowie Schlüssel festgelegt.

Die folgenden Beispiele zeigen die Verwendung des ER-Modells im Datenbankentwurf.

Beispiel: Kunden- und Projektverwaltung

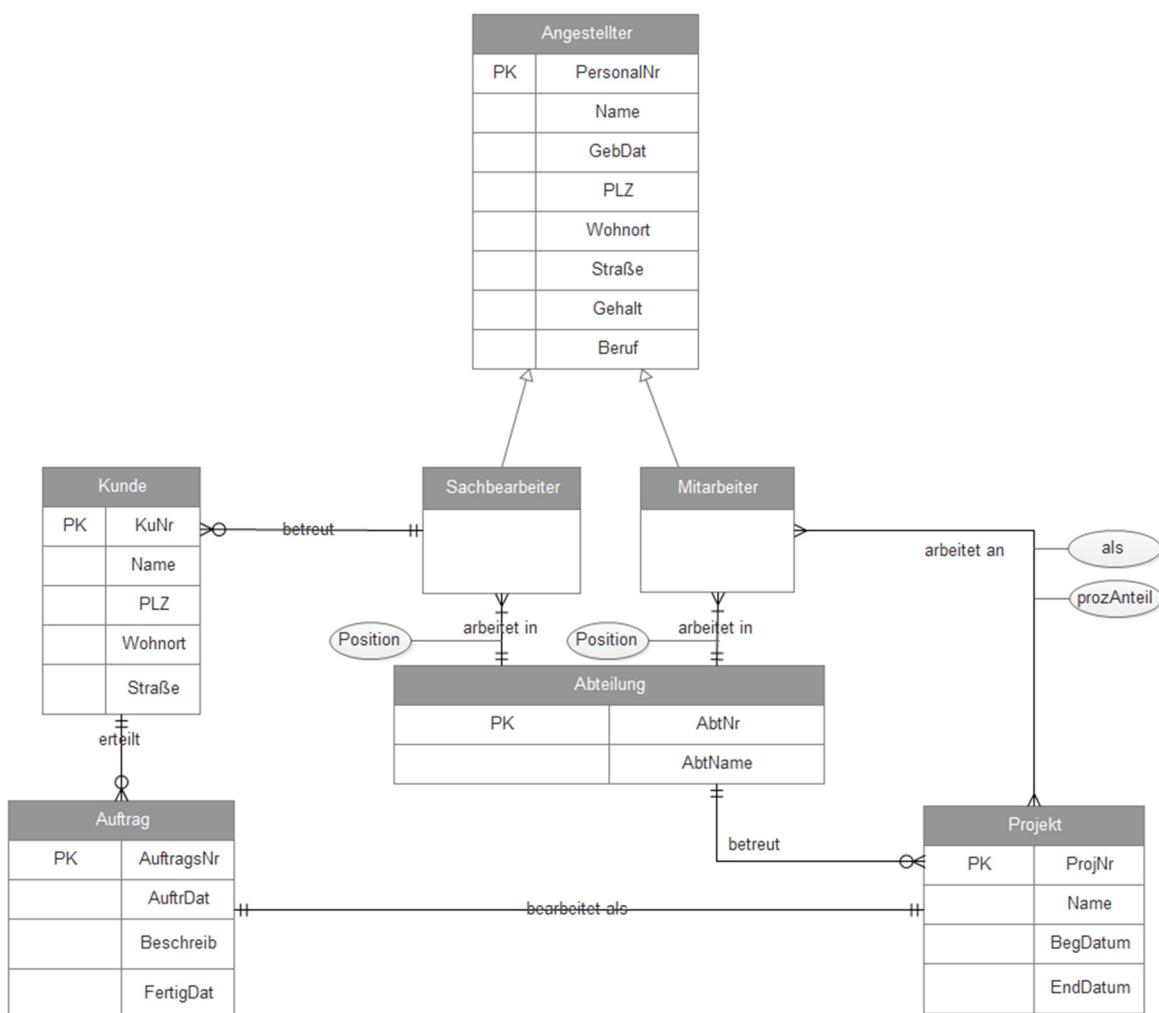
In einem Unternehmen (z. B. einer Softwarefirma) sind Sachbearbeiter und Mitarbeiter angestellt. Die Sachbearbeiter betreuen die Kunden und nehmen deren Aufträge entgegen. Der jeweilige Auftrag wird als Projekt bearbeitet und von einer Abteilung betreut. An dem Projekt arbeiten mehrere Mitarbeiter. Zur Verwaltung aller Informationen und Ereignisse soll eine Datenbank erstellt werden. In der folgenden Abbildung sind alle Elemente der Datenbank und deren Beziehungen untereinander zusammengestellt.



- ① Alle Informationen zu den Angestellten sind in dem Entity-Typ **ANGESTELLTER** abgelegt. Da Name und Geburtsdatum allein einen Angestellten nicht eindeutig identifizieren können, wird als identifizierendes Attribut und Primärschlüssel ein künstliches Attribut **PersonalNr** eingeführt.
- ②–④ Der Entity-Typ **ANGESTELLTER** lässt sich in zwei Teilmengen unterteilen, die **MITARBEITER** und die **SACHBEARBEITER**. Diese Spezialisierung wird über eine Is-a-Beziehung ausgedrückt.
- ⑤ Die **MITARBEITER** und die **SACHBEARBEITER** sind jeweils **einer** Abteilung zugeordnet. Der Entity-Typ **ABTEILUNG** wird durch das Attribut **AbtNr** (Abteilungsnummer) identifiziert.
- ⑥, ⑦ Die Assoziation zwischen den Entity-Typen **MITARBEITER** bzw. **ANGESTELLTER** und dem Entity-Typ **ABTEILUNG** kann durch eine 1:n-Beziehung dargestellt werden. In einer Abteilung können n Angestellte arbeiten. Die Beziehung besitzt das Attribut **Position**, welches die Stellung des Angestellten in der Abteilung ausdrückt.
- ⑧ Alle Kunden der Firma werden in dem Entity-Typ **KUNDE** zusammengefasst. Jeder Kunde hat eine Kundennummer (**KuNr**), die ihn identifiziert.
- ⑨ Von einem Sachbearbeiter werden mehrere Kunden betreut, was durch die 1:n-Beziehung zum Ausdruck kommt.
- ⑩, ⑪ Von einem Kunden kann kein, ein Auftrag oder können mehrere Aufträge erteilt werden, was durch die Angabe 0,n ausgesagt wird. Der Entity-Typ **AUFRAG** wird durch die Auftragsnummer (**AuftragsNr**) identifiziert. Das Auftragsdatum, eine Beschreibung des Auftrags und das Fertigstellungsdatum (Auslieferungsdatum) werden darin gespeichert.
- ⑫, ⑬ Der Auftrag wird in der Firma als Projekt bearbeitet. Die 1:1-Beziehung drückt aus, dass es zu jedem Auftrag genau ein Projekt gibt. Der Entity-Typ **PROJEKT** enthält den eigenen Namen und das Anfangs- und das Endatum der Bearbeitung. Sie wird durch die Projekt-nummer (**ProjNr**) identifiziert.

- ⑭ An einem Projekt arbeitet mindestens ein Mitarbeiter. Die Beziehung wird durch das Attribut *als* und das Attribut *prozAnteil* näher beschrieben. Das Attribut *als* gibt die Funktion an, die der Mitarbeiter in diesem Projekt übernimmt (z. B. Leiter, Entwickler ...). Das Attribut *prozAnteil* legt den prozentualen Anteil an der Arbeitszeit des Mitarbeiters fest, den er in ein Projekt investiert, da ein Mitarbeiter an mehreren Projekten beteiligt sein kann.
- ⑮ Eine Abteilung hat die Verantwortung für das Projekt, was durch die Beziehung *betreut* ausgedrückt wird. Nicht jede Abteilung betreut ein Projekt, wie z. B. die Abteilung *Kundenbetreuung*, was die Angabe 0,n erklärt. Dabei können aber Mitarbeiter aus verschiedenen Abteilungen an einem Projekt arbeiten.

In der Krähenfuss-Notation (erstellt mit Edraw Max Trial Version) ergibt sich für das Beispiel folgende Darstellung:



Textbeschreibung eines Datenmodells

Um eine kompakte Beschreibung Ihrer Entity-Typen, Attribute und Schlüsselfelder zu erhalten, können Sie die folgende textuelle Form verwenden. Zu Beginn stehen der Name des Entity-Typs und in Klammern die Attribute, wobei der oder die Primärschlüssel unterstrichen sind.

Entity-Typen

```
ANGESTELLTER(PersonalNr, Name, GebDat, PLZ, Wohnort, Strasse, Gehalt, Beruf)
MITARBEITER(PersonalNr, Name, GebDat, PLZ, Wohnort, Strasse, Gehalt, Beruf)*
SACHBEARBEITER(PersonalNr, Name, GebDat, PLZ, Wohnort, Strasse, Gehalt, Beruf)*
ABTEILUNG (AbtNr, AbtName)
KUNDE(KuNr, Name, PLZ, Wohnort, Strasse)
AUFTAG(AuftragsNr, AuftrDat, Beschreib, FertigDat)
PROJEKT (ProjNr, Name, BegDatum, EndDatum)
```

*Die Attribute erben diese Entity-Typen von dem Entity-Typ *ANGESTELLTER*.

Beziehungstypen

```
is-a(ANGESTELLTER, MITARBEITER, SACHBEARBEITER)
arbeitet_in(ABTEILUNG, SACHBEARBEITER, Position)
arbeitet_in(ABTEILUNG, MITARBEITER, Position)
betreut(SACHBEARBEITER, KUNDE)
erteilt(KUNDE, AUFTAG)
bearbeitet_als(AUFTAG, PROJEKT)
arbeitet_an(MITARBEITER, PROJEKT, als, prozAnteil)
betreut(ABTEILUNG, PROJEKT)
```

Vorgehensweisen

Im Folgenden werden zwei Möglichkeiten vorgestellt, wie dieses Modell schrittweise aufgebaut werden kann.

1. Möglichkeit

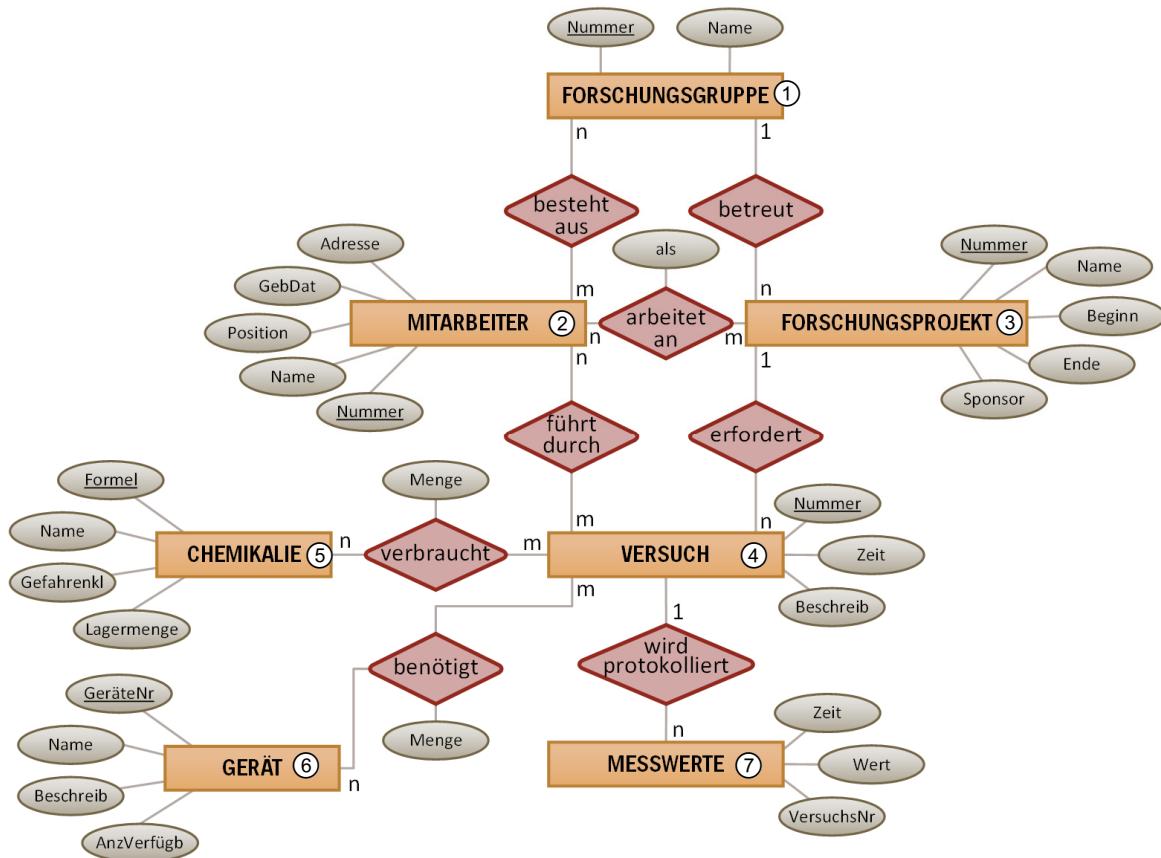
- ▶ Es werden die Entity-Typen *SACHBEARBEITER* und *MITARBEITER* beschrieben.
- ▶ Da beide Entity-Typen die gleichen Attribute besitzen, kann eine Generalisierung durchgeführt werden. Der Entity-Typ *ANGESTELLTER* wird gebildet.
- ▶ Alle Angestellten sind in Abteilungen aufgeteilt. So wird der Entity-Typ *ABTEILUNG* gebildet und die Beziehung zu den Angestellten hergestellt. Die Beziehung kann entweder zu den Angestellten oder zu den Mitarbeitern und Sachbearbeitern aufgebaut werden.
- ▶ Die Entity-Typen für Kunden und Auftrag werden gebildet und die Beziehung wird zwischen ihnen hergestellt.
- ▶ Da nur Sachbearbeiter Kunden betreuen, wird zwischen diesen beiden Entity-Typen eine Beziehung hergestellt.
- ▶ Die Auftragsbearbeitung erfolgt über ein Projekt. Der Entity-Typ *PROJEKT* wird erstellt.
- ▶ Vom Projekt aus gibt es eine Beziehung zum Auftrag, da es zu jedem Auftrag ein Projekt gibt. Eine weitere Beziehung besteht zur Abteilung, da eine Abteilung die Verantwortung für ein Projekt hat. Eine Abteilung kann 0 bis n Projekte betreuen. Auch zu den Mitarbeitern gibt es eine Beziehung, da ein oder mehrere Mitarbeiter, auch aus verschiedenen Abteilungen, an einem Projekt mitarbeiten. Ein Mitarbeiter kann an mehreren Projekten mitarbeiten.

2. Möglichkeit

- ▶ Für alle Angestellten wird ein Entity-Typ erstellt.
- ▶ Alle Angestellten sind in Abteilungen aufgeteilt. So wird der Entity-Typ **ABTEILUNG** gebildet und die Beziehung zu den Angestellten hergestellt.
- ▶ Die Entity-Typen für Kunden und Auftrag werden gebildet und die Beziehung zwischen ihnen wird hergestellt.
- ▶ Die Kunden werden nur von einem Teil der Angestellten, den Sachbearbeitern, betreut. Die anderen Mitarbeiter führen die Projektarbeit durch. Es wird eine Spezialisierung durchgeführt.
- ▶ Die Beziehung zwischen Sachbearbeitern und Kunden wird hergestellt.
- ▶ Der Entity-Typ **PROJEKT** wird wie in der ersten Möglichkeit erstellt und assoziiert.

Beispiel: Forschungsgruppe

In einer Forschungsgruppe arbeiten Mitarbeiter an verschiedenen Forschungsobjekten. Sie führen Versuchsreihen durch und protokollieren die Messwerte. Für die Versuche werden verschiedene Geräte und Chemikalien benötigt. Dieser Ausschnitt aus der Welt eines Forschungslabors wird im folgenden ER-Modell dargestellt.



- ① Der Entity-Typ **FORSCHUNGSGRUPPE** besitzt nur die Attribute *Nummer* und *Name*.
- ② Die Mitarbeiter sind in dem gleichnamigen Entity-Typ zusammengefasst. Das Attribut *Nummer* identifiziert eine Entität. Die angegebenen Attribute können gegebenenfalls noch erweitert werden. Zu einer Forschungsgruppe gehören mehrere Mitarbeiter, aber ein Mitarbeiter kann auch in mehreren Forschergruppen arbeiten, was durch die n:m-Beziehung verdeutlicht wird.

- ③ Von einer Forschungsgruppe werden verschiedene Forschungsprojekte betreut (1:n-Beziehung *betreut*). Ein Forschungsprojekt wird durch seine *Nummer* identifiziert. Es besitzt einen *Namen* und Attribute für den Beginn und das Ende der Forschungsarbeiten (*Beginn* und *Ende*). An einem Projekt können mehrere Mitarbeiter arbeiten, wobei ein Mitarbeiter an mehreren Projekten arbeiten kann (n:m-Beziehung *arbeitet_an*).
- ④ Die Mitarbeiter führen im Rahmen der Projektarbeit Versuche durch. An einem Versuch können mehrere Mitarbeiter arbeiten und es können durch Mitarbeiter mehrere Versuche durchgeführt werden (n:m-Beziehung *führt_durch*). Für ein Forschungsprojekt werden meist viele Versuche benötigt (1:n-Beziehung *erfordert*). Der Entity-Typ *VERSUCH* wird durch eine Nummer identifiziert. Die Beschreibung (*Beschreib*) des Versuchs und der Zeitpunkt (*Zeit*) sind ebenfalls Attribute.
- ⑤ Für einen Versuch werden Chemikalien in einer bestimmten Menge benötigt und Chemikalien können in mehreren Versuchen eingesetzt werden. Die n:m-Beziehung *verbraucht* besitzt aus diesem Grund das Attribut *Menge*. Der Entity-Typ *CHEMIKALIE* wird durch ihre Formel (*Formel*) identifiziert. Weitere Attribute sind der *Name*, die Gefahrenklasse (*Gefahrenkl*) und die verfügbare *Lagermenge*.
- ⑥ Weiterhin wird für den Versuch eine Versuchsanordnung aus verschiedenen Geräten benötigt und die Geräte können wiederum in mehreren Versuchen eingesetzt werden. In der n:m-Beziehung *benötigt* kann die Menge der benötigten Geräte angegeben werden. Im Entity-Typ *GERAET* besitzt ein Gerät eine *GeräteNr*, die es identifiziert. Jedes Gerät besitzt einen Namen und eine Beschreibung. Auch die Anzahl der verfügbaren Geräte (*AnzVerfüg*) kann angegeben werden.
- ⑦ Die Versuchsergebnisse werden in dem Entity-Typ *MESSWERTE* gespeichert. Für jede Messung werden die *Zeit* und der *Wert* notiert sowie die Nummer des Versuchs (*VersuchsNr*). Dieser Entity-Typ besitzt keinen Primärschlüssel. Alle Entitäten mit der gleichen Versuchsnummer gehören zu einem Versuch. Sie müssen nicht einzeln identifiziert werden. Ein solcher Entity-Typ wird auch als **schwacher Entity-Typ** bezeichnet.

Textbeschreibung eines Datenmodells

Entity-Typen

FORSCHUNGSGRUPPE(Nummer, Name)
MITARBEITER(Nummer, Name, GebDat, Position, Adresse)
FORSCHUNGSPROJEKT(Nummer, Name, Beginn, Ende, Sponsor)
VERSUCH(Nummer, Zeit, Beschreib)
CHEMIKALIE(Formel, Name, Gefahrenkl, Lagermenge)
GERAET(GeräteNr, Name, Beschreib, AnzVerfüg)
MESSWERTE (Zeit, Wert, VersuchsNr)

Beziehungstypen

besteht_aus(FORSCHUNGSGRUPPE, MITARBEITER)
betreut(FORSCHUNGSGRUPPE, FORSCHUNGSPROJEKT)
arbeitet_an(MITARBEITER, FORSCHUNGSPROJEKT)
führt_durch(MITARBEITER, VERSUCH)
erfordert(FORSCHUNGSPROJEKT, VERSUCH)
verbraucht(VERSUCH, CHEMIKALIE, Menge)
benötigt(VERSUCH, GERAET, Menge)
wird_protokolliert(VERSUCH, MESSWERTE)

Übersichtlichkeit der Darstellung

Bereits in den beiden nicht so umfangreichen Beispielen wird deutlich, dass bei Aufnahme aller Attribute in das Diagramm dieses sehr schnell unübersichtlich wird. Möglichkeiten, dies zu vermeiden bestehen darin:

- ✓ die Darstellung im Gesamtdiagramm auf die Schlüsselattribute sowie die für die Beziehungstypen relevanten Attribute zu beschränken,
- ✓ Diagrammauszüge anzufertigen, welche jeweils die komplette Anzahl der Attribute für den jeweiligen Ausschnitt enthalten,
- ✓ Beziehungstypen im Diagramm als Nummern darstellen. In einer separaten Legende werden diese Nummern dann mit Namen und Bedeutung der Beziehung aufgelistet.

2.5 Übung

Aufgaben zur Datenbanktheorie

Übungsdatei: --

Ergebnisdateien: [ER_Modell_Baufirma.pdf](#),
[ER_Modell_Lehrer.pdf](#)

1. Was verstehen Sie unter dem Datenbank-Lebenszyklus?
2. Welche Phasen werden beim Entwurf von Datenbanken durchlaufen?
3. Wozu dient das Entity-Relationship-Modell?
4. Welche Abstraktionskonzepte werden beim Datenbankentwurf angewendet? Durch welche Elemente des ER-Modells werden diese Abstraktionskonzepte realisiert?
5. Erstellen Sie ein ER-Modell für den folgenden Sachverhalt: Eine Baufirma möchte eine Datenbankanwendung einsetzen, um die Kundenaufträge und vorhandenen Ressourcen (Lager, Fahrzeuge, Fahrer) besser planen zu können.

In der Firma sind Büroangestellte, Fahrer, Arbeiter und Techniker angestellt.

Die Büroangestellten betreuen die Kunden und nehmen die Aufträge der Kunden entgegen.

Die Arbeiter und Fahrer führen die Aufträge aus. Der Fahrer benutzt dazu einen Lieferwagen oder einen Lkw. Für einen Auftrag werden ein Fahrer und ein oder mehrere Arbeiter benötigt. Für einige Aufträge werden Materialien aus dem Lager benötigt.

Der Techniker betreut den Fuhrpark, zu dem die Lieferwagen und Lkws gehören.

Versehen Sie die Entity-Typen des ER-Modells mit Attributen, die Sie für notwendig halten, und legen Sie auch Primärschlüssel fest. Beschriften Sie die Beziehungen und geben Sie die Kardinalitäten an.

6. Die Planung der Lehrveranstaltungen für Lehrer, Klassen und Räume soll in einem ER-Modell dargestellt werden. Zu beachten ist dabei, dass nicht jeder Unterricht in jedem beliebigen Raum abgehalten werden kann (z. B. Computer-Unterricht nur in Räumen, in denen sich Computer befinden). Alle Angehörigen der Bildungseinrichtung und die Zugehörigkeit zu den Abteilungen bzw. Klassen sind ebenfalls in das Modell zu integrieren. Die Lehrer bieten zu bestimmten Fächern Zusatzkurse an, die von einer bestimmten Anzahl an Schülern besucht werden können. Nehmen Sie auch diesen Aspekt mit in das Modell auf.
7. Geben Sie die Textbeschreibung der Entity-Typen zu Ihrem ER-Modell an.

3

Das relationale Datenmodell

3.1 Begriffe aus dem Bereich relationaler Datenbanken

Das relationale Datenmodell wurde 1970 von dem Mathematiker E. F. Codd entwickelt und mithilfe der Mengentheorie beschrieben. Dieses Modell bildet die Basis für relationale Datenbanken. Datenbanksysteme, die auf dem relationalen Datenmodell aufbauen, sind heute immer noch am weitesten verbreitet. Für relationale Datenbanken ist die Datenmanipulationssprache SQL entwickelt worden, die sich als Standard durchgesetzt hat.

Eine Datenbank besteht im relationalen Modell aus einer Menge von Relationen, in denen die logisch zusammengehörigen Daten gespeichert werden.

Relation

Eine Relation ist im Sinne einer relationalen Datenbank eine Menge von Tupeln (Datensätzen). Sie hat die Form einer **Tabelle** und ist damit eine Konstruktion aus Spalten und Zeilen. Sowohl Entitäten als auch Beziehungen des Entity-Relationship-Modells werden als Relationen modelliert. Datenbanktechnisch bezieht sich also eine Relation nicht auf die Beziehung (Verknüpfung) zwischen Tabellen. In einer Relation werden die Daten gespeichert. Diese Daten stellen die Informationen dar, die Sie verwalten wollen. Eine Relation ist gekennzeichnet durch ...

- ✓ einen eindeutigen Namen, beispielsweise: Kunde,
- ✓ mehrere Attribute (Spalten),
- ✓ keine bis beliebig viele Tupel (Tabellenzeilen oder Datensätze),
- ✓ einen einzigen Wert pro Attribut in einem Tupel (Tabellenzelle),
- ✓ einen Primärschlüssel, bestehend aus einem oder mehreren Attributen:
 - ✓ Dieser identifiziert jedes Tupel eindeutig.
 - ✓ Dessen Wert ändert sich während der Existenz des Datensatzes nicht.

Attribute und Tupel

Eine Tabelle besteht aus Spalten und Zeilen. Bei relationalen Datenbanken werden die Spalten **Attribute** oder auch **Felder** genannt. Die Zeilen der Tabelle werden als **Tupel** oder **Datensätze** bezeichnet.

Der Aufbau aller Tupel einer Tabelle ist gleich. Zum Beispiel werden in jedem Tupel Nummer, Name, PLZ, Ort, Straße und Hausnummer einer Person immer in dieser Reihenfolge abgespeichert. Der Inhalt der Tupel muss aber wegen der Mengendefinition der Relation immer unterschiedlich sein, d. h., es gibt keine zwei Tupel, die exakt gleich sind; sie unterscheiden sich in mindestens einem Attributwert (mindestens in dem Primärschlüssel).

Der Aufbau einer Relation wird auch die **Struktur** oder das **Schema** der Tabelle genannt. Die Gesamtheit der zu einer Datenbank gehörigen Relationenschemata heißt Schema der Datenbank. Die verschiedenen Datenbanksysteme haben unterschiedliche Formate für Relationen, die von den Anforderungen an die Namensgebung für Relationen und Attribute sowie von den Datentypen der Attribute abhängen.

Attribute (Spalten, Felder)					
Nummer	Name	PLZ	Ort	Straße	Hausnr.
1	Schulze	04173	Leipzig	Sassstr.	6
2	Meier	28199	Bremen	Hauptstr.	123
3	Müller	60320	Frankfurt	Dorfstr.	2

Aufbau einer Relation (Tabelle) mit ihrem Schema

Datentypen

Jedem Attribut wird ein bestimmter Datentyp zugeordnet. Meist stehen verschiedene Zahlen-typen, Zeichenketten (Text) und Datumstypen zur Verfügung. Der Datentyp bildet den Wertebereich (die Domäne) des Attributs. Meist lässt sich der Wertebereich aber noch durch weitere Angaben einschränken, z. B. „E–G“ oder „5–15“. Welche Datentypen und welche weiteren Einschränkungen möglich sind, hängt vom DBMS ab.

NULL-Werte oder leere Einträge

Die Attribute können entsprechend ihrem Wertebereich mit Werten belegt werden. Ein leerer Attributwert kann einen NULL-Wert haben. NULL-Werte entsprechen keinem Datentyp, sondern sind nur symbolische Werte und sind nicht dem numerischen Wert null gleichzusetzen. Ein NULL-Wert ist ein Indikatorbyte, das angibt, ob in einem Tupel für ein Attribut ein Wert eingetragen wurde oder nicht. Da er nur ein symbolischer Wert ist, kann er mit keinem anderen Wert verglichen werden, auch nicht mit einem anderen NULL-Wert. Ein leerer Wert dagegen entsteht, wenn z. B. ein Attributwert gelöscht wird.

Mengenschreibweise

In der Mengenschreibweise

$$R\{A_1, A_2, \dots, A_n\}$$

stellt R eine Relation mit den Attributen A_1 bis A_n , die paarweise verschieden sind, dar. Die Zahl n wird als Grad der Relation bezeichnet, der angibt, wie viele Attribute eine Relation besitzt.

$$R \subseteq D_1 \times D_2 \times \dots \times D_n$$

($D_i \times D_j$ symbolisiert das kartesische Produkt)

D_1, D_2, \dots, D_n sind die Wertebereiche der Attribute (die Domänen) und R ist eine n -stellige Relation. Die Tupel der Relation sind somit Elemente aus dem kartesischen Produkt über den Wertebereichen der Attribute. Als Tupel sind alle Kombinationen der zulässigen Werte für die Attribute möglich. Ein Tupel der Relation wird wie folgt dargestellt:

$r = (d_1, d_2, \dots, d_n) \in R$ ($d_i \in D_i, i = 1, \dots, n$, d_i ist der i -te Wert des Tupels)
bzw.

$$r \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n) \quad (\text{dom}(A_i) \text{ ist der Wertebereich des Attributs } A_i)$$

Beispiel

Eine Relation *ADRESSEN* besitzt die Attribute *Nummer, Name, Vorname, PLZ, Ort, Strasse, Hausnummer, Geburtsdatum*:

$$\text{ADRESSEN}(\text{Nummer}, \text{Name}, \text{Vorname}, \text{PLZ}, \text{Ort}, \text{Strasse}, \text{Hausnummer}, \text{Geburtsdatum})$$

Der Wertebereich des Attributes *Nummer* sind die ganzen Zahlen (Integer). Die Elemente des Namens und der Anschrift besitzen den Wertebereich einer Zeichenkette, die beispielsweise aus bis zu 5, 30 bzw. 100 verschiedenen Zeichen des Zeichenvorrats – Character(5), Character(30) bzw. Character(100) – bestehen darf. Das Geburtsdatum muss einem bestimmten Datumstyp entsprechen (Datum).

$$\text{ADRESSEN} \subseteq \text{Integer} \times \text{Character}(30) \times \text{Character}(30) \times \text{Character}(5) \times \text{Character}(100) \times \text{Character}(100) \times \text{Character}(5) \times \text{Datum}$$

Tupel dieser Relation sind z. B.

$$\begin{aligned} \text{adresse725} &= (725, \text{'Meier'}, \text{'David'}, \text{'10115'}, \text{'Berlin'}, \text{'Steinstr.'}, \text{'4'}, 17.04.1976) \\ \text{adresse123} &= (123, \text{'Schneider'}, \text{'Nina'}, \text{'1210'}, \text{'Wien'}, \text{'Buchenweg'}, \text{'4'}, 30.04.1982) \end{aligned}$$

Schlüssel

In einer Relation dürfen durch die Mengendefinition keine gleichen Tupel vorhanden sein. Somit ist jedes Tupel durch einen oder mehrere (gegebenenfalls auch alle) Attributwerte eindeutig identifizierbar. Die Menge der Attribute, die ein Tupel eindeutig identifizieren, heißt Schlüsselkandidaten. An Schlüssel wird die Minimalitätsanforderung gestellt, d. h., ein Schlüssel sollte so kurz wie möglich sein und es darf keine Teilmenge des Schlüssels bereits Schlüssel sein.

Beispiel

Zum Beispiel ist Relation *Kunde* wie folgt aufgebaut:

$$\text{KUNDE}(\underline{\text{KuNr}}, \text{Name}, \text{PLZ}, \text{Wohnort}, \text{Strasse})$$

Als Schlüssel für die Relation kommen infrage:

KuNr (die Kundennummer) oder
Name, PLZ, Strasse oder
Name, Wohnort, Strasse

Einer dieser Schlüssel wird als **Primärschlüssel** (Hauptschlüssel) definiert. In diesem Beispiel ist es das Attribut *KuNr*, da es der kürzeste von den infrage kommenden Schlüsseln (nur ein Attribut) ist. Gekennzeichnet wird der Primärschlüssel durch die Unterstreichung.

Um alle Tupel eindeutig zu identifizieren, kann einer Relation auch ein weiteres Attribut hinzugefügt werden, welches die Tupel durchnummert. Dadurch ist es möglich, die Tupel der Relation immer anhand dieses Attributs zu unterscheiden. Das Attribut wird in diesem Fall der Primärschlüssel der Relation. Jede Relation kann nur einen Primärschlüssel besitzen. Das Attribut, welches als Primärschlüssel dient, wird auch als **identifizierendes Attribut** bezeichnet (bzw. als identifizierende Attribute, wenn mehrere Attribute gemeinsam als Primärschlüssel dienen).

Ein Primärschlüssel einer Tabelle kann ein Attribut oder eine Attributkombination sein, deren Werte die Datensätze dieser Tabelle eindeutig identifizieren. Alle Attribute bzw. Attributkombinationen, deren Werte eindeutig sind, werden als **Schlüsselkandidaten** bezeichnet. Nur ein Schlüsselkandidat kann als Primärschlüssel festgelegt werden. Wenn eine Tabelle keine eindeutigen Datenfelder besitzt, sollte eine eindeutige **ID**-Nummer (Identifikationsnummer) als Schlüssel vergeben werden, beispielsweise eine Kundennummer für eine Relation *Kunden*.

Fremdschlüssel

Ein Fremdschlüssel ist ein Attribut in einer Relation, welches eine Beziehung zu einem Schlüsselfeld einer anderen Relation herstellt. Die Kennzeichnung des Fremdschlüssels kann über die Abkürzung FK oder das nachgestellte Zeichen '#' erfolgen.

Beispiel

AUFTRÄGE(AuftragsNr, Datum, ... <weitere Attribute>, Kundennummer #)

Beispielsweise befindet sich in einer Relation *Aufträge* in jedem Tupel eine Kundennummer des Kunden, der den Auftrag ausgelöst hat. Diese Kundennummer identifiziert einen Kunden in der Kundenrelation eindeutig. Die Kundennummer in der Auftragstabelle stellt somit einen Fremdschlüssel („fremder“ Schlüssel einer anderen Relation) dar.

Index

Häufig sollen die Tupel einer Relation in einer anderen Reihenfolge durchlaufen werden, als sie erfasst wurden, z. B. nach Postleitzahlen. Dazu muss die gesamte Relation nach diesen Postleitzahlen sortiert werden. Bei einer großen Anzahl von Tupeln kann dies eine lange Zeit in Anspruch nehmen. Um den Zugriff auf die Daten zu beschleunigen, können zusätzlich zum Primärschlüssel weitere Indizes angelegt werden, die auch als **Sekundärindizes** bzw. **Sekundärschlüssel** bezeichnet werden.

Bei Verwendung mehrerer Indizes wird z. B. beim Füllen einer Tabelle viel Zeit benötigt, um die Indizes zu aktualisieren. Günstig ist in diesem Fall, die Indizes erst nach dem Füllen der Tabelle zu aktualisieren.

3.2 Transformation des ER-Modells in ein relationales Modell

Das folgende ER-Modell soll in ein relationales Modell überführt werden.



Die Anzahl der Relationen, die aus dem ER-Modell erzeugt werden, hängt von den definierten Entitätsmengen und den Beziehungstypen zwischen den Entitätsmengen ab.

Entitätsmengen

Für jede Entitätsmenge wird eine Relation erstellt, welche für jedes Attribut eine Spalte besitzt. Die Primärschlüssel werden übernommen.

Relationenname	Relationenschema	Beispiel																																							
ABTEILUNG	<table border="1"> <thead> <tr> <th></th><th>Attribut</th><th>Datentyp</th></tr> </thead> <tbody> <tr> <td>!</td><td>AbteilungsNr</td><td>Zahl</td></tr> <tr> <td></td><td>Bezeichnung</td><td>Text</td></tr> </tbody> </table>		Attribut	Datentyp	!	AbteilungsNr	Zahl		Bezeichnung	Text	<table border="1"> <thead> <tr> <th>AbteilungsNr</th><th>Bezeichnung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Personal</td></tr> <tr> <td>2</td><td>Einkauf</td></tr> <tr> <td>3</td><td>Verkauf</td></tr> </tbody> </table>	AbteilungsNr	Bezeichnung	1	Personal	2	Einkauf	3	Verkauf																						
	Attribut	Datentyp																																							
!	AbteilungsNr	Zahl																																							
	Bezeichnung	Text																																							
AbteilungsNr	Bezeichnung																																								
1	Personal																																								
2	Einkauf																																								
3	Verkauf																																								
MITARBEITER	<table border="1"> <thead> <tr> <th></th><th>Attribut</th><th>Datentyp</th></tr> </thead> <tbody> <tr> <td>!</td><td>PersonalNr</td><td>Zahl</td></tr> <tr> <td></td><td>Name</td><td>Text</td></tr> <tr> <td></td><td>Vorname</td><td>Text</td></tr> <tr> <td></td><td>Gebdatum</td><td>Datum</td></tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		Name	Text		Vorname	Text		Gebdatum	Datum	<table border="1"> <thead> <tr> <th>PersonalNr</th><th>Name</th><th>Vorname</th><th>Gebdatum</th></tr> </thead> <tbody> <tr> <td>0001</td><td>Eichenau</td><td>Maria</td><td>15.05.1987</td></tr> <tr> <td>0002</td><td>Glahn</td><td>Stefanie</td><td>02.05.1978</td></tr> <tr> <td>0003</td><td>Kirsch</td><td>Karin</td><td>24.05.1988</td></tr> <tr> <td>0004</td><td>Conolly</td><td>Sean</td><td>26.04.1976</td></tr> <tr> <td>0005</td><td>Frawley</td><td>Lutz</td><td>09.09.1979</td></tr> </tbody> </table>	PersonalNr	Name	Vorname	Gebdatum	0001	Eichenau	Maria	15.05.1987	0002	Glahn	Stefanie	02.05.1978	0003	Kirsch	Karin	24.05.1988	0004	Conolly	Sean	26.04.1976	0005	Frawley	Lutz	09.09.1979
	Attribut	Datentyp																																							
!	PersonalNr	Zahl																																							
	Name	Text																																							
	Vorname	Text																																							
	Gebdatum	Datum																																							
PersonalNr	Name	Vorname	Gebdatum																																						
0001	Eichenau	Maria	15.05.1987																																						
0002	Glahn	Stefanie	02.05.1978																																						
0003	Kirsch	Karin	24.05.1988																																						
0004	Conolly	Sean	26.04.1976																																						
0005	Frawley	Lutz	09.09.1979																																						
PROJEKTE	<table border="1"> <thead> <tr> <th></th><th>Attribut</th><th>Datentyp</th></tr> </thead> <tbody> <tr> <td>!</td><td>ProjektNr</td><td>Zahl</td></tr> <tr> <td></td><td>Beschreibung</td><td>Text</td></tr> </tbody> </table>		Attribut	Datentyp	!	ProjektNr	Zahl		Beschreibung	Text	<table border="1"> <thead> <tr> <th>ProjektNr</th><th>Beschreibung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Kundenumfrage</td></tr> <tr> <td>2</td><td>Verkaufsmesse</td></tr> <tr> <td>3</td><td>Konkurrenzanalyse</td></tr> </tbody> </table>	ProjektNr	Beschreibung	1	Kundenumfrage	2	Verkaufsmesse	3	Konkurrenzanalyse																						
	Attribut	Datentyp																																							
!	ProjektNr	Zahl																																							
	Beschreibung	Text																																							
ProjektNr	Beschreibung																																								
1	Kundenumfrage																																								
2	Verkaufsmesse																																								
3	Konkurrenzanalyse																																								

1:1- und 1:n-Beziehungen

Für die Umsetzung der beiden Beziehungstypen gibt es zwei Möglichkeiten:

- ✓ Es wird eine neue Relation erzeugt, welche als Attribute (Spalten) die Primärschlüssel der beiden Relationen, die in Beziehung stehen, enthält. Außerdem kann die Relation beschreibende Attribute in einer zusätzlichen Spalte aufnehmen.
Diese Darstellungsform hat den Vorteil, dass keine NULL-Werte auftreten, und den Nachteil, dass eine weitere Relation benötigt wird.

Relationenname	Relationenschema	Beispiel																														
besteht_aus	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td>!</td> <td>AbteilungsNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Position</td> <td>Text</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl	!	AbteilungsNr	Zahl		Position	Text	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>AbteilungsNr</th> <th>Position</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>1</td> <td>Leiterin</td> </tr> <tr> <td>0002</td> <td>2</td> <td>Leiterin</td> </tr> <tr> <td>0003</td> <td>2</td> <td>Mitarbeiter</td> </tr> <tr> <td>0004</td> <td>3</td> <td>Leiter</td> </tr> <tr> <td>0005</td> <td>3</td> <td>Mitarbeiterin</td> </tr> </tbody> </table>	PersonalNr	AbteilungsNr	Position	0001	1	Leiterin	0002	2	Leiterin	0003	2	Mitarbeiter	0004	3	Leiter	0005	3	Mitarbeiterin
	Attribut	Datentyp																														
!	PersonalNr	Zahl																														
!	AbteilungsNr	Zahl																														
	Position	Text																														
PersonalNr	AbteilungsNr	Position																														
0001	1	Leiterin																														
0002	2	Leiterin																														
0003	2	Mitarbeiter																														
0004	3	Leiter																														
0005	3	Mitarbeiterin																														

- ✓ Eine der beiden Relationen wird um ein Attribut (Spalte) erweitert. Bei 1:1-Beziehungen ist es egal, welche der beiden Relationen erweitert wird. In einer 1:n-Beziehung wird die Relation erweitert, bei der das n steht. (Würde das Attribut an die Relation angefügt werden, bei der die 1 steht, würden Redundanzen auftreten, die Sie unbedingt vermeiden sollten.) Besitzt die Relation beschreibende Attribute, so werden diese zusätzlich noch bei der erweiterten Relation angegeben.

Bei dieser Darstellungsform können NULL-Werte auftreten. Sie hat aber den Vorteil, dass keine zusätzliche Relation benötigt wird. Für das obige Beispiel bedeutet dies, dass die Relation **MITARBEITER** erweitert wird.

Relationenname	Relationenschema	Beispiel																																																									
MITARBEITER	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Name</td> <td>Text</td> </tr> <tr> <td></td> <td>Vorname</td> <td>Text</td> </tr> <tr> <td></td> <td>Gebdatum</td> <td>Datum</td> </tr> <tr> <td></td> <td>AbtNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Position</td> <td>Text</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		Name	Text		Vorname	Text		Gebdatum	Datum		AbtNr	Zahl		Position	Text	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>Name</th> <th>Vorname</th> <th>Gebdatum</th> <th>AbtNr</th> <th>Position</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>Eichenau</td> <td>Maria</td> <td>15.05.1987</td> <td>1</td> <td>Leiterin</td> </tr> <tr> <td>0002</td> <td>Glahn</td> <td>Stefanie</td> <td>02.05.1978</td> <td>2</td> <td>Leiterin</td> </tr> <tr> <td>0003</td> <td>Kirsch</td> <td>Karin</td> <td>24.05.1988</td> <td>2</td> <td>Mitarb.</td> </tr> <tr> <td>0004</td> <td>Conolly</td> <td>Sean</td> <td>26.04.1976</td> <td>3</td> <td>Leiter</td> </tr> <tr> <td>0005</td> <td>Frawley</td> <td>Lutz</td> <td>09.09.1979</td> <td>3</td> <td>Mitarb.</td> </tr> </tbody> </table>	PersonalNr	Name	Vorname	Gebdatum	AbtNr	Position	0001	Eichenau	Maria	15.05.1987	1	Leiterin	0002	Glahn	Stefanie	02.05.1978	2	Leiterin	0003	Kirsch	Karin	24.05.1988	2	Mitarb.	0004	Conolly	Sean	26.04.1976	3	Leiter	0005	Frawley	Lutz	09.09.1979	3	Mitarb.
	Attribut	Datentyp																																																									
!	PersonalNr	Zahl																																																									
	Name	Text																																																									
	Vorname	Text																																																									
	Gebdatum	Datum																																																									
	AbtNr	Zahl																																																									
	Position	Text																																																									
PersonalNr	Name	Vorname	Gebdatum	AbtNr	Position																																																						
0001	Eichenau	Maria	15.05.1987	1	Leiterin																																																						
0002	Glahn	Stefanie	02.05.1978	2	Leiterin																																																						
0003	Kirsch	Karin	24.05.1988	2	Mitarb.																																																						
0004	Conolly	Sean	26.04.1976	3	Leiter																																																						
0005	Frawley	Lutz	09.09.1979	3	Mitarb.																																																						

Bei Erweiterung der Relation **ABTEILUNG** entstehen **redundante Daten**.

ABTEILUNG	Relationenschema	Beispiel																														
	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>AbteilungsNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Bezeichnung</td> <td>Text</td> </tr> <tr> <td></td> <td>PersonalNr</td> <td>Zahl</td> </tr> </tbody> </table>		Attribut	Datentyp	!	AbteilungsNr	Zahl		Bezeichnung	Text		PersonalNr	Zahl	<table border="1"> <thead> <tr> <th>AbteilungsNr</th> <th>Bezeichnung</th> <th>PersonalNr</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Personal</td> <td>0001</td> </tr> <tr> <td>2</td> <td>Einkauf</td> <td>0002</td> </tr> <tr> <td>2</td> <td>Einkauf</td> <td>0003</td> </tr> <tr> <td>3</td> <td>Verkauf</td> <td>0004</td> </tr> <tr> <td>3</td> <td>Verkauf</td> <td>0005</td> </tr> </tbody> </table>	AbteilungsNr	Bezeichnung	PersonalNr	1	Personal	0001	2	Einkauf	0002	2	Einkauf	0003	3	Verkauf	0004	3	Verkauf	0005
	Attribut	Datentyp																														
!	AbteilungsNr	Zahl																														
	Bezeichnung	Text																														
	PersonalNr	Zahl																														
AbteilungsNr	Bezeichnung	PersonalNr																														
1	Personal	0001																														
2	Einkauf	0002																														
2	Einkauf	0003																														
3	Verkauf	0004																														
3	Verkauf	0005																														

Ein NULL-Wert kann in diesem Beispiel auftreten, wenn ein Mitarbeiter keiner Abteilung zugeordnet ist.

m:n-Beziehungen

Bei einer m:n-Beziehung ist immer eine zusätzliche Relation erforderlich, um die betreffenden Entitäten zu verknüpfen. Diese Relation enthält die Primärschlüssel der beiden Relationen und kann zusätzlich noch beschreibende Attribute enthalten.

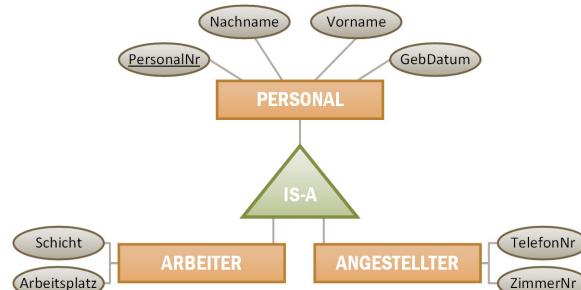
Für das obige Beispiel wird eine Relation *arbeitet_an* benötigt. Das Relationenschema wird aus den beiden Primärschlüsseln und den Attributen *Tätigkeit* und *prozAnteil* (prozentualer Anteil der Arbeitszeit) aufgebaut.

Relationenname	Relationenschema	Beispiel																																																			
<i>arbeitet_an</i>	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td>!</td> <td>ProjektNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Tätigkeit</td> <td>Text</td> </tr> <tr> <td></td> <td>prozAnteil</td> <td>Zahl</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl	!	ProjektNr	Zahl		Tätigkeit	Text		prozAnteil	Zahl	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>ProjektNr</th> <th>Tätigkeit</th> <th>prozAnteil</th> </tr> </thead> <tbody> <tr> <td>0002</td> <td>1</td> <td>Leiterin</td> <td>25</td> </tr> <tr> <td>0003</td> <td>1</td> <td>Sachbearbeiter</td> <td>50</td> </tr> <tr> <td>0004</td> <td>1</td> <td>Sachbearbeiter</td> <td>50</td> </tr> <tr> <td>0004</td> <td>2</td> <td>Leiter</td> <td>25</td> </tr> <tr> <td>0005</td> <td>2</td> <td>Präsentationsvorbereitung</td> <td>100</td> </tr> <tr> <td>0004</td> <td>3</td> <td>Leiter</td> <td>25</td> </tr> <tr> <td>0002</td> <td>3</td> <td>Sachbearbeiterin</td> <td>50</td> </tr> <tr> <td>0003</td> <td>3</td> <td>Sachbearbeiter</td> <td>50</td> </tr> </tbody> </table>	PersonalNr	ProjektNr	Tätigkeit	prozAnteil	0002	1	Leiterin	25	0003	1	Sachbearbeiter	50	0004	1	Sachbearbeiter	50	0004	2	Leiter	25	0005	2	Präsentationsvorbereitung	100	0004	3	Leiter	25	0002	3	Sachbearbeiterin	50	0003	3	Sachbearbeiter	50
	Attribut	Datentyp																																																			
!	PersonalNr	Zahl																																																			
!	ProjektNr	Zahl																																																			
	Tätigkeit	Text																																																			
	prozAnteil	Zahl																																																			
PersonalNr	ProjektNr	Tätigkeit	prozAnteil																																																		
0002	1	Leiterin	25																																																		
0003	1	Sachbearbeiter	50																																																		
0004	1	Sachbearbeiter	50																																																		
0004	2	Leiter	25																																																		
0005	2	Präsentationsvorbereitung	100																																																		
0004	3	Leiter	25																																																		
0002	3	Sachbearbeiterin	50																																																		
0003	3	Sachbearbeiter	50																																																		

Generalisierung/Spezialisierung (Is-a-Beziehungen)

Für die Überführung einer *Is-a*-Beziehung gibt es mehrere Möglichkeiten, die von dem jeweiligen Schema und von der Umgebung (dem Kontext) abhängig sind. In jedem Fall ist aber keine zusätzliche Relation für die Beziehung nötig.

- Es wird für jede Entitätsmenge eine Relation mit den relevanten Attributen angelegt. Den Teilmengen (Spezialisierungen) wird der Primärschlüssel der Obermenge (Generalisierung) als Fremdschlüssel hinzugefügt, um die Zuordnung zu sichern.

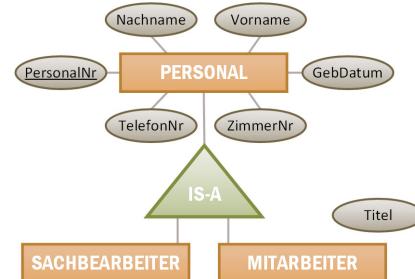


Relationenname	Relationenschema	Beispiel																																											
<i>PERSONAL</i>	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Name</td> <td>Text</td> </tr> <tr> <td></td> <td>Vorname</td> <td>Text</td> </tr> <tr> <td></td> <td>GebDatum</td> <td>Datum</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		Name	Text		Vorname	Text		GebDatum	Datum	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>Nachname</th> <th>Vorname</th> <th>GebDatum</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>Eichenau</td> <td>Maria</td> <td>15.05.1987</td> </tr> <tr> <td>0002</td> <td>Glahn</td> <td>Stefanie</td> <td>02.05.1978</td> </tr> <tr> <td>0003</td> <td>Kirsch</td> <td>Karin</td> <td>24.05.1988</td> </tr> <tr> <td>0004</td> <td>Conolly</td> <td>Sean</td> <td>26.04.1976</td> </tr> <tr> <td>0005</td> <td>Frawley</td> <td>Lutz</td> <td>09.09.1979</td> </tr> <tr> <td>0006</td> <td>...</td> <td>...</td> <td>...</td> </tr> </tbody> </table>	PersonalNr	Nachname	Vorname	GebDatum	0001	Eichenau	Maria	15.05.1987	0002	Glahn	Stefanie	02.05.1978	0003	Kirsch	Karin	24.05.1988	0004	Conolly	Sean	26.04.1976	0005	Frawley	Lutz	09.09.1979	0006
	Attribut	Datentyp																																											
!	PersonalNr	Zahl																																											
	Name	Text																																											
	Vorname	Text																																											
	GebDatum	Datum																																											
PersonalNr	Nachname	Vorname	GebDatum																																										
0001	Eichenau	Maria	15.05.1987																																										
0002	Glahn	Stefanie	02.05.1978																																										
0003	Kirsch	Karin	24.05.1988																																										
0004	Conolly	Sean	26.04.1976																																										
0005	Frawley	Lutz	09.09.1979																																										
0006																																										
<i>ARBEITER</i>	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Schicht</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Arbeitsplatz</td> <td>Zahl</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		Schicht	Zahl		Arbeitsplatz	Zahl	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>Schicht</th> <th>Arbeitsplatz</th> </tr> </thead> <tbody> <tr> <td>0011</td> <td>1</td> <td>01</td> </tr> <tr> <td>0012</td> <td>2</td> <td>02</td> </tr> <tr> <td>0015</td> <td>3</td> <td>02</td> </tr> <tr> <td>0019</td> <td>2</td> <td>01</td> </tr> <tr> <td>0037</td> <td>3</td> <td>01</td> </tr> </tbody> </table>	PersonalNr	Schicht	Arbeitsplatz	0011	1	01	0012	2	02	0015	3	02	0019	2	01	0037	3	01													
	Attribut	Datentyp																																											
!	PersonalNr	Zahl																																											
	Schicht	Zahl																																											
	Arbeitsplatz	Zahl																																											
PersonalNr	Schicht	Arbeitsplatz																																											
0011	1	01																																											
0012	2	02																																											
0015	3	02																																											
0019	2	01																																											
0037	3	01																																											

Relationenname	Relationenschema	Beispiel																														
ANGESTELLTER	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>ZimmerNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>TelefonNr</td> <td>Zahl</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		ZimmerNr	Zahl		TelefonNr	Zahl	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>ZimmerNr</th> <th>TelefonNr</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>1</td> <td>369</td> </tr> <tr> <td>0002</td> <td>8</td> <td>265</td> </tr> <tr> <td>0005</td> <td>3</td> <td>235</td> </tr> <tr> <td>0009</td> <td>5</td> <td>302</td> </tr> <tr> <td>0007</td> <td>3</td> <td>236</td> </tr> </tbody> </table>	PersonalNr	ZimmerNr	TelefonNr	0001	1	369	0002	8	265	0005	3	235	0009	5	302	0007	3	236
	Attribut	Datentyp																														
!	PersonalNr	Zahl																														
	ZimmerNr	Zahl																														
	TelefonNr	Zahl																														
PersonalNr	ZimmerNr	TelefonNr																														
0001	1	369																														
0002	8	265																														
0005	3	235																														
0009	5	302																														
0007	3	236																														

- ✓ Es wird nur eine Relation angelegt, die relevante Attribute der Teilmengen, sofern vorhanden, mit aufnimmt. Diese Möglichkeit der Umsetzung kann dann angewandt werden, wenn die Teilmengen keine oder wenige eigene Attribute haben.

An die Relation kann bei dieser Variante ein Attribut (z. B. Is-a) angefügt werden, welches für die Zuordnung steht.



Relationenname	Relationenschema	Beispiel																																																															
PERSONAL	<table border="1"> <thead> <tr> <th></th> <th>Attribut</th> <th>Datentyp</th> </tr> </thead> <tbody> <tr> <td>!</td> <td>PersonalNr</td> <td>Zahl</td> </tr> <tr> <td></td> <td>Name</td> <td>Text</td> </tr> <tr> <td></td> <td>Vorname</td> <td>Text</td> </tr> <tr> <td></td> <td>...</td> <td>...</td> </tr> <tr> <td></td> <td>Titel</td> <td>Text</td> </tr> <tr> <td></td> <td>IS-A</td> <td>Text</td> </tr> </tbody> </table>		Attribut	Datentyp	!	PersonalNr	Zahl		Name	Text		Vorname	Text			Titel	Text		IS-A	Text	<table border="1"> <thead> <tr> <th>PersonalNr</th> <th>Nachname</th> <th>Vorname</th> <th>...</th> <th>Titel</th> <th>IS-A</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>Eichenau</td> <td>Maria</td> <td>...</td> <td>Sachbearb.</td> <td></td> </tr> <tr> <td>0002</td> <td>Glahn</td> <td>Stefanie</td> <td>...</td> <td>Dipl.-Ing.</td> <td>Mitarb.</td> </tr> <tr> <td>0003</td> <td>Kirsch</td> <td>Karin</td> <td>...</td> <td></td> <td>Sachbearb.</td> </tr> <tr> <td>0004</td> <td>Conolly</td> <td>Sean</td> <td>...</td> <td>Dr. Ing.</td> <td>Mitarb.</td> </tr> <tr> <td>0005</td> <td>Frawley</td> <td>Lutz</td> <td>...</td> <td></td> <td>Sachbearb.</td> </tr> <tr> <td>0006</td> <td>...</td> <td>...</td> <td>...</td> <td></td> <td>...</td> </tr> </tbody> </table>	PersonalNr	Nachname	Vorname	...	Titel	IS-A	0001	Eichenau	Maria	...	Sachbearb.		0002	Glahn	Stefanie	...	Dipl.-Ing.	Mitarb.	0003	Kirsch	Karin	...		Sachbearb.	0004	Conolly	Sean	...	Dr. Ing.	Mitarb.	0005	Frawley	Lutz	...		Sachbearb.	0006
	Attribut	Datentyp																																																															
!	PersonalNr	Zahl																																																															
	Name	Text																																																															
	Vorname	Text																																																															
																																																															
	Titel	Text																																																															
	IS-A	Text																																																															
PersonalNr	Nachname	Vorname	...	Titel	IS-A																																																												
0001	Eichenau	Maria	...	Sachbearb.																																																													
0002	Glahn	Stefanie	...	Dipl.-Ing.	Mitarb.																																																												
0003	Kirsch	Karin	...		Sachbearb.																																																												
0004	Conolly	Sean	...	Dr. Ing.	Mitarb.																																																												
0005	Frawley	Lutz	...		Sachbearb.																																																												
0006																																																												

3.3 Normalisierung des Datenbankschemas

Ziel der Normalisierung des relationalen Datenbankschemas ist es,

- ✓ Anomalien zu beheben,
- ✓ Redundanzen zu vermeiden,
- ✓ einen übersichtlichen und möglichst einfachen Aufbau der Relationen zu erhalten,
- ✓ eine einfache Datenpflege zu ermöglichen.

Probleme beim Ändern, Einfügen und Löschen von Datensätzen (Anomalien)

Ausgehend von der Tatsache, dass eine Relation logisch zusammengehörige Daten enthält, kann beispielsweise folgende Relation angelegt werden: Für jeden Mitarbeiter werden die Personalnummer, der Name und Vorname sowie dessen Abteilung und seine Projektdaten erfasst. Alle Daten werden in einer einzigen Relation gespeichert. Der Primärschlüssel setzt sich aus der *PersonalNr* und der *ProjNr* zusammen. Da eine Person an mehreren Projekten mitarbeiten kann, ist die *PersonalNr* nicht mehr eindeutig und erfüllt somit nicht mehr das Kriterium des Primärschlüssels. Die Relation befindet sich in der ersten Normalform.

Relationenschema		Beispiel								
	Attribut	Daten-typ	PersonalNr	Name	Vorname	AbtNr	AbtBezeichnung	ProjNr	ProjektBeschreibung	Tätigkeit
!	PersonalNr	Zahl	0001	Eichenau	Maria	1	Personal			
	Name	Text	0002	Glahn	Stefanie	2	Einkauf	1	Kundenumfrage	Leiterin
	Vorname	Text	0002	Glahn	Stefanie	2	Einkauf	3	Konkurrenzanalyse	Sachbearbeiterin
	AbteilungsNr	Zahl	0003	Kirsch	Karin	2	Einkauf	1	Kundenumfrage	Sachbearb.
!	AbtBezeichnung	Text	0004	Conolly	Sean	3	Verkauf	2	Verkaufsmesse	Leiter
!	ProjNr	Zahl	0004	Conolly	Sean	3	Verkauf	3	Konkurrenzanalyse	Leiter
	ProjektBeschreibung	Text	0005	Frawley	Lutz	3	Verkauf	2	Verkaufsmesse	Präsent.-Vorbereitung
	Tätigkeit	Text								

Relation MITARBEITER in 1. Normalform

- ! Bei einigen Datenbanksystemen müssen Schlüsselattribute bei der Definition der Struktur der Relation direkt untereinander stehen.

Eine Relation befindet sich in der ersten Normalform (1NF), wenn jedes Attribut der Relation atomar ist. Das bedeutet, dass die Attribute keine weiteren Untergliederungen aufweisen dürfen bzw. dass für jedes Attribut eines Tupels nur ein Eintrag zulässig ist (z. B., dass keine Liste von Projektnummern angegeben werden kann). Die erste Normalform ist der Ausgangszustand für weitere Betrachtungen.

Das abgebildete Relationenschema weist erhebliche Schwachstellen auf. Werden Tupel (Datensätze) geändert, neue eingefügt oder alte gelöscht, können fehlerhafte Zustände auftreten. Diese werden durch die Datenredundanz hervorgerufen und führen zu Inkonsistenzen. Diese Fehler werden auch als Anomalien bezeichnet.

✓ Einfüge-Anomalie

Wenn Sie der Relation einen neuen Mitarbeiter hinzufügen, der zu diesem Zeitpunkt an keinem Projekt mitarbeitet, entstehen leere Datenfelder. Neben der Tatsache, dass dadurch Speicherplatz verschwendet wird, tritt das Problem auf, dass einem Attribut kein Wert zugewiesen wird, das Teil des Primärschlüssels ist, wie im Beispiel der Projektnummer (*ProjNr*). Beim Verwalten und Suchen dieses Datensatzes können Fehler auftreten. In einigen DBMS ist es gar nicht möglich, einen solchen Datensatz zu speichern.

✓ Lösch-Anomalie

Wenn Sie einen vorhandenen Mitarbeiter löschen, werden auch die zugehörigen Projektdateien gelöscht. Sind die Daten für dieses Projekt nur bei diesem Mitarbeiter gespeichert, gehen diese Daten verloren.

✓ Änderungs-Anomalie

Wenn sich der Familienname eines Mitarbeiters ändert, beispielsweise wenn sich der Name von *Glahn* zu *Schumann* ändert, müssen alle Datensätze geändert werden, die diesen Wert beinhalten. Wird der Familienname nur in einem Datensatz geändert, wird die Relation inkonsistent, da in diesem Fall zu einer Personalnummer zwei verschiedene Familiennamen existieren.

Abhängigkeiten

Bei der Normalisierung spielt die Beseitigung von Abhängigkeiten zwischen den Attributen einer Relation eine große Rolle. Es gibt funktionale und transitive Abhängigkeiten.

Funktionale Abhängigkeit

Für eine Relation $R(A_1, A_2, \dots, A_n)$ sind X und Y je eine echte Teilmenge der Attributmenge (z. B. $X = (A_4, A_5)$ und $Y = (A_8)$). Eine funktionale Abhängigkeit ($X \rightarrow Y$) liegt vor, wenn es keine zwei Tupel geben kann, in denen für gleiche X -Werte verschiedene Y -Werte auftreten können. Umgekehrt kann es aber für gleiche Y -Werte verschiedene X -Werte geben.

Beispiel

In der Relation

LIEFERUNG(LieferNr, ArtikelNr, Artikelname, Anzahl, Lieferdatum, Lieferfirma, AnschriftLieferfirma)

bestehen funktionale Abhängigkeiten zwischen den Attributen

$Anzahl \rightarrow LieferNr, ArtikelNr$	In einer Lieferung (<i>LieferNr</i>) werden verschiedene Artikel (<i>ArtikelNr</i>) in einer bestimmten Anzahl (<i>Anzahl</i>) geliefert.
$LieferNr \rightarrow Lieferdatum$	Eine Lieferung (mit der <i>LieferNr</i>) erfolgt an einem Tag (<i>Lieferdatum</i>).
$ArtikelNr \rightarrow Artikelname$	Zu einer Artikelnummer (<i>ArtikelNr</i>) gibt es immer genau einen Artikelnamen (<i>Artikelname</i>).
$ArtikelNr \rightarrow Lieferfirma$	Ein Artikel mit einer Nummer (<i>ArtikelNr</i>) wird von genau einer bestimmten Firma (<i>Lieferfirma</i>) geliefert.
$Lieferfirma \rightarrow AnschriftLieferfirma$	Jede Lieferfirma (<i>Lieferfirma</i>) hat genau eine Anschrift (<i>Anschrift Lieferfirma</i>).

Transitive Abhängigkeit

Eine transitive Abhängigkeit besteht, wenn ein Attribut a von einem Attribut b und ein Attribut b von einem Attribut c funktional abhängig ist. In diesem Fall ist a transitiv von c abhängig.

Das obige Beispiel enthält eine transitive Abhängigkeit. Die Lieferfirma hängt von der Artikelnummer ab, die Lieferfirmen-Anschrift aber von der Lieferfirma (*AnschriftLieferfirma* \rightarrow *Lieferfirma* \rightarrow *ArtikelNr*).

Der Normalisierungsprozess

Die Normalisierung eines relationalen Schemas wird in mehreren Stufen vollzogen. Dabei müssen die Daten in den Relationen in jeder Stufe bestimmte Bedingungen erfüllen. Das Resultat der Anwendung dieser Regeln wird als **Normalform** des Relationenschemas bezeichnet.

Beim Normalisierungsprozess werden die Daten einer Relation auf mehrere Relationen verteilt. Die einzelnen Stufen des Normalisierungsprozesses werden als erste bis fünfte Normalform bezeichnet. Oft ist es sinnvoll, die Normalisierung nur bis zur dritten Normalform durchzuführen.

Die weiteren Normalformen verkomplizieren danach die Verwaltung der Datenbank zu sehr, da viele kleine Relationen entstehen.

Wie weit die Normalisierung realisiert wird, ist auch immer unter dem Gesichtspunkt der Performance der Abarbeitung der SQL-Anfragen zu entscheiden. Zu weitgehende Normalisierungen werden unter Umständen aus Performancegründen wieder denormalisiert, um langen Laufzeiten entgegenzuwirken.

Nicht normalisierte Datenstruktur

Eine Datenstruktur wird als **nicht normalisiert** bezeichnet, wenn in einem Datensatz ein Attribut nicht atomar ist, sondern stattdessen eine Werteliste angegeben wird. Zum Beispiel arbeitet der Mitarbeiter *Richter* an den Projekten 1, 2 und 3.

PersonalNr	Nachname	Vorname	AbtNr	AbtBezeichng	ProjNr	ProjektBeschreibung	Tätigkeit
0004	Richter	Hans	3	Verkauf	1, 2, 3	Kundenumfrage, Konkurrenzanalyse, Verkaufsmesse	Sachbearbeiter, Leiter

mehrere Werte in einem Datenfeld

Diese Relation ist schwer auszuwerten, da einige Attribute mehrere Werte haben. Werden z. B. alle Mitarbeiter gesucht, die an Projekt 3 mitarbeiten, ist eine komplizierte Auswertung notwendig, da erst alle Werte des Attributs *ProjNr* untersucht werden müssen. Mitunter sind keine genauen Zuordnungen mehr möglich. Zum Beispiel lässt sich nicht mehr herausfinden, in welchem Projekt der Mitarbeiter *Richter* welche Tätigkeit ausübt.

Beispiel

In der folgenden schrittweisen Umsetzung wird die folgende Datenstruktur in die verschiedenen Normalformen überführt:

PersonalNr	Name	Vorname	AbtNr	AbtBezeichnung	ProjNr	ProjektBeschreibung	Tätigkeit
0001	Eichenau	Maria	1	Personal			
0002	Glahn	Stefanie	2	Einkauf	1, 3	Kundenumfrage, Konkurrenzanalyse	Leiterin, Sachbearbeiterin
0003	Kirsch	Karin	2	Einkauf	1	Kundenumfrage	Sachbearb.
0004	Conolly	Sean	3	Verkauf	2, 3	Verkaufsmesse, Konkurrenzanalyse	Leiter, Leiter
0005	Frawley	Lutz	3	Verkauf	2	Verkaufsmesse	Präsent.-Vorbereitung

1. Normalform (1NF)

Eine Relation befindet sich in der 1. Normalform, wenn ...

- ✓ sie zweidimensional ist, d. h. ein Gebilde aus Zeilen und Spalten,
- ✓ sich in jedem Datensatz nur Daten befinden, die zu einem Objekt der realen Welt gehören, und jeder Datensatz nur einmal vorkommt,
- ✓ sich in jeder Spalte nur Daten befinden, die einem Attribut entsprechen, und das Attribut nur einmal in der Relation vorkommt,
- ✓ für jedes Attribut nur ein Wert eingetragen ist.

Gehen Sie bei der Transformation einer nicht normalisierten Datenstruktur in die 1. Normalform wie folgt vor:

- Entfernen Sie alle Mehrfacheinträge in einem Attribut. Erstellen Sie pro Mitarbeiter/Projektkombination einen eigenen Datensatz. Dabei darf jedem Attribut eines Datensatzes höchstens ein Wert zugewiesen sein.

Im Ergebnis liegt die Datenstruktur in einer Tabelle mit einem Tupel pro Mitarbeiter/Projekt-kombination vor.

PersonalNr	Name	Vorname	AbtNr	AbtBezeichnung	ProjNr	ProjektBeschreibung	Tätigkeit
0001	Eichenau	Maria	1	Personal			
0002	Glahn	Stefanie	2	Einkauf	1	Kundenumfrage	Leiterin
0002	Glahn	Stefanie	2	Einkauf	3	Konkurrenzanalyse	Sachbearbeiterin
0003	Kirsch	Karin	2	Einkauf	1	Kundenumfrage	Sachbearb.
0004	Conolly	Sean	3	Verkauf	2	Verkaufsmesse	Leiter
0004	Conolly	Sean	3	Verkauf	3	Konkurrenzanalyse	Leiter
0005	Frawley	Lutz	3	Verkauf	2	Verkaufsmesse	Präsent.-Vorbereitung

Probleme

- ✓ Die Relation weist Redundanzen auf, z. B. treten Mitarbeiterdaten, Abteilungsnamen und Projektnamen im Beispiel mehrfach auf.
- ✓ Die Relation enthält voneinander unabhängige Sachgebiete, wie zum Beispiel Mitarbeiter, Abteilungen, Projekte.
- ✓ Daten können nicht eindeutig identifiziert werden. Beispielsweise kann der Abteilungsname Einkauf nur über eine Personal- und Projektnummer ermittelt werden.

2. Normalform (2NF)

Eine Relation befindet sich in der 2. Normalform, wenn die 1. Normalform erfüllt ist und wenn jedes Nicht-Schlüsselfeld vom ganzen Primärschlüssel (der auch aus mehreren Feldern bestehen kann) abhängig ist. Wichtig hierbei ist, dass Datenfelder nicht nur von einem Teilschlüsselfeld, sondern vom **gesamten** Schlüssel funktional abhängig sind.

Gehen Sie bei der Transformation von der 1. Normalform in die 2. Normalform wie folgt vor:

- Zerlegen Sie die Relation in kleinere Relationen, sodass in jeder Relation alle Nicht-Schlüsselfelder nur noch vom Primärschlüssel abhängen.
- Besteht ein Primärschlüssel aus mehreren Attributen, ist zu prüfen, ob es Attribute gibt, die eigentlich nur von einem Teil des Primärschlüssels abhängen. Ist das der Fall, so sind dieser Teil des Primärschlüssels und die zugehörigen Attribute in eine neue Relation zu bringen.

Aus der ursprünglichen Tabelle entstehen für das verwendete Beispiel drei Tabellen.

Werden die Attribute der obigen Relation *MITARBEITER* der Reihe nach untersucht, ergibt sich, dass die Attribute *Name*, *Vorname* nur von einem Teil des Primärschlüssels, von der *PersonalNr*, abhängen. Auch zwischen den Attributen *AbtNr* und *PersonalNr* ist eine eindeutige Beziehung erkennbar. Alle anderen Attribute hängen nicht oder nur zum Teil vom Attribut *PersonalNr* ab. Die Relation *MITARBEITER* besitzt nur noch vier Attribute.

PERSONAL (PersonalNr, Name, Vorname, AbtNr)

Des Weiteren ist das Attribut *AbtBezeichnung* nicht vom Primärschlüssel abhängig, sondern nur vom Attribut *AbtNr*. Kein weiteres Attribut hängt vom Attribut *AbtNr* ab. Die Relation *ABTEILUNG* wird gebildet.

ABTEILUNG (AbtNr, Name)

Die Projektbeschreibung ist nur von einem Teil des Primärschlüssels abhängig, vom Attribut *ProjNr*. Die Relation *PROJEKT* entsteht.

PROJEKT (ProjNr, Beschreibung)

Die Funktion, in der ein Mitarbeiter an einem Projekt arbeitet, geht nur aus der Kombination der Attribute *ProjNr* und *PersonalNr* hervor. So entsteht eine weitere Relation *ARBEITET_AN*.

ARBEITET_AN (ProjNr, PersonalNr, Tätigkeit)

	Attribut	Datentyp
!	PersonalNr	Zahl
	Name	Text
	Vorname	Text
	AbtNr	Zahl

	Attribut	Datentyp
!	AbtNr	Zahl
	Name	Text

	Attribut	Datentyp
	ProjNr	Zahl
	Beschreibung	Text

	Attribut	Datentyp
!	ProjNr	Zahl
!	PersonalNr	Zahl
	Tätigkeit	Text

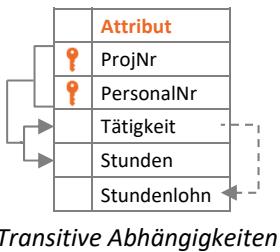
Relationen in 2. Normalform

3. Normalform (3NF)

Eine Tabelle befindet sich in der 3. Normalform, wenn alle Datenfelder nur vom gesamten Schlüssel abhängig sind und untereinander keine Abhängigkeiten auftreten. Sobald ein Nicht-Schlüsselfeld nur über ein anderes Nicht-Schlüsselfeld identifizierbar ist, wird von **transitiver Abhängigkeit** gesprochen. Transitive Abhängigkeiten verursachen ebenfalls Datenredundanz und -inkonsistenz.

Für das verwendete Beispiel befinden sich alle Tabellen der 2. Normalform bereits in der 3. Normalform, da keine transitiven Abhängigkeiten bestehen. Zur Darstellung der Umwandlung von der 2. in die 3. Normalform wird das Beispiel erweitert.

Die Relation **ARBEITET_AN** wird ergänzt und soll zusätzlich die Attribute **Stunden** und **Stundenlohn** besitzen. Der Stundenlohn wird von der Art der Tätigkeit bestimmt. Damit ist der Stundenlohn nicht direkt, sondern **transitiv** vom Schlüsselfeld (**ProjNr, PersonalNr**) abhängig. Ändert sich beispielsweise der Stundenlohn für die Tätigkeit **Bearbeitung**, muss eine Änderung in mehreren Tupeln vorgenommen werden.



PersonalNr	ProjektNr	Tätigkeit	Stunden	Stundenlohn
0002	1	Leitung	25	50,00
0003	1	Bearbeitung	55	30,00
0004	1	Bearbeitung	70	30,00
0004	2	Leitung	25	50,00
0005	2	Präsentationsvorbereitung	160	35,00
0004	3	Leitung	25	50,00
0002	3	Bearbeitung	80	30,00
0003	3	Bearbeitung	65	30,00

Gehen Sie bei der Transformation von der 2. Normalform in die 3. Normalform wie folgt vor:

- Entfernen Sie für die 3. Normalform alle transitiven Abhängigkeiten durch Teilen der Relation in mehrere Relationen, in denen alle Nicht-Schlüsselfelder direkt vom gesamten Schlüsselfeld abhängig sind.

Die Relation **ARBEITET_AN** wird in zwei Relationen zerlegt. Der Stundenlohn und die Tätigkeit werden einer neuen Relation **TÄTIGKEIT** zugeordnet.

Relation ARBEITET_AN	Attribut		<table border="1"> <thead> <tr> <th>PersonalNr</th><th>ProjektNr</th><th>TätigkeitsNr</th><th>Stunden</th></tr> </thead> <tbody> <tr> <td>0002</td><td>1</td><td>1</td><td>25</td></tr> <tr> <td>0003</td><td>1</td><td>2</td><td>55</td></tr> <tr> <td>0004</td><td>1</td><td>2</td><td>70</td></tr> <tr> <td>0004</td><td>2</td><td>1</td><td>25</td></tr> </tbody> </table>	PersonalNr	ProjektNr	TätigkeitsNr	Stunden	0002	1	1	25	0003	1	2	55	0004	1	2	70	0004	2	1	25
PersonalNr	ProjektNr	TätigkeitsNr	Stunden																				
0002	1	1	25																				
0003	1	2	55																				
0004	1	2	70																				
0004	2	1	25																				
PersonalNr																							
ProjNr																							
TätigkeitsNr																							
Stunden																							
Relation TÄTIGKEIT	Attribut		<table border="1"> <thead> <tr> <th>TätigkeitsNr</th> <th>Tätigkeit</th> <th>Stundenlohn</th> </tr> </thead> <tbody> <tr> <td>1</td><td>Leitung</td><td>50,00</td></tr> <tr> <td>2</td><td>Bearbeitung</td><td>30,00</td></tr> <tr> <td>3</td><td>Präsentationsvorbereitung</td><td>35,00</td></tr> </tbody> </table>	TätigkeitsNr	Tätigkeit	Stundenlohn	1	Leitung	50,00	2	Bearbeitung	30,00	3	Präsentationsvorbereitung	35,00								
TätigkeitsNr	Tätigkeit	Stundenlohn																					
1	Leitung	50,00																					
2	Bearbeitung	30,00																					
3	Präsentationsvorbereitung	35,00																					
TätigkeitsNr																							
Tätigkeit																							
Stundenlohn																							

Relationen in 3. Normalform

Zusätzliche Schlüsselfelder einfügen

Lange Textfelder sind für Schlüsselfelder ungeeignet, da sie mehr Speicherplatz für den Index benötigen. Außerdem zieht eine Änderung des Textes auch eine Änderung in den Relationen nach sich, die in Beziehung zu dieser Relation stehen. Abhilfe schafft das Hinzufügen eines weiteren Schlüsselfeldes.

Die Tabellen erhalten damit beispielsweise folgende Struktur:

ARBEITET_AN (ProjNr, PersonalNr, Stunden, **TätigkeitsNr**)

TÄTIGKEIT (**TätigkeitsNr**, Tätigkeit, Stundenlohn)

Weitere Normalformen

Es gibt weitere Normalformen, die in der Praxis aber kaum Bedeutung haben. Mit jeder weiteren Stufe der Normalisierung kommen weitere Relationen hinzu, und damit ist häufig ein Geschwindigkeitsverlust bei Abfragen und Transaktionen verbunden.

Sie sollten versuchen, bei der Normalisierung des relationalen Modells einen guten Kompromiss zwischen der Speicherung von abhängigen Daten (eventuell auch redundanten Daten) und der Verarbeitungsgeschwindigkeit zu finden.

Boyce-Codd-Normalform (BCNF)

Die Abhängigkeiten von einzelnen Schlüsseln oder Schlüsselattributen untereinander werden bis zur 3. Normalform (3NF) nicht berücksichtigt. In der Boyce-Codd-Normalform werden sie beseitigt.

Eine Relation befindet sich in Boyce-Codd-Normalform, wenn kein Attribut funktional abhängig von einer Attributgruppe ohne Schlüsseleigenschaft ist.

Das bedeutet, dass in einer Relation keine funktionalen Abhängigkeiten zwischen einem minimalen Schlüssel (nicht zusammengesetzten Schlüssel) und den Attributen bestehen dürfen. Die meisten Relationen der 3NF befinden sich bereits in BCNF. Nur die Relationen, die mehrere überlappende, minimale Schlüssel besitzen, die eine funktionale Abhängigkeit erzeugen und somit wiederum Redundanzen hervorrufen, sind zu überführen.

4. und 5. Normalform

In der 4. Normalform befinden sich Relationen, die sich in der 3NF befinden und die keine paarweise auftretenden mehrwertigen Abhängigkeiten der Attribute enthalten. Eine mehrwertige Abhängigkeit besteht, wenn mehrere Nichtschlüsselattribute direkt vom Schlüssel abhängig sind, aber nicht voneinander abhängig sind. In der nebenstehenden Tabelle ist sowohl das Attribut *ProjektNr* als auch das Attribut *Kinder* direkt vom Attribut *PersonalNr* abhängig.

PersonalNr	ProjektNr	Kinder
0002	1	Hans
0002	1	Gerd
0003	2	Johanna
0003	2	Jens
0003	3	Johanna
0003	3	Jens

Relation mit mehrwertigen Abhängigkeiten

Ist es möglich, eine Relation in zwei Relationen zu zerlegen und die ursprüngliche Relation durch einen Natural Join dieser beiden Relationen wiederherzustellen, befindet sich die Ausgangsrelation noch nicht in der 4. Normalform.

! Eine Zerlegung in die 4. Normalform ist nicht immer möglich bzw. nötig. Wenn zwischen den Attributen inhaltliche Zusammenhänge bestehen, kann sich die Relation bereits in der 4NF befinden.

Ist eine verlustlose Zerlegung in Einzelabhängigkeiten in der 4. Normalform nicht möglich, werden in der 5. Normalform weitere Primärschlüssel hinzugefügt. Das geschieht so lange, bis nur noch Einzelabhängigkeiten der Attribute von einem oder mehreren Primärschlüsseln bestehen (keine Join Dependencies). Kann eine Relation nicht durch einen Join aus Einzelrelationen erstellt werden, befindet sie sich in der 5. Normalform.

Diese beiden Normalformen führen, wie auch die Boyce-Codd-Normalform, zu einer größeren Menge von Relationen.

Beispiel

Die räumliche Zuordnung von Arbeitsgruppen und Arbeitsplätzen ist in einer Relation gespeichert. Diese Relation befindet sich in der dritten Normalform, da der einzige Primärschlüssel die Kombination aller drei Attribute ist. Es liegen aber Redundanzen vor, da jede Arbeitsgruppe mehrere Arbeitsplätze benötigt und es in jedem Raum mehrere Arbeitsplätze gibt. Der Primärschlüssel besitzt somit mehrwertige Abhängigkeiten. Diese lassen sich durch Aufteilung in zwei Tabellen beseitigen.

Arbeitsgruppe	Raum	Arbeitsplatz
1	1	1
1	1	2
1	1	3
2	1	4
2	1	5
3	3	1
3	3	2
3	3	3

Relation in 3. Normalform

Die beiden Ergebnisrelationen befinden sich dann in der 4. und ebenfalls in der 5. Normalform, da eine verlustfreie Zerlegung möglich ist.

Arbeitsgruppe_Raum	Arbeitsgruppe	Raum
11	1	1
21	2	1
33	3	3

Arbeitsgruppe_Raum	Arbeitsplatz
11	1
11	2
11	3
21	4
21	5
33	1
33	2
33	3

Relation in 4. und 5. Normalform

3.4 Theorie relationaler Sprachen

Sprachen, die mit relationalen Datenbanken arbeiten, müssen in der Lage sein, folgende Operationen auszuführen:

- ✓ Anlegen von neuen Relationen
- ✓ Verändern von Relationen
- ✓ Löschen von Relationen
- ✓ Erzeugen von Relationen aus vorhandenen Relationen mit ausgewählten Tupeln und ausgewählten Attributen

Eine Datenbanksprache besitzt also Operationen zum Abfragen sowie zum Anlegen, Verändern und Löschen der Datensätze einer Datenbank. Das **Retrieval** (das Auffinden der gewünschten Daten) bildet den Kern der Sprache; es ist der schwierigere Teil einer Datenbankoperation. Sind die Daten erst einmal gefunden, können sie relativ einfach geändert bzw. gelöscht werden.

PersonalNr	ProjektNr	Tätigkeit	Stunden
0002	1	Leitung	25
0003	1	Bearbeitung	55
0004	1	Bearbeitung	70
0004	2	Leitung	25
0005	2	Präsentation	160
0004	3	Leitung	25
0002	3	Bearbeitung	80
0003	3	Bearbeitung	65

Relation ARBEITET_AN

Die meisten relationalen Datenbanksprachen, wie z. B. SQL, setzen sich aus zwei Sprachparadigmen zusammen, der Relationenalgebra und den Relationenkalkülen. Die Beispiele in diesem Kapitel werden mithilfe folgender Relationen erläutert:

PersonalNr	Name	Vorname	Abteilung	Gebdatum
0001	Eichenau	Maria	1	15.05.1987
0002	Glahn	Stefanie	2	02.05.1978
0003	Kirsch	Karin	2	24.05.1988
0004	Conolly	Sean	3	26.04.1976
0005	Frawley	Lutz	3	09.09.1979

ProjektNr	Beschreibung
1	Kundenumfrage
2	Verkaufsmesse
3	Konkurrenzanalyse

Relation PROJEKT1

Relation MITARBEITER1

PersonalNr	Name	Vorname	Abteilung	Gebdatum
0011	Schneider	Jakob	1	06.02.1968
0012	Unterwegner	Daniel	3	12.04.1973

ProjektNr	Beschreibung
1	Kundenumfrage
4	Wirtschaftlichkeitsanalyse

Relation PROJEKT2

Relation MITARBEITER2

Relationenalgebra

Als Algebra wird im Bereich der Mathematik ein System bezeichnet, welches aus nicht leeren Mengen und Operationen auf diesen Mengen besteht. Die nicht leeren Mengen sind in der Relationenalgebra die Relationen und die Operationen darauf sind die Datenbankoperationen, z. B. Datenbankanfragen. Durch die Datenbankoperationen werden aus den vorhandenen Relationen (den Basisrelationen) neue Relationen erzeugt, die aber nicht in der Datenbank gespeichert werden. Im Folgenden werden Operationen vorgestellt, die für eine relationale Sprache unverzichtbar sind.

Projektion Symbol: π	<p>In der Projektion werden nur bestimmte Attribute einer Relation (Spalten einer Tabelle) in einer Relation dargestellt, die übrigen werden weggelassen. Doppelte Tupel werden dabei nicht entfernt.</p> <p>Beispiel: Gesucht sind die Attribute <i>Nachname</i>, <i>Vorname</i> und <i>Gebdatum</i> der Relation <i>Mitarbeiter1</i>.</p> <table border="1"> <thead> <tr> <th>Name</th><th>Vorname</th><th>Gebdatum</th></tr> </thead> <tbody> <tr> <td>Eichenau</td><td>Maria</td><td>15.05.1987</td></tr> <tr> <td>Glahn</td><td>Stefanie</td><td>02.05.1978</td></tr> <tr> <td>Kirsch</td><td>Karin</td><td>24.05.1988</td></tr> <tr> <td>Conolly</td><td>Sean</td><td>26.04.1976</td></tr> <tr> <td>Frawley</td><td>Lutz</td><td>09.09.1979</td></tr> </tbody> </table> <p>(Beispiel: Kapitel 7, <i>Datenabfrage2.sql</i>)</p>	Name	Vorname	Gebdatum	Eichenau	Maria	15.05.1987	Glahn	Stefanie	02.05.1978	Kirsch	Karin	24.05.1988	Conolly	Sean	26.04.1976	Frawley	Lutz	09.09.1979
Name	Vorname	Gebdatum																	
Eichenau	Maria	15.05.1987																	
Glahn	Stefanie	02.05.1978																	
Kirsch	Karin	24.05.1988																	
Conolly	Sean	26.04.1976																	
Frawley	Lutz	09.09.1979																	
Selektion Symbol: σ	<p>Die Selektion liefert eine Teilmenge aller Tupel einer Relation. Entsprechend der Selektionsbedingung werden diese Tupel ermittelt. Die Teilmenge enthält alle Attribute der Basisrelation.</p> <p>Beispiel: Gesucht sind alle Tupel aus der Relation <i>ARBEITET_AN</i>, in denen die Projektnummer den Wert 2 hat.</p> <table border="1"> <thead> <tr> <th>PersonalNr</th> <th>ProjektNr</th> <th>Tätigkeit</th> <th>Stunden</th> </tr> </thead> <tbody> <tr> <td>0004</td> <td>2</td> <td>Leitung</td> <td>25</td> </tr> <tr> <td>0005</td> <td>2</td> <td>Präsentation</td> <td>160</td> </tr> </tbody> </table> <p>(Beispiel: Kapitel 7, <i>Datenabfrage2.sql</i>)</p>	PersonalNr	ProjektNr	Tätigkeit	Stunden	0004	2	Leitung	25	0005	2	Präsentation	160						
PersonalNr	ProjektNr	Tätigkeit	Stunden																
0004	2	Leitung	25																
0005	2	Präsentation	160																

Vereinigung Symbol: \cup	<p>Wenn zwei Relationen A und B die gleichen Attribute (mit gleichen Namen und Wertebereichen) besitzen, ist es möglich, eine Vereinigung $V = A \cup B$ zu bilden. Die Vereinigung V enthält dann alle Tupel beider Mengen.</p> <p>Beispiel: Gesucht wird die Vereinigung der Relationen <i>MITARBEITER1</i> und <i>MITARBEITER2</i>.</p> $\text{MITARBEITER} := \text{MITARBEITER1} \cup \text{MITARBEITER2}$ <table border="1" data-bbox="531 460 1214 729"> <thead> <tr> <th>PersonalNr</th><th>Name</th><th>Vorname</th><th>Abteilung</th><th>Gebdatum</th></tr> </thead> <tbody> <tr><td>0001</td><td>Eichenau</td><td>Maria</td><td>1</td><td>15.05.1987</td></tr> <tr><td>0002</td><td>Glahn</td><td>Stefanie</td><td>2</td><td>02.05.1978</td></tr> <tr><td>0003</td><td>Kirsch</td><td>Karin</td><td>2</td><td>24.05.1988</td></tr> <tr><td>0004</td><td>Conolly</td><td>Sean</td><td>3</td><td>26.04.1976</td></tr> <tr><td>0005</td><td>Frawley</td><td>Lutz</td><td>3</td><td>09.09.1979</td></tr> <tr><td>0011</td><td>Schneider</td><td>Jakob</td><td>1</td><td>06.02.1968</td></tr> <tr><td>0012</td><td>Unterwegner</td><td>Daniel</td><td>3</td><td>12.04.1973</td></tr> </tbody> </table> <p>(Beispiel: Kapitel 10, <i>Union.sql</i>)</p>	PersonalNr	Name	Vorname	Abteilung	Gebdatum	0001	Eichenau	Maria	1	15.05.1987	0002	Glahn	Stefanie	2	02.05.1978	0003	Kirsch	Karin	2	24.05.1988	0004	Conolly	Sean	3	26.04.1976	0005	Frawley	Lutz	3	09.09.1979	0011	Schneider	Jakob	1	06.02.1968	0012	Unterwegner	Daniel	3	12.04.1973
PersonalNr	Name	Vorname	Abteilung	Gebdatum																																					
0001	Eichenau	Maria	1	15.05.1987																																					
0002	Glahn	Stefanie	2	02.05.1978																																					
0003	Kirsch	Karin	2	24.05.1988																																					
0004	Conolly	Sean	3	26.04.1976																																					
0005	Frawley	Lutz	3	09.09.1979																																					
0011	Schneider	Jakob	1	06.02.1968																																					
0012	Unterwegner	Daniel	3	12.04.1973																																					
Differenz Symbol: -	<p>Die Bildung der Differenz ist nur dann möglich, wenn die zwei Relationen A und B, analog zur Vereinigung, die gleichen Attribute (mit gleichen Namen und Wertebereichen) besitzen. Dann sind in der Ergebnisrelation $D := A - B$ alle Tupel aus A enthalten, die nicht in B enthalten sind.</p> <p>Beispiel: Die Anwendung der Differenz der Relationen <i>PROJEKT</i> und <i>PROJEKT2</i> ergibt folgende Ergebnisrelation:</p> $\text{DIFFERENZ} := \text{PROJEKT} - \text{PROJEKT2}$ <table border="1" data-bbox="531 1066 865 1179"> <thead> <tr> <th>ProjektNr</th><th>Beschreibung</th></tr> </thead> <tbody> <tr><td>2</td><td>Verkaufsmesse</td></tr> <tr><td>3</td><td>Konkurrenzanalyse</td></tr> </tbody> </table> <p>(vgl. Abschnitt 10.5 Schnitt- und Differenzmengen)</p>	ProjektNr	Beschreibung	2	Verkaufsmesse	3	Konkurrenzanalyse																																		
ProjektNr	Beschreibung																																								
2	Verkaufsmesse																																								
3	Konkurrenzanalyse																																								
Durchschnitt Symbol: \cap	<p>Der Durchschnitt der beiden Relationen A und B enthält nur die Tupel aus A, die auch in B enthalten sind. Voraussetzung ist wieder, dass die Namen und Wertebereiche der Attribute beider Relationen übereinstimmen.</p> <p>Beispiel: Der Durchschnitt der Relationen <i>PROJEKT</i> und <i>PROJEKT2</i> ergibt folgende Ergebnismenge:</p> $\text{Durchschnitt} := \text{PROJEKT} \cap \text{PROJEKT2}$ <table border="1" data-bbox="531 1471 865 1538"> <thead> <tr> <th>ProjektNr</th><th>Beschreibung</th></tr> </thead> <tbody> <tr><td>1</td><td>Kundenumfrage</td></tr> </tbody> </table> <p>(vgl. Abschnitt 10.5 Schnitt- und Differenzmengen)</p>	ProjektNr	Beschreibung	1	Kundenumfrage																																				
ProjektNr	Beschreibung																																								
1	Kundenumfrage																																								
Umbenennen Symbol: \leftarrow	<p>Attribute lassen sich in einer anderen Relation mit einem anderen Attributnamen darstellen. Die Domains (Wertebereiche) bleiben aber erhalten. So können Relationen für den Benutzer mit anderen, aussagekräftigeren Attributnamen angezeigt werden. Ein wichtiger Anwendungsfall der Umbenennung ist der, dass für einige Operationen gleiche Attributnamen vorausgesetzt werden, beispielsweise für eine Vereinigung zweier Relationen. Sollen zwei Relationen mit gleichen Wertebereichen, aber mit verschiedenen Attributnamen vereinigt werden, müssen die Attributnamen der einen Relation zuvor umbenannt werden.</p>																																								

Umbenennen Symbol: ←	<p>Beispiel: Bei den Attributen der Relation <i>ARBEITET_AN</i> soll <i>PersonalNr</i> in <i>Personalnummer</i> umbenannt werden.</p> <p><i>Ergebnisrelation := ARBEITET_AN</i> $\text{PersonalNr} \leftarrow \text{Personalnummer}$</p> <table border="1" data-bbox="531 359 1079 494"> <thead> <tr> <th>PersonalNr</th><th>ProjektNr</th><th>Tätigkeit</th><th>Stunden</th></tr> </thead> <tbody> <tr> <td>0002</td><td>1</td><td>Leitung</td><td>25</td></tr> <tr> <td>0003</td><td>1</td><td>Bearbeitung</td><td>55</td></tr> <tr> <td>...</td><td>...</td><td>...</td><td>...</td></tr> </tbody> </table> <p>(Beispiel: Kapitel 7, <i>Datenabfrage3.sql</i>)</p>	PersonalNr	ProjektNr	Tätigkeit	Stunden	0002	1	Leitung	25	0003	1	Bearbeitung	55																																	
PersonalNr	ProjektNr	Tätigkeit	Stunden																																															
0002	1	Leitung	25																																															
0003	1	Bearbeitung	55																																															
...																																															
Kartesisches Produkt Symbol: x	<p>Das kartesische Produkt zweier Relationen $K = A \times B$ wird wie in der Mengenlehre gebildet. Es werden alle Tupel der einen Relation mit allen Tupeln der anderen Relation kombiniert.</p> <p>Beispiel: Gesucht wird das kartesische Produkt der Relationen <i>MITARBEITER2</i> und <i>PROJEKT</i>. Welcher Mitarbeiter könnte an welchem Projekt mitarbeiten?</p> <p><i>Ergebnisrelation := MITARBEITER2 x PROJEKT</i></p> <table border="1" data-bbox="404 842 1357 1066"> <thead> <tr> <th>PersonalNr</th><th>Name</th><th>Vorname</th><th>Abteilung</th><th>Gebdatum</th><th>ProjektNr</th><th>Beschreibung</th></tr> </thead> <tbody> <tr> <td>0011</td><td>Schneider</td><td>Jakob</td><td>1</td><td>06.02.1968</td><td>1</td><td>Kundenumfrage</td></tr> <tr> <td>0011</td><td>Schneider</td><td>Jakob</td><td>1</td><td>06.02.1968</td><td>2</td><td>Verkaufsmesse</td></tr> <tr> <td>0011</td><td>Schneider</td><td>Jakob</td><td>1</td><td>06.02.1968</td><td>3</td><td>Konkurrenzanalyse</td></tr> <tr> <td>0012</td><td>Unterwegner</td><td>Daniel</td><td>3</td><td>12.04.1973</td><td>1</td><td>Kundenumfrage</td></tr> <tr> <td>0012</td><td>Unterwegner</td><td>Daniel</td><td>3</td><td>12.04.1973</td><td>1</td><td>Verkaufsmesse</td></tr> <tr> <td>0012</td><td>Unterwegner</td><td>Daniel</td><td>3</td><td>12.04.1973</td><td>1</td><td>Konkurrenzanalyse</td></tr> </tbody> </table> <p>(Beispiel: Kapitel 10, <i>CrossJoin.sql</i>)</p>	PersonalNr	Name	Vorname	Abteilung	Gebdatum	ProjektNr	Beschreibung	0011	Schneider	Jakob	1	06.02.1968	1	Kundenumfrage	0011	Schneider	Jakob	1	06.02.1968	2	Verkaufsmesse	0011	Schneider	Jakob	1	06.02.1968	3	Konkurrenzanalyse	0012	Unterwegner	Daniel	3	12.04.1973	1	Kundenumfrage	0012	Unterwegner	Daniel	3	12.04.1973	1	Verkaufsmesse	0012	Unterwegner	Daniel	3	12.04.1973	1	Konkurrenzanalyse
PersonalNr	Name	Vorname	Abteilung	Gebdatum	ProjektNr	Beschreibung																																												
0011	Schneider	Jakob	1	06.02.1968	1	Kundenumfrage																																												
0011	Schneider	Jakob	1	06.02.1968	2	Verkaufsmesse																																												
0011	Schneider	Jakob	1	06.02.1968	3	Konkurrenzanalyse																																												
0012	Unterwegner	Daniel	3	12.04.1973	1	Kundenumfrage																																												
0012	Unterwegner	Daniel	3	12.04.1973	1	Verkaufsmesse																																												
0012	Unterwegner	Daniel	3	12.04.1973	1	Konkurrenzanalyse																																												
Verbund (Join) Symbol: ⚤	<p>Nicht immer interessiert das gesamte kartesische Produkt, sondern nur ein bestimmter Teil davon. Eine solche Teilmenge liefert der Verbund. Er besitzt eine Bedingung, die eine Einschränkung der Ergebnismenge bewirkt. Es gibt verschiedene Möglichkeiten, zwei Relationen durch einen Verbund zu kombinieren.</p>																																																	
Natürlicher Verbund (Natural-Join)	<p>Zwei Relationen R1 und R2 werden über die Gleichheit zweier Attribute miteinander verbunden. Dabei ist a_i das i-te Attribut der Relation R1 und a_j ist das j-te Attribut der Relation R2.</p> $V := R1 \bowtie_{R1(a_i)=R2(a_j)} R2$ <p>In die Ergebnisrelation werden nur die Tupel übernommen, bei denen die Attribute a_i und a_j der Relationen R1 und R2 gleich sind. Das Vergleichsattribut ist in der Zielrelation nur einmal enthalten (obwohl es praktisch zweimal vorhanden ist).</p>																																																	

Natürlicher Verbund (Natural-Join) Fortsetzung	<p>Beispiel: Gesucht wird ein natürlicher Verbund zwischen den Relationen <i>MITARBEITER1</i> und <i>ARBEITET_AN</i> mit der Bedingung, dass das Attribut <i>PersonalNr</i> in beiden Relationen übereinstimmt (somit gibt es auch nur ein Attribut <i>PersonalNr</i>, obwohl es in beiden Relationen vorhanden ist).</p> <p><i>VERBUND := MITARBEITER1 ⋈_{PersonalNr= PersonalNr} ARBEITET_AN</i></p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th>PersonalNr</th><th>Name</th><th>Vorname</th><th>Abteilung</th><th>Gebdatum</th><th>ProjektNr</th><th>Beschreibung</th><th>Stunden</th></tr> </thead> <tbody> <tr><td>0002</td><td>Glahn</td><td>Stefanie</td><td>2</td><td>02.05.1978</td><td>1</td><td>Leitung</td><td>25</td></tr> <tr><td>0003</td><td>Kirsch</td><td>Karin</td><td>2</td><td>24.05.1988</td><td>1</td><td>Bearbeitung</td><td>55</td></tr> <tr><td>0004</td><td>Conolly</td><td>Sean</td><td>3</td><td>26.04.1976</td><td>1</td><td>Bearbeitung</td><td>70</td></tr> <tr><td>0004</td><td>Conolly</td><td>Sean</td><td>3</td><td>26.04.1976</td><td>2</td><td>Leitung</td><td>25</td></tr> <tr><td>0005</td><td>Frawley</td><td>Lutz</td><td>3</td><td>09.09.1979</td><td>2</td><td>Präsentation</td><td>160</td></tr> <tr><td>0004</td><td>Conolly</td><td>Sean</td><td>3</td><td>26.04.1976</td><td>3</td><td>Leitung</td><td>25</td></tr> <tr><td>0002</td><td>Glahn</td><td>Stefanie</td><td>2</td><td>02.05.1978</td><td>3</td><td>Bearbeitung</td><td>80</td></tr> <tr><td>0003</td><td>Kirsch</td><td>Karin</td><td>2</td><td>24.05.1988</td><td>3</td><td>Bearbeitung</td><td>65</td></tr> </tbody> </table> <p>(Beispiel: Kapitel 10, <i>NaturalJoin.sql</i>)</p>	PersonalNr	Name	Vorname	Abteilung	Gebdatum	ProjektNr	Beschreibung	Stunden	0002	Glahn	Stefanie	2	02.05.1978	1	Leitung	25	0003	Kirsch	Karin	2	24.05.1988	1	Bearbeitung	55	0004	Conolly	Sean	3	26.04.1976	1	Bearbeitung	70	0004	Conolly	Sean	3	26.04.1976	2	Leitung	25	0005	Frawley	Lutz	3	09.09.1979	2	Präsentation	160	0004	Conolly	Sean	3	26.04.1976	3	Leitung	25	0002	Glahn	Stefanie	2	02.05.1978	3	Bearbeitung	80	0003	Kirsch	Karin	2	24.05.1988	3	Bearbeitung	65
PersonalNr	Name	Vorname	Abteilung	Gebdatum	ProjektNr	Beschreibung	Stunden																																																																		
0002	Glahn	Stefanie	2	02.05.1978	1	Leitung	25																																																																		
0003	Kirsch	Karin	2	24.05.1988	1	Bearbeitung	55																																																																		
0004	Conolly	Sean	3	26.04.1976	1	Bearbeitung	70																																																																		
0004	Conolly	Sean	3	26.04.1976	2	Leitung	25																																																																		
0005	Frawley	Lutz	3	09.09.1979	2	Präsentation	160																																																																		
0004	Conolly	Sean	3	26.04.1976	3	Leitung	25																																																																		
0002	Glahn	Stefanie	2	02.05.1978	3	Bearbeitung	80																																																																		
0003	Kirsch	Karin	2	24.05.1988	3	Bearbeitung	65																																																																		

Weitere Verbundoperationen:	
Theta-Join	Im Theta-Join sind auch die Vergleichsoperatoren <, > ... zulässig. Das Vergleichsattribut ist zweimal vorhanden. (Beispiel: Kapitel 10, <i>ThetaJoin.sql</i>)
Equi-Join	Der Equi-Join erlaubt nur den Vergleichsoperator =, das Vergleichsattribut ist zweimal vorhanden. (Beispiele: Kapitel 10, <i>EinfacheVerknuepfung1.sql</i> und <i>InnerJoin.sql</i>)
Self-Join	Im Self-Join werden Tupel einer Relation miteinander verbunden. (Beispiel: Kapitel 11, <i>SelfJoin.sql</i>)
Semi-Join	Der Semi-Join als spezielle Form des Equi-Joins dient der Verknüpfung von verteilten Datenbanken über Operationen.
Outer-Join	Alle zuvor erwähnten Verbundoperationen gehören zu den Inner-Joins, die nur Tupel übernehmen, die die Bedingung erfüllen. Im Outer-Join werden auch die Tupel übernommen, für die kein entsprechendes Tupel der anderen Relation vorhanden ist. Im verwendeten Beispiel (natürlicher Verbund) erscheint beim Outer-Join der Mitarbeiter mit der <i>PersonalNr</i> 0001. Die Attribute <i>ProjektNr</i> , <i>Tätigkeit</i> und <i>Stunden</i> sind hier leer. Es gibt 3 Formen des Outer-Joins: den Left-Outer-Join (alle Tupel der linken Relation werden übernommen), den Right-Outer-Join (alle Tupel der rechten Relation werden übernommen) und den symmetrischen Outer-Join (Vereinigung des Left- und des Right-Outer-Joins). (Beispiel: Kapitel 10, <i>OuterJoin.sql</i>)

Relationenkalkül

Während die Relationenalgebra prozedural aufgebaut ist (durch Angabe einer Operation wird eine Ergebnismenge berechnet), wird im Relationenkalkül die Menge der auszuwählenden Tupel deskriptiv, ohne Angabe der anzuwendenden Operation aufgeführt. Das Relationenkalkül wird in zwei Ausprägungen unterteilt, in das Tupelkalkül (tupelorientiertes Relationenkalkül) und das Domainkalkül (werteorientiertes Relationenkalkül).

- ✓ Ein relationales Tupelkalkül wird in Anlehnung an die Prädikatenlogik in der Form

$$\{t \mid P(t)\}$$

angegeben. Dabei ist t die Tupelvariable und $P(t)$ das Prädikat, das erfüllt sein muss, damit ein Tupel in die Ergebnisrelation aufgenommen wird.

- ✓ Im Domainkalkül werden Variablen für Wertebereiche (nicht für Tupel) eingesetzt. Das Domainkalkül wird in der Form

$$\{[d_1, d_2, \dots, d_n] \mid P(d_1, d_2, \dots, d_n)\}$$

notiert; d_i ist eine Domainvariable, die für einen Attributwert steht, und P ist ein Prädikat.

Die Prädikate sind logische Ausdrücke, für deren Formulierung neben den Vergleichsoperatoren ($>$, $<$, $=$, \neq ...) die folgenden Zeichen verwendet werden:

Zeichen	Name, Bedeutung	Beispiel
\wedge	Logisches Und (AND)	$a \wedge b \rightarrow$ Der Ausdruck ist wahr, wenn a und b wahr sind.
\vee	Logisches Oder (OR)	$a \vee b \rightarrow$ Der Ausdruck ist wahr, wenn a oder b wahr ist.
\neg	Negation (NOT)	$\neg a \rightarrow$ Der Ausdruck ist wahr, wenn a falsch ist.
\forall	Allquantor, "für alle ... "	$(\forall x)b \rightarrow$ Für alle Werte x ist der Ausdruck b wahr.
\exists	Existenzquantor, "es existiert ..."	$(\exists x)b \rightarrow$ Es existiert ein Wert x , sodass der Ausdruck b wahr ist.

Beispiele

Es werden alle Personalnummern in der Relation *ARBEITET_AN* gesucht, die an dem Projekt mit der Nummer 1 mitarbeiten:

$$\{\text{ARBEITET_AN}.\text{PersonalNr} \mid \text{ARBEITET_AN}.\text{ProjektNr} = '1'\}$$

Es werden die Namen der Personen gesucht, die an dem Projekt mit der Nummer 2 mitarbeiten. Dazu werden die Relationen *MITARBEITER1* und *ARBEITET_AN* ausgewertet:

$$\{\text{MITARBEITER1}.\text{Name} \mid \exists \text{ARBEITET_AN}(\text{ARBEITET_AN}.\text{ProjektNr} = '2') \wedge (\text{ARBEITET_AN}.\text{PersonalNr} = \text{MITARBEITER1}.\text{PersonalNr})\}$$

Es werden die Namen aller Personen gesucht, die an dem Projekt mitarbeiten, das vom Mitarbeiter 0004 geleitet wird:

$$\begin{array}{ll} \text{ARBEITET_AN} \text{ as } A & A \text{ wird als Tupelvariable für den Typ ARBEITET_AN definiert.} \\ \text{ARBEITET_AN} \text{ as } B & B \text{ wird als Tupelvariable für den Typ ARBEITET_AN definiert.} \\ \{\text{MITARBEITER1}.\text{Name} \mid \exists A(A.\text{PersonalNr} = \text{MITARBEITER1}.\text{PersonalNr}) \wedge \exists B(B.\text{Tätigkeit} = 'Leitung' \wedge B.\text{PersonalNr} = '0004'))\} & \end{array}$$

Es werden die Namen aller Personen gesucht, die an keinem Projekt mitarbeiten:

ARBEITET_AN as A A wird als Tupelvariable für den Typ *ARBEITET_AN* definiert.
 $\{MITARBEITER1.Name \mid \forall A(A.\text{PersonalNr} \neq MITARBEITER1.\text{PersonalNr})\}$

3.5 Übung

Aufgaben zur Datenbanktheorie

Übungsdatei: --

Ergebnisdateien: *Uebung1.pdf*, *Uebung2.pdf*, *Uebung3.pdf*,
Uebung4.pdf, *Uebung5.pdf*, *Uebung6.pdf*

1. Die Daten eines Busreise-Unternehmens sollen in einer Datenbank gespeichert werden. Die folgende Abbildung zeigt die Daten in einer nicht normalisierten Relation:

Attribut	Datentyp	Beschreibung
FahrtNr	Zahl	Nummer der Fahrt
BusNr	Zahl	Nummer des Busses
Hersteller	Text	Herstellerfirma des Busses
AnzahlPlaetze	Zahl	Anzahl der Plätze im Bus
POLKZ	Zahl	Polizeiliches Kennzeichen des Busses
Erstzulassung	Datum	Datum der ersten Zulassung des Busses
Anschaffungspreis	Zahl (Währung)	Anschaffungspreis des Busses
gewartet_am	Datum	Datum der Wartung
gewartet_von	Text	Bus wurde gewartet von Firma
Maengel	Text	Bei der Wartung festgestellte Mängel
Kosten	Zahl (Währung)	Kosten für Wartung und Reparatur
FahrtZiel	Text	Ort, an den der Bus fahren soll
FahrtDatum	Datum	Abfahrtsdatum
Abfahrtzeit	Zeit	Abfahrtszeit
Fahrzeit	Zahl	Dauer der Fahrt
FahrerName	Text	Name des Fahrers
FahrerVorname	Text	Vorname des Fahrers
FahrerStrasse	Text	Straße, in der der Fahrer wohnt
FahrerPLZ	Text	PLZ der Anschrift des Fahrers
FahrerWohnort	Text	Wohnort des Fahrers
FahrerTelefon	Text	Telefonnummer des Fahrers
FahrerGehalt	Zahl (Währung)	Gehalt des Fahrers
gebuchtePlaetze	Zahl	Anzahl der für die Fahrt gebuchten Plätze

Entwerfen Sie ein Entity-Relationship-Modell für die vorgegebenen Daten. Versuchen Sie dabei, zusammengehörige Daten zu finden und diese in mehreren Relationen zusammenzufassen.

2. Überführen Sie dieses ER-Modell in das relationale Modell.
3. Normalisieren Sie den Datenbankentwurf (bis zur dritten Normalform). Achten Sie auf transitive Abhängigkeiten.
4. Was ist bei der Datenerfassung in Bezug auf die Schlüsselwerte zu beachten?
5. Welche Beziehungen bestehen zwischen den Relationen des Beispiels Busreise-Unternehmen? Wie werden diese Beziehungen im Relationenmodell umgesetzt?
6. Erstellen Sie für jede Relation eine Beispieldtabelle mit einigen Beispielwerten.

4

Datenbanken

4.1 Die Datenbankabfragesprache SQL

Die Datenbankabfragesprache SQL wurde aus der Abfragesprache SEQUEL (Structured English QUEry Language) der Firma IBM entwickelt. Mit der Entwicklung der Sprache wurde das Ziel verfolgt, auch für Nicht-Programmierer eine relativ einfache Sprache zur Abfrage von Datenbanken zur Verfügung zu stellen, die keine mathematischen Notationen (z. B. die Quantoren \forall und \exists) verwendet.

SQL entwickelte sich zum internationalen Standard und wurde 1986 vom ANSI (American National Standards Institute) als Standard SQL1 (SQL-86) festgelegt. Im Laufe der Jahre wurde die Sprache intensiv weiterentwickelt und ANSI veröffentlichte weitere Standards, die analog zu dem betreffenden Jahr mit SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011 und SQL:2016 bezeichnet wurden. Für SQL-92 wird allgemein auch die Bezeichnung SQL2 und für SQL:1999 die Bezeichnung SQL3 verwendet. Die aktuelle Version SQL:2016 wurde im Dezember des Jahres 2016 veröffentlicht. Der Standard wurde dabei im Vergleich zur Vorversion neben einer Reihe optionaler Ergänzungen um zwei Funktionen erweitert:

- ✓ Die Unterstützung von JSON

Die JavaScript Object Notation (JSON) dient als kompaktes Datenformat dem Datenaustausch zwischen Anwendungen und ist besonders im Bereich der Web- und mobilen Anwendungen weit verbreitet. Das Format besitzt eine einfach lesbare Textform, ist unabhängig von Programmiersprachen und wird in allen wichtigen Programmiersprachen durch sogenannte Parser zur Analyse der Daten unterstützt.

- ✓ Die Unterstützung des sogenannten *row pattern matching*

Diese Funktion dient der Analyse von Tabellenzeilen zur Erkennung von Ähnlichkeiten und Trends unter Verwendung regulärer Ausdrücke, befindet sich jedoch noch im Anfangsstadium. So werden nicht alle regulären Ausdrücke unterstützt und auch die praktische Anwendung ist zum Zeitpunkt der Erstellung dieses Buches nur eingeschränkt möglich, lediglich das DBMS Oracle unterstützt einige Möglichkeiten.

Im Jahr 2019 wurde der Standard um Teil 15, Multi-Dimensional Arrays (SQL/MDA, Verwendung von SQL in Verbindung mit mehrdimensionalen Feldern), ergänzt. Aktuell befindet sich Teil 16, SQL Property Graph Queries (SQL/PGQ, Graphen-basierte Datenabfragen), in der Entwicklung.

Die Spezifikation umfasst sowohl die Syntax und Semantik der Datenbankabfrage- und Datenmanipulationssprache SQL als auch Konzepte des DBS (Transaktionen, Zugriffsschutz, Sicherung der Integrität, Implementierung objektorientierter Funktionalität etc.).

Bei all den Weiterentwicklungen der Sprache basieren aktuelle Datenbanken in der Regel immer noch auf dem SQL2-Standard. Die Modifikationen der nachfolgenden Standards werden dabei in unterschiedlichem Maß in die Datenbanken implementiert.

Der Sprachumfang von SQL2 wird durch die drei Conformance Level untergliedert:

Entry-Level	Der Entry-Level wird von den meisten relationalen DBS unterstützt und entspricht nahezu dem ANSI-Standard von 1989. Er umfasst die grundlegenden Befehle zum Anlegen von Datenbanken und Tabellen, zu deren Bearbeitung und Verwaltung (z. B. CREATE-, SELECT-, SHOW-Anweisungen).
Intermediate-Level	Der Intermediate-Level bietet zusätzliche Funktionalität (z. B. einen zusätzlichen Datums- und Zeit-Datentyp, Mengenoperationen, dynamisches SQL).
Full-Level	Der Full-Level enthält Funktionen, die bis jetzt nur unvollständig in DBS implementiert sind (z. B. Constraints über mehrere Tabellen, Constraint-Modi, SELECT-Befehle in der FROM-Klausel).

Die Hersteller von DBMS implementieren meist die gesamte Funktionalität des SQL2-Entry-Levels und Teile der anderen Level. Häufig werden aber auch eigene Erweiterungen eingebaut, was die Portierung erschwert. Anpassungen lassen sich aber relativ leicht durchführen.

Ab SQL:1999 wird die Sprache nicht mehr in die drei Level eingeteilt. Ein Kernsystem Core-SQL enthält einen großen Teil des Funktionsumfangs der Sprache (entsprechend dem Full-Level). Zusätzlich werden Erweiterungen angeboten. Von diesen ursprünglich 14 Stück sind noch 10 relevant:

- ✓ Teil 1: SQL/Framework
- ✓ Teil 2: SQL/Foundation
- ✓ Teil 3: SQL/CLI (Call-Level Interface)
- ✓ Teil 4: SQL/PSM (Persistent Stored Modules)
- ✓ Teil 9: SQL/MED (Management of External Data)
- ✓ Teil 10: SQL/OLB (Object Language Bindings)
- ✓ Teil 11: SQL/Schemata (Information and Definition Schemas)
- ✓ Teil 13: SQL/JRT (SQL Routines and Types Using the Java TM Programming Language)
- ✓ Teil 14: SQL/XML (XML-Related Specifications)
- ✓ Teil 15: SQL/MDA (Multi-Dimensional Arrays)

In diesen Bereich gehört auch die Erweiterung zu JSON. Zusätzlich gibt es noch weitere SQL-Pakete für Multimedia und Anwendungen.

Für die Version SQL:2008 entschied die Standardisierungskommission, dass vier der neun Teile – SQL/CLI, SQL/MED, SQL/OLB und SQL/JRT – stabil sind und keine weitere Überarbeitung benötigen. Somit betreffen die späteren Neuerungen nur die Teile SQL/Framework, SQL/Foundation, SQL/ PSM, SQL/Schemata und SQL/XML.

Zwecks Einsicht können auf der Webseite <http://www.iso.org> die Definitionen der einzelnen Teile des Standards erworben werden. Zu erreichen ist Liste der Dokumente über die Suche nach '9075'.

SQL ist eine interaktive Sprache und erlaubt Ad-hoc-Abfragen von Datenbanken (Abfragen, die sofort ausgeführt werden). Über sogenannte SQL-Skripts können Sie mehrere SQL-Anweisungen automatisiert ausführen lassen. Außer zur interaktiven Eingabe wird SQL auch in Anwendungen zur Kommunikation mit einer Datenbank eingesetzt. Dabei können lesende wie auch schreibende Zugriffe durchgeführt werden.

Zum Sprachumfang von SQL gehören vier Befehlsgruppen:

DDL (Data Definition Language)	Erstellen von Datenbanken, Tabellen (Relationen) und Indizes
DQL (Data Query Language)	Abfragen von Daten
DML (Data Manipulation Language)	Anlegen, Ändern und Löschen von Datensätzen
DCL (Data Control Language)	Anlegen von Benutzern und Vergabe von Zugriffsrechten

4.2 Datenbank erstellen

Beim Aufbau einer Datenbank gehen Sie prinzipiell wie folgt vor:

- ▶ Legen Sie zuerst die Datenbank an.
Sie dient als Container für die Tabellen und alle weiteren Datenbankobjekte.
- ▶ Erstellen Sie danach die benötigten Tabellen.
Dabei legen Sie zunächst nur die Struktur der Tabellen fest, d. h., welche Spalten sie enthalten sollen und welche Datentypen darin gespeichert werden und legen den Primärschlüssel für die Tabelle fest.
- ▶ Füllen Sie jetzt die Tabellen mit den Daten.
Tabellen sind die einzigen Datenbankobjekte, in denen die Daten physisch gespeichert werden. Nun können Sie Auswertungen der Daten vornehmen oder weitere Aktionen ausführen, z. B. Daten ändern oder löschen.

Der erste Schritt beim Erstellen einer Datenbank ist das Anlegen der Datenbank selbst. In ihr werden später die Daten in Form von mehreren Tabellen gespeichert.

Sie können unbegrenzt viele Datenbanken erstellen und so die zu speichernden Daten nach Themengebieten zusammenfassen. Falls unterschiedliche Benutzer mit dem Datenbanksystem arbeiten, sollten Sie auf die korrekte Vergabe von Zugriffsrechten für die einzelnen Benutzer achten.

```
C:\WINDOWS\system32\cmd.exe
Aktive Codepage: 1252.
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.5.10-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> CREATE DATABASE mav;
Query OK, 1 row affected (0.237 sec)

MariaDB [(none)]> USE mav;
Database changed
MariaDB [mav]>
```

Erstellen und Auswählen einer neuen Datenbank in MariaDB

Über die Anweisung CREATE DATABASE erstellen Sie eine neue Datenbank. Diese Datenbank enthält noch keine eigenen Tabellen. Oft sind jedoch bereits einige Systemtabellen enthalten, die z. B. Informationen über die Datenbank und die Datenbankobjekte enthalten, sogenannte Metadaten.

Um eine Datenbank erstellen zu können, benötigen Sie das Ausführungsrecht für die CREATE-DATABASE-Anweisung. Der Administrator vergibt dieses Recht mit der Anweisung GRANT CREATE DATABASE. Der Benutzer, der die Datenbank erstellt, wird danach zu ihrem Eigentümer.

Datenbank in MariaDB erstellen

In MariaDB können Sie eine neue Datenbank entweder über den Datenbank-Client *mariadb monitor* erstellen, der zum Lieferumfang von MariaDB für die verschiedenen Betriebssysteme gehört, oder Sie verwenden eines der Hilfsprogramme mit grafischer Oberfläche, die es für MariaDB gibt.

Aufgrund ihrer Herkunft als Ableger von MySQL verwendet MariaDB an vielen Stellen Komponenten von MySQL, so dass sich dieser Name in der Bezeichnung von Zusatzprogrammen und Befehlen wiederfindet.

Über die folgenden Anweisungen erstellen Sie eine neue Datenbank in MariaDB und wählen diese aus.

①	CREATE DATABASE IF NOT EXISTS mav;
②	USE mav;

- ① Mit dieser Anweisung wird eine neue Datenbank mit dem Namen *mav* erstellt, falls diese noch nicht existiert.
- ② Die neue Datenbank wird ausgewählt, damit sie bearbeitet werden kann. Es kann immer nur eine Datenbank ausgewählt sein.

Das Erstellen einer Datenbank in MariaDB bedeutet das Erzeugen eines neuen Verzeichnisses im Dateisystem mit dem entsprechenden Datenbanknamen. Je nach verwendetem Betriebssystem wird daher zwischen Groß- und Kleinschreibung unterschieden, z. B. unter Linux. Die Datenbankverzeichnisse befinden sich im Verzeichnis ... \data des MariaDB-Installationsverzeichnisses oder des MariaDB-Benutzerverzeichnisses (unter Windows beispielsweise C:\Program Files\ MariaDB 10.5\data).

Syntax

```
CREATE DATABASE [IF NOT EXISTS] datenbankname;
```

- ✓ Mit der SQL-Anweisung CREATE werden neue Datenbankobjekte erzeugt. Danach erfolgt die Angabe, welches Objekt erzeugt werden soll, in diesem Fall eine neue Datenbank (DATABASE). Zum Abschluss müssen Sie den gewünschten Namen der Datenbank angeben.
- ✓ Falls die Datenbank bereits existiert, wird eine Fehlermeldung ausgegeben. Um dies zu umgehen, kann die Anweisung IF NOT EXISTS hinzugefügt werden. In diesem Fall wird die Datenbank nur dann erstellt, falls sie noch nicht existiert.

Für die Vergabe des Datenbanknamens unter MariaDB gelten die folgenden Regeln:

- ✓ Der Name darf maximal 64 Zeichen umfassen.
- ✓ Es sind alle Zeichen erlaubt, die in einem Verzeichnisnamen des betreffenden Betriebssystems verwendet werden dürfen. Nicht erlaubt ist grundsätzlich das Zeichen /.

Datenbank mit PostgreSQL erstellen

In PostgreSQL können Sie eine Datenbank mit dem Datenbank-Client *psql* erstellen, der zum Lieferumfang von PostgreSQL für die verschiedenen Betriebssysteme gehört, oder Sie verwenden eines der Hilfsprogramme mit grafischer Oberfläche, die es für PostgreSQL gibt.

Über die folgende Anweisung wird eine Datenbank mit PostgreSQL erstellt:

```
① CREATE DATABASE mav  
    WITH OWNER = "postgres";
```

- ① Mit dieser Anweisung wird eine neue Datenbank mit dem Namen *mav* erstellt. PostgreSQL erwartet dabei die Angabe eines Datenbankdateinamens für die Datenbank. Mit den optionalen Parametern WITH OWNER wird der Besitzer der Datenbank festgelegt. Es können später weitere Benutzer für die Datenbank angelegt werden.

Syntax

```
CREATE DATABASE datenbankname  
  [WITH OWNER benutzername  
   TEMPLATE template  
   ENCODING kodierung  
   TABLESPACE tablespaces];
```

- ✓ Über die SQL-Anweisung CREATE werden neue Datenbankobjekte erzeugt. Danach erfolgt die Angabe, welches Objekt erzeugt werden soll; in diesem Fall eine Datenbank (DATABASE).
- ✓ Optional kann beim Erstellen ein Besitzer der Datenbank festgelegt werden. Dazu wird der Parameter OWNER verwendet. Der Besitzer muss bereits angelegt sein. Wenn Sie stattdessen den Standardwert verwenden möchten (in dem Fall der Benutzer, der den Befehl ausführt), geben Sie DEFAULT ein.
- ✓ Mit dem Parameter TEMPLATE kann eine Datenbankvorlage gewählt werden, die für die Erstellung der neuen Datenbank verwendet werden soll.

- ✓ Der Parameter `ENCODING` ermöglicht Ihnen die Festlegung einer Zeichenkodierung (z. B. `SQL_ASCII`). Der Standardwert ist hier die Kodierung der gewählten Datenbankvorlage.
- ✓ Möchten Sie einen anderen Tablespace angeben als in der Datenbankvorlage, können Sie über den Parameter `TABLESPACE` individuelle Einstellungen vornehmen.

Für die Vergabe des Datenbanknamens unter PostgreSQL gelten die folgenden Regeln:

- ✓ Der Datenbankname unterliegt den Regeln des verwendeten Dateisystems. Es sind alle Zeichen erlaubt, die in einem Verzeichnisnamen des verwendeten Betriebssystems verwendet werden können. Nicht erlaubt ist grundsätzlich das Zeichen `/`.
- ✓ Der Name darf maximal 64 Zeichen umfassen.

4.3 Datenbank anzeigen und auswählen

Datenbank anzeigen

Wenn Sie den Namen einer bestimmten Datenbank nicht kennen, können Sie sich eine Liste der vorhandenen Datenbanken anzeigen lassen.

In MariaDB zeigen Sie mit der Anweisung `SHOW DATABASES` alle bestehenden Datenbanken an.

```
MariaDB [mav]> SHOW DATABASES;
+-----+
| Database |
+-----+
| bibliothek |
| information_schema |
| mav |
| mysql |
| performance_schema |
| test |
| uebungen |
+-----+
7 rows in set (0.084 sec)

Anzeige der vorhandenen
Datenbanken in MariaDB
```

Syntax

```
SHOW DATABASES;
```

In PostgreSQL geben Sie den Befehl "`\l`" (Backslash und kleines L) ein, um eine Auflistung aller Datenbanken sowie deren Eigentümer anzeigen zu lassen.

Name	Eigentümer	Kodierung	Liste der Datenbanken		
			Sortierfolge	Zeichentyp	Zugriffsprivilegien
bibliothek	postgres	UTF8	German_Germany.1252	German_Germany.1252	
mav	postgres	UTF8	German_Germany.1252	German_Germany.1252	
postgres	postgres	UTF8	German_Germany.1252	German_Germany.1252	
template0	postgres	UTF8	German_Germany.1252	German_Germany.1252	
template1	postgres	UTF8	German_Germany.1252	German_Germany.1252	=c/postgres + postgres=CTc/postgres
uebungen (6 Zeilen)	postgres	UTF8	German_Germany.1252	German_Germany.1252	=c/postgres + postgres=CTc/postgres

Anzeige der vorhandenen Datenbanken in PostgreSQL

Syntax

```
\l
```

Datenbank auswählen in MariaDB

Bevor Sie mit einer bestimmten Datenbank und den darin befindlichen Datenbankobjekten arbeiten können, müssen Sie diese auswählen.

In MariaDB erfolgt die Angabe eines berechtigten Benutzers bereits beim Aufruf des Datenbank-Clients, eine bestimmte Datenbank ist dann jedoch noch nicht aktiviert. Um die zu bearbeitende Datenbank auszuwählen, verwenden Sie die Anweisung **USE**. Danach folgt die Angabe des Datenbanknamens.

```
① USE mav;
```

- ① Die Datenbank *mav* wird ausgewählt.

Syntax der Anweisung **USE** in MariaDB

```
USE datenbankname;
```

- ✓ Die Anweisung **USE** wählt die angegebene Datenbank aus. Alle weiteren Anweisungen beziehen sich auf diese Datenbank.

Datenbank wechseln mit PostgreSQL

In PostgreSQL erfolgt die Angabe eines berechtigten Benutzers und der betreffenden Datenbank bereits beim Aufruf des Datenbank-Clients. Das heißt, Sie sind bereits mit der Datenbank verbunden, mit der Sie sich anmelden. Möchten Sie eine andere Datenbank zur Bearbeitung wählen, geben Sie "`\c datenbankname`" ein.

```
postgres=# \c
Sie sind jetzt verbunden mit der Datenbank „postgres“ als Benutzer „postgres“.
postgres=#
```

Datenbank wechseln mit PostgreSQL

Syntax in PostgreSQL

```
\c datenbankname;
```

- ✓ Die Anweisung `\c` wählt die angegebene Datenbank aus. Alle weiteren Anweisungen beziehen sich auf diese Datenbank.

4.4 Datenbank löschen

Über die Anweisung DROP DATABASE wird eine Datenbank mit allen enthaltenen Daten gelöscht. Dabei wird keine Warnmeldung ausgegeben, der Vorgang kann nicht rückgängig gemacht werden. Mit diesem Befehl werden die Katalogeinträge der Datenbank sowie das Verzeichnis inklusive der Daten gelöscht.

Syntax in MariaDB

```
DROP DATABASE [IF EXISTS] datenbankname;
```

- ✓ Der Name der Datenbank folgt am Ende der Anweisung.
- ✓ Die Anweisung IF EXISTS verhindert das Auftreten von Fehlermeldungen, falls die Datenbank nicht existiert.

Syntax in PostgreSQL

```
DROP DATABASE datenbankname;
```

- ✓ Der Name der Datenbank folgt am Ende der Anweisung.

4.5 Übung

Neue Datenbanken erstellen und löschen

Übungsdatei: --

Ergebnisdatei: *Uebung.sql*

1. Erstellen Sie eine Datenbank mit dem Namen *Uebungen*. Verwenden Sie dazu den entsprechenden Datenbank-Client.
2. Erstellen Sie eine Datenbank mit dem Namen *testdb*.
3. Wechseln Sie zur Datenbank *testdb*.
4. Löschen Sie die Datenbank *testdb*.

5

Tabellen erstellen und verwalten

5.1 Tabellen erstellen

Tabellen (Relationen) sind die einzigen Objekte einer Datenbank, in denen die Daten gespeichert werden. Jeder Zugriff auf die Daten erfolgt über die Tabellen. Bei jeder Abfrage oder Auswertung muss der Name der Tabelle angegeben werden. Die Datenbank dient dabei als Container für die logisch zusammengehörigen Tabellen. Der Tabellename muss innerhalb einer Datenbank eindeutig sein, kann aber in mehreren Datenbanken eines DBS verwendet werden.

Eine Tabelle besteht aus einzelnen Feldern (Datenfeldern, Attributen, Spalten). Durch die Namen und Datentypen der Felder wird die Struktur bzw. das Schema der Tabelle festgelegt. Zusammengehörige Daten werden in eine Zeile der Tabelle eingetragen. Sie bilden einen Datensatz bzw. ein Tupel.

Um eine Tabelle erstellen zu können, benötigen Sie das Ausführungsrecht für die Anweisung `CREATE TABLE`. Der Administrator vergibt dieses Recht mit der Anweisung `GRANT CREATE TABLE`. Der Benutzer, der die Tabelle erstellt, wird danach zu ihrem Eigentümer.

Einfache Tabellen erstellen

Eine Tabelle mit den dazugehörigen Datenfeldern erstellen Sie mit der Anweisung `CREATE TABLE`. Die Anweisung besitzt sehr viele zusätzliche Optionen, die z. B. das Erstellen von Indizes oder das Definieren von Standardwerten erlauben.

Beispiel: *Mitarbeiter.sql*

Im Folgenden wird eine Tabelle mit vier Feldern zum Speichern von Mitarbeiterdaten erzeugt.

```
① CREATE TABLE t_ma
② (id INTEGER NOT NULL,
③      vname VARCHAR(100),
④      name VARCHAR(100),
⑤      adr TEXT);
```

- ① Die neue Tabelle erhält den Namen `t_ma`. In Klammern folgt die Definition der Datenfelder.

- ② Das erste Datenfeld heißt *id* und ist vom Typ INTEGER. Es kann ganze Zahlen mit einer Länge von vier Bytes speichern. Da anhand der Einträge im Feld *id* die einzelnen Datensätze identifiziert werden sollen, wird zusätzlich festgelegt, dass das Feld immer einen Wert enthalten muss (NOT NULL, vgl. nachfolgende Tabelle).
- ③ Die Felder *vname* und *name* werden mit dem Datentyp VARCHAR und einer maximalen Länge von 100 Zeichen definiert. Bei diesem Datentyp werden nur so viele Zeichen in der Tabelle gespeichert wie tatsächlich benötigt.
- ④ Hier wird zu Demonstrationszwecken ein Datenfeld vom Typ TEXT angelegt. Es ermöglicht das Speichern größerer Textmengen mit variabler Länge. In der praktischen Umsetzung spaltet man die Adresse in einzelne Felder verschiedener Länge vom Datentyp VARCHAR, beispielsweise *plz*, *ort*, *str* und *hnr*, auf und speichert die Adresse nicht in einem einzigen Datenfeld.

- ✓ Damit Sie einen Tabellennamen besser von anderen Datenbankobjekten (z. B. Abfragen oder Stored Procedures) unterscheiden können, empfiehlt es sich, ein Präfix als Kennzeichen voranzustellen, z. B. *t_*.
- ✓ Wie lang der Name eines Datenbankobjektes sein kann, ist vom jeweiligen DBS abhängig.

```
MariaDB [mav]> CREATE TABLE t_ma
->   (id INTEGER NOT NULL,
->    vname VARCHAR(100),
->    name VARCHAR(100),
->    adr TEXT);
Query OK, 0 rows affected (1.85 sec)

MariaDB [mav]>
```

Erstellen einer neuen Tabelle mit MariaDB

Syntax

```
CREATE TABLE tabellenname
  (datenfeld1 datentyp1 [DEFAULT standardwert1 | NULL | NOT NULL]
  [AUTO_INCREMENT] ,
  ...
  datenfeldX datentypX [DEFAULT standardwertX | NULL | NOT NULL]
  [AUTO_INCREMENT] ,
  PRIMARY KEY(datenfeldname)) ;
```

- ✓ Mit der Anweisung CREATE TABLE wird eine neue leere Tabelle in der aktiven Datenbank erstellt. Danach folgt der gewünschte Tabellename. Dieser muss eindeutig sein.
- ✓ In runden Klammern folgen die Definitionen der einzelnen Datenfelder. Für jedes Datenfeld müssen dabei ein Name und ein Datentyp angegeben werden. Es muss mindestens ein Datenfeld definiert werden.
- ✓ Mit der Angabe PRIMARY KEY kann ein Datenfeld als Primärschlüssel festgelegt werden. Der Primärschlüssel ermöglicht es, einen Datensatz eindeutig zu identifizieren.

Folgende optionale Parameter sind bei der Definition von Datenfeldern möglich:

! Der Wert NULL bedeutet, dass ein Datenfeld keinen Inhalt besitzt. Er ist nicht mit der Zahl 0 und je nach Datenbanksystem auch nicht mit einer leeren Zeichenkette identisch. NULL-Werte können entstehen, wenn beim Einfügen von Daten in eine Tabelle für bestimmte Spalten keine Werte angegeben werden. In einer Abfrage kann auf den Wert NULL geprüft werden.

NULL	Mit diesem Parameter wird festgelegt, dass das Datenfeld standardmäßig keinen Wert (auch nicht 0 oder eine leere Zeichenkette) enthält und ein Datensatz in diesem Feld auch keinen Wert enthalten muss.
NOT NULL	Mit diesem Parameter wird die Eingabe eines Wertes für das entsprechende Datenfeld erzwungen . Die Angabe NOT NULL ist für Schlüsselfelder unbedingt anzugeben.
DEFAULT standardwert	Der Parameter DEFAULT definiert einen Standardwert für das Datenfeld. Erhält dieses Datenfeld bei der Eingabe der Daten keinen Wert, wird der Standardwert verwendet. Mit NOT NULL definierte Datenfelder benötigen immer einen Standardwert.
AUTO_INCREMENT (MariaDB) SERIAL (PostgreSQL)	Der Wert dieses Datenfeldes wird automatisch beim Anlegen eines neuen Datensatzes aus dem Wert des Datenfeldes des vorherigen Datensatzes plus eins errechnet. Dieser Wert kann vom Benutzer nicht geändert werden. Diese Einstellung ist besonders für den Primärschlüssel empfehlenswert, da dadurch automatisch ein eindeutiger Schlüsselwert erzeugt wird.
INVISIBLE (MariaDB)	Die Werte der Datenfelder der Spalte werden bei einer einfachen Datenabfrage mit SELECT * (Kapitel 7) nicht im Ergebnis angezeigt. Beim Hinzufügen von Werten zur Tabelle muss für die Spalte kein Wert angegeben werden, es sei denn, sie wird explizit im Einfügebefehl INSERT (Kapitel 6) aufgeführt.

In verschiedenen Datenbanksystemen sind nicht immer alle angegebenen Möglichkeiten zur Definition einer Tabellenstruktur vorhanden. Es ist auch möglich, dass diese einen anderen Namen für die betreffende Einstellung verwenden.

MariaDB bietet seit der Version 10.3 zusätzlich die Möglichkeit, mittels der Option **WITH SYSTEM VERSIONING** versionierbare Tabellen zu erstellen. Seit der Version SQL:2011 sind diese Bestandteil des Standards. Versionierbare Tabellen ermöglichen die Speicherung des Verlaufs aller Änderungen. Dadurch sind sowohl eine Datenanalyse zu jedem Zeitpunkt als auch die Prüfung von Änderungen und der Vergleich von Daten zu verschiedenen Zeitpunkten möglich. Häufige Anwendungsfälle sind beispielsweise die Sicherstellung der Erfüllung gesetzlicher Anforderungen zur Speicherung von Daten für x Jahre oder die Wiederherstellung der Daten zu einem bestimmten Zeitpunkt.

Beispiele für die Definition von Datenfeldern

Definition in der CREATE - TABLE-Anweisung	Erklärung
ort VARCHAR (100)	Das Datenfeld <i>ort</i> wird mit dem Datentyp VARCHAR definiert und ist maximal 100 Zeichen lang.
id INTEGER NOT NULL	Für das Datenfeld <i>id</i> wird der Integer-Datentyp zum Speichern von ganzen Zahlen festgelegt. Die Angabe NOT NULL erzwingt – da kein DEFAULT-Wert festgelegt wird – die Eingabe eines Wertes beim Anlegen eines neuen Datensatzes.

Definition in der CREATE-TABLE-Anweisung	Erklärung
anzahl DEFAULT 1	Für das Datenfeld <i>anzahl</i> wird ein Standardwert 1 definiert.
ort DEFAULT "Berlin"	Im Datenfeld <i>ort</i> wird der Standardwert <i>Berlin</i> verwendet.

5.2 Datentypen festlegen

Beim Erstellen einer Tabelle muss für jedes Datenfeld ein Datentyp angegeben werden. Der verwendete Datentyp entscheidet, ob in einem Feld Zahlen, Zeichenketten oder andere Daten gespeichert werden. SQL bietet eine große Anzahl an Datentypen, im Folgenden werden einige typische Datentypen beispielhaft vorgestellt. Einige der vorgestellten Datentypen werden nicht von allen DBMS unterstützt.

Numerische Datentypen für ganzzahlige Werte

In Feldern dieses Datentyps werden ganzzahlige Werte (Integer-Zahlen) ohne Kommastellen gespeichert.

Datentyp	Speicherbedarf	Wertebereich
SMALLINT	2 Byte	-32768 bis +32767
INTEGER	4 Byte	-2147483648 bis +2147483647
BIGINT	8 Byte	-9223372036854775808 bis +9223372036854775808

Integer-Datentypen eignen sich besonders für eindeutige Identifikationsnummern, z. B. Primärschlüssel einer Tabelle. Der Zugriff auf solche Datenfelder erfolgt besonders schnell, da Prozessoren in Computern für Zahlenwerte optimiert sind.

Numerische Datentypen für Fließkommazahlen

Mit diesen Datentypen können Sie Zahlenwerte mit Nachkommastellen speichern. Die Datentypen unterscheiden sich dabei im Wertebereich.

Datentyp	Speicherbedarf	Wertebereich
FLOAT (n)	4 oder 8 Byte	Die Gesamtstellenanzahl kann durch n festgelegt werden. Je nach Angabe von n können 7 bis 15 signifikante Stellen plattformunabhängig gespeichert werden.
REAL	4 Byte	7 signifikante Stellen (keine plattformunabhängige Speicherung)
DOUBLE PRECISION	8 Byte	15 signifikante Stellen (keine plattformunabhängige Speicherung)

Die Datentypen können einen Zahlenwert mit der angegebenen Anzahl an Gleitkommastellen speichern. Alle Ziffern nach der Anzahl signifikanter Stellen werden abgeschnitten, daher wird bei diesen Datentypen nicht immer die genaue Größe gespeichert, sondern die näherungsweise Größe.

Numerische Datentypen für Festkommazahlen

Diese Datentypen eignen sich für das Speichern formatierter Zahlen mit einer festen Anzahl von Nachkommastellen. Damit können z. B. Währungsangaben oder Messwerte gespeichert werden.

Datentyp	Erklärung
NUMERIC (Präzision, Skalierung)	Der Parameter Präzision (1-15) legt die Gesamtzahl der signifikanten Stellen der Zahl fest. Der Parameter Skalierung (1-15) bestimmt die Anzahl der Nachkommastellen, die kleiner oder gleich der Gesamtzahl der Stellen sein muss. Der Unterschied zwischen NUMERIC und DECIMAL liegt darin, dass NUMERIC die exakte Anzahl und DECIMAL die minimale Anzahl der signifikanten Stellen gibt.
DECIMAL (Präzision, Skalierung)	
Beispiel: DECIMAL (8 , 3) speichert Zahlen zwischen -99999,999 und 99999,999	

Datentypen für Datums- und Zeitwerte

Für Datums- und Zeitwerte gibt es spezielle Datentypen. Abhängig vom Datenbank-Server werden unterschiedliche Wertebereiche zugelassen.

Datentyp	Speicherbedarf	Wertebereich
DATE	4 Byte	unterschiedlich: 01.01.1000 bis 31.12.9999 (MariaDB) 4713 BC bis 5874897 AD (PostgreSQL)
TIME	3 bzw. 8 Byte	unterschiedlich: -838:59:59 bis 838:59:59 (MariaDB) 00:00:00 bis 23:59:99 (PostgreSQL)
TIMESTAMP	4 bzw. 8 Byte	unterschiedlich: '1970-01-01 00:00:01' bis '2038-01-19 03:14:07' (MariaDB) 4713 v.u.Z. bis 1465001 u.Z. (PostgreSQL)

- ✓ Bei der Eingabe können Datumswerte in verschiedenen Formaten angegeben werden.
Die Umwandlung in das interne Format erfolgt automatisch.
- ✓ Zeitwerte geben Sie in der Form *hh:mm:ss* an.

- ✓ Bei Timestamps kann über die Angabe eines Zusatzwertes die Genauigkeit gesteuert werden (bis zur Speicherung von Microsekunden).

In MariaDB erfolgt dies über YYYY : MM : DD HH : MM : SS [. fraction], wobei fraction ein Wert zwischen 000000 und 999999 ist.

In PostgreSQL wird zwischen Timestamps mit und ohne Zeitzone unterschieden. Die Angabe zur Steuerung der Genauigkeit der Sekunden erfolgt über einen Zusatzwert p im Bereich zwischen 0 und 6 vor der Angabe des Timestamps: TIMESTAMP [(p)] YYYY : MM : DD HH : MM : SS.

Eingabeformat	Beispiel
tt.mm.jj	12.06.21
tt-mmm-jj	12-JUN-21
mm-tt-jj	06-12-21

Für die Angabe des Monatsnamens können Sie folgende Syntax verwenden: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec. Die Groß- und Kleinschreibung wird dabei nicht beachtet.

Jahresangaben können zwei- oder vierstellig erfolgen. Bei zweistelliger Angabe wird automatisch das aktuelle Jahrhundert angenommen.

Datentypen für Zeichen und Texte

SQL sieht verschiedene Datentypen vor, um Textinformationen zu speichern. Es gibt dabei Datentypen mit variabler und fester Länge.

Datentyp	Erklärung
CHAR (Länge) Länge = 1 .. 255	Dieser Datentyp dient zum Speichern beliebiger Textinformationen. Die maximale Länge wird dabei als Parameter übergeben. Unabhängig von der tatsächlichen Länge der gespeicherten Information wird stets die bei der Definition angegebene Anzahl Zeichen gespeichert.
VARCHAR (Länge) Länge = 1 .. 255	Zum Speichern beliebiger Textinformationen wird auch dieser Typ verwendet. Auch hier wird die maximale Länge als Parameter übergeben. In ein Datenfeld dieses Typs eingegebener Text wird in seiner tatsächlichen Länge gespeichert; in zwei zusätzlichen Bytes wird die Länge des Textes abgelegt.
BLOB (mariaDB)	Dieser Datentyp (Binary Large Objects) wird zum Speichern großer, auch binärer Datenmengen eingesetzt, z. B. sehr großer Textdateien, Grafiken, Bilder oder Videos.
BYTEA (PostgreSQL)	Bei PostgreSQL gibt es den Datentyp BLOB nicht, dafür kann jedoch der Datentyp BYTEA verwendet werden, der vergleichbar mit BLOB ist.
TEXT	Wie auch in BLOB-Feldern können Sie hier größere Informationsmengen mit variabler Länge speichern. Der Unterschied zu BLOB-Feldern liegt in der anderen Sortierreihenfolge, die bei TEXT-Feldern unabhängig von der Groß- und Kleinschreibung ist.

Da der Datentyp VARCHAR zwei zusätzliche Bytes für die Längenspeicherung benötigt, sollten Sie ihn nicht für die Definition sehr kurzer Datenfelder einsetzen. In einigen Datenbanksystemen erfolgt der Zugriff auf die mit fester Länge gespeicherten CHAR-Datenfelder schneller als auf VARCHAR-Datenfelder. VARCHAR-Datenfelder eignen sich besonders für die Speicherung von unterschiedlich langen Texten.

Logischer Datentyp

SQL bietet auch die Möglichkeit, logische Werte zu speichern.

Datentyp	Erklärung
BOOLEAN	Logische Werte Ja/Nein bzw. True/False. <code>NULL</code> ist ebenfalls möglich und wird als UNKNOWN interpretiert.

Datentypumwandlung

Datentypen lassen sich ineinander umwandeln. Das ist z. B. bei Vergleichen von Werten unterschiedlichen Datentyps oder bei Abfragen erforderlich, in denen zwei Tabellen über Datenfelder unterschiedlichen Typs verknüpft werden. Es wird zwischen impliziter (automatischer) und expliziter Typumwandlung unterschieden.

Eine implizite Typumwandlung findet beispielsweise statt, wenn Sie eine Zahl vom Typ `SMALLINT` mit einer `INTEGER`-Zahl vergleichen. Die Konvertierung erfolgt immer vom „niedrigeren“ zum „höheren“ Datentyp, z. B. von `SMALLINT` zu `INTEGER`.

Explizit wird eine Typumwandlung über die Funktion `CAST` durchgeführt.

Syntax der `CAST`-Anweisung

CAST (Wert **AS** Datentyp)

- ✓ Die Funktion wandelt den angegebenen Wert in den Datentyp um, der nach dem Schlüsselwort **AS** aufgeführt ist.
- ✓ Es können folgende Typen ineinander umgewandelt werden:

NUMERIC	→	CHAR, VARCHAR, DATE
CHAR, VARCHAR	→	NUMERIC, DATE
DATE	→	CHAR, VARCHAR, DATE
BLOB, TEXT	→	nicht möglich

5.3 Constraints in Tabellen verwenden

Beim Erstellen von Tabellen können Sie bereits einfache Bedingungen für die zu speichernden Daten definieren. Dazu gehört z. B., dass NULL-Werte nicht zugelassen sein sollen, dass ein Feld als Primärschlüssel dienen soll oder dass eine Gültigkeitsprüfung für einen Wertebereich eines Datenfeldes durchgeführt werden soll. Wenn diese sogenannten Constraints (engl. für Zwang, Nebenbedingung) bei der Datenbearbeitung nicht erfüllt sind, wird eine Fehlermeldung ausgegeben.

Sie können Constraints entweder bei der Definition einer Spalte festlegen (Spalten-Constraint) oder nach Angabe aller Spalten als Zusatz angeben (Tabellen-Constraint).

Beispiel: Produkte.sql

```
CREATE TABLE t_produkte
(p_id INTEGER NOT NULL,
 name VARCHAR(50), lagernr INTEGER, anzahl INTEGER,
PRIMARY KEY(p_id));
```

- ✓ Die Spalte *p_id* beinhaltet die Bedingung, dass NULL-Werte nicht zugelassen sind (Spalten-Constraint).
- ✓ Nach Angabe aller Spalten wird die Bedingung festgelegt, dass die Spalte *p_id* als Primärschlüssel dient (Tabellen-Constraint).

In diesem Beispiel könnte der Primärschlüssel auch als Spalten-Constraint festgelegt werden.

```
CREATE TABLE t_produkte
(p_id INTEGER PRIMARY KEY,
 name VARCHAR(50), lagernr INTEGER, anzahl INTEGER);
```

Integritätsregeln für Tabellen mit der Anweisung **CONSTRAINT** festlegen

Möchten Sie für Ihre Tabellen eine bestimmte Gültigkeitsprüfung angeben, die die Integritätsregeln Ihrer Datenbank gewährleistet, können Sie die Anweisung **CONSTRAINT** verwenden, die nach der Definition der Datenfelder angegeben wird. Bei der Erstellung der Integritätsregeln gilt:

- ✓ Ein Datenfeld kann mehrere **CONSTRAINT**-Anweisungen besitzen.
- ✓ Eine **CONSTRAINT**-Anweisung kann sich auf mehrere Datenfelder beziehen.
- ✓ Wenn die Prüfung einer **CONSTRAINT**-Anweisung den Wert FALSE ergibt, werden keine Daten hinzugefügt bzw. geändert.
- ✓ **CONSTRAINT**-Anweisungen können nachträglich für bestehende Tabellen hinzugefügt werden.

MariaDB erlaubt die Syntax, ignoriert jedoch reine **CONSTRAINT**-Anweisungen ohne eine Fremdschlüsselbeziehung.

Beispiel: *MitarbeiterAbteilungen.sql*

In der Tabelle *t_ma_abt* wird die Beziehung zwischen den Mitarbeitern und den Abteilungen gespeichert. Jeder Mitarbeiternummer wird eine Abteilungsnummer zugeordnet. Der Wert einer Mitarbeiternummer muss immer größer als 1 sein. Diese Bedingung wird als Gültigkeitsprüfung definiert.

Einige DBS, wie zum Beispiel PostgreSQL, akzeptieren den Wert NULL.

```
CREATE TABLE t_ma_abt
  (id INTEGER NOT NULL,
   abtname VARCHAR(25) DEFAULT "Produktion",
   ma_nr INTEGER,
   CONSTRAINT mpruef CHECK(ma_nr > 1));
```

- ① An dieser Stelle wird für das Datenfeld *abtname* der Standardwert *Produktion* definiert. Wenn bei einer späteren Dateneingabe keine Angabe erfolgt, wird dieser Wert verwendet.
- ② Das Datenfeld *ma_nr* wird als Integer-Zahl deklariert.
- ③ Zusätzlich erfolgt nach der Definition der Datenfelder die Angabe einer Gültigkeitsprüfung (*mpruef*). Der eingegebene Wert für das Feld *ma_nr* wird nur akzeptiert, wenn er größer ist als 1.

```
MariaDB [mav]> CREATE TABLE t_ma_abt
->   (id INTEGER NOT NULL,
->     abtname VARCHAR(25) DEFAULT 'Produktion',
->     ma_nr INTEGER,
->     CONSTRAINT mpruef CHECK(ma_nr > 1));
Query OK, 0 rows affected (0.11 sec)
```

Erstellen der Tabelle *t_ma_abt* innerhalb der Datenbank *mav* in MariaDB

Syntax

```
CREATE TABLE tabellenname
  (datenfeld1 datentyp1 [DEFAULT standardwert1 | NULL | NOT NULL]
  [AUTO_INCREMENT],
  ...
  datenfeldX datentypX [DEFAULT standardwertX | NULL | NOT NULL]
  [AUTO_INCREMENT],
  CONSTRAINT constraintname1 CHECK (gültigkeitsprüfung),
  ...
  PRIMARY KEY(datenfeldname));
```

- ✓ Die Tabellendefinition erfolgt mit der Anweisung CREATE TABLE. Danach werden in runden Klammern die Datenfelder und Tabellenoptionen angegeben.
- ✓ Die Angabe einer Gültigkeitsbedingung wird mit dem Schlüsselwort CONSTRAINT eingeleitet. Danach wird der gewünschte Name der Bedingung angegeben. Nach dem Schlüsselwort CHECK folgt in runden Klammern die Gültigkeitsbedingung.

Die Gültigkeitsbedingung muss erfüllt (wahr) sein, damit der Wert in der Tabelle gespeichert wird. Die folgende Tabelle zeigt einige Möglichkeiten:

Bedingung	Erklärung
nummer <= 100	wahr, wenn nummer kleiner oder gleich 100 ist
name NOT LIKE "%Ä%"	wahr, wenn name den Buchstaben Ä nicht enthält
abteilung IN ("Einkauf", "Verkauf")	wahr, wenn abteilung den Wert <i>Einkauf</i> oder <i>Verkauf</i> besitzt

5.4 Domänen verwenden

Domänen definieren

Domänen (engl. Domains) sind benutzerdefinierte Datentypen, die Sie einmalig definieren und in verschiedenen Tabellen zum Definieren von Feldtypen verwenden können. Mit der Verwendung von Domänen kann die Definition von Integritätsregeln deutlich vereinfacht werden, da die Regeln dort nur einmalig und nicht bei jeder Tabellendefinition angegeben werden müssen. Im Falle einer Änderung müssen Sie nicht jede Tabelle einzeln bearbeiten, sondern nur die entsprechende Domäne verändern. Domänen sind in der gesamten Datenbank verfügbar, in der sie definiert wurden.

Der MariaDB-Server unterstützt die Verwendung von Domänen nicht.

Beispiel

```

① CREATE DOMAIN d_ma_nr AS INTEGER;
② CREATE DOMAIN d_benutzer AS VARCHAR(15) DEFAULT "gast";
③ CREATE DOMAIN d_abt AS VARCHAR(15)
    CHECK (VALUE IN
    ("Einkauf", "Verkauf", "Marketing", "Verwaltung", "Produktion"));

```

- ① Die Domäne *d_ma_nr* wird als Integer-Datentyp definiert.
- ② Die Domäne *d_benutzer* wird mit dem Datentyp VARCHAR und 15 Zeichen Länge definiert. Zusätzlich wird ein Standardwert angegeben.
- ③ Auch die Domäne *d_abt* wird als VARCHAR festgelegt. Darüber hinaus wird eine Prüfung integriert, die sicherstellt, dass der zu speichernde Wert in der angegebenen Liste enthalten ist.

Syntax

```
CREATE DOMAIN domänenname AS datentyp [DEFAULT standardwert];
```

- ✓ Die Definition einer Domäne wird mit den Schlüsselwörtern CREATE DOMAIN eingeleitet. Danach folgen der Name der Domain und nach der Angabe AS der gewünschte Datentyp.
- ✓ Ein Standardwert wird mit dem Schlüsselwort DEFAULT angegeben.
- ✓ Wie bei der Namensvergabe von Tabellen vorgeschlagen, können Sie Domänen mit dem Präfix *d_* kennzeichnen.

Eine Domäne kann neben dem Datentyp auch mit verschiedenen Bedingungen und Prüfungen definiert werden. Dafür wird das Schlüsselwort CHECK verwendet.

```
CREATE DOMAIN domänename AS datentyp CHECK (gültigkeitsbedingung);
```

- ✓ Mit dem Schlüsselwort CHECK kann eine Bedingung hinzugefügt werden. In Klammern folgt danach ein Ausdruck, der einen Wahrheitswert (wahr oder falsch) liefert. Liefert der Ausdruck den Wert Falsch, wird die entsprechende Operation nicht durchgeführt.

Domänen verwenden

Sie können Domänen in Tabellen auf die gleiche Weise wie Standarddatentypen verwenden.

Beispiel

Es wird eine neue Domäne definiert, die in einer Tabellendefinition eingesetzt wird. Der Vorteil des Einsatzes einer Domäne ist in diesem Fall, dass Sie die Liste der möglichen Werte erweitern können, ohne die Tabellendefinition zu ändern.

①	CREATE DOMAIN d_abt AS VARCHAR(15) CHECK (VALUE IN ("Einkauf", "Verkauf", "Marketing", "Verwaltung", "Produktion"));
②	CREATE TABLE t_ma_abt (id INTEGER ,
③	abtname d_abt, ma_nr INTEGER);

- ① Hier wird die Domäne *d_abt* mit dem Datentyp VARCHAR definiert. Zusätzlich wird eine Bedingung angegeben, die sicherstellt, dass nur die angegebenen Abteilungsnamen verwendet werden können.
- ② Mit der Anweisung CREATE TABLE wird die Tabelle *t_ma_abt* definiert. Sie soll für jeden Mitarbeiter die zugehörige Abteilung speichern.
- ③ An dieser Stelle wird für das Datenfeld *abtname* als Datentyp die Domäne *d_abt* verwendet.

Domänen ändern

Der Datentyp einer Domäne kann nicht geändert werden, da darauf auch die Typen der Datenfelder in den Tabellen beruhen. Sie können jedoch den Standardwert und die Gültigkeitsprüfung über die Anweisung ALTER DOMAIN verändern.

Beispiel

Die Definition der Domäne *d_abt* wird geändert. Es wird ein neuer Standardwert festgelegt und über die Anweisung DROP CONSTRAINT die bisherige Gültigkeitsprüfung der Domäne entfernt.

```
ALTER DOMAIN d_abt SET DEFAULT "Produktion";  
ALTER DOMAIN d_abt DROP CONSTRAINT;
```

Syntax

```
ALTER DOMAIN domänename [SET DEFAULT standardwert]
  [DROP DEFAULT]
  [ADD CHECK (gültigkeitsbedingung) ]
  [DROP CONSTRAINT] ;
```

- ✓ Über die Anweisung ALTER DOMAIN können Sie eine Domäne bearbeiten. Als erster Parameter wird dabei der Domänenname angegeben.
- ✓ Mit dem danach folgenden Parameter wird angegeben, was geändert oder gelöscht werden soll.

SET DEFAULT standardwert	Legt einen neuen Standardwert fest
DROP DEFAULT	Löscht den bisherigen Standardwert
ADD CHECK (gültigkeitsbedingung)	Fügt eine Gültigkeitsbedingung hinzu
DROP CONSTRAINT	Entfernt die bestehende Gültigkeitsprüfung

Beachten Sie, dass Sie keine vorhandene Gültigkeitsprüfung bearbeiten oder ergänzen können. Sie müssen zuerst die bestehende Prüfung mit DROP CONSTRAINT löschen und danach eine neue Prüfung definieren.

Domänen löschen

Eine Domäne löschen Sie mit der Anweisung DROP DOMAIN. Danach können Sie die Domäne beispielsweise mit einem anderen Datentyp neu erstellen. Um eine Domäne zu löschen, darf sie in keiner Tabelle als Datentyp verwendet werden. Die entsprechende Datenfelddefinition muss zuerst gelöscht werden.

Syntax

```
DROP DOMAIN domänename [CASCADE/RESTRICT] ;
```

- ✓ Die Anweisung DROP DOMAIN löscht eine Domäne. Als Parameter muss ein gültiger Domänenname angegeben werden.
- ✓ Der Parameter CASCADE löscht alle Objekte, die von der Domäne abhängig sind.
- ✓ Mit der Anweisung RESTRICT (Standardwert) können Sie verhindern, dass die Domäne gelöscht wird, sofern noch andere Objekte von ihr abhängen.

5.5 Vorhandene Tabellen anzeigen, ändern und löschen

Vorhandene Tabellen anzeigen

Über die Anweisung SHOW TABLES können Sie die in der Datenbank vorhandenen Tabellen auflisten. Die Abbildung zeigt das Ergebnis der Auflistung unter MariaDB.

```
MariaDB [mav]> SHOW TABLES;
+-----+
| Tables_in_mav |
+-----+
| produkte
| t_ma
| t_ma_abt
+-----+
3 rows in set (0.01 sec)

MariaDB [mav]>
```

Syntax

Anzeige der Tabellen

```
SHOW TABLES [FROM datenbankname] [LIKE "muster"] ;
```

- ✓ Die Anweisung SHOW TABLES zeigt eine Liste der vorhandenen Tabellen in der aktuell geöffneten Datenbank an.
- ✓ Mit dem Schlüsselwort FROM kann eine andere Datenbank benannt werden.
- ✓ Mithilfe von LIKE kann die Anzeige der Tabellen auf ein bestimmtes Muster im Tabellennamen beschränkt werden. Durch Verwendung des Musters m% werden beispielsweise nur die Tabellen angezeigt, die mit dem Buchstaben m beginnen.

Der Befehl SHOW TABLES ist in PostgreSQL nicht hinterlegt. Sie können sich in psql die Tabellen der aktuellen Datenbank jedoch anzeigen lassen, indem Sie den Befehl "\dt" bzw. "\d" eingeben.

Tabellenstruktur ändern

Die Struktur einer Tabelle können Sie jederzeit über die Anweisung ALTER TABLE ändern. Sie können ...

- ✓ Datenfelder hinzufügen oder löschen,
- ✓ Datenfelder umbenennen,
- ✓ Datenfelddefinitionen verändern,
- ✓ Gültigkeitsprüfungen hinzufügen oder löschen,
- ✓ Schlüssel und Indizes hinzufügen oder löschen.

Mit dem Schlüsselwort ADD fügen Sie einer bestehenden Tabelle ein neues Datenfeld hinzu. Die bereits in der Tabelle vorhandenen Datensätze enthalten danach ein neues leeres Datenfeld, das gegebenenfalls den festgelegten Standardwert enthält. Für das Löschen eines Datenfelds, Schlüssels oder Indizes wird die Anweisung DROP verwendet.

Beispiel: AlterTableBeispiele.sql

Im Folgenden werden einige Anweisungen zum Ändern der Tabellenstruktur vorgestellt.

①	ALTER TABLE t_lager ADD artikel VARCHAR(20) DEFAULT "unbekannt";
---	--

```

② ALTER TABLE t_ma_abt
    ADD PRIMARY KEY (id);

③ ALTER TABLE t_lager
    ADD CONSTRAINT ppruef CHECK (preis > 0);

④ ALTER TABLE t_lager
    DROP wert;

⑤ ALTER TABLE t_lager RENAME COLUMN anzahl TO stueck;

```

- ① Der Tabelle *t_lager* wird das neue Datenfeld *artikel* mit dem Standardwert *unbekannt* hinzugefügt.
- ② Mit dieser Anweisung wird das Feld *id* der Tabelle *t_ma_abt* nachträglich als Primärschlüssel definiert.
- ③ Hier wird eine neue Gültigkeitsbedingung für die Tabelle *t_lager* definiert. Der Wert für das Feld *preis* wird bei der Eingabe überprüft; er muss größer als 0 sein.
- ④ Mithilfe der Anweisung **DROP** wird das angegebene Datenfeld in der Tabelle gelöscht.
- ⑤ Die Spalte *anzahl* wird in *stueck* umbenannt (wird in MariaDB seit Version 10.5.2 unterstützt)



Beim Löschen eines Datenfelds werden die enthaltenen Werte in allen Datensätzen gelöscht.

Syntax

```

ALTER TABLE tabellename
    [ADD datenfelddefinition]
    [ADD indexdefinition]
    [ADD CONSTRAINT constraintname CHECK (gültigkeitsbedingung) ]
    [DROP objektnname];

```

- ✓ Für das Ändern einer Tabelle wird die Anweisung **ALTER TABLE** verwendet. Danach folgt der Name der zu ändernden Tabelle.
- ✓ Mit dem Schlüsselwort **ADD** wird das Hinzufügen eines Datenfelds, Indizes oder Schlüssels oder einer Gültigkeitsbedingung eingeleitet.
- ✓ Mit der Klausel **DROP** wird ein Datenfeld, Index, Schlüssel oder eine Gültigkeitsbedingung gelöscht.
- ✓ Die Definition eines Datenfelds, Indexes, Schlüssels oder einer Gültigkeitsbedingung folgt der gleichen Syntax wie die Erstellung einer Tabelle.
- ✓ Gültigkeitsprüfungen gelten nur für die nach der Änderung neu hinzugefügten oder geänderten Datensätze.

PostgreSQL prüft hinzugefügte Gültigkeitsprüfungen für vorhandene Datensätze und lehnt die Änderung bei einem Verstoß der vorhandenen Daten gegen die neue Gültigkeitsprüfung ab.

```
mav=# SELECT * FROM t_lager;
 id | bezeichnung
----+-
 1 | Hefter
 2 | Ordner
(2 Zeilen)

mav=# ALTER TABLE t_lager ADD CONSTRAINT p_id CHECK (id>10);
FEHLER: Check-Constraint „p_id“ wird von irgendeiner Zeile verletzt
mav=# ALTER TABLE t_lager ADD CONSTRAINT p_id CHECK (id>10) NOT VALID;
ALTER TABLE
mav=# insert into t_lager(id, bezeichnung) values (3, 'Stift');
FEHLER: neue Zeile für Relation „t_lager“ verletzt Check-Constraint „p_id“
DETAIL: Fehlgeschlagene Zeile enthält (3, Stift).
mav=# insert into t_lager(id, bezeichnung) values (11, 'Stift');
INSERT 0 1
mav=# ALTER TABLE t_lager VALIDATE CONSTRAINT p_id;
FEHLER: Check-Constraint „p_id“ wird von irgendeiner Zeile verletzt
mav=#

```

Verhalten von CONSTRAINT-Prüfungen in PostgreSQL

Durch die Klausel NOT VALID ist es möglich, die Überprüfung der vorhandenen Daten zu unterdrücken. Mittels der Anweisung VALIDATE CONSTRAINT kann zu einem späteren Zeitpunkt geprüft werden, ob die zum Zeitpunkt der Änderung vorhandenen Daten der neuen Gültigkeitsprüfung entsprechen.

Anzeige der Tabellenstruktur

Möchten Sie sich die Tabellenstruktur in der Konsole anzeigen lassen, geben Sie bei PostgreSQL den Befehl "`\d tabellenname`" ein.

Tabelle »public.t_lager«				
Spalte	Typ	Sortierfolge	NULL erlaubt?	Vorgabewert
id	integer		not null	
stueck	integer			
preis	numeric			

Indexe:
"t_lager_pkey" PRIMARY KEY, btree (id)

Tabellenstruktur der Tabelle t_lager in PostgreSQL anzeigen

Bei MariaDB können Sie sich die Tabellenstruktur mithilfe des Befehls `DESCRIBE <Tabellenname>` anzeigen lassen. Der Befehl `SHOW TABLE STATUS` liefert eine Gesamtübersicht über die Tabellen der Datenbank, jedoch keine Informationen zu deren Struktur.

Eine alternative Möglichkeit besteht in der Verwendung des Befehls `SHOW CREATE TABLE <Tabellenname>`, welcher die Anweisung zeigt, die notwendig wäre, um die Tabelle in ihrer vorhandenen Form zu erstellen.

```
MariaDB [mav]> SHOW CREATE TABLE t_ma \G;
***** 1. row *****
Table: t_ma
Create Table: CREATE TABLE `t_ma` (
  `id` int(11) NOT NULL,
  `vname` varchar(100) DEFAULT NULL,
  `name` varchar(100) DEFAULT NULL,
  `adr` text DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

Tabellenstruktur in der Listenansicht (Option \G) des notwendigen Erstellungsbefehls der Tabelle

Tabellen löschen

Zum Löschen einer Tabelle verwenden Sie die Anweisung `DROP TABLE`. Dabei werden die Definition der Tabellenstruktur und alle in der Tabelle gespeicherten Datensätze gelöscht.

Syntax

```
DROP TABLE tabellename;
```

- ✓ Mit der Anweisung DROP TABLE wird die angegebene Tabelle mit allen enthaltenen Daten gelöscht.

Beim Löschen einer Tabelle müssen Sie folgende Besonderheiten beachten:

- ✓ Wenn andere Datenbankobjekte die zu löschen Tabelle verwenden oder referenzieren, z. B. durch eine Sicht (VIEW) oder eine gespeicherte Prozedur (STORED PROCEDURE), ist das Löschen nicht möglich. Es müssen zuerst alle Referenzen entfernt werden.
- ✓ Sie müssen entweder Eigentümer der Tabelle sein oder über die notwendigen Rechte zum Löschen der Tabelle verfügen.
- ✓ Es dürfen keine Transaktionen aktiv sein, die die betreffende Tabelle verwenden.

5.6 Übung

Tabellen erstellen und löschen

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*,
Uebung3.sql

1. Erstellen Sie eine Datenbank mit dem Namen *testdb*.
Wechseln Sie zu dieser neuen Datenbank.
Erstellen Sie eine neue Tabelle *t_artikel* mit den Datenfeldern *id*, *name* und *preis*.
Verwenden Sie geeignete Datentypen.
Löschen Sie die Tabelle und anschließend die Datenbank.
2. Wechseln Sie zur Datenbank *Uebungen*.
Erstellen Sie eine Tabelle *t_person* mit den Datenfeldern *id*, *vname* und *name*.
Verwenden Sie dabei geeignete Datentypen.
Für alle Datenfelder soll eine Eingabe erforderlich sein. Erweitern Sie die Definition der Datenfelder für diesen Zweck.
Fügen Sie ein neues Datenfeld *beschäftigt_seit* in die Tabelle ein. Es soll einen Datumswert speichern.
Löschen Sie das Datenfeld wieder.
3. Wechseln Sie zur Datenbank *Uebungen*.
Löschen Sie die in Übung 2 erstellte Tabelle *t_person*.
Erstellen Sie eine geeignete Domäne, die als Datentyp für die Speicherung von Vorname und Nachname verwendet werden soll. Es soll für das Datenfeld unbedingt eine Eingabe erforderlich sein. Nennen Sie die Domäne *Namen*.
Erstellen Sie die Tabelle *t_person* mit folgenden Datenfeldern: *id*, *vname*, *name*, *strasse*, *hnr*, *plz* und *ort*. Verwenden Sie dabei geeignete Datentypen bzw. die neu definierte Domäne.

6

Daten einfügen, aktualisieren, löschen

6.1 Daten einfügen

Einfügen eines Datensatzes

Um Datensätze in eine Tabelle einzufügen, verwenden Sie die Anweisung `INSERT INTO`. Dabei ist es möglich, einen vollständigen Datensatz oder nur Werte für ausgewählte Datenfelder einzufügen.

Falls die Tabelle ein Datenfeld enthält, das mit dem Parameter `AUTO_INCREMENT` (MariaDB) bzw. `SERIAL` (PostgreSQL) definiert wurde, wird das Feld beim Einfügen automatisch um den Wert 1 erhöht.

Um Daten in eine Tabelle einzufügen, müssen Sie Besitzer der Tabelle sein bzw. über die entsprechenden Rechte verfügen. Der Administrator vergibt diese Rechte mit der `GRANT-INSERT`-Anweisung.

In der Praxis werden Sie nur in seltenen Fällen mithilfe der `INSERT-INTO`-Anweisung Datensätze manuell in eine Tabelle einfügen. In der Regel erfolgt die Dateneingabe über eine entsprechende grafische Oberfläche, wie z. B. phpmyAdmin, phppgAdmin oder SQuirreL.

Beispiel: *DatensatzEinfuegen.sql*

Zuerst wird die Tabelle `t_ma_dt` für die Speicherung der Mitarbeiter in Deutschland erstellt. Danach werden drei Datensätze in diese Tabelle eingefügt.

```
① CREATE TABLE t_ma_dt
    (id INTEGER NOT NULL AUTO_INCREMENT,
     name VARCHAR(50), vname VARCHAR(50), str VARCHAR(150),
     hnr VARCHAR(5), plz VARCHAR(5), ort VARCHAR(50), gebdat DATE,
     land varchar(4), PRIMARY KEY(id));
② INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat,
    land) VALUES ("Illner", "Markus", "Goetheweg Str.", "55",
    "70173", "Stuttgart", "1976-12-23", "D");
```

```
(3) INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat,
    land) VALUES ('Schubert', 'Steve', 'Albert-Schweitzer-Str.', ,
    '10', '70174', 'Stuttgart', '1982-02-15', 'D');

(4) INSERT INTO t_ma_dt (name, vname) VALUES ("Sattke", "Beatrice");
```

- ① Die benötigte Tabelle *t_ma_dt* wird mit den entsprechenden Datenfeldern definiert. Das Feld *id* wird dabei als Primärschlüssel festgelegt, dessen Wert automatisch erhöht wird. Das Beispiel bildet die Schreibweise von MariaDB ab. Bei PostgreSQL würde hier der Parameter SERIAL stehen.
- ② An dieser Stelle wird ein vollständiger Datensatz in die Tabelle eingefügt. Die Daten, die in Textfeldern gespeichert werden, sind in Anführungszeichen eingeschlossen. Da der Wert für das Feld *id* automatisch erstellt wird, kann er bei der Eingabe entfallen.
- ③ Auch hier wird ein Datensatz in die Tabelle eingefügt. Statt Anführungszeichen kann zur Eingabe von Textinformationen auch ein Apostroph verwendet werden.
- ④ Ein unvollständiger Datensatz, der nur aus dem Namen und Vornamen besteht, wird eingefügt. Die anderen Werte sind gleich dem NULL-Wert, sie können durch eine spätere Aktualisierung nachgetragen werden.

Alphanumerische Werte, Textinformationen bzw. Datums- und Zeitangaben müssen innerhalb der INSERT-INTO-Anweisung in Anführungszeichen oder Apostrophe eingeschlossen werden. Bei numerischen Werten kann dies entfallen.

Syntax

```
INSERT INTO tabellenname (feld1, ..., feldX)
VALUES (wert1, ..., wertX);
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern INSERT INTO eingeleitet. Danach folgt der Name der gewünschten Tabelle.
- ✓ In runden Klammern werden, durch Kommata getrennt, die Namen der Datenfelder angegeben, in die Werte eingefügt werden sollen.
- ✓ Nach dem Schlüsselwort VALUES folgt in runden Klammern die Angabe der einzelnen Werte. Diese müssen in der gleichen Reihenfolge angegeben werden, wie die zugehörigen Datenfelder in der Liste davor. Dadurch ist es möglich, eine beliebige Reihenfolge für die Datenfelder zu wählen.
- ✓ Textinformationen, Datums- und Zeitwerte sind in Apostrophe (') oder Anführungszeichen (") einzuschließen. Beachten Sie dabei, dass nicht alle Datenbanksysteme Anführungszeichen akzeptieren.

Sie brauchen nicht alle Datenfelder einer Tabelle mit Daten zu füllen. Die nicht aufgeführten Felder enthalten nach dem Einfügen automatisch entweder keinen Wert (NULL – nicht zu verwechseln mit der Zahl Null) oder den definierten Standardwert. Es gelten dabei die folgenden Regeln:

Angabe bei der Definition des Datenfelds	Resultat, wenn kein Wert für das Datenfeld angegeben wird
DEFAULT standardwert	Im Datenfeld wird der angegebene Standardwert gespeichert.
NULL	Das Datenfeld bleibt leer. Es enthält den Wert NULL.
NOT NULL	Es wird eine Fehlermeldung ausgegeben, da das Datenfeld einen Wert enthalten muss.
Keine Angabe	Das Datenfeld bleibt leer. Es enthält den Wert NULL.

Soll in einem Datenfeld kein Wert gespeichert werden, können Sie ihm in der **INSERT-INTO**-Anweisung auch direkt den Wert **NULL** zuweisen.

Abfrage der eingefügten Datensätze

Um das erfolgreiche Einfügen der neuen Datensätze zu überprüfen, können Sie eine einfache Datenabfrage durchführen.

Beispiel

Mit der folgenden **SELECT**-Anweisung werden Datensätze aus der Tabelle **t_ma_dt** abgefragt. Mit der Angabe des Zeichens * wird die Anzeige aller Datenfelder erreicht.

```
SELECT * FROM t_ma_dt;
```

```
MariaDB [mav]> select * from t_ma_dt;
+---+---+---+---+---+---+---+---+---+
| id | name | vname | str          | hnr | plz | ort        | gebdat | land |
+---+---+---+---+---+---+---+---+---+
| 1  | Illner | Markus | Goetheweg Str. | 55  | 70173 | Stuttgart | 1976-12-23 | D    |
| 2  | Schubert | Steve | Albert-Schweitzer-Str. | 10  | 70174 | Stuttgart | 1982-02-15 | D    |
| 3  | Sattke | Beatrice | NULL           | NULL | NULL | NULL       | NULL    | NULL  |
+---+---+---+---+---+---+---+---+---+
3 rows in set (0.00 sec)
```

Abfrageergebnis zu den neuen Datensätzen der Tabelle **t_ma_dt**

In Kapitel 7 (*Einfache Datenabfragen*) wird die **SELECT**-Anweisung ausführlich erläutert.

Einfügen mehrerer Datensätze

Die zweite Form der **INSERT-INTO**-Anweisung ermöglicht das Einfügen mehrerer Datensätze. Die einzufügenden Daten werden dabei mithilfe einer Abfrage aus einer existierenden Tabelle gewonnen.

Beispiel: *MehrereDatensaetzeEinfuegen.sql*

Im ersten Schritt werden mehrere Datensätze in die Tabelle **t_ma_dt** eingefügt. Danach wird die Tabelle **t_ma_frankfurt** erzeugt, die nicht notwendigerweise die gleiche Struktur wie die Tabelle **t_ma_dt** besitzen muss. Dieser werden danach über eine **INSERT-INTO**-Anweisung Daten hinzugefügt.

```

① INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat, land) VALUES ("Meier", "Susann", "Hauptweg", "4", "60385", "Frankfurt", "1972-04-21", "D");
INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat, land) VALUES ("Brauer", "Claudia", "Bahnhofstr.", "156", "60386", "Frankfurt", "1971-04-28", "D");
INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat, land) VALUES ("Yildiz", "Umit", "Universitätsstr.", "24", "60322", "Frankfurt", "1974-08-29", "D");
INSERT INTO t_ma_dt (name, vname, str, hnr, plz, ort, gebdat, land) VALUES ("Becker", "Sebastian", "Heidestr.", "150", "60385", "Frankfurt", "1994-08-14", "D");
② CREATE TABLE t_ma_frankfurt
(id INTEGER NOT NULL AUTO_INCREMENT, name VARCHAR(50), vname VARCHAR(50), str VARCHAR(150), hnr VARCHAR(5), plz VARCHAR(5), ort VARCHAR(50), gebdat DATE, land varchar(4), PRIMARY KEY(id));
③ INSERT INTO t_ma_frankfurt
(name, vname, str, hnr, plz, ort, gebdat, land)
SELECT name, vname, str, hnr, plz, ort, gebdat, land
FROM t_ma_dt WHERE ort = "Frankfurt";

```

- ① Zuerst werden vier neue Datensätze in die Tabelle *t_ma_dt* eingefügt. Diese sollen dann später ausgewählt werden.
- ② An dieser Stelle wird die neue Tabelle *t_ma_frankfurt* definiert. Sie soll nur Mitarbeiter aus Frankfurt aufnehmen.
- ③ Mit der **INSERT-INTO**-Anweisung erfolgt das Einfügen der Datensätze in die neue Tabelle. Dazu werden in einer Abfrage alle Datensätze aus der Tabelle *t_ma_dt* ausgewählt, bei denen der Inhalt des Feldes *ort* dem Wert *Frankfurt* entspricht. Die **SELECT**-Anweisung liefert vier Datensätze, die sofort in die Tabelle eingefügt werden.

```

MariaDB [mav]> INSERT INTO t_ma_frankfurt (name, vname, str, hnr, plz, ort, gebdat, land)
->   SELECT name, vname, str, hnr, plz, ort, gebdat, land
->   FROM t_ma_dt WHERE ort = "Frankfurt";
Query OK, 4 rows affected (0.02 sec)
Records: 4  Duplicates: 0  Warnings: 0

```

Einfügen der Datensätze

```

MariaDB [mav]> SELECT * FROM t_ma_frankfurt;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | name | vname | str | hnr | plz | ort | gebdat | land |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | Meier | Susann | Hauptweg | 4 | 60385 | Frankfurt | 1972-04-21 | D |
| 2 | Brauer | Claudia | Bahnhofstr. | 156 | 60386 | Frankfurt | 1971-04-28 | D |
| 3 | Yildiz | Umit | Universitätsstr. | 24 | 60322 | Frankfurt | 1974-08-29 | D |
| 4 | Becker | Sebastian | Heidestr. | 150 | 60385 | Frankfurt | 1994-08-14 | D |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

Ergebnis der Abfrage

Syntax

```
INSERT INTO tabellenname (feld1, ..., feldX)
SELECT [* | datenfelder] FROM tabellenname [WHERE bedingung];
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern **INSERT INTO** eingeleitet. Danach folgt der Name der gewünschten Tabelle.
- ✓ In runden Klammern folgen, durch Kommata getrennt, die Namen der Datenfelder, in die Werte eingefügt werden sollen. Die Reihenfolge muss nicht der Reihenfolge der Datenfelder bei der Definition der Tabelle entsprechen.
- ✓ Danach wird eine gültige **SELECT**-Anweisung angegeben, die als Abfrage einen oder mehrere Datensätze aus einer Tabelle liefert. Der Aufbau der zurückgelieferten Datensätze muss dabei mit den Angaben in der Datenfeldliste übereinstimmen.

Die Datenfeldliste kann auch entfallen. In diesem Fall müssen die Anzahl und die Reihenfolge der durch die Abfrage zurückgelieferten Datenfelder mit der der Tabelle übereinstimmen, in die diese eingefügt werden sollen. Dies betrifft auch die einfache **INSERT-INTO**-Anweisung.

6.2 Daten aktualisieren

Beim Ausführen der **INSERT-INTO**-Anweisung wird stets ein neuer Datensatz erzeugt und in der Tabelle gespeichert. Sie können jedoch auch gezielt existierende Datensätze bearbeiten und verändern. Zu diesem Zweck stellt SQL die **UPDATE**-Anweisung zur Verfügung.

Mit der **UPDATE**-Anweisung können Sie einen oder auch mehrere Datensätze gleichzeitig aktualisieren. Die Auswahl der betreffenden Datensätze erfolgt dabei über eine **WHERE**-Klausel. Wenn Sie keine Bedingung angeben, werden alle Datensätze aktualisiert.

! Um Daten in einer Tabelle zu verändern, müssen Sie Besitzer der Tabelle sein bzw. über die entsprechenden Rechte verfügen. Der Administrator vergibt diese Rechte mit der **GRANT-UPDATE**-Anweisung. Werden in dem **UPDATE**-Befehl noch andere Tabellen angesteuert, z. B. über die **WHERE**-Bedingung, müssen Sie für diese Tabellen über das Recht **SELECT** verfügen.

Beispiel: *DatenAktualisieren.sql*

In den folgenden **UPDATE**-Anweisungen werden die Daten in verschiedenen Tabellen verändert.

```
(1) UPDATE t_ma_dt SET str = "Hauptstr.", "hnr" = "17"
    WHERE name = "Meier" AND vname = "Susann";
(2) UPDATE t_ma_frankfurt SET ort = "Frankfurt/Main";
(3) UPDATE t_lager SET stueck = stueck + 100 WHERE stueck < 100;
```

- ① Es wird der Inhalt der Datenfelder *str* und *hnr* aktualisiert. Die Auswahl des betreffenden Datensatzes erfolgt über die Felder *name* und *vname*, die beide den angegebenen Wert besitzen müssen.

- ② Der Inhalt des Datenfelds *ort* wird in *Frankfurt/Main* geändert. Da keine Bedingung zur Eingrenzung der Datensätze angegeben ist, werden alle Datensätze der Tabelle aktualisiert.
- ③ Mit dieser Anweisung wird für alle Datensätze der Tabelle *t_lager*, für die die angegebene Bedingung gilt (Stückzahl kleiner als 100), der aktuelle Wert des Felds *stueck* um den Wert 100 erhöht.

Bei der Aktualisierung eines Datensatzes müssen Sie keine absoluten Werte angeben. Sie können auch Rechenoperationen mit dem aktuellen Datenfeldinhalt durchführen, z. B.

feld = feld + 20.

Syntax MariaDB

```
UPDATE tabellenname SET feld1 = wert1, . . . , feldX = wertX FROM
tabellennamen [WHERE bedingung];
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort UPDATE. Danach wird der Name der gewünschten Tabelle angegeben.
- ✓ Mit dem Schlüsselwort SET wird die Wertzuweisung eingeleitet. Sie erfolgt stets nach dem Muster *datenfeldname = wert*. Der Wert kann dabei eine Zahl, ein Text, ein berechneter Ausdruck oder eine Unterabfrage sein, abhängig vom Datentyp des Datenfeldes. Für die Änderung mehrerer Werte werden die Datenfeld-Wert-Paare durch Komma getrennt.
- ✓ Wenn Sie in der Anweisung WHERE zusätzlich Spalten anderer Tabellen eingeben möchten, müssen Sie in der Anweisung FROM die betreffenden Tabellennamen angeben.
- ✓ Nach der Wertzuweisung können Sie mithilfe der Anweisung WHERE eine Bedingung angeben. Die Aktualisierung erfolgt dann nur bei den Datensätzen, die diese Bedingung erfüllen.
- ✓ Wird keine Bedingung angegeben, erfolgt die Wertzuweisung bei allen Datensätzen.

Syntax PostgreSQL

Die Syntax von PostgreSQL ist größtenteils identisch, bis auf den Zusatz ONLY, mit dem Sie steuern können, dass nur die Felder der ausgewählten Tabelle geändert werden sollen. Lassen Sie den Ausdruck ONLY weg, werden auch die entsprechenden Zeilen in den selektierten Tabellen geändert.

```
UPDATE [ONLY] tabellenname SET feld1 = wert1, . . . , feldX = wertX
FROM tabellennamen [WHERE bedingung];
```

Beispiele für die Wertzuweisung

Wertzuweisung in der UPDATE-Anweisung	Erläuterung
SET bezeichnung = "Gerätenummer"	Zuweisen einer Textinformation
SET anzahl = 200	Speichern der Zahl 200 im Feld <i>anzahl</i>

Wertzuweisung in der UPDATE-Anweisung	Erläuterung
<code>SET anzahl = anzahl * 2</code>	Der aktuelle Wert des Felds <i>anzahl</i> wird mit zwei multipliziert und wieder im Feld <i>anzahl</i> gespeichert.
<code>SET bezeichnung = (SELECT artikel FROM t_lager WHERE id = 20)</code>	Mittels einer Unterabfrage wird aus der Tabelle <i>t_lager</i> ein Datensatz abgefragt und der ermittelte Wert im Datenfeld <i>bezeichnung</i> gespeichert.

Bei Wertzuweisungen durch Unterabfragen müssen Sie die Abfrage so definieren, dass Sie nur einen Datensatz zurückliefert. MariaDB unterstützt keine Unterabfragen in der UPDATE-Anweisung.

6.3 Bedingtes Einfügen / Aktualisieren von Daten

Seit dem Standard SQL:2003 gibt es die Möglichkeit, Daten in Abhängigkeit vom vorhandenen Datenbestand entweder hinzuzufügen oder, wenn sie in Abhängigkeit einer angegebenen Bedingung bereits schon in einer Tabelle vorhanden sind, zu aktualisieren. Die im Standard dafür beschriebene Anweisung heißt MERGE. Häufig wird für die Funktionalität die Bezeichnung *upsert* verwendet.

Die Umsetzung in den einzelnen DBMS ist sehr unterschiedlich, nur einzelne – beispielsweise Oracle und DB2 – haben die im Standard vorgegebene Syntax direkt implementiert.

In MariaDB wird die Funktion über die Syntax `INSERT ... ON DUPLICATE KEY UPDATE` realisiert. In PostgreSQL besteht die Möglichkeit, mittels `INSERT INTO ... ON CONFLICT` Daten entweder einzufügen oder zu aktualisieren.

Beispiel: *BedingtesEinfuegenAendern.sql*

In den folgenden Anweisungen wird in PostgreSQL entweder ein neuer Datensatz hinzugefügt oder ein vorhandener entsprechend der Bedingung aktualisiert.

```

① INSERT INTO t_lager(id, preis) VALUES (8, 1)
    ON CONFLICT DO NOTHING;
② INSERT INTO t_lager(id, preis) VALUES (8, 2)
    ON CONFLICT (id) DO UPDATE SET preis = 2;

```

- ① Es soll zur Tabelle *t_lager* ein Datensatz mit den Werten 8 für das Feld *id* und 1 für das Feld *preis* hinzugefügt werden. Falls es bereits ein Datensatz existiert, beim das Feld *id* den Wert 8 besitzt soll keine Aktion ausgeführt werden.
- ② Hier wird für die identische Situation bestimmt, dass bei einem schon vorhandenen Datensatz der Wert des Feldes *preis* geändert werden soll.

```

uebungen=# select * from t_lager WHERE id = 8;
+---+---+---+
| id | stueck | preis |
+---+---+---+
| 8  |    25  |     1  |
+---+---+---+
(1 Zeile)

uebungen=# INSERT INTO t_lager(id, preis) VALUES (8, 1) ON CONFLICT DO NOTHING;
INSERT 0 0
uebungen=# select * from t_lager WHERE id = 8;
+---+---+---+
| id | stueck | preis |
+---+---+---+
| 8  |    25  |     1  |
+---+---+---+
(1 Zeile)

uebungen=# INSERT INTO t_lager(id, preis) VALUES (8, 2) ON CONFLICT (id) DO UPDATE SET preis = 2;
INSERT 0 1
uebungen=# select * from t_lager WHERE id = 8;
+---+---+---+
| id | stueck | preis |
+---+---+---+
| 8  |    25  |     2  |
+---+---+---+
(1 Zeile)

```

Anwendung des Befehls im Beispiel

6.4 Daten löschen

Mithilfe der `DELETE`-Anweisung können Sie einen oder mehrere Datensätze einer Tabelle löschen. Die Auswahl der Datensätze kann dabei wie bei der `UPDATE`-Anweisung mit einer `WHERE`-Klausel eingegrenzt werden.

Um Daten in einer Tabelle löschen zu können, müssen Sie Besitzer der Tabelle sein bzw. über die entsprechenden Rechte verfügen. Der Administrator vergibt diese Rechte mit der `GRANT-DELETE`-Anweisung.

Werden in der Anweisung noch weitere Tabellen angesteuert, müssen Sie über das Recht `SELECT` für diese Tabellen verfügen.

Beispiel: *DatenLoeschen.sql*

Es werden im Folgenden einige Möglichkeiten zum Löschen von Datensätzen mit verschiedenen Bedingungen vorgestellt.

①	<code>DELETE FROM t_lager WHERE stueck = 0;</code>
②	<code>DELETE FROM t_ma_dt WHERE ort = "Frankfurt";</code>
③	<code>DELETE FROM t_ma_frankfurt;</code>

- ① Alle Datensätze der Tabelle `t_lager` werden gelöscht, bei denen der Wert im Feld `stueck` gleich `0` ist.
- ② Es werden alle Datensätze aus der Tabelle `t_ma_dt` gelöscht, bei denen als Ort der Wert `Frankfurt` gespeichert wurde.
- ③ Da keine Bedingung angegeben wurde, werden alle Datensätze der Tabelle `t_ma_frankfurt` gelöscht.

Syntax MariaDB

```
DELETE FROM tabellenname [WHERE bedingung] ;
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern **DELETE FROM** eingeleitet. Danach folgt der gewünschte Tabellename.
- ✓ Um den Löschvorgang auf einen oder mehrere bestimmte Datensätze einzuschränken, kann eine **WHERE**-Klausel angegeben werden.

Syntax PostgreSQL

Wie schon bei dem Befehl **UPDATE** ist auch hier die Syntax von PostgreSQL größtenteils identisch mit der von MariaDB, bis auf den Zusatz **ONLY**, mit dem Sie steuern können, dass nur die Felder der ausgewählten Tabelle gelöscht werden sollen. Lassen Sie den Ausdruck **ONLY** weg, werden auch die entsprechenden Zeilen in den selektierten Tabellen gelöscht.

```
DELETE FROM [ONLY] tabellenname [WHERE bedingung] ;
```



Wenn Sie keine Bedingung angeben, werden ohne Nachfrage alle Datensätze der Tabelle gelöscht. Als Resultat verbleibt eine leere Tabelle.

6.5 Übung

Daten eingeben und löschen

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*

1. Wechseln Sie zur Datenbank *Uebungen*.

Fügen Sie der Tabelle *t_lager* ein neues Datenfeld hinzu. Es soll Artikelnamen mit maximal 100 Zeichen Länge speichern. Wählen Sie einen geeigneten Datentyp und benennen Sie das Feld mit *name*.

Fügen Sie folgende Datensätze in die Tabelle *t_lager* ein.

ID	Name	Stückzahl	Preis
1	Kugelschreiber	200	2,99
2	Ordner	123	2,50
3	Heftklammern	423	0,99
4	Bleistift	170	0,99
5	Umschläge B6	230	0,80

Fügen Sie einen Datensatz für den Artikel *Schreibblock A4* mit dem Preis 1,99 und der ID 6 hinzu. Für diesen Artikel ist keine Stückzahl bekannt.

Führen Sie eine einfache Datenabfrage durch, um das erfolgreiche Einfügen der neuen Datensätze zu überprüfen.

Ändern Sie die Stückzahl der vorhandenen Bleistifte von 170 auf 270.

Ändern Sie den Preis der Ordner von 2,50 auf 2,80.

Löschen Sie den Datensatz *Schreibblock A4*.

2. Wechseln Sie zur Datenbank *Uebungen*.

Fügen Sie folgende Datensätze in die Tabelle *t_ma_dt* ein.

Name	Vorname	Straße	Haus-nummer	PLZ	Ort	Geburtsdatum	Land
Haas	Martina	Blumenweg	23	63065	Offenbach	01.06.1972	D
Richter	Carsten	Frankfurter Str.	16	63067	Offenbach	18.10.1982	D
Seiler	Janine	Goethestr.	61	63067	Offenbach	11.11.1990	D
Hartmann	Jochen	Berliner Str.	23	60528	Frankfurt	25.03.1979	D
Goldbach	Martin	Frankfurter Str.	6	60529	Frankfurt	06.07.1981	D
Naumann	Norbert	Goethestr.	161	60594	Frankfurt	06.11.1972	D

In die Tabelle *t_ma_frankfurt* sollen alle Datensätze aus der Tabelle *t_ma_dt* eingefügt werden, bei denen der Ort den Wert *Frankfurt* oder *Offenbach* besitzt.

Erstellen Sie zu diesem Zweck eine `INSERT`-Anweisung mit einer geeigneten Unterabfrage. Die `WHERE`-Klausel soll dabei folgendermaßen lauten:

```
ort = "Frankfurt" OR ort = "Offenbach"
```

Überprüfen Sie mit einer einfachen Datenabfrage, ob die neuen Datensätze in die Tabelle *t_ma_frankfurt* eingefügt wurden.

Ändern Sie in den Datensätzen der Tabelle *t_ma_frankfurt* den Wert *Frankfurt* in *Frankfurt/Main*.

7

Einfache Datenabfrage

7.1 Grundlagen zu einfachen Datenabfragen

Einführung zur **SELECT**-Anweisung

Das gezielte Abfragen der gespeicherten Informationen und Daten kann eine schwierige Aufgabe beim Arbeiten mit einer Datenbank sein. In SQL wird dazu die **SELECT**-Anweisung verwendet. Sie ist die komplexeste SQL-Anweisung, besitzt viele optionale Erweiterungen und wird in der Praxis am häufigsten genutzt.

Die **SELECT**-Anweisung ähnelt einer Frage bzw. Aufforderung an das Datenbanksystem, die gewünschten Daten zu liefern. Die Frage können Sie dabei sehr einfach halten oder sehr komplex formulieren, z. B. mit vielschichtigen Bedingungen. Je nach Qualität der Anfrage erhalten Sie danach Daten, die schlecht oder sehr gut zu Ihren Erfordernissen passen. Als Ergebnis liefert die **SELECT**-Abfrage im Regelfall die Daten in Tabellenform.

Die Vor- und Familiennamen aller Mitarbeiter aus der Tabelle *t_ma* können Sie z. B. mit einer recht einfachen Anweisung abfragen.

```
SELECT vname, name FROM t_ma;
```

Diese Anweisung übermittelt dem Datenbanksystem die Aufforderung, alle Daten aus den Datenfeldern *vname* und *name* zu liefern, die in der Tabelle *t_ma* gespeichert sind.

SELECT-Anweisung verwenden

Mithilfe der **SELECT**-Anweisung können Sie alle oder ausgewählte Datenfelder von Tabellen abfragen. Die **SELECT**-Anweisung muss die gewünschten Daten dabei nicht nur aus einer einzigen Tabelle ermitteln. Es sind auch verknüpfte Abfragen über zwei oder mehr Tabellen möglich.

Die mit einer **SELECT**-Anweisung abgefragten Daten können Sie gezielt über sogenannte **WHERE**-Bedingungen eingrenzen. Außerdem ist das Sortieren und Gruppieren des Abfrageergebnisses möglich.

Beispiel: Datenabfrage1.sql

Mit dieser Anweisung werden alle Datenfelder und alle Datensätze aus der Tabelle `t_ma_dt` gelesen und angezeigt.

```
SELECT * FROM t_ma_dt;
```

MariaDB [mav]> SELECT * FROM t_ma_dt;									
id	name	vname	str	hnr	plz	ort	gebdat	land	
1	Illner	Markus	Goetheweg Str.	55	70173	Stuttgart	1976-12-23	D	
2	Schubert	Steve	Albert-Schweizer-Str.	10	70174	Stuttgart	1982-02-15	D	
3	Sattke	Beatrice	NULL	NULL	NULL	NULL	NULL	NULL	
9	Meier	Susann	Hauptweg	4	60385	Frankfurt	1972-04-21	D	
10	Brauer	Claudia	Bahnhofstr.	156	60386	Frankfurt	1971-04-28	D	
11	Yildiz	Umit	Universitätsstr.	24	60322	Frankfurt	1974-08-29	D	
12	Becker	Sebastian	Heidestr.	150	60385	Frankfurt	1994-08-14	D	
13	Hartmann	Jochen	Berliner Str.	23	60528	Frankfurt	19/9-03-25	D	

Die Abfrage liefert in MariaDB eine übersichtliche Darstellung der Tabelle

Die einfachste SELECT-Anweisung lautet `SELECT * FROM Tabellenname`. Damit rufen Sie schnell alle Datensätze einer gewünschten Tabelle ab.

Beispiel: Datenabfrage2.sql

Mit der folgenden Anweisung werden nur die Datenfelder `name`, `vname`, `plz` und `ort` aus der Tabelle `t_ma_dt` ausgelesen. Gleichzeitig wird die Abfrage auf alle Datensätze begrenzt, bei denen der Wert im Feld `plz` mit der Ziffer 6 beginnt und bei denen der Wert im Feld `gebdat` ein Datum nach dem 01.01.1975 beinhaltet. Die Datensätze werden bei der Anzeige aufsteigend nach der Postleitzahl sortiert.

Bei dieser Form der Abfrage handelt es sich um eine Projektion und – bedingt durch die gleichzeitige Beschränkung auf ausgewählte Datensätze – um eine Selektion (vgl. Abschnitt 3.4: *Theorie relationaler Sprachen*, Übersicht der Operationen der Relationenalgebra, Stichworte 'Projektion', 'Selektion').

```
SELECT name, vname, plz, ort FROM t_ma_dt
 WHERE plz LIKE '6%' AND gebdat > "1975-01-01" ORDER BY plz;
```

MariaDB [mav]> SELECT name, vname, plz, ort FROM t_ma_dt -> WHERE plz LIKE '6%' AND gebdat > "1975-01-01" ORDER BY plz;			
name	vname	plz	ort
Becker	Sebastian	60385	Frankfurt
Hartmann	Jochen	60528	Frankfurt

Die Abfrage liefert nur eine Auswahl der vorhandenen Datensätze

Syntax einer einfachen Datenabfrage

```
SELECT [DISTINCT] * | Datenfeld [, Datenfeld] FROM Tabellenname
[WHERE Bedingung]
[GROUP BY Datenfeld [, Datenfeld] [HAVING Bedingung] ]
[ORDER BY Datenfeld [, Datenfeld] [ASC|DESC]]
[LIMIT [Start, ] Anzahl];
```

- ✓ Die Anweisung wird mit dem Schlüsselwort SELECT eingeleitet.
- ✓ Um doppelte (identische) Datensätze zu vermeiden, können Sie nach dem Schlüsselwort SELECT die Angabe DISTINCT hinzufügen.
- ✓ Danach folgt die Angabe der Namen der gewünschten Datenfelder. Die Datenfeldnamen müssen den bei der Tabellendefinition angegebenen Namen entsprechen. Die Feldnamen werden durch Kommata getrennt. Das Zeichen * steht für alle Datenfelder der Tabelle.
- ✓ Nach dem Schlüsselwort FROM folgt die Angabe der gewünschten Tabelle.

Alle folgenden Angaben sind optional und können entfallen.

- ✓ Die WHERE-Klausel, gefolgt von einer Bedingung, schränkt die Datensatzauswahl ein. Es werden nur die Datensätze geliefert, für welche die Bedingung zutrifft.
- ✓ Mit GROUP BY, gefolgt vom Namen eines Datenfeldes oder einer Datenfeldliste, können die Daten nach diesen Feldern gruppiert werden.
- ✓ Über die HAVING-Klausel kann die gruppierte Abfrage durch eine Bedingung eingeschränkt werden. Die Angabe von Aggregatfunktionen in der Bedingung ist hier möglich.
- ✓ Über das Schlüsselwort ORDER BY, gefolgt von einem oder mehreren Datenfeldnamen, wird die Sortierung der Ausgabedaten beeinflusst.
- ✓ Mit dem Schlüsselwort LIMIT kann die Anzahl der zurückgelieferten Datensätze begrenzt werden. Gleichzeitig ist dabei die Ausgabe ab einem bestimmten Startdatensatz möglich.

Die Reihenfolge der Feldnamen in der SELECT-Abfrage ist beliebig und nicht an die Anordnung der Datenfelder in der Tabelle gebunden. Sie können daher die Daten in der Ergebnisliste in die von Ihnen gewünschte Ordnung bringen.

Besondere Möglichkeit bei der Datenausgabe

Spalten umbenennen

Als Spaltenüberschrift in der Ausgabe der Daten wird stets der Name des Datenfeldes verwendet. Sie können mit dem Schlüsselwort AS für jedes Datenfeld der Abfrage einen Ersatznamen (Alias) definieren. Auf diese Weise können Sie z. B. die Datenausgabe durch aussagekräftigere Feldnamen benutzerfreundlicher gestalten oder in komplexen Abfragen den Eingabeaufwand durch abgekürzte Feldnamen reduzieren.

Beispiel: *Datenabfrage3.sql*

Es wird eine Abfrage ausgewählter Datensätze und Datenfelder mit neuen aussagekräftigen Feldnamen erzeugt. Die Festlegung eines anderen Spaltennamens entspricht der Operation Umbenennen der Relationenalgebra (vgl. Abschnitt 3.4: *Theorie relationaler Sprachen*, Übersicht der Operationen der Relationenalgebra, Stichwort 'Umbenennen').

① **SELECT name AS "Familienname", vname AS "Vorname", plz AS "Postleitzahl", gebdat AS "Geburtsdatum" FROM t_ma_dt WHERE gebdat < "1975-01-01";**

- ① Hier werden für die einzelnen Elemente der SELECT-Anweisung Ersatznamen definiert. Diese Namen erscheinen in der Datenausgabe als Spaltenüberschriften.

```
MariaDB [mav]> SELECT name AS "Familienname", vname AS "Vorname", plz AS "Postleitzahl",
->     gebdat AS "Geburtsdatum" FROM t_ma_dt
-> WHERE gebdat < "1975-01-01";
+-----+-----+-----+-----+
| Familienname | Vorname | Postleitzahl | Geburtsdatum |
+-----+-----+-----+-----+
| Meier       | Susann   | 60385      | 1972-04-21  |
| Brauer      | Claudia  | 60386      | 1971-04-28  |
| Yildiz      | Umit     | 60322      | 1974-08-29  |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Ersatznamen können z. B. die Lesbarkeit der Datenausgabe verbessern



- ✓ Entspricht der Ersatzname einem SQL-Schlüsselwort, z. B. ALTER oder UPDATE, oder soll der Name Leerzeichen enthalten, müssen Sie ihn in Anführungszeichen (" ") oder Apostrophe (' ') setzen. Andernfalls wird bei der Ausführung eine Fehlermeldung erzeugt.
- ✓ Wenn Sie bei PostgreSQL in einer Abfrage mit Anführungszeichen arbeiten, sollten Sie diese durchgängig für alle Namen verwenden, da Sie sonst u. U. unerwünschte Ergebnisse erhalten.
- ✓ Da bei PostgreSQL Großbuchstaben in Abfragen ignoriert werden, müssen Sie Namen, die mit einem Großbuchstaben beginnen sollen, immer in Anführungszeichen setzen, nur dann werden sie auch mit Großbuchstaben wiedergegeben.

Konstante Werte ausgeben

In manchen Fällen ist es notwendig, zusätzlich zu den Datenfeldern einen konstanten Wert in den Ergebnisdaten zu übergeben. Solche Werte werden in der Liste der Datenfelder in Apostrophe bzw. Anführungszeichen gesetzt, um sie von den Feldnamen zu unterscheiden.

Beispiel: *Datenabfrage4.sql*

Die folgende Abfrage liefert in der ersten Spalte einen konstanten Wert.

① **SELECT 'Mitarbeiter' AS 'angestellt als', vname, name
FROM t_ma_dt WHERE id < 4;**

- ① Der konstante Wert *Mitarbeiter* wird in Anführungszeichen gesetzt. Er wird für jeden Datensatz des Ergebnisses der Abfrage in dieser Form übernommen. Als Spaltenüberschrift wird *angestellt als* festgelegt. Wird die Spaltenüberschrift nicht explizit mithilfe von AS umbenannt, erscheint dort ebenfalls der konstante Wert.

```
MariaDB [mav]> SELECT 'Mitarbeiter' AS 'angestellt als', vname, name FROM t_ma_dt
-> WHERE id < 4;
+-----+-----+
| angestellt als | vname   | name    |
+-----+-----+
| Mitarbeiter    | Markus   | Illner  |
| Mitarbeiter    | Steve    | Schubert|
| Mitarbeiter    | Beatrice | Sattke |
+-----+-----+
3 rows in set (0.00 sec)
```

Ausgabe eines konstanten Wertes

Anzahl der Datensätze beschränken

Die SELECT-Abfrage liefert standardmäßig alle Datensätze einer Tabelle bzw. alle Datensätze, auf die die angegebene Bedingung zutrifft. Bei sehr vielen Datensätzen kann dies lange dauern und es werden meist auch nicht sofort alle Datensätze als Ergebnis benötigt. Mit dem Schlüsselwort `LIMIT` können Sie die Anzahl der zurückgelieferten Datensätze begrenzen.

Beispiel: *Datenabfrage5.sql*

Die Abfrage liefert die Vornamen und Familiennamen aller Mitarbeiter, bei denen die Postleitzahl mit 6 beginnt. Die Ergebnismenge wird mit der Angabe `LIMIT 10` auf die ersten 10 Datensätze beschränkt.

```
SELECT vname, name FROM t_ma_dt WHERE plz LIKE "6%" LIMIT 10;
```

Beispiel: *Datenabfrage6.sql*

Durch die Angabe von `LIMIT` können Sie zusätzlich die Ausgabe der Datensätze ab einer bestimmten Position in der Ergebnismenge der Abfrage bestimmen. Um beispielsweise 10 Datensätze ab dem 5. Datensatz anzuzeigen, verwenden Sie die Angabe von `LIMIT 5,10`.

```
SELECT vname, name FROM t_ma_dt WHERE plz LIKE "6%" LIMIT 5,10;
```

Nicht alle Datenbanksysteme unterstützen die `LIMIT`-Funktion.

Doppelte Datensätze vermeiden

Um doppelte Datensätze zu vermeiden, können Sie nach dem Schlüsselwort `SELECT` die Angabe `DISTINCT` hinzufügen. Dabei ist es ausreichend, wenn sich die Daten in einem Datenfeld unterscheiden, um von der Datenbank als unterschiedlich erkannt zu werden.

Beispiel: *DatenabfrageDistinct.sql*

Im Folgenden wird der Unterschied zwischen den Ergebnismengen bei Verwendung von `DISTINCT` gezeigt.

①	<code>SELECT plz, ort FROM t_ma LIMIT 10;</code>
②	<code>SELECT DISTINCT plz, ort FROM t_ma LIMIT 10;</code>

- ① Bei dieser Abfrage werden alle Postleitzahlen und Orte aus der Tabelle `t_ma` ermittelt und die ersten 10 Datensätze ausgegeben.

- ② Durch die Angabe des Schlüsselwortes DISTINCT werden keine doppelten Datensätze ausgegeben, sodass sich eine andere Ergebnismenge ergibt.

MariaDB [uebungen]> SELECT plz,ort FROM t_ma LIMIT 10;	
plz	ort
22307	Hamburg
21029	Hamburg
1060	wien
8000	Zürich
60323	Frankfurt
1080	wien
21029	Hamburg
1210	wien
22159	Hamburg
10179	Berlin

Ausgabe mit doppeltem Datensatz

MariaDB [uebungen]> SELECT DISTINCT plz,ort FROM t_ma LIMIT 10;	
plz	ort
22397	Hamburg
10209	Hamburg
1060	wien
8000	Zürich
60323	Frankfurt
1080	wien
1210	wien
22159	Hamburg
10179	Berlin
60388	Frankfurt

Ausgabe der Abfrage ohne doppelten Datensatz

Berechnungen ausführen

In einer SELECT-Abfrage können Sie verschiedene Berechnungen durchführen. Die Ergebnisse werden wie der Inhalt eines Datenfeldes behandelt. Für Berechnungen können Sie alle numerischen Datenfelder der Tabelle sowie konstante Werte verwenden.

Beispiel: *DatenabfrageBerechnung.sql*

Es wird der Wert der gelagerten Artikel aus Anzahl und Stückpreis berechnet.

①	<code>SELECT id, preis, stueck, preis*stueck AS "Lagerwert" FROM t_lager LIMIT 10;</code>
---	---

- ① In der Abfrage werden die Werte der Datenfelder *preis* und *stueck* multipliziert, um den Wert der gelagerten Artikel zu ermitteln.

MariaDB [uebungen]> SELECT id, preis, stueck, preis*stueck AS Lagerwert FROM t_lager LIMIT 10;			
id	preis	stueck	Lagerwert
22	22	267	5874
23	10	100	1000
33	8.5	134	1139
38	35.8	89	3186.1999320983887
45	9.5	156	1482
46	12	322	3864
47	24.8	46	1140.7999649047852
48	5.5	245	1347.5
49	14.6	144	2102.4000549316406
50	34	45	1530

Abfrage mit berechneten Werten

Für die Berechnungen akzeptiert das Datenbanksystem nur numerische Datenfelder, z. B. vom Typ Integer oder Float. Wenn Sie Berechnungen mit anderen Datentypen versuchen, wird eine Fehlermeldung ausgegeben.

Folgende Operatoren können Sie für Berechnungen in SELECT-Abfragen in SQL verwenden:

Für die Ausführungsreihenfolge werden Operatoren nach Prioritäten eingeteilt. Dabei gilt wie in der Mathematik, dass Punktrechnung vor Strichrechnung ausgeführt wird. Die Ausführungsreihenfolge kann jedoch durch Klammerpaare () festgelegt werden, wobei der Inhalt der Klammern immer zuerst ausgewertet wird.

Operator	Erklärung
-	Subtraktion
+	Addition
*	Multiplikation
/	Division
%	Modulo

7.2 Bedingungen definieren

Einführung

Mithilfe der WHERE-Klausel können Sie die mit einer SELECT-Abfrage zurückgelieferten Daten einschränken bzw. filtern. Dies ist vor allem bei sehr großen Datenmengen wichtig und wird von Datenbanken auch bei mehreren Millionen Datensätzen sehr schnell ausgeführt, wenn diese dafür optimiert wurden (z. B. durch Verwendung von Indizes).

Für das Definieren der WHERE-Bedingung steht in SQL eine große Anzahl verschiedener Operatoren und Funktionen zur Verfügung. Einzelne Bedingungen können darüber hinaus mithilfe von logischen Verknüpfungen miteinander verbunden werden.

Möglichkeiten für WHERE-Bedingungen finden Sie in der folgenden Tabelle.

Bedingung	Beispiel	Erklärung
Vergleichsoperatoren	preis < 100 name = "Meier"	Vergleicht den Wert eines Datenfelds mit einem vorgegebenen Wert
Bereichsprüfung	preis BETWEEN 10 AND 100	Prüft, ob der Wert eines Feldes innerhalb eines bestimmten Bereichs liegt
Elementprüfung	abteilung IN ("Einkauf", "Verkauf")	Prüft, ob der Wert eines Feldes sich in der angegebenen Liste befindet
Mustervergleich	name LIKE "M%"	Überprüft einen Feldinhalt auf Übereinstimmung mit einem angegebenen Muster
NULL-Wertprüfung	preis IS NULL	Prüft einen Feldinhalt auf den Wert NULL (Datenfeld enthält keinen Wert)
Logische Operatoren	preis < 100 AND preis > 10 name <> "Meier" OR name <> "Mayer"	Verknüpfen mehrerer Bedingungen
Existenzbedingungen	EXISTS (SELECT * FROM t_lager WHERE t_artikel.name = t_lager.name)	Durch Existenzbedingungen wird getestet, ob die Antworttabelle einer Unteranfrage leer ist oder nicht. Es ist möglich, Existenzbedingungen zu negieren (NOT EXISTS).

In WHERE-Bedingungen können Sie nur Datenfelder und konstante Werte verwenden. Ersatznamen, die Sie in der Datenfeldliste einer SELECT-Anweisung mit dem Schlüsselwort AS definiert haben, sind nicht möglich, da Sie der Datenbank zu diesem Zeitpunkt noch nicht bekannt sind.

Vergleichsoperatoren

Beim Vergleich werden zwei Ausdrücke über einen Operator miteinander verglichen. Dabei können zwei Datenfelder oder ein Datenfeld und ein konstanter Ausdruck miteinander verglichen werden.

SQL besitzt die in der nebenstehenden Tabelle aufgelisteten Operatoren.

Für den Vergleich von Textwerten sind nur die Operatoren = und <>, die auf Gleichheit oder Ungleichheit überprüfen, erlaubt. Für detailliertere Textvergleiche nach bestimmten Mustern können Sie den LIKE-Operator verwenden.

Operator	Erklärung
<	kleiner als
>	größer als
<>	ungleich
=	ist gleich
>=	größer oder gleich
<=	kleiner oder gleich

Beispiel: BedingungVergleich.sql

In diesen Anweisungen wird die Verwendung verschiedener Vergleichsoperatoren gezeigt.

```

① SELECT * FROM t_lager WHERE preis < 20;
② SELECT * FROM t_ma WHERE ort = "Hamburg";
③ SELECT * FROM t_ma WHERE ort <> "Berlin";

```

- ① Die Abfrage liefert alle Datensätze, bei denen der Preis kleiner ist als 20.
 - ② Hier wird die Tabelle *t_ma* nach allen Datensätzen gefiltert, bei denen der Ort *Hamburg* ist.
 - ③ Die Ergebnismenge der Abfrage enthält alle Datensätze, bei denen der Ort nicht *Berlin* ist.
- ✓ Bei Vergleichen mit numerischen Werten mit Kommastrichen müssen Sie die Dezimalstellen mit einem Punkt statt mit einem Komma trennen, z. B. 12.5 statt 12,5.
 - ✓ Wenn bei einem Vergleich von Textwerten die Groß- bzw. Kleinschreibung keine Rolle spielt, können Sie den Wert eines Datenfeldes vor dem Vergleich mithilfe der Funktionen UPPER bzw. LOWER in Groß- bzw. Kleinschreibung überführen.

Beispiel: BedingungVergleichText.sql

Im folgenden Vergleich soll die Groß-/Kleinschreibweise ignoriert werden. Dies ist beispielsweise dann sinnvoll, wenn Werte wie *Hamburg*, *HAMBURG*, *hamburg* auftreten können, die zwar alle den gleichen Ort bezeichnen, aber unterschiedlich geschrieben wurden.

```
① SELECT * FROM t_ma WHERE UPPER(ort) = 'HAMBURG';
```

- ① Mit der Funktion UPPER wird der Wert des Datenfeldes *ort* vor dem Vergleich in Großbuchstaben gewandelt.

Einige Systeme unterscheiden nicht zwischen Groß- und Kleinschreibung, wie z. B. MariaDB und PostgreSQL. Sie erhalten mit und ohne Aufruf der Funktionen das gleiche Ergebnis.

Bereichsprüfung

Die Operatoren BETWEEN und IN erleichtern das Definieren von Wertebereichen oder Wertelisten, indem sie mehrere verbundene Vergleichsausdrücke ersetzen. Mit dem Operator BETWEEN prüfen Sie, ob der Wert eines Datenfelds in einem bestimmten Wertebereich liegt. Der Operator ersetzt dabei die Bedingung `x >= Wert1 AND x <= Wert2`.

Beispiel: *BedingungOperatorBetween.sql*

Die ersten beiden Ausdrücke verwenden unterschiedliche Operatoren, um alle Artikel mit einem Preis zwischen 10 und 100 zu ermitteln. Im dritten Ausdruck wird ein alphabetischer Bereich angegeben.

①	<code>SELECT * FROM t_lager WHERE preis >= 10 AND preis <= 100;</code>
②	<code>SELECT * FROM t_lager WHERE preis BETWEEN 10 AND 100;</code>
③	<code>SELECT * FROM t_ma WHERE name BETWEEN 'Be' AND 'Bo';</code>

- ① Bei dieser Bedingung wird mit zwei Vergleichen geprüft, ob der Preis größer oder gleich 10 und gleichzeitig kleiner oder gleich 100 ist.
- ② Der BETWEEN-Operator vereinfacht den Ausdruck ①.
- ③ Auch ein alphabetischer Bereich kann auf diese Weise angegeben werden. Es werden hier die Datensätze ausgewählt, bei denen die Namen zwischen *Be* und *Bo* liegen. Das sind alle Namen, die mit *Be* beginnen, bis zu den Namen, die mit *Bn* beginnen. Auch der Wert *Bo* wird als richtig erkannt. Namen, die mit *Bo* beginnen, werden nicht mit ausgewählt, da sie außerhalb der Bereichsgrenzen liegen.

Syntax

<code>SELECT ... WHERE Datenfeld BETWEEN Untergrenze AND Obergrenze;</code>

- ✓ Zuerst wird das zu überprüfende Datenfeld angegeben. Danach folgen das Schlüsselwort BETWEEN und die Angabe der unteren und der oberen Grenze, die mit AND verbunden werden.
- ✓ Die Werte für Unter- und Obergrenze sind im Wertebereich enthalten.

Elementprüfung

Sie können auch prüfen, ob der Wert eines Datenfeldes in einer Liste von Werten enthalten ist. Dazu definieren Sie eine Werteliste, die alle möglichen Werte enthält. Mithilfe des IN-Operators testen Sie in einer Bedingung, ob der Wert in der Liste enthalten ist.

Beispiel: *BedingungOperatorIn.sql*

Die beiden folgenden Ausdrücke verwenden unterschiedliche Operatoren, um die Mitarbeiter aus drei verschiedenen Orten zu ermitteln.

①	<code>SELECT * FROM t_ma WHERE ort = 'Berlin' OR ort = 'Wien' OR ort = 'Hamburg';</code>
②	<code>SELECT * FROM t_ma WHERE ort IN ('Berlin', 'Wien', 'Hamburg');</code>

- ① Hier wird durch drei jeweils durch OR-Verknüpfungen miteinander verbundene Vergleiche geprüft, ob der Wert des Datenfeldes *ort* gleich *Berlin*, *Wien* oder *Hamburg* ist.
- ② Übersichtlicher wird die gleiche Bedingung mit dem IN-Operator formuliert. Die Liste enthält die drei gewünschten Orte.

Syntax

```
SELECT ... WHERE Datenfeld IN (Werteliste) ;
```

- ✓ Zuerst erfolgt die Angabe des gewünschten Datenfeldes, gefolgt vom Schlüsselwort **IN**. Danach wird, in runde Klammern gesetzt, eine Liste mit den gewünschten Werten angegeben.
- ✓ Die einzelnen Werte der Liste werden durch Kommata getrennt. Textwerte müssen in Anführungszeichen (" ") oder Apostrophe (') eingeschlossen werden.

Mustervergleich

Speziell für Textwerte existiert der **LIKE**-Operator, der einen flexiblen Vergleich mit einem vorgegebenen Muster ermöglicht. Dies ist besonders dann nützlich, wenn Ihnen als Suchkriterium nur ein Teil einer Information zur Verfügung steht.

In dem Vergleichsmuster können Sie zwei verschiedene Platzhalterzeichen verwenden.

Platzhalter	Erklärung	Beispiel	Mögliches Ergebnis
%	Platzhalter steht für kein, ein oder mehrere beliebige Zeichen	name LIKE "F%" name LIKE "%son" name LIKE "%ill%"	Funke, Franz, Fahrmann Benson, Jenson, Morrison Miller, Filler, Ofillson
_	Platzhalter steht für exakt ein beliebiges Zeichen	name LIKE "M_ller" name LIKE "____"	Müller, Möller, Miller Adas, Funk, Tier

Microsoft Access verwendet statt des Platzhalters "%" das Zeichen "*".

Beispiel: *BedingungOperatorLike.sql*

Durch die Verwendung des Operators **LIKE** werden Mitarbeiter der Tabelle *t_ma* durch verschiedene Mustervergleiche selektiert.

```
① SELECT * FROM t_ma WHERE name LIKE "Me%";  
② SELECT * FROM t_ma WHERE name LIKE "M%er";
```

- ① Die Abfrage liefert alle Mitarbeiter, deren Name mit *Me* beginnt.
- ② Die Abfrage liefert alle Mitarbeiter, deren Name mit *M* beginnt und auf *er* endet.

MariaDB [uebungen]> SELECT * FROM t_ma WHERE name LIKE 'Me%';										
id	name	vname	str	plz	ort	① abtnr	hnr	gebdat	land	
26	Meier	Kerstin	Nordstr.	1120	Wien	4	6	1966-12-27	AT	
34	Meyer	Matthias	Schulstr.	8038	Zürich	6	6	1972-05-23	CH	
36	Meyer	Peter	Am Ring	60594	Frankfurt	1	6	1974-08-13	D	

3 rows in set (0.00 sec)

MariaDB [uebungen]> SELECT * FROM t_ma WHERE name LIKE 'M%er';										
id	name	vname	str	plz	ort	② abtnr	hnr	gebdat	land	
26	Meier	Kerstin	Nordstr.	1120	wien	4	6	1966-12-27	AT	
34	Meyer	Matthias	Schulstr.	8038	Zürich	6	6	1972-05-23	CH	
36	Meyer	Peter	Am Ring	60594	Frankfurt	1	6	1974-08-13	D	
38	Möller	Jochen	Am Kirchhof	22111	Hamburg	3	23	1978-09-19	D	
58	Maier	Frank	Mittelstr.	60320	Frankfurt	1	37	1978-01-12	D	

5 rows in set (0.00 sec)

Auswahl aller Namen nach einem bestimmten Muster

Wenn in den zu vergleichenden Werten die als Platzhalterzeichen definierten Zeichen % bzw. _ vorkommen, müssen Sie mit dem Schlüsselwort **ESCAPE** ein Zeichen als sogenanntes Fluchtzeichen (Escape-Zeichen) kennzeichnen. Mit dem Fluchtzeichen können Sie in einem Muster festlegen, dass das folgende Zeichen kein Platzhalter ist.

Beispiel: *BedingungOperatorLikeEscape.sql*

Es soll nach allen Mitarbeitern in Abteilungen gesucht werden, deren Name mit *Abt_* beginnt. Da im Muster *Abt_* der Unterstrich verwendet wird, dieses Zeichen jedoch standardmäßig als Platzhalter dient, muss ein Fluchtzeichen definiert werden.

① **SELECT * FROM t_abt WHERE name LIKE "Abt\ _%" ESCAPE "\ "**

- ① Mithilfe von **ESCAPE** wird das Zeichen \ als Fluchtzeichen definiert. Im Muster wird durch das Voranstellen dieses Zeichens festgelegt, dass der Unterstrich in diesem Fall nicht als Platzhalter zu sehen ist. Das folgende Prozent-Zeichen arbeitet jedoch ganz normal als Platzhalterzeichen.

MariaDB [uebungen]> SELECT * FROM t_abt WHERE name LIKE 'Abt\ _%' ESCAPE '\ ';	
id	name
5	Abt_Organisation

1 row in set (0.01 sec)

Auswahl aller Abteilungsnamen, die mit der Zeichenkette *Abt_* beginnen

Syntax

SELECT ... WHERE Datenfeld LIKE "Muster" [ESCAPE "Zeichen"] ;

- ✓ Zuerst erfolgt die Angabe des gewünschten Datenfelds. Danach folgen das Schlüsselwort **LIKE** und das anzuwendende Muster in Anführungszeichen oder Apostrophen.
- ✓ Falls im Muster eines der Zeichen % oder _ verwendet werden muss, wird mit dem Schlüsselwort **ESCAPE** ein Fluchtzeichen definiert. Dieses Zeichen steht, gefolgt von einem Leerzeichen, in Anführungszeichen oder Apostrophen.
- ✓ Im Muster wird das Fluchtzeichen dem gewünschten Zeichen, durch ein Leerzeichen getrennt, vorangestellt.

Logische Operatoren

Mithilfe von logischen Operatoren können Sie mehrere Bedingungen miteinander verbinden. Auf diese Weise erhalten Sie sehr komplexe Suchkriterien.

SQL ermöglicht den Einsatz der folgenden logischen Operatoren:

AND	UND-Verknüpfung: Beide Bedingungen müssen erfüllt sein.
OR	ODER-Verknüpfung: Mindestens eine der Bedingungen muss erfüllt sein.
NOT	NICHT-Operator: Die nachfolgende Bedingung wird negiert (aus wahr wird falsch und aus falsch wird wahr).

Beispiel: *BedingungOperatorLogik.sql*

In den folgenden Anweisungen werden mehrere Bedingungen mit logischen Operatoren verbunden.

```

① SELECT * FROM t_ma WHERE name LIKE 'M%' AND ort <> 'Berlin';
② SELECT * FROM t_ma WHERE name LIKE 'M%' AND
                           (ort = 'Berlin' OR ort = 'Hamburg');
③ SELECT * FROM t_ma WHERE NOT (name LIKE "M%") ;

```

- ① Die Abfrage liefert alle Mitarbeiter, deren Name mit *M* beginnt und die nicht in *Berlin* wohnen.
- ② Mit dieser Abfrage werden alle Mitarbeiter ermittelt, deren Name mit *M* beginnt und die in *Berlin* oder *Hamburg* wohnen.
- ③ Hier werden alle Mitarbeiter gesucht, deren Name nicht mit *M* beginnt.

Sie können mehrere logische Verknüpfungen beliebig kombinieren. Damit die Bedingung dabei übersichtlich bleibt und so von der Datenbank interpretiert wird, wie Sie es wünschen, sollten Sie auf die korrekte Klammersetzung achten.

7.3 Abfrageergebnisse gruppieren

Um die Datensätze einer Tabelle nach einem oder mehreren Kriterien zusammenzufassen (Gruppen zu bilden), verwenden Sie die GROUP-BY-Anweisung. Auf diese Weise suchen Sie alle Datensätze, die in einem bestimmten Datenfeld den gleichen Wert besitzen, und fassen diese zu einer Gruppe zusammen.

Das Gruppieren ermöglicht das Kombinieren von normalen Datenfeldern und Aggregatfunktionen in einer SELECT-Abfrage, was sonst eine Fehlermeldung erzeugen würde. Aggregatfunktionen dienen beispielsweise dazu, die Anzahl der selektierten Datensätze oder den Minimal- bzw. Maximalwert einer Spalte zu ermitteln. In einer SELECT-Abfrage ohne Gruppierung bezieht sich die Aggregatfunktion auf die gesamte Abfrage. Verwenden Sie Aggregatfunktionen in gruppierten Abfragen, beziehen sich diese auf die Gruppen und nicht auf die gesamte Abfrage.

Beispiel: *Gruppierung.sql*

Es soll ermittelt werden, wie viele Mitarbeiter aus jedem Ort kommen. Dazu werden alle Datensätze nach dem jeweiligen Ort gruppiert (zusammengefasst) und danach wird die Anzahl der Werte der Spalte "name" je Gruppe gezählt.

① **SELECT** **ort AS Wohnort, COUNT(name) FROM t_ma GROUP BY ort;**

- ① Die Datensätze der Tabelle werden mithilfe der Anweisung GROUP BY nach dem Ort gruppiert. Für jede Gruppe wird mit der Aggregatfunktion COUNT die Anzahl der Mitarbeiter ermittelt, die in diesem Ort wohnen. Der Name des Feldes, auf welches die Aggregatfunktion angewendet werden soll, folgt dem Funktionsnamen in runden Klammern.

MariaDB [uebungen]> SELECT ort AS Wohnort, COUNT(name) FROM t_ma GROUP BY ort;	
Wohnort	COUNT(name)
Berlin	10
Bern	3
Frankfurt	9
Hamburg	18
Wien	16
Zürich	10

6 rows in set (0.00 sec)

Durch die Gruppierung sind interessante statistische Auswertungen möglich

In gruppierten Abfragen kann die Menge der Datensätze über die HAVING-Klausel eingeschränkt werden. In der HAVING-Klausel ist im Unterschied zur WHERE-Bedingung die Verwendung von Aggregatfunktionen möglich, was in der Praxis häufig benötigt wird.

Syntaktisch ist die Verwendung der HAVING-Klausel ohne eine Gruppierung mittels der GROUP-BY-Anweisung möglich. Im Regelfall werden aber beide gemeinsam eingesetzt. Dabei ist die mit HAVING definierte Bedingung nach der GROUP-BY-Anweisung anzugeben.

Beispiel: *Having.sql*

Es soll ermittelt werden, wie viele Mitarbeiter aus jedem Ort kommen. Dabei sollen jedoch nur die Orte ausgegeben werden, aus denen mindestens 10 Mitarbeiter stammen. In der zweiten Beispielanweisung werden alle Namen gesucht, die mehr als einmal in der Tabelle enthalten sind.

① **SELECT** **ort AS Wohnort, COUNT(name) FROM t_ma GROUP BY ort **HAVING** COUNT(name) >= 10;**
 ② **SELECT** **name, COUNT(name) FROM t_ma GROUP BY name **HAVING** COUNT(name) > 1;**

- ① Mithilfe der HAVING-Klausel wird festgelegt, dass nur die Orte ausgegeben werden, aus denen zehn oder mehr Mitarbeiter kommen.
- ② Diese Anweisung gibt alle Namen zurück, die mehr als einmal in der Tabelle t_ma enthalten sind.

MariaDB [uebungen]> SELECT ort AS Wohnort, COUNT(name) FROM t_ma -> GROUP BY ort HAVING COUNT(name)>=10;	
Wohnort	COUNT(name)
Berlin	10
Hamburg	18
Wien	16
Zürich	10

4 rows in set (0.00 sec)

In der Auswertung erscheinen nur Orte, in denen **zehn und mehr** Mitarbeiter wohnen

MariaDB [uebungen]> SELECT name, COUNT(name) FROM t_ma -> GROUP BY name HAVING COUNT(name) > 1;	
name	COUNT(name)
Berger	2
Meyer	2

2 rows in set (0.00 sec)

In der Auswertung erscheinen nur Namen, die **mindestens zweimal** vorkommen

Syntax

```
SELECT ... GROUP BY Datenfeld[, Datenfeld, ...] [HAVING Bedingung];
```

- ✓ Mithilfe der GROUP-BY-Klausel lassen sich Abfragen nach einem oder mehreren Datenfeldern gruppieren. Dabei wird erst nach dem ersten Datenfeld gruppiert, innerhalb der ersten Gruppierung nach dem zweiten Datenfeld usw.
- ✓ Mit der HAVING-Klausel kann die Ergebnismenge in einer gruppierten Abfrage eingeschränkt werden. In der Bedingung können Aggregatfunktionen angegeben werden, was bei der WHERE-Klausel nicht zulässig ist.

7.4 Abfrageergebnisse sortieren

Die SELECT-Anweisung liefert die Datensätze in keiner definierten Reihenfolge. Meist verwendet die Datenbank dabei die Reihenfolge, in der die Daten in die Tabelle eingegeben wurden. Um das Abfrageergebnis nach einem oder mehreren Datenfeldern zu sortieren, können Sie die ORDER-BY-Klausel verwenden.

Beispiel: *DatenSortieren.sql*

In den folgenden Abfragen werden mehrere Möglichkeiten des Sortierens der Ergebnisdaten einer Abfrage gezeigt.

①	SELECT vname, name, plz, ort FROM t_ma ORDER BY name, vname;
②	SELECT id, stueck FROM t_lager ORDER BY preis DESC LIMIT 15;
③	SELECT ort AS Wohnort, COUNT(name) AS "Anzahl Mitarbeiter" FROM t_ma GROUP BY ort ORDER BY ort;

- ① In dieser Anweisung wird zuerst nach dem Familiennamen und dann nach dem Vornamen der Mitarbeiter sortiert.
- ② Mit dem Schlüsselwort DESC wird eine absteigende Sortierung erreicht. Alle Datensätze der Tabelle *t_lager* werden ausgewählt und nach dem Preis in absteigender Reihenfolge sortiert. Der Preis wird dabei nicht angezeigt. Die Anzeige wird auf 15 Datensätze beschränkt.
- ③ Die Sortieranweisung lässt sich auch bei gruppierten Daten anwenden. Hier wird die Ausgabeliste nach dem Ort sortiert.

MariaDB [uebungen]> SELECT id, stueck FROM t_lager -> ORDER BY preis DESC;	
id	stueck
38	89
50	45
47	46
22	267
49	144
46	322
23	100
45	156
33	134
48	245

10 rows in set (0.00 sec)

Absteigende Sortierung nach dem Artikelpreis

MariaDB [uebungen]> SELECT ort AS Wohnort, COUNT(name) AS "Anzahl Mitarbeiter" -> FROM t_ma GROUP BY ort ORDER BY ort;	
wohnort	Anzahl Mitarbeiter
Berlin	10
Bern	3
Frankfurt	9
Hamburg	18
Wien	16
Zurich	10

6 rows in set (0.00 sec)

Anzahl der Mitarbeiter aus den jeweiligen Orten, sortiert nach Ortsnamen

Syntax

```
SELECT ... ORDER BY Datenfeld[, Datenfeld, ...] [ASC|DESC];
```

- ✓ Das Abfrageergebnis wird nach dem angegebenen Datenfeld geordnet. Sind mehrere Datenfelder angegeben, wird erst nach dem ersten, dann nach dem zweiten usw. sortiert.
- ✓ Standardmäßig wird in aufsteigender Reihenfolge sortiert, ASC (ascending) braucht somit nicht angegeben zu werden. Um eine absteigende Sortierreihenfolge zu erhalten, geben Sie DESC (descending) an.

Eine Sortierung nach Aggregatfunktionen ist nur in einigen Datenbanksystemen möglich (wie z. B. in MariaDB oder PostgreSQL). Sie können alternativ durch Speichern der Ergebnisse einer Funktion in einem eigenen Datenfeld oder durch Stored Procedures zu dem gewünschten Ergebnis kommen.

7.5 Übung

Abfragen in Datenbanken durchführen

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*

1. Wechseln Sie zur Datenbank *Uebungen*.

Erstellen Sie eine Datenabfrage für die Tabelle *t_ma*, um die Vornamen und Familiennamen aller Mitarbeiter zu ermitteln. Begrenzen Sie das Abfrageergebnis auf 15 Datensätze.

Erweitern Sie die Abfrage, damit Sie zusätzlich die Postleitzahl und den Ort der Mitarbeiter erhalten.

Definieren Sie für alle Felder der Abfrage sinnvolle Ersatznamen.

Ändern Sie die Abfrage, sodass Sie nur Mitarbeiter aus Hamburg und Berlin erhalten.

Verwenden Sie dazu die IN-Anweisung. Zusätzlich sollen die Mitarbeiter vor dem Jahr 1980 geboren sein. Lassen Sie sich die Ergebnisse der Abfrage nach Familiennamen der Mitarbeiter sortieren.

2. Wechseln Sie zur Datenbank *Uebungen*.

Ermitteln Sie aus der Tabelle *t_lager* für jeden Artikel den Bruttopreis, indem Sie zu dem in der Tabelle gespeicherten Preis 19 % Mehrwertsteuer hinzufügen.

Gruppieren Sie die Tabelle anhand der Stückzahl und ermitteln Sie in einer Abfrage die Anzahl der Artikel, die mit einer bestimmten Stückzahl vorhanden sind.

Schränken Sie die Abfrage auf Stückzahlen mit weniger als 10 Artikeln ein.

Sortieren Sie das Abfrageergebnis wenn möglich absteigend nach der Stückzahl.

8

Schlüsselfelder und Indices

8.1 Einführung zu Schlüsseln und Indizes

In vielen Datenbanksystemen werden Schlüssel und Indizes als gleichwertig behandelt. Für das relationale Datenmodell sind nur Schlüssel von Bedeutung. In der Datenbank wird jedoch für jeden Schlüssel automatisch ein Index angelegt.

Schlüssel	Durch einen Schlüssel wird jeder Datensatz einer Tabelle eindeutig identifiziert . Ein Schlüssel wird aus einem Datenfeld oder einer Kombination von Datenfeldern der Tabelle gebildet. Wie auch im relationalen Datenmodell können Schlüssel benutzt werden, um Beziehungen zwischen logisch zusammengehörigen Tabellen herzustellen.
Indizes	Ein Index wird für ein bestimmtes Datenfeld oder auch mehrere Datenfelder angelegt, nach denen häufig sortiert oder in denen häufig gesucht wird. Er ähnelt dem Inhaltsverzeichnis eines Buches und ist in Form eines B*-Baumes aufgebaut. Das Durchsuchen der Datensätze wird durch einen Index zum Teil erheblich beschleunigt. Für einen Schlüssel wird vom DBMS meist automatisch ein Index erstellt. Für eine Tabelle können Sie mehrere Indizes definieren.

- ✓ Bei den Schlüsseln im relationalen Datenmodell werden Primär-, Sekundär- und Fremd-schlüssel unterschieden. Nicht in jedem Datenbanksystem sind alle Schlüsselarten verfügbar.
- ✓ Für jede Tabelle können mehrere Indizes erstellt werden, jedoch nur ein Primärschlüssel.

Primärschlüssel

Der Primärschlüssel ist ein Schlüssel, der einen Datensatz eindeutig kennzeichnet, z. B. eine Kundennummer. Häufig handelt es sich dabei um eine fortlaufende Nummer. Das Datenbanksystem kann diese Nummern in der Regel automatisch erzeugen.

Jede Tabelle kann nur einen Primärschlüssel enthalten. Alle weiteren Schlüssel sind Sekundär-schlüssel. Im relationalen Datenmodell muss für jede Tabelle zwingend ein Primärschlüssel definiert werden. Dieses Schlüsselfeld darf niemals leer sein.

Sekundärschlüssel

Ähnlich wie beim Primärschlüssel müssen sich die Datensätze auch in den Sekundärschlüsseln unterscheiden. Sie dienen dazu, Redundanzen in den gespeicherten Daten zu vermeiden, da in einem Schlüsselfeld ein Wert nicht mehrfach vorkommen darf. Besitzt eine Tabelle *Abteilungen* beispielsweise die Felder *ID* und *Name*, kann das Feld *ID* zur eindeutigen Kennzeichnung der Datensätze verwendet werden (ist demnach Primärschlüssel). Damit als Abteilungsname eine Abteilung nicht mehrfach verwendet werden kann, wird das Feld *Name* zusätzlich als Sekundärschlüssel definiert.

Jede Tabelle kann mehrere Sekundärschlüssel enthalten. Diese Schlüsselfelder dürfen niemals leer sein und sie müssen sich in jedem Datensatz unterscheiden.

Fremdschlüssel

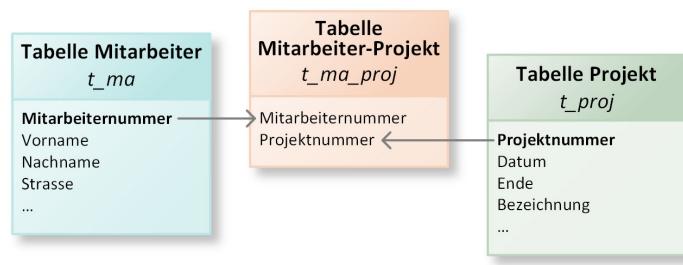
Ein Fremdschlüssel bezeichnet die Übereinstimmung eines Datenfeldes in einer Tabelle mit dem Primärschlüssel einer anderen Tabelle. Im relationalen Datenmodell werden häufig sogenannte Beziehungs-Tabellen verwendet, die zwei Tabellen miteinander verbinden.

Beispiel

Es existieren die Tabellen *t_ma* und *t_proj*. In der Tabelle *t_ma* werden alle Daten der Mitarbeiter eines Unternehmens gespeichert. Jeder Datensatz ist mit einer eindeutigen ID-Nummer (Mitarbeiternummer) als Primärschlüssel gekennzeichnet. Die Tabelle *t_proj* speichert die im Unternehmen laufenden und abgeschlossenen Projekte. Auch hier wird jeder Datensatz mit einem Primärschlüssel (der Projektnummer) gekennzeichnet.

Jeder Mitarbeiter ist einem oder mehreren Projekten zugeordnet, es besteht eine n:m-Beziehung zwischen beiden Tabellen.

Durch eine weitere Tabelle *t_ma_proj* wird die Referenz hergestellt. Hier werden die Mitarbeiternummern mit den zugehörigen Projektnummern gespeichert.



Primärschlüssel-Fremdschlüssel-Beziehung am Beispiel einer n:m-Beziehung

Da die Felder *Mitarbeiternummer* und *Projektnummer* in den jeweiligen Tabellen Primärschlüssel sind, entsteht eine Fremdschlüsselbeziehung. In der Tabelle *t_ma_proj* werden sie daher als Fremdschlüssel bezeichnet.

Indizes

Indizes erleichtern die Suche nach Datensätzen, insbesondere bei sehr großen Datenmengen. Standardmäßig durchsucht das Datenbanksystem bei der Suche nach bestimmten Werten in einer oder mehreren Spalten nacheinander alle Datensätze einer Tabelle. Existiert für eine oder mehrere Spalten jedoch ein Index, dann wird dieser zur Suche herangezogen und beschleunigt die Suche.

- ✓ Ein Index bringt vor allem dann große Geschwindigkeitsvorteile, wenn die Datensätze sehr viele Daten enthalten oder die Tabelle sehr groß ist.
- ✓ Beachten Sie, dass beim Einfügen, Löschen und Ändern von Datensätzen alle definierten Indizes aktualisiert werden müssen. Dies kann beim Füllen von Tabellen sehr viel Zeit in Anspruch nehmen.

Erzeugen Sie die Indizes für umfangreiche Tabellen erst nach dem Einfügen der Datensätze, da diese Operation sonst länger dauern kann.

8.2 Schlüsselfelder festlegen und bearbeiten

Primärschlüssel definieren und löschen

Primärschlüssel werden in der Regel beim Erstellen einer Tabelle definiert. Dazu können Sie die Anweisung PRIMARY KEY verwenden.

Beispiel: *PrimaerschluesselErstellen.sql*

Das Feld *id* der Tabelle *t_proj* wird als Primärschlüssel definiert. Da das Primärschlüsselfeld keine NULL-Werte enthalten darf, wird es mit NOT NULL definiert. Beim Einfügen neuer Datensätze soll es automatisch weiterzählen.

```

① CREATE TABLE t_proj
②   (id INTEGER NOT NULL AUTO_INCREMENT,
      name VARCHAR(50),
      beginn DATE,
      ende DATE,
      PRIMARY KEY(id));
    
```

- ① Die Tabelle *t_proj* wird erstellt. Sie soll die Namen der verschiedenen Projekte mit einer eindeutigen Projektnummer (*id*) speichern.
- ② Das Feld *id* soll jeden Datensatz eindeutig kennzeichnen und als Primärschlüssel eingesetzt werden. Da es ausschließlich ganze Zahlen aufnehmen soll, wird es vom Typ INTEGER deklariert. Die Anweisung NOT NULL legt fest, dass das Feld niemals leer sein darf. Die Angabe AUTO_INCREMENT legt fest, dass der Datenfeldinhalt bei jedem neuen Datensatz um den Wert eins erhöht wird.
- ③ Durch die Angabe von PRIMARY KEY wird das Feld *id* als Primärschlüssel definiert.

Syntax für die Definition des Primärschlüssels

Wenn Sie den Primärschlüssel für eine Spalte anlegen möchten, können Sie direkt nach der betreffenden Spaltendefinition die Option PRIMARY KEY eingeben (Spalten-Constraint).

```
CREATE TABLE tabellename
  (primärschlüsselfeld datentyp PRIMARY KEY . . . ,
  . . . );
```

- ✓ Der Primärschlüssel wird als normales Datenfeld in der CREATE-TABLE-Anweisung deklariert.
- ✓ Durch die Option PRIMARY KEY in der betreffenden Spaltendefinition wird bereits für die Spalte festgelegt, dass sie eindeutig sein muss und keine NULL-Werte enthalten darf. Die Angabe NOT NULL entfällt daher.

Neben der oben beschriebenen Syntax haben Sie auch die Möglichkeit, den Primärschlüssel erst nach der Angabe aller Spalten festzulegen (Tabellen-Constraint), beispielsweise, wenn der Primärschlüssel aus mehreren Spalten bestehen soll. In diesem Fall sieht die Syntax folgendermaßen aus.

```
CREATE TABLE tabellename
  (primärschlüsselfeld1 datentyp . . . NOT NULL,
  primärschlüsselfeld2 datentyp . . . NOT NULL,
  . . .
PRIMARY KEY(primärschlüsselfeld1, primärschlüsselfeld2) ;
```

- ✓ Der Primärschlüssel wird als normales Datenfeld in der CREATE-TABLE-Anweisung deklariert. Da der Schlüssel niemals leer sein darf, muss in manchen DBMS die Option NOT NULL verwendet werden.
- ✓ Am Ende der Tabellendefinition wird mit der Angabe PRIMARY KEY das entsprechende Datenfeld als Primärschlüssel gekennzeichnet. Der Name des Datenfeldes wird dabei in runde Klammern eingeschlossen. Werden mehrere Datenfelder als Primärschlüssel festgelegt, sind diese durch Kommata getrennt anzugeben.

Werden Primärschlüssel über mehrere Datenfelder angelegt, müssen sich die Datensätze in mindestens einem der Datenfelder unterscheiden. Beachten Sie, dass sich durch komplexe Primärschlüssel viele Operationen verlangsamen (Bearbeitung der Daten, Abfragen) und mehr Platz zu deren Speicherung benötigt wird. Da Primärschlüssel zum Verknüpfen von Tabellen benötigt werden, sollten diese möglichst einen Datentyp besitzen, der wenig Speicherplatz verwendet.

- ✓ Mit der Angabe AUTO_INCREMENT wird in MariaDB erreicht, dass der Wert eines Feldes beim Einfügen eines neuen Datensatzes automatisch erhöht wird. Derartige Datenfelder eignen sich gut als Primärindex.
- ✓ Bei PostgreSQL haben Sie die Möglichkeit, als Datentyp SERIAL zu wählen, wenn Sie eine Spalte erzeugen möchten, die automatisch eine eindeutige Nummerierung erstellt. SERIAL ist kein echter Datentyp, sondern eine abgekürzte Schreibweise für eine Spalte, die den Datentyp INTEGER mit einem sequenziellen Vorgabewert erstellt und NULL-Werte nicht zulässt.

```
CREATE TABLE tabellename
  (primärschlüsselfeld SERIAL PRIMARY KEY . . . ,
  . . . );
```

Primärschlüssel nachträglich hinzufügen oder löschen

Der Primärschlüssel wird meist bereits beim Erstellen einer Tabelle definiert. In einigen Fällen kann es jedoch notwendig sein, die Definition nachträglich zu ändern bzw. den Primärschlüssel einer bestehenden Tabelle hinzuzufügen. Dies können Sie mit der ALTER-TABLE-Anweisung erreichen. Mit der gleichen Anweisung ist es auch möglich, einen vorhandenen Primärschlüssel zu löschen.

Syntax

```
ALTER TABLE tabellenname ADD PRIMARY KEY (datenfeldname) ;
ALTER TABLE tabellenname DROP PRIMARY KEY;
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern **ALTER TABLE** eingeleitet. Danach folgt der Tabellename.
- ✓ Die **ALTER-TABLE**-Anweisung erlaubt verschiedene Änderungen an der Tabellenstruktur. Mit der Angabe **ADD PRIMARY KEY** wird der in runden Klammern angegebene Datenfeldname als Primärschlüssel definiert.
- ✓ Besteht der Primärschlüssel aus mehreren Datenfeldern, werden diese durch Kommata getrennt.
- ✓ Mit der Angabe **DROP PRIMARY KEY** wird ein vorhandener Primärschlüssel gelöscht.

Das nachträgliche Hinzufügen eines Primärschlüssels ist nur möglich, wenn das entsprechende Datenfeld mit der Angabe **NOT NULL** definiert wurde. Gleichzeitig dürfen die Datensätze keinen doppelten Wert in diesem Feld enthalten.

Sekundärschlüssel erstellen und löschen

Neben dem Primärschlüssel können Sie mehrere Sekundärschlüssele definieren, die sich aus beliebig vielen Datenfeldern zusammensetzen können. Diese Sekundärschlüssele werden mit der Anweisung **UNIQUE** beim Erstellen einer Tabelle definiert.

Beispiel: *SekundaerschluesselErstellen.sql*

Es wird eine Tabelle *t_artikel* mit einem Primärschlüssel und zwei Sekundärschlüsseln erzeugt.

```
CREATE TABLE t_artikel
  (id INTEGER NOT NULL AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL, code VARCHAR(30) NOT NULL,
  lieferant INTEGER, bemerkung VARCHAR(100),
  PRIMARY KEY(id),
  UNIQUE namecodeindex (name, code));
```

- ① Das Datenfeld *id* soll als Primärschlüssel verwendet werden. Dadurch darf es keine NULL-Werte enthalten. Außerdem soll es automatisch weiterzählen, wenn ein neuer Datensatz eingefügt wird. Zu diesem Zweck wird es bei MariaDB mit der Angabe **NOT NULL** und **AUTO_INCREMENT** definiert.

- ② Für jeden Artikel sollen ein spezifischer Name und ein Code gespeichert werden. Da sich die Artikel in mindestens einem dieser Daten unterscheiden, können beide als Sekundärschlüssel eingesetzt werden. Da für alle Schlüssel keine NULL-Werte erlaubt sind, wird auch hier die Angabe NOT NULL verwendet.
- ③ Der Primärschlüssel wird an dieser Stelle festgelegt.
- ④ Mit der UNIQUE-Klausel werden die Felder *name* und *code* als Sekundärschlüssel definiert. Optional kann bei MariaDB ein Name für diesen Schlüssel angegeben werden (nicht bei PostgreSQL). In diesem Fall soll er *namecodeindex* heißen.

Alle Schlüsselfelder dürfen niemals leer bleiben. Daher müssen Sie bei der Definition der Datenfelder die Angabe NOT NULL verwenden.

Sie können die Anweisung für eine Spalte direkt bei der Definition des betreffenden Datenfeldes angeben (Spalten-Constraint).

Syntax

```
CREATE TABLE tabellename
  (datenfeld1 datentyp1 [schlüsselname] UNIQUE . . . ,
   . . . );
```

- ✓ Die Sekundärschlüssel werden als normale Datenfelder in der CREATE-TABLE-Anweisung deklariert.
- ✓ Durch die Option UNIQUE in dem betreffenden Datenfeld wird bereits für die Spalte festgelegt, dass sie als Sekundärschlüssel dienen soll.
- ✓ Optional kann ein Name für den Sekundärschlüssel angegeben werden (nicht in allen DBS möglich, PostgreSQL unterstützt diese Möglichkeit nicht). Wird nichts angegeben, verwendet die Datenbank in der Regel den Namen des ersten Datenfeldes als Schlüsselnamen.

Neben der oben beschriebenen Syntax haben Sie auch die Möglichkeit, den Sekundärschlüssel erst nach der Angabe aller Spalten festzulegen. Dies ist beispielsweise nötig, wenn der Schlüssel aus mehreren Spalten bestehen soll. In diesem Fall sieht die Syntax folgendermaßen aus:

Syntax

```
CREATE TABLE tabellename
  (datenfeld1 datentyp1 . . . ,
   . . .
   UNIQUE [schlüsselname] (datenfeldliste));
```

- ✓ Am Ende der Tabellendefinition werden mit der Angabe UNIQUE die entsprechenden Datenfelder als Sekundärschlüssel gekennzeichnet. Die Namen der Datenfelder werden dabei durch Kommata getrennt und in runde Klammern eingeschlossen.

Sekundärschlüssel nachträglich hinzufügen oder löschen

Genau wie Primärschlüssel können Sie einer Tabelle auch nachträglich Sekundärschlüssel hinzufügen oder diese löschen. Dazu verwenden Sie die ALTER-TABLE-Anweisung.

Syntax

```
ALTER TABLE tabellenname ADD UNIQUE [schlüsselname]
(datenfeldliste);
ALTER TABLE tabellenname DROP INDEX schlüsselname;
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern **ALTER TABLE** eingeleitet. Danach folgt der Tabellename.
- ✓ Die **ALTER-TABLE**-Anweisung erlaubt verschiedene Änderungen an der Tabellenstruktur. Mit der Angabe **ADD UNIQUE** können Sie ein oder mehrere Datenfelder als Sekundärschlüssel definieren. Zusätzlich ist es in manchen DBS möglich, einen Schlüsselnamen festzulegen.
- ✓ Mit der Angabe **DROP INDEX** wird ein vorhandener Sekundärschlüssel gelöscht. Dazu ist der Name des Schlüssels notwendig.

Das nachträgliche Hinzufügen eines Sekundärschlüssels ist nur möglich, wenn die entsprechenden Datenfelder mit der Angabe **NOT NULL** definiert wurden. Gleichzeitig müssen alle Datensätze in diesem Feld unterschiedliche Werte besitzen. Ist dies nicht der Fall, erhalten Sie eine Fehlermeldung.

Fremdschlüssel erstellen

Im relationalen Datenmodell werden die Datenbestände auf mehrere Tabellen aufgeteilt, um sie möglichst effizient zu speichern und Redundanzen zu vermeiden. In einigen Tabellen werden dadurch nur Verweise auf die Primärschlüssel anderer Tabellen gespeichert. Diese Verweise werden Fremdschlüssel genannt. Sie werden mit der Anweisung **FOREIGN KEY/REFERENCES** beim Erstellen einer Tabelle definiert.

Damit die **referentielle Integrität** gewahrt bleibt, dürfen keine Datensätze aus einer Tabelle gelöscht werden, ohne dabei auch alle Verweise auf diese Datensätze in anderen Tabellen zu entfernen. Die Verweise würden sonst auf einen nicht mehr vorhandenen Datensatz zeigen. Sie dürfen beispielsweise keinen Mitarbeiter aus der Mitarbeitertabelle löschen, bevor Sie die Verweise auf diesen Mitarbeiter aus der Projektverwaltung entfernt haben.

Durch die Definition von Fremdschlüsseln beachtet die Datenbank automatisch die Regeln der referentiellen Integrität und verweigert entsprechende **INSERT**-, **DELETE**- und **UPDATE**-Anweisungen mit einer Fehlermeldung.



Nicht alle Datenbanksysteme unterstützen Fremdschlüssel in vollem Umfang.

Beispiel: *FremdschluesselErstellen.sql*

Im Folgenden wird eine Tabelle *t_ma_proj* mit zwei Fremdschlüsseln erstellt, um die Verbindung zwischen den Tabellen *t_ma* und *t_proj* herzustellen. Wenn Sie neue Datensätze in diese Tabelle einfügen, müssen Sie gültige ID-Nummern der beiden anderen Tabellen verwenden. Ansonsten erhalten Sie eine Fehlermeldung.

```
CREATE TABLE t_ma_proj
  (ma_id INTEGER NOT NULL,
   proj_id INTEGER NOT NULL,
   FOREIGN KEY (ma_id) REFERENCES t_ma (id),
   FOREIGN KEY (proj_id) REFERENCES t_proj (id));
```

- ① Die Tabelle enthält zwei Datenfelder, um eine Zuordnung eines Projektes zu einem Mitarbeiter zu ermöglichen. Da die Felder Verweise auf die Primärschlüssel der jeweiligen Tabellen speichern sollen, werden sie mit den gleichen Datentypen wie diese (beide INTEGER) und der Angabe NOT NULL deklariert.
- ② Die Tabelle enthält zwei Fremdschlüssele. Der erste weist das Datenfeld *ma_id* als Verweis auf das Feld *id* in der Tabelle *t_ma* aus.
- ③ Der zweite Fremdschlüssel definiert das Feld *proj_id* als Verweis auf das Feld *id* in der Tabelle *t_proj*.



Sie können für jede Tabelle mehrere Fremdschlüssele erstellen. Achten Sie dabei darauf, dass Sie in einer Tabelle keinen Fremdschlüssel auf eine Tabelle erstellen, die wiederum einen Fremdschlüssel enthält, der auf die erste Tabelle verweist. Diese sogenannten zirkulären Referenzen können zu Problemen beim Einfügen und Löschen von Datensätzen führen.

Syntax für das Erstellen von Fremdschlüssen

Wenn Sie den Fremdschlüssel für eine Spalte anlegen möchten, können Sie direkt nach der betreffenden Spaltendefinition die Option REFERENCES eingeben (Spalten-Constraint).

```
CREATE TABLE tabellename
  (datenfeld1 datentyp1,
   datenfeld 2 datentyp 2
   REFERENCES tabellename (datenfeldname)
   [ON UPDATE referenzoption] [ON DELETE referenzoption]),
  datenfeld 3 datentyp 3);
  ...;
```

- ✓ Der Fremdschlüssel wird bei der betreffenden Spaltendefinition der CREATE-TABLE-Anweisung deklariert.
- ✓ Nach der Angabe des Datenfeldes, das auf einen Primärschlüssel einer anderen Tabelle verweist, folgt die Anweisung REFERENCES und der Name der Tabelle, auf die verwiesen wird. In runden Klammern folgt dann der Name des Primärschlüsselfeldes dieser Tabelle.
- ✓ Wenn benötigt, können danach verschiedene Optionen angegeben werden, die das Verhalten bei Fremdschlüsselverletzungen steuern.

Neben der oben beschriebenen Syntax haben Sie auch die Möglichkeit, den Fremdschlüssel erst nach der Angabe aller Spalten festzulegen (Tabellen-Constraint). In diesem Fall sieht die Syntax folgendermaßen aus.

```
CREATE TABLE tabellename
  (datenfeld1 datentyp1 . . . ,
   . . .
   FOREIGN KEY (datenfeldname)
   REFERENCES tabellename (datenfeldname)
   [ON UPDATE referenzoption] [ON DELETE referenzoption]) . . .;
```

- ✓ Die Fremdschlüsse werden am Ende der CREATE-TABLE-Anweisung deklariert.
- ✓ Nach der Angabe FOREIGN KEY folgt in runden Klammern der Name des Datenfeldes, das auf einen Primärschlüssel einer anderen Tabelle verweist.
- ✓ Danach folgen das Schlüsselwort REFERENCES und der Name der Tabelle, auf die verwiesen wird. In runden Klammern folgt dann der Name des Primärschlüsseldes dieser Tabelle.
- ✓ Wenn benötigt, können danach verschiedene Optionen angegeben werden, die das Verhalten bei Fremdschlüsselverletzungen steuern.
- ✓ Durch Kommata getrennt können mehrere Fremdschlüsseldefinitionen angegeben werden.

Optionen für das Verhalten bei Fremdschlüsselverletzungen

Beim Einfügen eines Datensatzes in eine Tabelle mit Fremdschlüssen prüft das Datenbanksystem, ob eine Entsprechung in der referenzierten Tabelle vorhanden ist. Ist dies nicht der Fall, wird eine Fehlermeldung ausgegeben.

Beispiel

In die Tabelle *t_ma_proj* wird ein neuer Datensatz eingefügt. Dieser verweist auf eine Projektnummer (Projektnummer 5), die in der Tabelle *t_proj* nicht existiert (es sind nur die Projektnummern 1 bis 4 vorhanden). Das Datenbanksystem verweigert die Ausführung der SQL-Anweisung, damit die referentielle Integrität erhalten bleibt.

```
MariaDB [uebungen]> SELECT * FROM t_proj;
+----+-----+-----+
| id | name          | beginn | ende   |
+----+-----+-----+
| 1  | Buchprojekt    | 2021-03-01 | 2021-08-20 |
| 2  | Renovierung des Pausenraumes | 2021-02-01 | 2021-09-28 |
| 3  | Anlegen eines Pflichtenheftes | 2021-02-10 | 2021-09-21 |
| 4  | Kundenumfrage    | 2021-03-01 | 2021-09-30 |
+----+-----+-----+
4 rows in set (0.027 sec)

MariaDB [uebungen]> INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(1, 5);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails ('uebungen'.`t_ma_proj`, CONSTRAINT `t_ma_proj_ibfk_2` FOREIGN KEY (`proj_id`) REFERENCES `t_proj` (`id`) ON DELETE CASCADE)
MariaDB [uebungen]> _
```

Einfügen einer nicht gültigen Projektnummer in eine Tabelle mit Fremdschlüssen unter MariaDB

Für das Löschen oder Aktualisieren eines Datensatzes, auf den noch Referenzen in anderen Tabellen bestehen, können Sie das Verhalten der Datenbank differenzierter steuern. Dazu verwenden Sie bei der Definition des Fremdschlüssels die Angabe ON DELETE option bzw. ON UPDATE option.

Folgende Optionen sind dabei möglich:

NO ACTION	Die Ausführung der Anweisung wird abgebrochen.
CASCADE	Beim Löschen werden alle Datensätze in anderen Tabellen, die auf diesen Datensatz verweisen, ebenfalls gelöscht. Beim Aktualisieren werden alle referenzierten Datensätze in anderen Tabellen ebenfalls geändert.

SET DEFAULT	Alle referenzierten Datenfelder werden auf den DEFAULT-Wert (Standardwert) zurückgesetzt.
SET NULL	Alle referenzierten Datenfelder werden auf den Wert NULL gesetzt.

Wenn Sie keine Option angeben, wird standardmäßig jede Anweisung abgebrochen, die die referentielle Integrität verletzt.

Beispiel: *FremdschlüsselDatensatzLoeschen.sql*

Das Projekt mit der ID 4 ist abgeschlossen und soll aus der Tabelle *t_proj* gelöscht werden. Damit in der Tabelle *t_ma_proj* keine Datensätze mit ungültigen Verweisen auf dieses Projekt gespeichert bleiben, wird dem Fremdschlüssel die Option ON DELETE CASCADE hinzugefügt.

```

① DROP TABLE t_ma_proj;
CREATE TABLE t_ma_proj
    (ma_id INTEGER NOT NULL,
     proj_id INTEGER NOT NULL,
     FOREIGN KEY (ma_id) REFERENCES t_ma (id) ON DELETE CASCADE,
     FOREIGN KEY (proj_id) REFERENCES t_proj (id)
        ON DELETE CASCADE);
③ INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(2, 1);
INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(5, 1);
INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(8, 1);
INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(11, 1);
INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(14, 4);
INSERT INTO t_ma_proj (ma_id, proj_id) VALUES(18, 4);
④ DELETE FROM t_proj WHERE id = 4;

```

- ① Die bereits vorhandene Tabelle *t_ma_proj* wird gelöscht und nochmals neu erstellt.
- ② Mit der zusätzlichen Option ON DELETE CASCADE wird erreicht, dass alle Datensätze der Tabelle automatisch gelöscht werden, wenn in der referenzierten Tabelle ein verknüpfter Datensatz gelöscht wird.
- ③ In die Tabelle *t_ma_proj* werden verschiedene Datensätze eingefügt. Zwei Datensätze verweisen auf das Projekt mit der Id 4.
- ④ An dieser Stelle wird in der Tabelle *t_proj* das Projekt mit der Id 4 gelöscht. Alle Datensätze in der Tabelle *t_ma_proj*, die sich auf diesen Datensatz beziehen, werden ebenfalls gelöscht.

```

MariaDB [uebungen]> SELECT * FROM t_ma_proj;
+-----+-----+
| ma_id | proj_id |
+-----+-----+
| 2     | 1      |
| 5     | 1      |
| 8     | 1      |
| 11    | 1      |
| 14    | 4      |
| 18    | 4      |
+-----+-----+
6 rows in set (0.00 sec)

MariaDB [uebungen]> DELETE FROM t_proj WHERE id = 4;
Query OK, 1 row affected (0.00 sec)

MariaDB [uebungen]> SELECT * FROM t_ma_proj;
+-----+-----+
| ma_id | proj_id |
+-----+-----+
| 2     | 1      |
| 5     | 1      |
| 8     | 1      |
| 11    | 1      |
+-----+-----+
4 rows in set (0.00 sec)

```

Die verknüpften Datensätze der Tabelle t_ma_proj werden automatisch gelöscht

Fremdschlüssel nachträglich hinzufügen

Mithilfe der ALTER-TABLE-Anweisung können Sie einer Tabelle auch nachträglich Fremdschlüssel hinzufügen. Dies kann z. B. dann sinnvoll sein, wenn Sie die Tabelle mit den Fremdschlüsseln anlegen möchten, bevor Sie die referenzierte Tabelle erstellen.

Syntax

```
ALTER TABLE tabellename ADD FOREIGN KEY (datenfieldname)
REFERENCES tabellename (datenfieldname)
[ON UPDATE referenzoption] [ON DELETE referenzoption];
```

- ✓ Die Anweisung wird mit den Schlüsselwörtern **ALTER TABLE** eingeleitet. Danach folgt der Tabellennname.
- ✓ Die **ALTER-TABLE**-Anweisung erlaubt verschiedene Änderungen an der Tabellenstruktur. Mit der Angabe **ADD FOREIGN KEY** wird ein neuer Fremdschlüssel definiert. Danach folgt in runden Klammern der Name des Datenfeldes in der aktuellen Tabelle, das auf einen Primärschlüssel einer anderen Tabelle verweist.
- ✓ Danach folgen das Schlüsselwort **REFERENCES** und der Name der Tabelle, auf die verwiesen wird. In runden Klammern folgt der Name des Primärschlüsselfeldes dieser Tabelle.
- ✓ Wenn benötigt, können danach verschiedene Optionen angegeben werden, die das Verhalten bei Fremdschlüsselverletzungen steuern.

8.3 Indizes

Indizes erstellen

Die Suche in großen Datenbeständen kann mit Indizes erheblich beschleunigt werden. Ein Index ähnelt dem Inhaltsverzeichnis eines Buches, in dem Suchbegriffe geordnet vorliegen und auf die entsprechenden Seiten des Buches verwiesen wird. Liegt ein Index für eine Tabelle vor, durchsucht das DBMS nicht mehr sequenziell alle Datensätze der Tabelle, sondern nur den kompakteren Index, und es wählt gezielt die passenden Datensätze aus. Indizes werden meist in gesonderten Dateien gespeichert. Mit der Anweisung **CREATE INDEX** wird ein Index erstellt.

Beispiel: *IndexErstellen.sql*

In zwei separaten Anweisungen werden eine Tabelle und ein Index für ein Feld dieser Tabelle definiert.

```
① CREATE TABLE t_produkt
    (id INTEGER NOT NULL, name VARCHAR(50),
     bemerk VARCHAR(100), datum DATE, PRIMARY KEY (id));
② INSERT INTO t_produkt ...
③ CREATE INDEX i_produkt_name ON t_produkt (name);
④ SELECT id, name FROM t_produkt WHERE name LIKE "S%"
    ORDER BY name;
```

- ① Die Tabelle wird über die CREATE-TABLE-Anweisung erstellt. Das Datenfeld *id* soll als Primärschlüssel eingesetzt werden.
- ② In die Tabelle *t_produkt* werden einige Datensätze eingefügt.
- ③ An dieser Stelle wird ein Index mit dem Namen *i_produkt_name* für das Feld *name* der Tabelle erstellt.
- ④ Wird in einer WHERE-Bedingung das Datenfeld *name* verwendet, kommt der Index automatisch zum Einsatz.

Bei PostgreSQL müssen statt Anführungszeichen Hochkommata verwendet werden (LIKE 'S%').

Der Indexname wird nur benötigt, um den Index zu einem späteren Zeitpunkt löschen zu können. Der Index wird von der Datenbank automatisch verwendet, wenn die Suche über ein oder mehrere mit einem Index versehene Felder verläuft. Ob ein Index bei der Ausführung einer Abfrage wirklich genutzt wird, kann nicht garantiert werden, da dies von der jeweiligen Abfrage-Engine unter anderem aufgrund von Statistiken entschieden wird. In der Regel wird jedoch auf den Index zurückgegriffen.

Beachten Sie, dass der Index richtungsabhängig ist. Standardmäßig wird der Index immer in aufsteigender Suchrichtung für das entsprechende Datenfeld erstellt. Einen absteigenden Index erreichen Sie mit der Anweisung CREATE DESC INDEX. Diese Anweisung wird von MariaDB jedoch nicht unterstützt.

Syntax

```
CREATE INDEX indexname ON tabellenname (datenfeldname [ASC|DESC] );
```

- ✓ Die Anweisung zum Erstellen eines neuen Index beginnt mit CREATE INDEX.
- ✓ Danach folgt der gewünschte Indexname. Dieser Name wird nur zur internen Verwaltung der Indizes verwendet. Über den Indexnamen kann ein Index später wieder gelöscht werden.
- ✓ Nach dem Schlüsselwort ON werden der Name der Tabelle sowie in runden Klammern der Name des Datenfelds angegeben, für das der Index erzeugt werden soll. Falls ein absteigender Index gewünscht wird, kann die Anweisung um das Schlüsselwort DESC erweitert werden.

Über die Anweisung SHOW INDEX FROM table können Sie sich alle definierten Indizes der Datenbank anzeigen lassen. In der Konsole von PostgreSQL können Sie die Indizes mit dem Befehl "\di" anzeigen lassen.

Richtlinien für das Erstellen von Indizes

Es ist nicht zweckmäßig, für alle Datenfelder einer Tabelle Indizes zu erstellen. In manchen Fällen kann das Suchverhalten dadurch verlangsamt werden. Für das Erstellen der Indizes sollten Sie folgende Richtlinien beachten:

- ✓ Für Datenfelder, die an Primär-, Sekundär- oder Fremdschlüsseln beteiligt sind, werden von den meisten Datenbanksystemen automatisch Indizes erstellt. Sie benötigen daher keinen zusätzlichen Index.

- ✓ Für Datenfelder, deren Werte sich nur wenig unterscheiden (z. B. ein Feld zum Speichern der Anrede Herr und Frau), bringt ein Index keine Geschwindigkeitsvorteile.
- ✓ Bei Tabellen mit wenigen Daten bringt ein Index ebenfalls kaum Geschwindigkeitsvorteile.
- ✓ Erstellen Sie Indizes für Tabellen, in denen große Datenmengen gefiltert, gruppiert oder sortiert werden müssen.
- ✓ Für sich häufig ändernde Datenfelder sollten keine Indizes erstellt werden, da der Index mit jeder Datenaktualisierung ebenfalls aktualisiert werden muss.
- ✓ Indizes sind nur für sehr häufig in Abfragen benötigte Datenfelder von Vorteil. Für selten verwendete Felder wird nur unnötig Speicherplatz belegt und Rechenzeit zum Aktualisieren des Indexes in Anspruch genommen.
- ✓ Für Datenfelder, die auf- und absteigend sortiert werden müssen, müssen Sie zwei verschiedene Indizes erstellen.
- ✓ Aggregatfunktionen nutzen keinen Index, da z. B. zur Summenbildung alle Felder durchlaufen werden müssen.

Indizes löschen

Mit der DROP-INDEX-Anweisung können Sie einen bestehenden Index löschen.

Syntax

```
DROP INDEX indexname [ON tabellenname] ;
```

- ✓ Der Index wird mit der Anweisung `DROP INDEX` gelöscht. Dazu ist der Name des Indexes notwendig.
- ✓ Beim Datenbanksystem MariaDB ist die Angabe der zugehörigen Tabelle notwendig.

Wenn Sie eine sehr große Datenmenge in eine Tabelle einfügen oder sie daraus löschen wollen, können Sie die Indizes der Tabelle vorher löschen und nach dem betreffenden Vorgang wieder erstellen. Diese Vorgehensweise ist wesentlich schneller, als wenn bei jedem Einfüge- oder Löschvorgang der Index aktualisiert werden müsste.

8.4 Übung

Primär- und Fremdschlüssel erstellen

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*

1. Erstellen Sie eine neue Datenbank *Bibliothek*.

Wechseln Sie in diese Datenbank.

Erstellen Sie die Tabelle *t_buecher* mit folgenden Datenfeldern:

isbn, Text mit maximal 13 Zeichen
titel, *autor*, Text mit maximal 100 Zeichen
auflage, *preis*.

Wählen Sie geeignete Datentypen.

Definieren Sie das Feld *isbn* als Primärschlüssel.

Erstellen Sie die Tabelle *t_leser* mit folgenden Datenfeldern:

nr, *name*, *vname*, *gebdat* und *adr*.

Wählen Sie geeignete Datentypen.

Definieren Sie das Feld *nr* als Primärschlüssel.

Erstellen Sie eine weitere Tabelle *t_verleih*, die eine Verbindung zwischen den Lesern und den Büchern herstellt. Sie soll folgende Datenfelder enthalten:

isbn, *leser*, *datum*.

Wählen Sie geeignete Datentypen.

Erstellen Sie eine geeignete Fremdschlüsselbeziehung zu den anderen beiden Tabellen.

Beim Löschen eines Buches oder eines Lesers sollen auch alle zugehörigen Datensätze dieser Tabelle gelöscht werden.

2. Wechseln Sie zur Datenbank *Bibliothek*.

Definieren Sie einen Index für das Datenfeld *autor* der Tabelle *t_buecher*.

Erstellen Sie eine geeignete Abfrage, die den Index verwendet.

Definieren Sie einen weiteren Index für das Datenfeld *titel* der Tabelle *t_buecher*.

Nennen Sie eine Abfrage, bei der dieser Index verwendet wird.

Definieren Sie einen kombinierten Index für die Datenfelder *isbn* und *leser* der Tabelle *t_verleih*.

Wie lautet eine Abfrage, bei der der Index verwendet wird?

Erstellen Sie einen Index für die Suche im Feld *name* der Tabelle *t_leser*.

Lassen Sie sich alle Indizes anzeigen.

Löschen Sie den Index für die Suche im Feld *name* der Tabelle *t_leser*.

9

Funktionen in Abfragen

9.1 Standard-Funktionen in SQL

Standard- und Nichtstandard-Funktionen

Bei der Arbeit mit den Daten eines Datenbanksystems kann der Einsatz von mathematischen Funktionen oder Funktionen für Zeichenketten nützlich sein. Mit diesen Funktionen können Sie bereits bei der Abfrage von Daten bestimmte Bearbeitungen vornehmen. Oder Sie machen eine Bedingung in einer SELECT- oder UPDATE-Anweisung vom Resultat einer Funktion abhängig.

Die meisten Datenbanksysteme stellen eine weit größere Bandbreite an Funktionen für viele Einsatzbereiche zur Verfügung, als im ANSI-SQL-Standard verbindlich definiert ist. MariaDB und PostgreSQL verfügen beispielsweise über sehr viele integrierte Funktionen, die vom Datenbanksystem mit hoher Geschwindigkeit ausgeführt werden.

In diesem Kapitel werden nur Funktionen vorgestellt, die von den meisten Datenbanksystemen angeboten werden. Mehr Informationen zu den nicht standardisierten Funktionen finden Sie in den Anleitungen zu Ihrem Datenbanksystem.

Aggregatfunktionen

Im Standard-SQL sind nur die Aggregatfunktionen definiert. Sie dienen vor allem der statistischen Auswertung der gespeicherten Daten.

In vielen Fällen werden nicht die einzelnen Datensätze einer Abfrage benötigt, sondern beispielsweise nur eine Information über die Anzahl der ermittelten Datensätze oder über einen Minimal- bzw. Maximalwert. Für diesen Zweck existieren in SQL Aggregatfunktionen, die Berechnungen über alle oder ausgewählte Datensätze durchführen. In runden Klammern wird bei den Aggregatfunktionen das gewünschte Datenfeld angegeben, auf das die Funktion angewendet werden soll.

In einigen Datenbanksystemen kann das Durchlaufen aller Datensätze großer Tabellen trotz eines definierten Indexes sehr lange dauern, da auf jeden Datensatz zugegriffen werden muss. In diesem Fall sollten Sie statt des Einsatzes einer Aggregatfunktion die Verwendung von Triggern vorziehen.

Beispiel: Aggregatfunktionen1.sql

Die folgenden Abfragen verwenden verschiedene Aggregatfunktionen.

```

① SELECT COUNT(id) AS Anzahl, AVG(preis) AS "Preis Durchschnitt",
       MIN(preis) AS "Preis Minimum",
       MAX(preis) AS "Preis Maximum" FROM t_lager;
② SELECT COUNT(id) AS Anzahl, AVG(preis) AS "Preis Durchschnitt"
       FROM t_lager WHERE stueck > 100;
```

- ① Mithilfe der verschiedenen Aggregatfunktionen werden die Anzahl der Artikel, der Durchschnittspreis, der minimale und der maximale Preis ermittelt.

```

Sie sind jetzt verbunden mit der Datenbank »uebungen« als Benutzer »postgres«.
uebungen=# SELECT COUNT(id) AS Anzahl, AVG(preis) AS "Preis Durchschnitt",
uebungen=#      MIN(preis) AS "Preis Minimum", MAX(preis) AS "Preis Maximum" FROM t_lager;
anzahl | Preis Durchschnitt | Preis Minimum | Preis Maximum
-----+-----+-----+-----+
 11   | 15.09090909090909 |          2 |        35
(1 Zeile)
```

Statistische Daten über die Tabelle t_lager in PostgreSQL

- ② Die angegebenen Aggregatfunktionen ermitteln die Anzahl der Artikel und den Durchschnittspreis der Produkte, von denen sich mehr als 100 Stück im Lager befinden.

```

uebungen=# SELECT COUNT(id) AS Anzahl, AVG(preis) AS "Preis Durchschnitt" FROM t_lager
uebungen=# WHERE stueck > 100;
anzahl | Preis Durchschnitt
-----+-----
  6   | 10.1666666666666667
(1 Zeile)
```

Statistische Berechnungen über einen Teil der Datensätze der Tabelle t_lager in PostgreSQL

Die folgenden Aggregatfunktionen stehen zur Verfügung:

Funktion	Erklärung	Beispiel
COUNT()	Liefert die Anzahl der Werte in der Ergebnismenge einer SELECT-Abfrage bzw. einer Gruppierung	<code>SELECT ma_id, COUNT(proj_id) FROM t_ma_proj GROUP BY ma_id;</code>
COUNT(DISTINCT)	Liefert die Anzahl der unterschiedlichen Werte in einer Abfrage oder Gruppierung	<code>SELECT COUNT(DISTINCT plz) FROM t_ma;</code>
AVG()	Liefert den Durchschnittswert eines Datenfeldes der Abfrage oder Gruppierung	<code>SELECT AVG(stueck) FROM t_lager;</code>
MIN() MAX()	Liefert den kleinsten bzw. größten Wert eines Datenfeldes der Abfrage oder Gruppierung	<code>SELECT MIN(preis) FROM t_lager;</code>
SUM()	Liefert die Summe der Werte eines Datenfeldes in der Abfrage oder Gruppierung	<code>SELECT SUM(preis) FROM t_lager WHERE stueck>10;</code>



In Abfragen können entweder nur Aggregatfunktionen oder nur Datenfelder angegeben werden, eine Mischung wie in dem folgenden Beispiel verursacht eine Fehlermeldung. Eine Ausnahme stellen gruppierte Abfragen dar.

```
SELECT ort AS Wohnort, COUNT(name) FROM t_ma;
```

Diese Anweisung enthält ein Datenfeld und eine Aggregatfunktion und verursacht somit eine Fehlermeldung.

```
uebungen=# SELECT ort AS Wohnort, COUNT(name) FROM t_ma;
FEHLER: Spalte »t_ma.ort« muss in der GROUP-BY-Klausel erscheinen oder in einer Aggregatfunktion verwendet werden
ZEILE 1: SELECT ort AS Wohnort, COUNT(name) FROM t_ma;
^
```

Fehlermeldung in PostgreSQL

Wenn Sie in einer SELECT-Abfrage eine GROUP-BY-Anweisung angeben, gruppieren Sie die Datensätze der Tabelle anhand der Werte des angegebenen Datenfeldes. In gruppierten SELECT-Abfragen ist eine Kombination von Datenfeldern und Aggregatfunktionen möglich. Die Aggregatfunktionen werden jeweils auf diese Gruppen angewendet. Fehlt die GROUP-BY-Anweisung, beziehen sich die Funktionen auf alle Datensätze der Abfrage.

Beispiel: Aggregatfunktionen2.sql

In den folgenden drei SELECT-Anweisungen wird die Wirkung der Aggregatfunktion COUNT() mit und ohne GROUP-BY-Anweisung gezeigt.

```
① SELECT * FROM t_ma_proj;
② SELECT COUNT(ma_id) FROM t_ma_proj;
③ SELECT proj_id, COUNT(ma_id) FROM t_ma_proj GROUP BY proj_id;
```

- ① Mit dieser einfachen SQL-Anweisung werden alle Datensätze der Tabelle zurückgeliefert.
- ② Durch Verwenden der Funktion COUNT() ohne GROUP-BY-Anweisung wird die Anzahl der Datensätze in der Tabelle ermittelt.
- ③ Die Tabelle wird mithilfe von GROUP BY nach dem Datenfeld proj_id gruppiert. Für jede dieser Gruppen wird die Anzahl der Datensätze und damit die Anzahl der Mitarbeiter pro Projekt ermittelt.

```
MariaDB [uebungen]> SELECT COUNT(ma_id) FROM t_ma_proj;
+-----+
| COUNT(ma_id) |
+-----+
|       6      |
+-----+
1 row in set (0.00 sec)
```

Ermitteln der Anzahl der Datensätze
in der Tabelle

```
MariaDB [uebungen]> SELECT proj_id,COUNT(ma_id) FROM t_ma_proj
-> GROUP BY proj_id;
+-----+-----+
| proj_id | COUNT(ma_id) |
+-----+-----+
|      1   |        4      |
|      2   |        2      |
+-----+-----+
2 rows in set (0.00 sec)
```

Ermitteln der Anzahl der Mitarbeiter pro Projekt

Aggregatfunktionen in Bedingungen

Aggregatfunktionen können nicht in einer WHERE-Klausel verwendet werden. Eine Auswahl der Datensätze über das Ergebnis einer Aggregatfunktion ist jedoch mithilfe der HAVING-Klausel möglich.

Beispiel: Aggregatfunktionen3.sql

Die gruppierte Abfrage wird durch eine HAVING-Klausel eingeschränkt.

```
① SELECT proj_id, COUNT(ma_id) FROM t_ma_proj
      GROUP BY proj_id HAVING COUNT(ma_id)>2;
```

- ① Durch das Hinzufügen der HAVING-Klausel werden nur die Projekte angezeigt, in denen mehr als zwei Mitarbeiter tätig sind.

Die HAVING-Klausel wird im Gegensatz zur WHERE-Klausel von den meisten Datenbanksystemen nicht optimiert und führt daher zu einer längeren Ausführungszeit der Abfrage. Sie sollten die HAVING-Klausel nur dann verwenden, wenn Sie Aggregatfunktionen oder Ersatznamen in der Bedingung benötigen.

Filtern von zu aggregierenden Datensätzen

Der im SQL-Standard definierte Zusatz FILTER ermöglicht, die in einer Aggregation auszuwertenden Datensätze nach einem logischen Kriterium zu filtern. PostgreSQL unterstützt diese Möglichkeit seit der Version 9.4.

In MariaDB ist die Anwendung derzeit noch nicht möglich.

Beispiel: Aggregatfunktionen4.sql

Durch den Einsatz des Filters werden in einer Abfrage sowohl die Anzahl aller als auch der Artikel, von denen mehr als 200 Stück vorrätig sind, ermittelt.

```
① SELECT COUNT(id) AS Anzahl FROM t_lager;
② SELECT COUNT(id) AS Anzahl FROM t_lager WHERE stueck > 200;
③ SELECT COUNT(id) AS Anzahl, COUNT(id) FILTER (WHERE stueck>200)
      AS AnzahlGrößer200 FROM t_lager;
```

- ① Über die Aggregatfunktion COUNT wird die Anzahl der Datensätze ermittelt. Mittels AS wird eine Spaltenüberschrift definiert.
- ② Die WHERE Bedingung schränkt die Menge der berücksichtigten Datensätze insgesamt ein. Es werden nur die Datensätze gezählt, von denen mehr als 200 Stück auf Lager sind.
- ③ Der Zusatz FILTER ermöglicht die Ermittlung beider Zahlen in einer Abfrage. Er wird nach der Aggregatfunktion unter Angabe der Filterbedingung mittels WHERE angegeben. Danach steht der anzugezeigende Spaltenname.

```

uebungen=# SELECT COUNT(id) AS Anzahl from t_lager;
          ^
anzahl
-----
 16
(1 Zeile)

uebungen=# SELECT COUNT(id) AS Anzahl from t_lager WHERE stueck > 200;
          ^
anzahl
-----
   6
(1 Zeile)

uebungen=# SELECT COUNT(id) AS Anzahl, COUNT(id) FILTER (WHERE stueck>200) AS AnzahlGrößer200 from t_lager;
          +-----+
anzahl | anzahlgrößer200
-----+-----+
 16    |      6
(1 Zeile)

uebungen=#

```

9.2 Nicht standardisierte Funktionen

Funktionen verwenden

Um das Ergebnis einer Funktion zu erhalten, müssen Sie eine `SELECT`-Abfrage verwenden. Im einfachsten Fall wird dabei der Funktionsaufruf nach dem Schlüsselwort `SELECT` angegeben.

Soll sich die Funktion auf den Datenbestand einer Tabelle beziehen, ist die Angabe der `FROM`-Klausel notwendig. Der Funktionsaufruf erfolgt im Bereich der Datenfeldliste zwischen den Schlüsselwörtern `SELECT` und `FROM`.

Als Argument der Funktion können Sie verwenden:

- ✓ Datenfelder der Tabelle bzw. der Tabellen, auf die sich die Abfrage bezieht
 - ✓ Konstante Werte
 - ✓ Ergebnisse aus Berechnungen oder Ergebnisse anderer Funktionen (Schachtelung ist möglich)
 - ✓ Sie können die in diesem Abschnitt erläuterten Funktionen im Gegensatz zu Aggregatfunktionen auch in `WHERE`-Bedingungen anwenden. Dazu verknüpfen Sie das Ergebnis einer Funktion durch Vergleichsoperatoren mit einem konstanten Wert oder einem Datenfeld.
 - ✓ Einige Funktionen können auch in anderen Anweisungen eingesetzt werden, wie beispielsweise als Wert in der `VALUES`-Klausel der `INSERT`-Anweisung.
- Zum Beispiel: `INSERT INTO ... VALUES (rand(), ...);`
- ✓ `SELECT`-Anweisungen ohne die `FROM`-Klausel werden nicht von allen DBMS akzeptiert.

Beispiel: *FunktionenVerwenden.sql*

In den drei folgenden Abfragen werden verschiedene Funktionen verwendet.

- | | |
|---|--|
| ① | <code>SELECT mod(24, 5);</code> |
| ② | <code>SELECT id, round(preis) FROM t_lager WHERE stueck > 200;</code> |
| ③ | <code>SELECT name, vname FROM t_ma WHERE lower(name) = "meyer";</code> |

- ① Die einfachste Form zum Ausführen einer Funktion ist der Aufruf mithilfe der SELECT-Anweisung ohne die Angabe einer Tabelle.
- ② Die Funktion `round` liefert in dieser Abfrage den gerundeten Preis der Artikel im Lager. Mit der Bedingung wird die Abfrage auf Artikel beschränkt, von denen mehr als zweihundert Stück im Lager sind.
- ③ Hier wird die Funktion in einer Bedingung angewendet. Die Abfrage liefert alle Datensätze der Tabelle, bei denen der Name *Meyer* lautet. Damit der Vergleich nicht von der Schreibweise mit Groß- oder Kleinbuchstaben abhängig ist, wird der Inhalt des Feldes `name` zuerst mithilfe der Funktion `lower` in Kleinbuchstaben umgewandelt.

```

MariaDB [uebungen]> SELECT mod(24,5);
+-----+
| mod(24,5) |
+-----+
|        4   |
+-----+
1 row in set (0.00 sec)

MariaDB [uebungen]> SELECT id, round(preis) FROM t_lager WHERE stueck > 200;
+----+-----+
| id | round(preis) |
+----+-----+
| 22 |          22  |
| 46 |          12  |
| 48 |           6  |
+----+-----+
3 rows in set (0.00 sec)

MariaDB [uebungen]> SELECT name, vname FROM t_ma WHERE lower(name) = "meyer";
+-----+-----+
| name | vname  |
+-----+-----+
| Meyer | Matthias |
| Meyer | Peter   |
+-----+-----+
2 rows in set (0.00 sec)

```

Einsatz von Funktionen

Mathematische Funktionen

Alle Funktionen, die das Datenbanksystem zur Verfügung stellt, dürfen nur im Bereich der Datenfeldliste einer SELECT-Abfrage verwendet werden. Die verschiedenen mathematischen Funktionen können Sie auf numerische Datenfelder, z. B. auf den INTEGER- oder FLOAT-Datentyp, anwenden.

Funktion	Erklärung	Beispiel
<code>abs(zahl)</code>	Gibt den absoluten Wert einer Zahl wieder, d. h. den Wert ohne Vorzeichen	<pre> MariaDB [uebungen]> SELECT abs(-4.4); +-----+ abs(-4.4) +-----+ 4.4 +-----+ </pre>
<code>ceiling(zahl)</code> MariaDB <code>ceil(Zahl)</code> PostgreSQL	Rundet den angegebenen Wert auf die nächste größere ganze Zahl	<pre> MariaDB [uebungen]> SELECT ceiling(100.6); +-----+ ceiling(100.6) +-----+ 101 +-----+ </pre>
<code>floor(zahl)</code>	Rundet den angegebenen Wert auf die nächste kleinere ganze Zahl	<pre> MariaDB [uebungen]> SELECT floor (100.6); +-----+ floor (100.6) +-----+ 100 +-----+ </pre>
<code>round(zahl)</code>	Rundet die angegebene Dezimalzahl nach den üblichen mathematischen Regeln auf eine ganze Zahl	<pre> MariaDB [uebungen]> SELECT round (3.45); +-----+ round (3.45) +-----+ 3 +-----+ </pre>
<code>round(zahl, stellen)</code>	Rundet die angegebene Dezimalzahl nach den üblichen mathematischen Regeln auf die angegebene Anzahl Dezimalstellen.	<pre> MariaDB [uebungen]> SELECT round (24.457,2); +-----+ round (24.457,2) +-----+ 24.46 +-----+ </pre>

Funktion	Erklärung	Beispiel
<code>log(zahl)</code> MariaDB <code>ln(zahl)</code> PostgreSQL	Ermittelt den natürlichen Logarithmus	MariaDB [uebungen]> SELECT log(16); +-----+ log (16) +-----+ 2.772588722239781 +-----+
<code>mod(zahl1, zahl2)</code>	Liefert den Rest der Ganzahldivision von <code>zahl1</code> durch <code>zahl2</code>	MariaDB [uebungen]> SELECT mod(20,6); +-----+ mod(20,6) +-----+ 2 +-----+
<code>pi()</code>	Liefert den Wert von PI	MariaDB [uebungen]> SELECT pi(); +-----+ pi() +-----+ 3.141593 +-----+
<code>rand()</code> MariaDB <code>random()</code> PostgreSQL	Liefert eine zufällige Zahl im Bereich 0 bis 1	MariaDB [uebungen]> SELECT rand(); +-----+ rand() +-----+ 0.6502966532952121 +-----+
<code>sign(zahl)</code>	Ermittelt das Vorzeichen einer Zahl und liefert daraufhin den Wert -1 bzw. 0 oder 1	MariaDB [uebungen]> SELECT sign(-23.5); +-----+ sign(-23.5) +-----+ -1 +-----+
<code>sin(zahl)</code> <code>cos(zahl)</code> <code>tan(zahl)</code>	Berechnet die entsprechenden Winkelfunktionen für den angegebenen Wert	MariaDB [uebungen]> SELECT sin(0.5); +-----+ sin(0.5) +-----+ 0.479425538604203 +-----+
<code>sqrt(zahl)</code>	Ermittelt die Quadratwurzel für die übergebene Zahl	MariaDB [uebungen]> SELECT sqrt(16); +-----+ sqrt(16) +-----+ 4 +-----+

Viele Datenbanksysteme stellen noch weitere mathematische Funktionen zur Verfügung, beispielsweise zum Berechnen weiterer Winkelfunktionen.

- ✓ Sie können mehrere Funktionen über mathematische Operatoren miteinander verbinden, um komplexere Berechnungen auszuführen. Das Argument einer Funktion kann dabei auch ein numerisches Datenfeld oder das Ergebnis einer Berechnung sein.
- ✓ Um ganzzahlige Zufallszahlen im Bereich von 1 bis 1000 zu erzeugen, können Sie den Aufruf `round(rand() * 1000) + 1` verwenden.

(Bei PostgreSQL heißt der Ausdruck `round(random() * 1000) + 1`.)

Funktionen für Zeichenketten

Die folgenden Funktionen dienen zur Verarbeitung von Textwerten und Zeichenketten (engl. strings), z. B. von Datenfeldern des Typs VARCHAR. Sie können in der Regel bis zu 32767 Zeichen verarbeiten.

Manche Funktionen benötigen als Argument die Position (den Index) eines Zeichens innerhalb der Zeichenkette, mit dem begonnen werden soll. Das erste Zeichen einer Zeichenkette besitzt die Position 1.

Überschrift	Überschrift	Beispiel
ascii(string)	Liefert den ASCII-Wert des ersten Zeichens der Zeichenkette	<pre>MariaDB [uebungen]> SELECT ascii("T"); +-----+ ascii("T") +-----+ 84 +-----+</pre>
char(wert) MariaDB chr(wert) PostgreSQL	Wandelt den angegebenen ASCII-Wert in das entsprechende Zeichen um	<pre>MariaDB [uebungen]> SELECT char("84"); +-----+ char("84") +-----+ T +-----+</pre>
length(string)	Ermittelt die Länge der Zeichenkette	<pre>MariaDB [uebungen]> SELECT length("Guten Tag"); +-----+ length("Guten Tag") +-----+ 9 +-----+</pre>
lower(string) upper(string)	Wandelt die angegebene Zeichenkette in Klein- bzw. Großbuchstaben um	<pre>MariaDB [uebungen]> SELECT lower("Guten Tag"); +-----+ lower("Guten Tag") +-----+ guten tag +-----+</pre>
ltrim(string) rtrim(string)	Entfernt alle führenden bzw. alle abschließenden Leerzeichen in der Zeichenkette	<pre>MariaDB [uebungen]> SELECT ltrim(" Guten Tag"); +-----+ ltrim(" Guten Tag") +-----+ Guten Tag +-----+</pre>
substring(string, anfang ,länge) substr(string, anfang, ende)	Liefert ein Teilstück der angegebenen Zeichenkette, definiert durch Anfangsposition und Länge bzw. Endposition	<pre>MariaDB [uebungen]> SELECT substring("Einen guten Tag", 7, 3); +-----+ substring("Einen guten Tag", 7, 3) +-----+ gut +-----+</pre>

9.3 Übung

Mit Funktionen arbeiten

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*

1. Wechseln Sie zur Datenbank *Bibliothek*.

Erstellen Sie jeweils eine Abfrage, um die Anzahl der Datensätze in der Tabelle *t_buecher* und der Tabelle *t_leser* zu ermitteln.

Gruppieren Sie eine Abfrage über die Tabelle *t_verleih* so, dass ihnen angezeigt wird, wie viele Bücher die einzelnen Leser ausgeliehen haben. Lassen Sie die Lesernummer und die Anzahl der Bücher ausgeben.

Sortieren Sie die Ausgabe absteigend nach der Lesernummer.

Beschränken Sie die Abfrage auf die Leser, die mehr als ein Buch ausgeliehen haben.

Lassen Sie sich die Namen der Leser und die Anzahl der Zeichen der Namen ausgeben.

Ermitteln Sie die ISBN-Nummern und den aufgerundeten Preis der vorhandenen Bücher.

2. Erzeugen Sie eine neue Tabelle *t_mess*, in der Messwerte erfasst werden sollen.

Die Tabelle soll die Felder *nr* und *wert* haben. Das Feld *nr* soll ein Selbstzählfeld sein.

Füllen Sie die Tabelle mit 11 Datensätzen. Für die Messwerte (Spalte *wert*) sind dabei Zufallszahlen zwischen 1 und 1000 zu generieren.

Erstellen Sie eine Abfrage, die folgende statistische Berechnungen über die Spalte *wert* durchführt: Anzahl, nach mathematischen Regeln gerundeter Durchschnittswert, Minimum und Maximum.

Legen Sie aussagekräftige Spaltenüberschriften fest.

10

Datenabfragen für mehrere Tabellen

10.1 Tabellen verknüpfen

Datenbestände in mehreren Tabellen

Bei relationalen Datenbanken werden die Daten in der Regel auf mehrere, logisch zusammengehörige Tabellen verteilt. In einer Abfrage müssen diese Daten so zusammengefügt werden, dass eine brauchbare Ergebnismenge entsteht. Eine wichtige Rolle spielen dabei Primär- und Fremdschlüssel, die die Verbindung von Datensätzen in mehreren Tabellen vereinfachen. Die Verknüpfung von mehreren Tabellen miteinander ist ein wichtiger Aspekt des relationalen Datenmodells.

Die SELECT-Abfrage kann auf verschiedene Weise erweitert werden, um den Zugriff auf mehrere Tabellen gleichzeitig zu ermöglichen. Die Abfragen können sich auf zwei oder auch mehr Tabellen beziehen.

Verknüpfung von Tabellen über Mengenoperationen

Die Datenbestände in den einzelnen Tabellen stellen Mengen von Datensätzen (Tupeln) dar. Durch SQL-Anweisungen können diese unterschiedlich miteinander verknüpft werden. Voraussetzung für die Ausführung einer Mengenoperation ist, dass beide Tabellen vereinigungsverträglich sind. Das bedeutet, beide Tabellen müssen die gleiche Struktur besitzen, Feldnamen und Wertebereiche müssen übereinstimmen. Wenn Sie zwei oder mehr vereinigungsverträgliche Mengen addieren, erhalten Sie einen Datenbestand, der alle Datensätze aus allen Tabellen enthält. Sie können auch Schnittmengen bilden, um nur solche Datensätze zu ermitteln, die in allen beteiligten Tabellen vorhanden sind. Folgende drei Mengenoperationen werden beim Verknüpfen von Tabellen am häufigsten eingesetzt:

Vereinigungs-menge 	Verbindung von zwei oder mehr Tabellen zu einem Datenbestand Beispiel: <i>Alle Datensätze aus der Projekttabelle von Frankfurt und aus der Projekttabelle von Leipzig werden zusammengefasst.</i>
Schnittmenge 	Aus zwei oder mehr Mengen werden nur die Datensätze herausgesucht, die in allen Mengen gleich sind. Die Schnittmenge entspricht dem Durchschnitt der Relationenalgebra. Beispiel: <i>Es werden die Datensätze aus der Projekttabelle von Frankfurt und aus der Projekttabelle von Berlin herausgesucht, die gleich sind (nur die Projekte, an denen sowohl in Frankfurt als auch in Berlin gearbeitet wird).</i>

Differenzmenge 	Aus zwei oder mehr Mengen werden alle die Datensätze ermittelt, die zwar in der einen Menge, jedoch nicht in der anderen Menge enthalten sind. Beispiel: <i>Die Datensätze aus der Projekttabelle von Frankfurt, die nicht in der Projekttabelle von Leipzig vorhanden sind (nur die Projekte, an denen nur in Frankfurt gearbeitet wird)</i>
--	--

Verbund von Tabellen (Joins)

Der Verbund mehrerer Tabellen wird als **Join** (engl. verbinden, vereinigen) bezeichnet. Über Joins werden die Datensätze aus zwei oder mehreren Tabellen kombiniert. Der umfangreichste Join ist das kartesische Produkt zweier Tabellen (auch als Cross-Join bezeichnet). Da dieses aber selten ein sinnvolles Ergebnis liefert, erfolgt in den verschiedenen Joins eine Einschränkung der ausgewählten Datensätze. Dazu werden jeweils ein oder mehrere Felder beider Tabellen miteinander verglichen. Die Datensätze, die der Vergleichsbedingung genügen, werden in die Ergebnistabelle aufgenommen. Es gibt verschiedene Arten von Joins, die sich durch die Kriterien unterscheiden, nach denen sie Spalten und Datensätze in die Ergebnismenge (-tabelle) übernehmen.

Als Beispiel dienen die folgenden Tabellenausschnitte, die durch die verschiedenen Joins verbunden werden.

ID	Name	Vorname	Nr.
1	Naumann	Steve	1
2	Baumann	Janine	1
3	Hartmann	Heike	7
4	Naumann	Jens	11

Tabelle A

Nr	Abteilung
1	Einkauf
7	Personal
10	EDV

Tabelle B

Cross-Join	Mit diesem Verbund wird das kartesische Produkt beider Tabellen gebildet. Das heißt, jeder Datensatz der einen Tabelle wird mit jedem Datensatz der anderen Tabelle kombiniert.																																				
Theta-Join	<p>Es werden bestimmte Datensätze aus dem kartesischen Produkt zweier Tabellen durch eine Bedingung ausgewählt. In dieser Bedingung wird eine Spalte aus der einen und eine Spalte aus der anderen Tabelle über eine logische Operation verglichen, z. B. $A.Nr < B.Nr$. Wird auf Gleichheit geprüft, so erhalten Sie eine Spezialform des Theta-Join, den Equi-Join.</p> <p>Für das Beispiel $A.Nr < B.Nr$ erhalten Sie die folgende Ergebnistabelle:</p> <table border="1"> <thead> <tr> <th>ID</th> <th>Name</th> <th>Vorname</th> <th>Nr</th> <th>Nr</th> <th>Abteilung</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Naumann</td> <td>Steve</td> <td>1</td> <td>7</td> <td>Personal</td> </tr> <tr> <td>1</td> <td>Naumann</td> <td>Steve</td> <td>1</td> <td>10</td> <td>EDV</td> </tr> <tr> <td>2</td> <td>Baumann</td> <td>Janine</td> <td>1</td> <td>7</td> <td>Personal</td> </tr> <tr> <td>2</td> <td>Baumann</td> <td>Janine</td> <td>1</td> <td>10</td> <td>EDV</td> </tr> <tr> <td>3</td> <td>Hartmann</td> <td>Heike</td> <td>7</td> <td>10</td> <td>EDV</td> </tr> </tbody> </table>	ID	Name	Vorname	Nr	Nr	Abteilung	1	Naumann	Steve	1	7	Personal	1	Naumann	Steve	1	10	EDV	2	Baumann	Janine	1	7	Personal	2	Baumann	Janine	1	10	EDV	3	Hartmann	Heike	7	10	EDV
ID	Name	Vorname	Nr	Nr	Abteilung																																
1	Naumann	Steve	1	7	Personal																																
1	Naumann	Steve	1	10	EDV																																
2	Baumann	Janine	1	7	Personal																																
2	Baumann	Janine	1	10	EDV																																
3	Hartmann	Heike	7	10	EDV																																

Inner-Join = Equi-Join	<p>Die Datensätze aus beiden Tabellen werden verbunden, wenn ein oder mehrere gemeinsame Felder den gleichen Wert haben (deshalb auch Equi-Join – äquivalent). Dieser Join ist der am häufigsten verwendete Verbund. Für das Beispiel A.Nr = B.Nr erhalten Sie die folgende Ergebnistabelle:</p> <table border="1"> <thead> <tr> <th>ID</th><th>Name</th><th>Vorname</th><th>Nr</th><th>Nr</th><th>Abteilung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Naumann</td><td>Steve</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>2</td><td>Baumann</td><td>Janine</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>3</td><td>Hartmann</td><td>Heike</td><td>7</td><td>7</td><td>Personal</td></tr> </tbody> </table>	ID	Name	Vorname	Nr	Nr	Abteilung	1	Naumann	Steve	1	1	Einkauf	2	Baumann	Janine	1	1	Einkauf	3	Hartmann	Heike	7	7	Personal												
ID	Name	Vorname	Nr	Nr	Abteilung																																
1	Naumann	Steve	1	1	Einkauf																																
2	Baumann	Janine	1	1	Einkauf																																
3	Hartmann	Heike	7	7	Personal																																
Natural-Join	<p>Der Natural-Join arbeitet wie der Inner-Join – mit dem Unterschied, dass in der Ergebnistabelle keine identischen Spalten enthalten sind.</p> <table border="1" data-bbox="473 658 1140 804"> <thead> <tr> <th>ID</th> <th>Name</th> <th>Vorname</th> <th>Nr</th> <th>Abteilung</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Naumann</td> <td>Steve</td> <td>1</td> <td>Einkauf</td> </tr> <tr> <td>2</td> <td>Baumann</td> <td>Janine</td> <td>1</td> <td>Einkauf</td> </tr> <tr> <td>3</td> <td>Hartmann</td> <td>Heike</td> <td>7</td> <td>Personal</td> </tr> </tbody> </table>	ID	Name	Vorname	Nr	Abteilung	1	Naumann	Steve	1	Einkauf	2	Baumann	Janine	1	Einkauf	3	Hartmann	Heike	7	Personal																
ID	Name	Vorname	Nr	Abteilung																																	
1	Naumann	Steve	1	Einkauf																																	
2	Baumann	Janine	1	Einkauf																																	
3	Hartmann	Heike	7	Personal																																	
Left-Outer-Join Linke Inklusions-Verknüpfung	<p>Von der ersten (linken) Tabelle werden alle Datensätze in die Ergebnismenge aufgenommen. Von der zweiten (rechten) Tabelle werden nur die dazugehörigen Datensätze übernommen. Die Felder der zweiten Tabelle bleiben leer, wenn kein passender Datensatz vorhanden ist.</p> <table border="1" data-bbox="473 979 1224 1170"> <thead> <tr> <th>ID</th><th>Name</th><th>Vorname</th><th>Nr</th><th>Nr</th><th>Abteilung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Naumann</td><td>Steve</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>2</td><td>Baumann</td><td>Janine</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>3</td><td>Hartmann</td><td>Heike</td><td>7</td><td>7</td><td>Personal</td></tr> <tr> <td>4</td><td>Naumann</td><td>Jens</td><td>11</td><td>NULL</td><td>NULL</td></tr> </tbody> </table>	ID	Name	Vorname	Nr	Nr	Abteilung	1	Naumann	Steve	1	1	Einkauf	2	Baumann	Janine	1	1	Einkauf	3	Hartmann	Heike	7	7	Personal	4	Naumann	Jens	11	NULL	NULL						
ID	Name	Vorname	Nr	Nr	Abteilung																																
1	Naumann	Steve	1	1	Einkauf																																
2	Baumann	Janine	1	1	Einkauf																																
3	Hartmann	Heike	7	7	Personal																																
4	Naumann	Jens	11	NULL	NULL																																
Right-Outer-Join Rechte Inklusions-Verknüpfung	<p>Von der zweiten (rechten) Tabelle werden alle Datensätze in die Ergebnismenge aufgenommen. Von der ersten (linken) Tabelle werden nur die dazugehörigen Datensätze übernommen. Die Felder der ersten Tabelle bleiben leer, wenn kein passender Datensatz vorhanden ist.</p> <table border="1" data-bbox="473 1349 1224 1540"> <thead> <tr> <th>ID</th><th>Name</th><th>Vorname</th><th>Nr</th><th>Nr</th><th>Abteilung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Naumann</td><td>Steve</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>2</td><td>Baumann</td><td>Janine</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>3</td><td>Hartmann</td><td>Heike</td><td>7</td><td>7</td><td>Personal</td></tr> <tr> <td>NULL</td><td>NULL</td><td>NULL</td><td>NULL</td><td>10</td><td>EDV</td></tr> </tbody> </table>	ID	Name	Vorname	Nr	Nr	Abteilung	1	Naumann	Steve	1	1	Einkauf	2	Baumann	Janine	1	1	Einkauf	3	Hartmann	Heike	7	7	Personal	NULL	NULL	NULL	NULL	10	EDV						
ID	Name	Vorname	Nr	Nr	Abteilung																																
1	Naumann	Steve	1	1	Einkauf																																
2	Baumann	Janine	1	1	Einkauf																																
3	Hartmann	Heike	7	7	Personal																																
NULL	NULL	NULL	NULL	10	EDV																																
Full-Outer-Join = Full-Join	<p>Der Full-Join ist eine Kombination aus dem Left-Outer-Join und dem Right-Outer-Join. Alle Datensätze beider Tabellen werden in die Ergebnismenge übernommen. Passen Datensätze aus beiden Tabellen laut Vergleichsoperation zusammen, so werden sie verbunden.</p> <table border="1" data-bbox="473 1709 1224 1922"> <thead> <tr> <th>ID</th><th>Name</th><th>Vorname</th><th>Nr</th><th>Nr</th><th>Abteilung</th></tr> </thead> <tbody> <tr> <td>1</td><td>Naumann</td><td>Steve</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>2</td><td>Baumann</td><td>Janine</td><td>1</td><td>1</td><td>Einkauf</td></tr> <tr> <td>3</td><td>Hartmann</td><td>Heike</td><td>7</td><td>7</td><td>Personal</td></tr> <tr> <td>4</td><td>Naumann</td><td>Jens</td><td>11</td><td>NULL</td><td>NULL</td></tr> <tr> <td>NULL</td><td>NULL</td><td>NULL</td><td>NULL</td><td>10</td><td>EDV</td></tr> </tbody> </table>	ID	Name	Vorname	Nr	Nr	Abteilung	1	Naumann	Steve	1	1	Einkauf	2	Baumann	Janine	1	1	Einkauf	3	Hartmann	Heike	7	7	Personal	4	Naumann	Jens	11	NULL	NULL	NULL	NULL	NULL	NULL	10	EDV
ID	Name	Vorname	Nr	Nr	Abteilung																																
1	Naumann	Steve	1	1	Einkauf																																
2	Baumann	Janine	1	1	Einkauf																																
3	Hartmann	Heike	7	7	Personal																																
4	Naumann	Jens	11	NULL	NULL																																
NULL	NULL	NULL	NULL	10	EDV																																

Semi-Join	<p>Um einen Semi-Join zu erhalten, werden zwei Tabellen über einen Natural-Join verbunden. Anschließend erfolgt eine Projektion auf die Spalten der ersten Tabelle.</p> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <thead> <tr> <th>ID</th><th>Name</th><th>Vorname</th><th>Nr</th></tr> </thead> <tbody> <tr> <td>1</td><td>Naumann</td><td>Steve</td><td>1</td></tr> <tr> <td>2</td><td>Baumann</td><td>Janine</td><td>1</td></tr> <tr> <td>3</td><td>Hartmann</td><td>Heike</td><td>7</td></tr> </tbody> </table>	ID	Name	Vorname	Nr	1	Naumann	Steve	1	2	Baumann	Janine	1	3	Hartmann	Heike	7																																		
ID	Name	Vorname	Nr																																																
1	Naumann	Steve	1																																																
2	Baumann	Janine	1																																																
3	Hartmann	Heike	7																																																
Self-Join	<p>Der Self-Join ist einer der zuvor genannten Joins, bei dem nicht zwei verschiedene Tabellen miteinander verbunden werden, sondern zweimal dieselbe.</p> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <thead> <tr> <th>ID</th> <th>Name</th> <th>Vorname</th> <th>Nr</th> <th>Chef</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Naumann</td> <td>Steve</td> <td>1</td> <td>NULL</td> </tr> <tr> <td>2</td> <td>Baumann</td> <td>Janine</td> <td>1</td> <td>1</td> </tr> <tr> <td>3</td> <td>Hartmann</td> <td>Heike</td> <td>7</td> <td>NULL</td> </tr> <tr> <td>4</td> <td>Naumann</td> <td>Jens</td> <td>11</td> <td>NULL</td> </tr> </tbody> </table> <p><i>Tabelle A um eine Spalte erweitert</i></p> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <thead> <tr> <th>ID</th> <th>Name</th> <th>Vorname</th> <th>Nr</th> <th>Chef</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Naumann</td> <td>Steve</td> <td>1</td> <td>NULL</td> </tr> <tr> <td>2</td> <td>Baumann</td> <td>Janine</td> <td>1</td> <td>Naumann</td> </tr> <tr> <td>3</td> <td>Hartmann</td> <td>Heike</td> <td>7</td> <td>NULL</td> </tr> <tr> <td>4</td> <td>Naumann</td> <td>Jens</td> <td>11</td> <td>NULL</td> </tr> </tbody> </table> <p><i>Self-Join auf Tabelle A</i></p>	ID	Name	Vorname	Nr	Chef	1	Naumann	Steve	1	NULL	2	Baumann	Janine	1	1	3	Hartmann	Heike	7	NULL	4	Naumann	Jens	11	NULL	ID	Name	Vorname	Nr	Chef	1	Naumann	Steve	1	NULL	2	Baumann	Janine	1	Naumann	3	Hartmann	Heike	7	NULL	4	Naumann	Jens	11	NULL
ID	Name	Vorname	Nr	Chef																																															
1	Naumann	Steve	1	NULL																																															
2	Baumann	Janine	1	1																																															
3	Hartmann	Heike	7	NULL																																															
4	Naumann	Jens	11	NULL																																															
ID	Name	Vorname	Nr	Chef																																															
1	Naumann	Steve	1	NULL																																															
2	Baumann	Janine	1	Naumann																																															
3	Hartmann	Heike	7	NULL																																															
4	Naumann	Jens	11	NULL																																															

In SQL2 (Entry Level) gibt es für einige der vorgestellten Joins Befehle. Joins, für die kein Befehl existiert, lassen sich über eine Kombination einer anderen Join-Art mit einer Selektion oder Projektion nachbilden.

10.2 Einfaches Verknüpfen von Tabellen

Mithilfe einer SELECT-Anweisung können Sie zwei oder mehr Tabellen verbinden, indem Felder aus mehreren Tabellen selektiert werden. Dazu lassen sich in der WHERE-Klausel Bedingungen für die Verknüpfung der beteiligten Tabellen angeben. Es werden jeweils zwei Felder aus unterschiedlichen Tabellen verglichen. Ist die Bedingung für zwei Datensätze der beteiligten Tabellen erfüllt, so werden diese miteinander verbunden.

Auf diese Weise können Sie verschiedene Arten von Joins realisieren. Wenn Sie in der WHERE-Bedingung keine Felder angeben, die miteinander verknüpft sind, erhalten Sie einen Cross-Join. Beim Cross-Join ist in der Ergebnismenge jeder Datensatz der einen Tabelle mit jedem Datensatz der anderen Tabelle verknüpft.

Beispiel: EinfacheVerknuepfung1.sql

In der Abfrage werden die zwei Tabellen *t_ma* und *t_abt* verknüpft, sodass ein einfacher Equi-Join entsteht (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, Stichwort 'Equi-Join'). Sie erhalten eine Ergebnismenge, die alle Datensätze beinhaltet, in denen die Felder *abtnr* und *id* der beiden Tabellen die gleichen Werte besitzen. Es werden somit bestimmte Felder der Tabellen selektiert.

```

① SELECT t_ma.vname, t_ma.name, t_ma.abtnr,
②   t_abt.* FROM t_ma, t_abt
③   WHERE t_ma.abtnr = t_abt.id;

```

- ① In der Datenfeldliste der SELECT-Anweisung werden die gewünschten Felder definiert. Damit die Zuordnung der Datenfelder zu der Tabelle eindeutig ist, wird der Name der Tabelle, gefolgt von einem Punkt, vorangestellt.
- ② Aus der Tabelle *t_abt* sollen alle Datenfelder angezeigt werden. Dazu kann auch hier der Platzhalter * verwendet werden. Im FROM-Bereich werden die beiden Tabellen angegeben.
- ③ In der WHERE-Bedingung wird der Zusammenhang zwischen den Tabellen definiert. In diesem Fall muss im Feld *abtnr* der Tabelle *t_ma* und im Feld *id* der Tabelle *t_abt* der gleiche Wert zu finden sein.

MariaDB [uebungen]> SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.* -> FROM t_ma, t_abt -> WHERE t_ma.abtnr = t_abt.id LIMIT 15;					
vname	name	abtnr	id	name	ort
Peter	Fuchs	3	3	Verkauf	Hamburg
Lilly	Baumann	6	6	Controlling	Bern
Norbert	Dorff	4	4	Produktion	Wien
Saskia	Bayerle	7	7	F&E	Zürich
Sebastian	Berger	1	1	Einkauf	Frankfurt
Karin	Kirsch	4	4	Produktion	Wien
Roland	Bergstein	3	3	Verkauf	Hamburg
Lisa	Schwondorf	4	4	Produktion	Wien
Johann	Luxemburg	3	3	Verkauf	Hamburg
Dilara	Ülkü	5	5	Abt_Organisation	Berlin
Annabell	Mannschatz	1	1	Einkauf	Frankfurt
Erwin	Nöller	3	3	Verkauf	Hamburg
Constantin	Brio	6	6	Controlling	Bern
Andreas	Eppel	4	4	Produktion	Wien
Andrea	Classmann	7	7	F&E	Zürich

Zuordnen von Abteilungen zu den Mitarbeitern über eine einfache Verknüpfung von zwei Tabellen

Syntax der SELECT-Anweisung für einen Equi-Join

```

SELECT datenfeldliste FROM tabellename1, tabellename2, ...
WHERE tabellename1.datenfeld = tabellename2.datenfeld [AND ...];

```

- ✓ Die Verknüpfung der Tabellen erfolgt in einer SELECT-Anweisung. In der Datenfeldliste werden die gewünschten Datenfelder bzw. der Platzhalter * angegeben. Falls die Namen der Felder nicht eindeutig sind (gleiche Feldnamen in mehreren Tabellen), muss der Tabellennamen mit einem Punkt vorangestellt werden.
- ✓ Nach dem Schlüsselwort FROM werden durch Kommata getrennt die Namen der Tabellen angegeben, die miteinander verknüpft werden sollen.
- ✓ In der WHERE-Bedingung müssen die übereinstimmenden Datenfelder der Tabellen angegeben werden.

Beispiele für Datenfeldlisten in Joins finden Sie in der folgenden Tabelle.

Beispiel	Erklärung
<code>SELECT * FROM tabelle1, tabelle2 ...</code>	Liefert alle Felder aus allen beteiligten Tabellen
<code>SELECT tabelle1.feld1, tabelle1.feld2, tabelle2.* ...</code>	Liefert die angegebenen Felder der Tabelle <i>tabelle1</i> und alle Felder der Tabelle <i>tabelle2</i>
<code>SELECT tabelle1.*, tabelle2.* ...</code>	Gleichbedeutend mit der Angabe <code>SELECT *</code>

Wenn der Name eines Datenfeldes in der Datenfeldliste oder der WHERE-Klausel eindeutig ist, weil er nur in einer Tabelle vorkommt, können Sie auf die Angabe des Tabellennamens verzichten.

Neben der Bedingung, unter welcher die Felder verknüpft werden, können in der WHERE-Klausel auch weitere Auswahlkriterien für die Abfrage angegeben werden.

Beispiel: *EinfacheVerknuepfung2.sql*

In diesem Beispiel werden zwei Tabellen verknüpft und die Datensätze durch eine zusätzliche Bedingung gefiltert.

```
① SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.*
     FROM t_ma, t_abt
② WHERE t_ma.abtnr = t_abt.id AND t_abt.ort = "Frankfurt";
```

- ① Die Abfrage wird wie im vorigen Beispiel als Equi-Join mit zwei beteiligten Tabellen gebildet.
- ② Neben der ersten Bedingung mit den übereinstimmenden Feldern der beiden Tabellen wird hier noch ein zusätzliches Auswahlkriterium angegeben. Damit sollen die Datensätze herausgefiltert werden, bei denen im Feld *ort* der Tabelle *t_abt* der Wert *Frankfurt* zu finden ist.

```
MariaDB [uebungen]> SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.*
->   FROM t_ma, t_abt
->   WHERE t_ma.abtnr = t_abt.id AND t_abt.ort = "Frankfurt";
+-----+-----+-----+-----+-----+
| vname | name | abtnr | id | name |
+-----+-----+-----+-----+-----+
| Sebastian | Berger | 1 | 1 | Einkauf |
| Annabell | Mannschatz | 1 | 1 | Einkauf |
| Peter | Meyer | 1 | 1 | Einkauf |
| Frank | Maier | 1 | 1 | Einkauf |
| Chris | Schmadtko | 1 | 1 | Einkauf |
| Annemarie | Segebrecht | 1 | 1 | Einkauf |
| Sophie | Brauer | 1 | 1 | Einkauf |
| Sabine | Carstedt | 1 | 1 | Einkauf |
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

Auswahl über zwei Tabellen mit einer zusätzlichen Bedingung

Die beiden verknüpften Datenfelder in der WHERE-Klausel können auch mit einem anderen Operator als dem Vergleichsoperator verbunden werden. In diesem Fall entsteht ein Theta-Join. So können Sie beispielsweise alle Datensätze ermitteln, die in dem verknüpften Feld nicht übereinstimmen oder in denen der Wert der ersten Spalte größer ist als der Wert der zweiten Spalte.

Ersatznamen für Tabellen definieren

Beim Verwenden von Verknüpfungen ist es an vielen Stellen notwendig, den Tabellennamen anzugeben. So müssen Sie z. B. die Datenfelder in der Datenfeldliste oder der WHERE-Klausel spezifizieren. Da dies besonders bei sehr langen Tabellennamen unübersichtlich wird, können Sie mit dem Schlüsselwort AS einen Ersatznamen definieren. Beim Self-Join ist dies sogar zwingend notwendig.

Beispiel: *VerknüpfungErsatznamen.sql*

```
① SELECT m.vname, m.name, m.abtnr, a.*  
      FROM t_ma AS m, t_abt AS a  
      WHERE m.abtnr = a.id;
```

- ① Mithilfe des Schlüsselwortes AS wird für die Tabelle *t_ma* der Name m und für die Tabelle *t_abt* der Name a definiert.
- ✓ Wenn Sie Ersatznamen für Tabellen definiert haben, müssen Sie in der Abfrage diese Namen in der Datenfeldliste, der WHERE-Bedingung und in weiteren Klauseln zwingend verwenden. Die richtigen Tabellennamen sind in dieser Abfrage ungültig.

Eine andere Möglichkeit, den Ersatznamen festzulegen, ist die, den Aliasnamen ohne das Schlüsselwort AS direkt hinter dem Tabellennamen anzugeben.

```
SELECT m.vname, m.name, m.abtnr, a.* FROM t_ma m, t_abt a  
WHERE m.abtnr = a.id;
```

10.3 Tabellen verknüpfen mit JOIN

Eine andere Möglichkeit, Tabellen miteinander zu verknüpfen, bietet die SELECT-Anweisung mit der JOIN-Klausel. Damit sind die meisten Verknüpfungsarten realisierbar. Die Datensätze stammen dabei aus zwei oder mehr Tabellen.

Da eine Verknüpfung mit JOIN übersichtlicher und weniger fehleranfällig ist, als eine einfache Verknüpfung und gleichzeitig das relationale Datenmodell besser widerspiegelt, sollten Sie diese der WHERE-Klausel vorziehen.

Cross-Join

Beim Cross-Join wird das kartesische Produkt aus den beiden Tabellen gebildet (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, Stichwort 'Kartesisches Produkt'). Häufig sind die Ergebnisse dieser Operation nicht sehr aussagekräftig. Werden die Datensätze aus den Tabellen aber durch eine WHERE-Klausel und Projektion eingeschränkt, lassen sich auch sinnvolle Ergebnismengen ermitteln.

Beispiel: CrossJoin.sql

Welche Mitarbeiter aus Bern könnten an welchen Projekten mitarbeiten?

```
① SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_proj.* FROM t_ma
② CROSS JOIN t_proj WHERE t_ma.ort = 'Bern';
```

- ① Die Felder *vname*, *name* und *abtnr* aus der Tabelle *t_ma* und alle Felder der Tabelle *t_proj* werden in der Ergebnismenge angezeigt.
- ② Die Tabelle *t_ma* und die Tabelle *t_proj* werden durch einen Cross-Join verknüpft. Durch die WHERE-Klausel wird die Ergebnismenge auf die Mitarbeiter aus Bern beschränkt.

MariaDB [uebungen]> SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_proj.* FROM t_ma -> CROSS JOIN t_proj WHERE t_ma.ort = 'Bern';						
vname	name	abtnr	id	name	beginn	ende
Constantin	Brio	6	1	Buchprojekt	2021-03-01	2021-08-20
Constantin	Brio	6	2	Renovierung des Pausenraumes	2021-02-01	2021-09-28
Constantin	Brio	6	3	Anlegen eines Pflichtenheftes	2021-02-10	2021-09-21
Steve	Walther	6	1	Buchprojekt	2021-03-01	2021-08-20
Steve	Walther	6	2	Renovierung des Pausenraumes	2021-02-01	2021-09-28
Steve	Walther	6	3	Anlegen eines Pflichtenheftes	2021-02-10	2021-09-21
Jacqueline	Seiler	5	1	Buchprojekt	2021-03-01	2021-08-20
Jacqueline	Seiler	5	2	Renovierung des Pausenraumes	2021-02-01	2021-09-28
Jacqueline	Seiler	5	3	Anlegen eines Pflichtenheftes	2021-02-10	2021-09-21

9 rows in set (0.051 sec)

Ergebnismenge des Full-Join

Syntax

```
SELECT datenfelder FROM tabelle1 CROSS JOIN tabelle2;
```

- ✓ Die Verknüpfung erfolgt in einer SELECT-Anweisung. Die Datenfelder, die in der Ergebnismenge enthalten sein sollen, werden in der Datenfeldliste angegeben. Auch der Platzhalter * kann angegeben werden.
- ✓ Nach dem Schlüsselwort FROM folgt der Name der ersten Tabelle. Nach der Angabe CROSS JOIN wird der Name der zweiten Tabelle angegeben.
- ✓ Zusätzlich können eine WHERE-Klausel und weitere Klauseln der SELECT-Anweisung verwendet werden (z. B. ORDER BY, GROUP BY).

Inner-Join (Equi-Join)

Die Verknüpfung von zwei oder mehr Tabellen über ein gemeinsames Feld (oder auch mehrere gemeinsame Felder) lässt sich in der SELECT-Anweisung über die Angabe der Schlüsselwörter INNER JOIN realisieren. Die Werte der gemeinsamen Felder müssen beim Equi-Join gleich sein, damit die Datensätze verbunden und in die Ergebnismenge aufgenommen werden.

Beispiel: *InnerJoin.sql*

Analog zu Beispiel *EinfacheVerknüpfung1.sql* sollen die zwei Tabellen *t_ma* und *t_abt* verknüpft werden, aber nun mithilfe der INNER-JOIN-Klausel.

```

① | SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.id, t_abt.name
② |   FROM t_ma INNER JOIN t_abt
③ |     ON t_ma.abtnr = t_abt.id
④ |   WHERE t_ma.ort = 'Zürich';

```

- ① In der Datenfeldliste der SELECT-Anweisung werden die gewünschten Felder aus den einzelnen Tabellen angegeben.
- ② Die Verknüpfung erfolgt über die Schlüsselwörter INNER JOIN. Dabei wird der Name der ersten Tabelle über diese Klausel mit dem Namen der zweiten Tabelle verknüpft.
- ③ Das übereinstimmende Feld beider Tabellen wird in diesem Fall nicht mehr in der WHERE-Klausel definiert. Stattdessen folgt die Klausel ON, die den Zusammenhang zwischen den Tabellen herstellt. Die Felder *abtnr* der Tabelle *t_ma* und *id* der Tabelle *t_abt* sind die gemeinsamen Felder oder Join-Felder der beiden Tabellen.
- ④ Um die Ergebnismenge einzuschränken, werden nur Mitarbeiter aus Zürich ausgewählt.

```

MariaDB [uebungen]> SELECT t_ma.vname, t_ma.name, t_ma.abtnr, t_abt.id, t_abt.name
->   FROM t_ma INNER JOIN t_abt
->     ON t_ma.abtnr = t_abt.id
->   WHERE t_ma.ort = 'Zürich';
+-----+-----+-----+-----+-----+
| vname | name | abtnr | id  | name |
+-----+-----+-----+-----+-----+
| Saskia | Bayerle |    7 | 7   | F&E
| Andrea | Classmann | 7 | 7   | F&E
| Gudrun | Wolff | 7 | 7   | F&E
| Björn | Holzhäußer | 2 | 2   | Marketing
| Matthias | Meyer | 6 | 6   | Controlling
| Sean | Conolly | 7 | 7   | F&E
| Barbara | Blücher | 3 | 3   | Verkauf
| Liane | Färber | 7 | 7   | F&E
| Lars | Trieschmann | 3 | 3   | Verkauf
| Gerlinde | Eberspächer | 7 | 7   | F&E
+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)

```

Erstellen eines Inner-Joins über zwei Tabellen mit der JOIN-Klausel

Syntax

```

SELECT datenfeldliste FROM tabellename1 INNER JOIN tabellename2
      ON tabellename1.datenfeld = tabellename2.datenfeld [WHERE ...];

```

- ✓ Die Verknüpfung erfolgt in einer SELECT-Anweisung. Die Datenfelder, die in der Ergebnismenge enthalten sein sollen, werden in der Datenfeldliste angegeben. Auch der Platzhalter * kann verwendet werden.
- ✓ Nach dem Schlüsselwort FROM folgt der Name der ersten Tabelle. Nach den Schlüsselwörtern INNER JOIN folgt der Name der zweiten Tabelle.
- ✓ Die zu vergleichenden Datenfelder beider Tabellen werden nach dem Schlüsselwort ON angegeben.
- ✓ Zusätzlich können eine WHERE-Klausel und weitere Klauseln der SELECT-Anweisung verwendet werden (z. B. ORDER BY, GROUP BY).

Inner-Joins über mehr als zwei Tabellen

Es besteht auch die Möglichkeit, mehr als zwei Tabellen über Joins miteinander zu verknüpfen. Dabei werden zunächst zwei Tabellen mit JOIN verknüpft, die eine (Zwischen-)Ergebnismenge erzeugen. Diese wird wiederum über JOIN mit einer weiteren Tabelle verknüpft usw.

Unter Verwendung des relationalen Datenmodells sind in der Praxis beispielsweise auch Joins über mehr als 10 Tabellen keine Seltenheit.

Beispiel: *InnerJoinMehrereTabellen.sql*

Es soll in einer Abfrage ermittelt werden, welche Mitarbeiter an welchen Projekten beteiligt sind. Die Namen der Mitarbeiter sind in der Tabelle *t_ma* gespeichert, die Namen der Projekte in der Tabelle *t_proj*.

Da ein Mitarbeiter in mehreren Projekten tätig sein kann, ist eine zusätzliche Tabelle *t_ma_proj* notwendig, um die Beziehungen zwischen den Projekten und den Mitarbeitern herzustellen. Um gleichzeitig Mitarbeitername und Projektname zu selektieren, ist eine Abfrage über drei Tabellen notwendig.

```
SELECT m.vname, m.name, p.name, p.ende
①   FROM t_ma_proj AS mp INNER JOIN t_ma AS m
②     ON mp.ma_id = m.id INNER JOIN t_proj AS p
③       ON mp.proj_id = p.id
      ORDER BY m.name;
```

- ① Hier wird der erste Inner-Join definiert. Für die Tabellennamen werden dabei Ersatznamen angegeben.
- ② Der nächste INNER JOIN-Befehl verknüpft die Ergebnismenge der beiden ersten Tabellen zusätzlich mit der Tabelle *t_proj*. Auch hier wird eine ON-Klausel angegeben.
- ③ Die Datensätze werden bei der Ausgabe nach dem Mitarbeiternamen sortiert.

```
MariaDB [uebungen]> SELECT m.vname, m.name, p.name, p.ende
->   FROM t_ma_proj AS mp INNER JOIN t_ma AS m
->     ON mp.ma_id = m.id INNER JOIN t_proj AS p ON mp.proj_id = p.id
->   ORDER BY p.name;
+-----+-----+-----+-----+
| vname | name | name | ende |
+-----+-----+-----+-----+
| Roland | Bergstein | Buchprojekt | 2021-08-20 |
| Saskia | Bayerle | Buchprojekt | 2021-08-20 |
| Lisa | Schwönsdorf | Buchprojekt | 2021-08-20 |
| Lilly | Baumann | Buchprojekt | 2021-08-20 |
+-----+-----+-----+-----+
4 rows in set (0.019 sec)
```

Das Ergebnis der verknüpften Abfrage über drei Tabellen

Bei sehr vielen verknüpften Tabellen können Sie zur besseren Übersicht die einzelnen Teile des Inner-Joins mit runden Klammern zusammenfassen. Dabei wird der Ausdruck

tabellename1 **INNER JOIN** tabellename2 **ON** bedingung

als ein Inner-Join in Klammern gesetzt.

Syntax

```
SELECT datenfelder FROM ((tabelle1 INNER JOIN tabelle2
ON bedingung) INNER JOIN tabelle3 ON bedingung) ...;
```

- ✓ Jeder Inner-Join erzeugt eine Datenmenge. Diese kann mit einer anderen Tabelle über einen weiteren Inner-Join und der zugehörigen ON-Klausel verknüpft werden kann.
- ✓ Die Klammern um die einzelnen Inner-Joins können entfallen. Sie dienen nur der besseren Übersichtlichkeit.

Natural-Join

Ein Inner-Join wird zum Natural-Join, wenn gleiche Spalten entfernt werden (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, Stichwort 'natürlicher Verbund'). Den Natural-Join gibt es auch als Left-Natural- und Right-Natural-Join. Eine WHERE- oder ON-Bedingung ist beim Natural-Join nicht vorhanden. Voraussetzung ist das Vorhandensein namensgleicher Spalten.

Die Syntax NATURAL JOIN wird nicht von allen Datenbanksystemen unterstützt. Sie können einen Natural-Join auf einfache Art erzeugen, indem Sie einen Inner-Join verwenden und eine Projektion hinzufügen.

Dem Vorteil der Methode – dem einfachen Aufbau der Bedingung – stehen auch Nachteile gegenüber. Zum einen ist die Abfrage langsamer, zum anderen ist die Ausgabe des Ergebnisses aufgrund der nicht zuordenbaren Spaltenbezeichnung des Vergleichsfeldes oft unklarer.

Beispiel: *NaturalJoin.sql*

①	<code>SELECT * FROM t_m_p INNER JOIN t_proj ON t_m_p.id = t_proj.id;</code>
②	<code>SELECT * FROM t_m_p NATURAL JOIN t_proj;</code>

- ① Der Inner-Join wird über die namensgleichen Spalten *id* der Tabellen *t_m_p* und *t_proj* ausgeführt.
- ② Alternativ kann bei dieser Konstellation ein Natural-Join benutzt werden. Eine Projektion ist nicht notwendig, doppelte Spalten werden automatisch entfernt.

```
MariaDB [uebungen]> SELECT * FROM t_m_p INNER JOIN t_proj ON t_m_p.id = t_proj.id;
+-----+-----+-----+-----+-----+
| ma_id | id   | id   | name  | beginn | ende   |
+-----+-----+-----+-----+-----+
|     2  | 1    | 1    | Buchprojekt | 2021-03-01 | 2021-08-20 |
|     5  | 1    | 1    | Buchprojekt | 2021-03-01 | 2021-08-20 |
|     8  | 1    | 1    | Buchprojekt | 2021-03-01 | 2021-08-20 |
|    11  | 1    | 1    | Buchprojekt | 2021-03-01 | 2021-08-20 |
+-----+-----+-----+-----+-----+
4 rows in set (0.001 sec)

MariaDB [uebungen]> SELECT * FROM t_m_p NATURAL JOIN t_proj;
+-----+-----+-----+-----+-----+
| id   | ma_id | name  | beginn | ende   |
+-----+-----+-----+-----+-----+
|     1  |     2  | Buchprojekt | 2021-03-01 | 2021-08-20 |
|     1  |     5  | Buchprojekt | 2021-03-01 | 2021-08-20 |
|     1  |     8  | Buchprojekt | 2021-03-01 | 2021-08-20 |
|     1  |    11  | Buchprojekt | 2021-03-01 | 2021-08-20 |
+-----+-----+-----+-----+-----+
4 rows in set (0.001 sec)
```

Unterschied zwischen einem Inner- und einem Natural-Join

Theta-Join

Ein Theta-Join wird wie ein Equi-Join bzw. Inner-Join erstellt, es wird aber in der ON-Klausel nicht auf Gleichheit geprüft, sondern ein anderer logischer Operator eingesetzt. Als logische Operatoren sind die Zeichen $>$, $<$, $>=$, $<=$, $<>$ erlaubt.

Beispiel: *ThetaJoin.sql*

Es werden Mitarbeiter aus der Tabelle *t_ma* gesucht, die älter sind als Mitarbeiter der Tabelle *t_ma_dt*.

```

① SELECT t_ma.name, t_ma.vname, t_ma.gebdat
    t_ma_dt.name, t_ma_dt.vname, t_ma_dt.gebdat
② FROM t_ma INNER JOIN t_ma_dt
③ ON t_ma.gebdat < t_ma_dt.gebdat LIMIT 10

```

- ① Es sollen die Felder *name*, *vname* und *gebdat* beider Tabellen angezeigt werden.
- ② Die Tabellen werden über das Schlüsselwort **INNER JOIN** verbunden.
- ③ In der ON-Klausel wird nicht auf Gleichheit, sondern auf 'größer als' verglichen. Mit der Angabe **LIMIT 10** wird die Anzeige auf die ersten 10 Datensätze beschränkt.

MariaDB [uebungen]> SELECT t_ma.name, t_ma.vname, t_ma.gebdat, t_ma_dt.name, t_ma_dt.vname, t_ma_dt.gebdat FROM t_ma INNER JOIN t_ma_dt ON t_ma.gebdat < t_ma_dt.gebdat LIMIT 10;					
name	vname	gebdat	name	vname	gebdat
Fuchs	Peter	1969-08-11	Illner	Markus	1976-12-23
Fuchs	Peter	1969-08-11	Schubert	Steve	1982-02-15
Dorff	Norbert	1968-12-23	Illner	Markus	1976-12-23
Dorff	Norbert	1968-12-23	Schubert	Steve	1982-02-15
Bayerle	Saskia	1976-06-05	Illner	Markus	1976-12-23
Bayerle	Saskia	1976-06-05	Schubert	Steve	1982-02-15
Schwönsdorf	Lisa	1981-11-15	Schubert	Steve	1982-02-15
Mannschatz	Annabell	1967-07-06	Illner	Markus	1976-12-23
Mannschatz	Annabell	1967-07-06	Schubert	Steve	1982-02-15
Nöller	Erwin	1980-05-04	Schubert	Steve	1982-02-15

10 rows in set (0.00 sec)

Outer-Join

Beim Inner-Join werden in der Ergebnismenge nur die Datensätze aufgeführt, die über die Join-Felder in beiden verknüpften Tabellen eine Übereinstimmung haben. Wenn Sie jedoch **alle** Datensätze einer Tabelle benötigen und diese um die Daten aus der verknüpften Tabelle erweitern wollen, ist ein Outer-Join die bessere Wahl. Ist in der verknüpften Tabelle kein passender Datensatz vorhanden, so werden die Datenfelder dieser Tabelle mit NULL-Werten belegt (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, Stichwort 'Outer-Join').

Beim Outer-Join wird zwischen einem Verknüpfen von links und von rechts unterschieden. Damit können sie festlegen, ob die Tabelle links oder rechts des Schlüsselworts **JOIN** vollständig übernommen wird. Die entsprechenden Join-Typen werden mit **LEFT OUTER JOIN** für das Verknüpfen von links und **RIGHT OUTER JOIN** für das Verknüpfen von rechts bezeichnet.

Beispiel: OuterJoin.sql

Es soll eine Mitarbeiterliste erzeugt werden, die für jeden Mitarbeiter die entsprechende Abteilung enthält. Durch eine Umstrukturierung im Unternehmen sind jedoch noch nicht alle Mitarbeiter in der Datenbank einer Abteilung zugeordnet. Mit einem Inner-Join erhalten Sie daher nur die Mitarbeiter, für die ein Datensatz in der Abteilungstabelle existiert. Durch das Anwenden eines Outer-Joins wird eine vollständige Mitarbeiterliste erstellt.

```
SELECT m.vname, m.name, a.name
①   FROM t_ma AS m LEFT OUTER JOIN t_abt AS a
②   ON m.abtnr = a.id ORDER BY m.name DESC LIMIT 15;
```

- ① Die Tabelle `t_ma` stellt die linke Tabelle dar, aus der alle Datensätze verwendet werden. Durch die Angabe von `LEFT OUTER JOIN` wird sie mit der Tabelle `t_abt` verknüpft.
- ② In der `ON`-Klausel werden die beiden Join-Felder verbunden. Zusätzlich soll die Ausgabe der Datensätze absteigend nach den Mitarbeiternamen sortiert werden. Mit der Angabe `LIMIT 15` wird die Anzeige auf die ersten 15 Datensätze beschränkt.

MariaDB [uebungen]> SELECT m.vname, m.name, a.name -> FROM t_ma AS m LEFT OUTER JOIN t_abt AS a -> ON m.abtnr = a.id -> ORDER BY m.name DESC LIMIT 15;		
vname	name	name
Norbert	Zielecki	Verkauf
Dilara	Ülkü	Abt_Organisation
Gudrun	Wolff	F&E
Steve	Walther	Controlling
Daniel	Unterwegner	Produktion
Tobias	Untergärtner	Verkauf
Lars	Trieschmann	Verkauf
Ansgar	Stifter	Verkauf
Hanna	Stern	Marketing
Jacqueline	Seiler	Abt_Organisation
Annemarie	Segebrecht	Einkauf
Andrea	Seeau	Marketing
Lisa	Schwönsdorf	Produktion
Gabriele	Schuster	Verkauf
Chris	Schmidtke	Einkauf

15 rows in set (0.00 sec)

Die Mitarbeiterliste als Ergebnis des Outer-Joins

Das gleiche Ergebnis wie im obigen Beispiel erhalten Sie, wenn Sie die Namen der Tabellen vertauschen und die Verknüpfung durch die Angabe von `RIGHT OUTER JOIN` realisieren.

Syntax

```
SELECT datenfeldliste FROM tabellename1 LEFT|RIGHT OUTER JOIN
```

```
tabellename2 ON bedingung ...;
```

- ✓ Der Outer-Join wird ähnlich dem Inner-Join gebildet. Nach dem Schlüsselwort `FROM` folgt der Name der ersten Tabelle. Über die Angabe von `LEFT OUTER JOIN` bzw. `RIGHT OUTER JOIN` wird diese mit der zweiten Tabelle verknüpft.
- ✓ Nach dem Schlüsselwort `ON` wird die Beziehung zwischen den Join-Feldern angegeben.
- ✓ Die Tabelle vor dem `OUTER JOIN` wird als linke Tabelle, die andere als rechte Tabelle bezeichnet. Je nachdem, ob `LEFT` oder `RIGHT OUTER JOIN` angegeben ist, wird die linke oder rechte Tabelle vollständig in das Abfrageergebnis übernommen.
- ✓ Sie können Inner- und Outer-Joins auch mischen. Dies ist jedoch schwer nachvollziehbar und verkompliziert das Erzeugen der gewünschten Ergebnisdatenmenge.
- ✓ Auch Outer-Joins können über mehr als zwei Tabellen definiert werden. Sie können Klammern setzen, um die Übersichtlichkeit zu erhöhen und die Reihenfolge der Joins gezielt zu steuern.

Falls Sie einmal nicht das gewünschte Ergebnis bei einem Join erhalten, sollten Sie die Tabellenreihenfolge verändern.

Eine Tabelle mit sich selbst verknüpfen (Self-Join)

Es müssen nicht immer zwei verschiedene Tabellen miteinander verbunden werden. Sie können auch eine Tabelle mit sich selbst verknüpfen und einen sogenannten Self-Join erzeugen. Dabei müssen Sie für die Tabelle zwei verschiedene Ersatznamen angeben.

Der Self-Join ist immer dann sinnvoll, wenn sich ein Feld einer Tabelle auf ein anderes in der gleichen Tabelle bezieht. Sie können Self-Joins aber auch verwenden, um die Daten einer Tabelle zu analysieren.

Beispiel: *SelfJoin.sql*

In der Tabelle *t_abt* werden die Abteilungsnamen und die jeweiligen Orte gespeichert. Um alle Abteilungen herauszufinden, an deren Standorten sich weitere Abteilungen befinden, kann ein Self-Join verwendet werden.

```
① SELECT a1.name,a2.name,a1.ort FROM t_abt AS a1
    INNER JOIN t_abt AS a2 ON a1.ort = a2.ort
    WHERE a1.id <> a2.id;
```

- ① Die Tabelle *t_abt* wird über den Befehl **INNER JOIN** mit sich selbst verknüpft. Dabei erhält sie die Ersatznamen *a1* und *a2*. Da alle Abteilungen gesucht werden, an deren Standort sich eine zweite Abteilung befindet, ist das Auswahlkriterium in der **ON**-Klausel der Ort.
- ② Damit ein sinnvolles Ergebnis entsteht, wird zusätzlich eine Bedingung angegeben, die bewirkt, dass nicht zwei gleiche Datensätze miteinander verbunden werden.

MariaDB [uebungen]> SELECT a1.name,a2.name,a1.ort -> FROM t_abt AS a1 -> INNER JOIN t_abt AS a2 ON a1.ort = a2.ort -> WHERE a1.id <> a2.id;		
name	name	ort
Abt_Organisation	Marketing	Berlin
Personal	Marketing	Berlin
F&E	Produktion	Wien
Marketing	Abt_Organisation	Berlin
Personal	Abt_Organisation	Berlin
Produktion	F&E	Wien
Marketing	Personal	Berlin
Abt_Organisation	Personal	Berlin

8 rows in set (0.00 sec)

Abteilungen, die sich am gleichen Ort befinden

10.4 Zwei Tabellen vereinigen

Mit Joins verknüpfen Sie zwei oder mehr Tabellen so, dass bestimmte Datensätze aus den verschiedenen Tabellen miteinander kombiniert werden. Zwei oder mehr Datensätze werden jeweils in einen Datensatz der Ergebnismenge aufgenommen. Meist ist die Verknüpfung von einem Vergleich bestimmter Felder der Tabellen abhängig.

Neben dieser Möglichkeit können Sie auch zwei Abfragen vereinigen, sodass eine Ergebnismenge entsteht, die sowohl Datensätze der ersten als auch der zweiten Abfrage enthält (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, 'Vereinigung'). Es wird die Vereinigungsmenge beider Abfragen gebildet. Umfassen die Abfragen die ganzen Tabellen, so wird die Vereinigungsmenge der Tabellen als Ergebnismenge geliefert. Dazu werden zwei **SELECT**-Anweisungen über das Schlüsselwort **UNION** miteinander verbunden. Beide Abfragen müssen dabei exakt die gleichen Datenfelder zurückliefern (Vereinigungsverträglichkeit).

Beispiel für Vereinigungsmengen *Union.sql*

Die Tabellen *t_ma* und *t_ma_dt* sollen vereinigt werden. Aus der Tabelle *t_ma* sind aber nur die Mitarbeiter aus *Bern* aufzunehmen. In der Ergebnismenge werden die Felder *name* und *ort* benötigt.

```
① SELECT name, ort FROM t_ma WHERE ort = 'Bern'  
②       UNION SELECT name, ort FROM t_ma_dt;
```

- ① Die erste SELECT-Anweisung ermittelt die Datensätze aus der Tabelle *t_ma*, bei denen der Ort *Gera* ist. Es werden nur die Felder *name* und *ort* projiziert.
- ② Über das Schlüsselwort UNION wird die Ergebnismenge der zweiten Abfrage angefügt. Die zweite Abfrage liefert die Felder *name* und *ort* der Tabelle *t_ma_dt*.

```
MariaDB [uebungen]> SELECT name, ort,  
->     FROM t_ma WHERE ort = 'Bern',  
->     UNION SELECT name, ort FROM t_ma_dt;  
+-----+-----+  
| name | ort  |  
+-----+-----+  
| Brio | Bern |  
| Walther | Bern |  
| Seiler | Bern |  
| Illner | Stuttgart |  
| Schubert | Stuttgart |  
| Satcke | NULL |  
+-----+-----+  
6 rows in set (0.00 sec)
```

Ergebnis der Vereinigung in MariaDB

10.5 Schnitt- und Differenzmengen

Auf die gleiche Weise wie die Vereinigungsmenge lassen sich Schnitt- und Differenzmengen erzeugen (vgl. Abschnitt 3.4, Übersicht der Operationen der Relationenalgebra, 'Durchschnitt', 'Differenz'). Bei PostgreSQL werden zwei Abfragen dazu über die Schlüsselwörter **INTERSECT** bzw. **EXCEPT** verbunden.

Beispiel

Aus den beiden Tabellen *t_ma* und *t_ma_dt* werden die gemeinsamen Mitarbeiter gesucht. Dazu wird eine Schnittmenge über das Schlüsselwort **INTERSECT** erzeugt.

```
SELECT name, ort FROM t_ma  
  INTERSECT SELECT name, ort FROM t_ma_dt;
```

```
uebungen=# SELECT name, ort FROM t_ma  
uebungen=#   INTERSECT SELECT name, ort FROM t_ma_dt;  
 name | ort  
-----+-----  
 Brauer | Frankfurt  
(1 Zeile)
```

Ergebnis der Schnittmenge bei PostgreSQL

Es werden alle Mitarbeiter gesucht, die nur in der Tabelle *t_ma* enthalten sind, nicht aber in der Tabelle *t_ma_dt*.

```
SELECT name, ort FROM t_ma  
  EXCEPT SELECT name, ort FROM t_ma_dt;
```

Diese beiden Operationen stehen nicht in jedem Datenbanksystem zur Verfügung, da sie zum SQL2 Intermediate Level gehören.

10.6 Unterabfragen

Unterabfragen ermitteln aus anderen Tabellen Informationen, welche für die eigentliche Abfrage einer Tabelle benötigt werden. Dabei wird die Unterabfrage als innere Abfrage bezeichnet, welche in die äußere – die eigentliche Abfrage – eingebettet wird.

Als Ergebnis liefert die Unterabfrage einen einzelnen Wert, eine Liste von Werten oder eine ganze Tabelle. Dieses Ergebnis wird in der äußeren Abfrage als Selektionskriterium in der Bedingung verwendet. Unterabfragen werden immer in Klammern eingefasst. Diese Schreibweise wird nicht von allen DBMS zwingend vorgeschrieben, ist jedoch auf Grund der besseren Lesbarkeit der SQL-Anweisung üblich.



Unterabfragen sind langsame Operationen. Wenn möglich sollten stattdessen Joins verwendet werden.

Einfache Unterabfragen

Für die Verknüpfungen zwischen den Relationen der Datenbank werden häufig Ziffern als IDs genutzt. Soll eine Selektion in einer Tabelle nach einem bestimmten Kriterium erfolgen, ist die Kenntnis der ID des Kriteriums notwendig. Häufig ist jedoch nicht diese, sondern nur die eigentliche Bezeichnung des Kriteriums bekannt. Mittels einer Unterabfrage kann die ID ermittelt und für die eigentliche Selektion verwendet werden.

Beispiel

Es sollen alle in der Tabelle `t_ma` erfassten Mitarbeiter der Abteilung *Controlling* ausgegeben werden. Die Verknüpfung zur Abteilung erfolgt in der Tabelle über den Eintrag der Abteilungs-ID `abtnr`. Da diese nicht bekannt ist, wird sie über eine Unterabfrage zuerst ermittelt und anschließend in der WHERE-Bedingung der äußeren Abfrage verwendet.

```

① SELECT vname, name, ort FROM t_ma
② WHERE abtnr =
      (SELECT id FROM t_abt WHERE name = 'Controlling');
  
```

- ① Die SELECT-Anweisung ermittelt die Felder `vname`, `name` und `ort` aus der Tabelle `t_ma`.
- ② In der WHERE-Bedingung wird die Abteilungsnummer verwendet. Dafür wird das Feld `abtnr` der Tabelle `t_ma` mit dem Ergebnis der Unterabfrage verglichen. Diese liefert das Feld `id` aus der Tabelle `t_abt` für den Datensatz, der im Datenfeld `name` den Eintrag *Controlling* enthält.

MariaDB [uebungen]> SELECT vname, name, ort FROM t_ma -> WHERE abtnr = -> (SELECT id FROM t_abt WHERE name = 'Controlling');		
vname	name	ort
Lilly	Baumann	Hamburg
Constantin	Brio	Bern
Klaus	Fiedler	Frankfurt
Matthias	Meyer	Zürich
Steve	Walther	Bern

5 rows in set (0.00 sec)

Ergebnis in MariaDB

Unterabfragen mit EXISTS

Mittels des Schlüsselworts `EXISTS` für die Unterabfrage kann getestet werden, ob bestimmte Daten in einer zweiten Tabelle vorhanden sind. Als Ergebnis der Unterabfrage werden keine konkreten Daten, sondern einer der Werte `TRUE` oder `FALSE` zurückgegeben.

Alternativ zu `EXISTS` ist die Verwendung von `NOT EXISTS` möglich.

Beispiel

Die Abfrage soll nur die Abteilungen ermitteln, für die in der Tabelle `t_ma` Mitarbeiter erfasst sind. Dazu wird mittels des Zusatzes `EXISTS` in der Unterabfrage geprüft, ob in der Tabelle `t_ma` im Feld `abtnr` ein Eintrag für die jeweilige Abteilungs-ID existiert.

```
① SELECT name FROM t_abt WHERE EXISTS  
②   (SELECT * FROM t_ma WHERE t_ma.abtnr = t_abt.id);
```

- ① In der Bedingung der `SELECT`-Anweisung wird der Zusatz `EXISTS` angegeben.
- ② Die Unterabfrage prüft, ob es in der Tabelle `t_ma` mindestens einen Datensatz gibt, welcher im Feld `abtnr` den Wert des Feldes `id` aus den Datensätzen der Tabelle `t_abt` besitzt. Ist diese Bedingung wahr, wird die Abteilung in der äußeren Abfrage ausgewählt.

```
MariaDB [uebungen]> SELECT name FROM t_abt WHERE EXISTS  
->   (SELECT * FROM t_ma WHERE t_ma.abtnr = t_abt.id);  
+-----+  
| name |  
+-----+  
| Einkauf |  
| Marketing |  
| Verkauf |  
| Produktion |  
| Abt_Organisation |  
| Controlling |  
| F&E |  
+-----+  
7 rows in set (0.00 sec)
```

Ergebnis der Vereinigung in MariaDB

10.7 Übung

Abfragen über mehrere Tabellen durchführen

Übungsdatei: --

Ergebnisdateien: *Uebung1.sql*, *Uebung2.sql*

1. Wechseln Sie zur Datenbank *Bibliothek*.

Erstellen Sie eine einfache Verknüpfung der Tabellen *t_leser* und *t_verleih*, um für jeden Leser die ausgeliehenen Bücher zu ermitteln. Lassen Sie sich in der Abfrage die Vor- und Familiennamen der Leser und die ISBN-Nummer anzeigen.

Verwenden Sie die gleiche Abfrage und ergänzen Sie diese um eine Sortierung nach Familiennamen und Vornamen der Leser.

Erstellen Sie die gleiche Abfrage unter Verwendung eines Joins.

Zeigen Sie nun zusätzlich die Namen der Bücher an, die in der Tabelle *t_buecher* abgelegt sind. Erweitern Sie dazu die SELECT-Anweisung um eine Verknüpfung zu dieser Tabelle.

2. Wechseln Sie zur Datenbank *Uebungen*.

Erstellen Sie eine Abfrage über die Tabellen *t_lager* und *t_artikel*, um alle am Lager vorrätigen Artikel aufzulisten. Verwenden Sie für die Tabellen passende Ersatznamen und lassen Sie sich die Artikelnummer, den Artikelnamen, den Lieferanten und die Stückzahl anzeigen.

Wandeln Sie die Abfrage so um, dass Sie alle Artikel erhalten, egal ob Sie am Lager sind oder nicht. Falls vorhanden, sollen neben der Artikelnummer, dem Artikelnamen und der Lieferantennummer auch die Stückzahl und der Preis angezeigt werden.

Erweitern Sie die Abfrage so, dass zusätzlich zu den bisherigen Feldern noch der Name der Lieferanten zurückgeliefert wird. Dabei sollen nur die Artikel berücksichtigt werden, die am Lager sind. Sortieren Sie die Ausgabe nach den Artikelnamen.

11

Sichten

11.1 Vordefinierte Abfragen

Eine wichtige Eigenschaft der Tabellen eines Datenbanksystems ist deren physikalische Existenz als strukturierte Sammlung von Daten auf einem Speichermedium. Oft werden jedoch nur bestimmte Datensätze oder Datenfelder einer Tabelle oder mehrerer verknüpfter Tabellen benötigt.

Abfragen sind eine Möglichkeit, die benötigten Daten bereitzustellen. Abfragen müssen aber immer wieder eingegeben werden. Um diese sich wiederholende Arbeit einzusparen, können Sie vordefinierte Abfragen, sogenannte **Sichten** (engl. **views**), verwenden. Sichten sind gespeicherte SELECT-Anweisungen, die bis auf ihre Definition keinen zusätzlichen Speicherplatz benötigen. Sie verwenden automatisch die aktuellen Datensätze der zugrunde liegenden Tabellen. Sichten entsprechen der Benutzersicht auf die Daten (vgl. Abschnitt 1.3, *3-Ebenen Modell*, Externe Ebene – Benutzersicht).

Mit Sichten können Sie auch einfacher mit verknüpften Tabellen arbeiten, da sie die Arbeit mit den verknüpften Daten wie mit einer einzigen Tabelle ermöglichen. Gleichzeitig erlauben sie, den Zugriff auf die Datensätze einer Tabelle für bestimmte Benutzer einzuschränken. Durch das Verwenden von Sichten kann der Zugriff auf die Daten ausschließlich auf die Benutzer beschränkt werden, für die sie relevant sind.

Sichten stellen Ergebnismengen von Abfragen dar, werden aber in SQL wie Tabellen verwendet. Sie können Daten aus vorhandenen Sichten abfragen und teilweise auch Datensätze mithilfe von Sichten aktualisieren bzw. neu einfügen. Alle Datenfelder, die beim Erstellen einer Sicht ausgewählt wurden, sind auch in der weiteren Verwendung der Sicht nicht vorhanden. Die Struktur der Tabellen, auf der die Sicht beruht, wird dabei jedoch nicht verändert.

Nicht alle Datenbanksysteme erlauben das Verwenden von Sichten bzw. sie unterstützen nicht alle Möglichkeiten von Sichten.

11.2 Sichten erstellen

Sichten werden mit der SQL-Anweisung `CREATE VIEW` erstellt. Dabei wird ein Name angegeben, über den Sie im weiteren Verlauf auf die Sicht zugreifen können.

Damit Sie Sichten besser von anderen Elementen des Datenbanksystems unterscheiden können, sollten Sie bei der Angabe des Namens das Präfix `v_` voranstellen. Dadurch grenzen Sie Sichten insbesondere von Tabellen ab.

Beispiel: *SichtErstellen.sql*

Die Tabelle `t_ma` enthält die Daten aller Mitarbeiter des Unternehmens. Die Beschäftigten der Abteilung 1 sollen jedoch nur Zugriff auf die Mitarbeiter der eigenen Abteilung erhalten. Zu diesem Zweck wird die Sicht `v_ma_abt1` erstellt, die zu Anschauungszwecken jedoch nur die Abteilungsnummer sowie den Vor- und Familiennamen der Mitarbeiter enthält.

```
(1) CREATE VIEW v_ma_abt1 AS
(2)     SELECT abtnr, vname, name FROM t_ma WHERE abtnr = 1;
(3)     SELECT * FROM v_ma_abt1;
```

- ① Mit der `CREATE-VIEW`-Anweisung wird die Sicht `v_ma_abt1` definiert.
- ② Nach dem Schlüsselwort `AS` folgt die `SELECT`-Anweisung, die der Sicht zugrunde liegen soll. Nur die hier angegebenen Datenfelder sind bei der späteren Verwendung der Sicht verfügbar.
- ③ In einer Abfrage kann die Sicht wie eine Tabelle verwendet werden. Hier werden alle Datensätze der Sicht abgerufen. In diesem Fall sind das die Abteilungsnummer und die Namen aller Mitarbeiter der Abteilung 1.

MariaDB [uebungen]> SELECT * FROM v_ma_abt1;		
abtnr	vname	name
1	Sebastian	Berger
1	Annabell	Mannschatz
1	Peter	Meyer
1	Frank	Maier
1	Chris	Schmadtko
1	Annemarie	Segebrecht
1	Sophie	Brauer
1	Sabine	Carstedt

8 rows in set (0.00 sec)

Abrufen aller Datensätze der Sicht
„v_ma_abt1“ in MariaDB

Abfragen für Sichten

In der `SELECT`-Anweisung, die einer Sicht zugrunde liegen soll, sind beliebige `WHERE`-Klauseln gestattet. Auch Abfragen über mehrere Tabellen (Joins) sind möglich. Nicht zulässig sind dagegen folgende Klauseln:

- ✓ `GROUP BY` und `HAVING`
- ✓ `ORDER BY`
- ✓ `UNION`

Die Angabe der Datenfeldnamen in der Abfrage ist optional. Um alle Felder einzuschließen, können Sie den Operator `*` verwenden. Ebenfalls zulässig sind Ersatznamen (Aliasnamen).

In folgenden Fällen ist die Angabe von Ersatznamen für Datenfelder zwingend notwendig:

- ✓ wenn die Namen der Datenfelder bei Abfragen über mehrere Tabellen nicht eindeutig sind,
- ✓ wenn in der Abfrage Ausdrücke oder Aggregatfunktionen verwendet werden.



Es besteht zusätzlich die Möglichkeit, Sichten zu erstellen, in deren SELECT-Anweisung ebenfalls auf eine Sicht zugegriffen wird. Beachten Sie jedoch, dass dies zu schwer durchschaubaren Abfragestrukturen führt.

Sichten über mehrere Tabellen

In umfangreichen Datenbeständen kann unter der Verteilung der Daten auf mehrere verschiedene Tabellen die Übersicht leiden. Mit Sichten, die auf Joins beruhen und die Daten aus mehreren Tabellen zusammenführen, kann die Übersichtlichkeit verbessert werden.

Beispiel: *SichtErstellenJoin.sql*

Um zu ermitteln, welche Mitarbeiter an welchen Projekten beteiligt sind, wird eine Abfrage über die Tabellen *t_ma*, *t_ma_proj* und *t_proj* benötigt. Zusätzlich werden aus allen Tabellen nur bestimmte Datenfelder verwendet.

```

① CREATE VIEW v_ma_proj
②   (MitarbeiterNr, Mitarbeitername, ProjektNr, Projektname) AS
③     SELECT m.id, m.name, p.id, p.name FROM t_ma_proj mp
        INNER JOIN t_ma m ON mp.ma_id = m.id
        INNER JOIN t_proj p ON mp.proj_id = p.id;
④   SELECT * FROM v_ma_proj WHERE ProjektNr = 1;

```

- ① Die Sicht wird mit der Anweisung CREATE VIEW erstellt. Um sie von der Tabelle *t_ma_proj* zu unterscheiden, wird das Präfix *v_* vorangestellt.
- ② In der Sicht müssen alle Datenfelder der beteiligten Tabellen einen eindeutigen Namen erhalten. Da in der vorliegenden Abfrage die Namen *id* und *name* doppelt vorkommen, werden für sie spezielle Ersatznamen festgelegt.
- ③ An dieser Stelle folgt die Angabe der zugrunde liegenden Abfrage über die drei Tabellen mithilfe von Inner-Joins.
- ④ Die probeweise Abfrage über die neu erstellte Sicht liefert die Namen aller Mitarbeiter, die am Projekt mit der Nummer 1 arbeiten. Ohne diese Sicht wäre dafür die aufwendige Abfrage (③) inklusive der WHERE-Klausel aus ④ notwendig gewesen.

MariaDB [uebungen]> SELECT * FROM v_ma_proj WHERE ProjektNr = 1;			
MitarbeiterNr	Mitarbeitername	ProjektNr	Projektname
2	Baumann	1	Buchprojekt
5	Bayerle	1	Buchprojekt
8	Bergstein	1	Buchprojekt
11	Schwondorf	1	Buchprojekt

4 rows in set (0.00 sec)

Die Abfrage der Sicht liefert die Daten aus drei Tabellen

Datenfelder in der Sicht benennen

Die Namen aller Datenfelder in einer Sicht müssen eindeutig sein. Insbesondere bei Abfragen, die sich über mehrere Tabellen erstrecken, können Namen von Datenfeldern mehrmals vorkommen. In solchen Fällen müssen Sie für die Datenfelder der Sicht neue Datenfeldnamen definieren. Die Angabe der Datenfeldnamen erfolgt in runden Klammern nach dem Namen der Sicht.

Syntax

```
CREATE VIEW viewname [ (datenfeldliste) ] AS SELECT . . . ;
```

- ✓ Die Anweisung wird von den Schlüsselwörtern CREATE VIEW eingeleitet. Danach folgt der Name der Sicht.
- ✓ In runden Klammern erfolgt die Angabe der Datenfeldnamen der Sicht. Die Reihenfolge entspricht den Datenfeldern in der SELECT-Anweisung.
- ✓ Falls die Namen der Datenfelder in der Abfrage eindeutig sind, kann die Datenfeldliste entfallen. In diesem Fall werden die Namen aus der Abfrage verwendet.
- ✓ Nach dem Schlüsselwort AS folgt die Angabe der SELECT-Anweisung. Diese kann eine WHERE-Klausel enthalten und mehrere Tabellen verknüpfen.
- ✓ In der Abfrage dürfen die Klauseln GROUP BY, HAVING, ORDER BY und UNION nicht verwendet werden.

Um eine Sicht zu erstellen, müssen Sie über das entsprechende Recht verfügen. Dieses wird mit der Anweisung GRANT CREATE VIEW gewährt. Zusätzlich benötigen Sie Leserechte für alle an der Abfrage beteiligten Tabellen.

11.3 Sichten löschen

Sichten werden mit der Anweisung DROP VIEW gelöscht. Die Anweisung löscht ausschließlich die Definition der Sichten. Die zugrunde liegenden Tabellen werden dabei nicht verändert. Um eine Sicht zu löschen, müssen Sie der Besitzer sein oder über die entsprechenden Zugriffsrechte verfügen.

Beispiel

Mit der Anweisung DROP VIEW wird die Sicht v_ma_abt1 gelöscht.

```
DROP VIEW v_ma_abt1;
```

Syntax

```
DROP VIEW viewname;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern DROP VIEW. Danach folgt der Name der Sicht.
- ✓ Falls die Sicht zu diesem Zeitpunkt durch andere SQL-Anweisungen, z. B. eine SELECT-Anweisung, verwendet wird, kann sie nicht gelöscht werden.
- ✓ Falls Sichten existieren, die auf der gelöschten Sicht basieren, werden diese ebenfalls gelöscht.

11.4 Daten über Sichten einfügen, ändern und löschen

Daten über Sichten einfügen und ändern

Unter bestimmten Voraussetzungen können Sichten in manchen Systemen auch zum Eingeben und Ändern von Daten verwendet werden. Bei MariaDB existieren für die Verwendung von INSERT-, UPDATE- und DELETE-Anweisungen jedoch in Sichten viele Einschränkungen, unter anderen:

- ✓ In der SELECT-Anweisungen der Sicht darf nicht das Schlüsselwort DISTINCT verwendet werden.
- ✓ Die Sicht darf sich nur auf eine Tabelle beziehen. Joins sind beim Ändern von Daten nicht gestattet.
- ✓ Die Abfrage darf keine Unterabfragen in der Bedingung verwenden.

Das Verändern der Datensätze in einer Sicht bedeutet das tatsächliche Ändern der entsprechenden Datensätze in der zugrunde liegenden Tabelle.

Wird ein Datensatz in eine Sicht eingefügt, die nur ausgewählte Felder einer Tabelle enthält, werden die restlichen Felder dieses Datensatzes (die nicht in der Sicht enthalten sind) in der Tabelle mit dem Wert NULL gefüllt. Voraussetzung dafür ist, dass diese Felder den Wert NULL akzeptieren (d. h., dass sie nicht mit NOT NULL definiert wurden) oder ein entsprechender Vorgabewert (mit dem Schlüsselwort DEFAULT) festgelegt wurde.

PostgreSQL unterstützt diese Anweisungen für einfache Sichten mit aktualisierbaren Spalten, die sich direkt auf eine Tabellenspalte beziehen. Daten können auch (seit der Version 9.4) in Sichten mit nicht-aktualisierbaren Spalten, wie beispielsweise einer auf zwei Spalten beruhenden Berechnung, eingefügt oder verändert werden. Für komplexe Sichten, welche auf der Abfrage mehrerer Tabellen basieren, können Sie durch die programminternen Regeln ON INSERT (bzw. ON UPDATE oder ON DELETE) DO-INSTEAD-Datenänderungen herbeiführen. Mehr dazu finden Sie in der Dokumentation von PostgreSQL.

Beispiel: *SichtDatenÄndern.sql*

Es wird eine neue Sicht definiert, die auf Datenfeldern der Tabelle *t_ma* beruht. Da über die Sicht Daten in die Tabelle eingefügt werden sollen, müssen sich alle Felder in der Sicht befinden, die keine NULL-Werte akzeptieren, wie z. B. das Primärschlüsselfeld.

```

① CREATE VIEW v_ma_abt AS
    SELECT id, vname, name, abtnr FROM t_ma;
② UPDATE v_ma_abt SET abtnr = 2 WHERE abtnr IS NULL;
③ INSERT INTO v_ma_abt (id, name, vname, abtnr)
    VALUES(102, 'Melzer', 'Martin', 1);

```

- ① Es wird eine neue Sicht auf Basis der Tabelle *t_ma* erstellt. Da das Feld *id* als Primärschlüssel definiert wurde und einen Wert enthalten muss, wird es in die Sicht aufgenommen. Nur so ist das Einfügen neuer Datensätze in die Tabelle möglich.
- ② Mit der UPDATE-Anweisung wird in allen Datensätzen, die noch keinen Wert im Feld *abtnr* enthalten (abtnr IS NULL), der Wert 2 eingefügt.
- ③ Ein neuer Datensatz wird wie bei Tabellen mit der INSERT-INTO-Anweisung eingefügt. Dem Primärschlüssel *id* muss dabei ein eindeutiger Wert zugewiesen werden.

Daten über Sichten löschen

Mithilfe der `DELETE`-Anweisung können Sie für eine Sicht Datensätze in der Basistabelle löschen.

Daten beim Verändern überprüfen

Wird in der `SELECT`-Anweisung, die der Sicht zugrunde liegt, eine `WHERE`-Klausel angegeben, wird die Situation beim Einfügen, Ändern oder Löschen von Daten der Sicht undurchsichtiger. Sie können zwar auch jetzt diese Operationen ausführen, die Sicht enthält jedoch nur die Datensätze, auf die die angegebene Bedingung zutrifft. Wird beispielsweise ein Datensatz eingefügt, der nicht der Bedingung entspricht, lässt sich durch das Anzeigen der Datensätze der Sicht nicht überprüfen, ob die Operation erfolgreich war.

Wollen Sie erreichen, dass nur Datensätze eingefügt, gelöscht oder geändert werden können, die in der Sicht auch angezeigt werden, fügen Sie der Definition der Sicht die Klausel `WITH CHECK OPTION` hinzu. So wird beim Bearbeiten der Daten ebenfalls die Bedingung aus der `SELECT`-Anweisung beachtet.

In PostgreSQL ist diese Option seit der Version 9.4 implementiert.

Beispiel: *SichtDatenÄndernCheck.sql*

Im Folgenden wird eine Sicht erstellt, die entsprechend der Bedingung in der Abfrage prüft, ob bei Änderungsoperationen im Datenfeld `abtnr` der Wert 1 angegeben wurde. Ist dies nicht der Fall, wird eine Fehlermeldung angezeigt.

```

① CREATE VIEW v_ma_abt1 AS
    SELECT * FROM t_ma WHERE abtnr = 1
② WITH CHECK OPTION;
③ INSERT INTO v_ma_abt1
    (id, name, vname, abtnr)
    VALUES(103, 'Funke', 'Franziska', 2);
④ INSERT INTO v_ma_abt1
    (id, name, vname, abtnr)
    VALUES(103, 'Funke', 'Franziska', 1);

```

- ① Die Sicht enthält alle Datenfelder der Tabelle `t_ma` und umfasst alle Mitarbeiter der Abteilung 1.
- ② Am Ende der `CREATE-VIEW`-Anweisung erfolgt die Angabe des Befehls zur Datenüberprüfung.
- ③ Hier soll ein Datensatz eingefügt werden, der im Feld `abtnr` den Wert 2 besitzt. Dies widerspricht der `WHERE`-Klausel in der Definition der Sicht, die nur Datensätze mit dem Wert 1 im Feld `abtnr` zulässt. Das DBMS erzeugt eine Fehlermeldung.
- ④ Diese `INSERT`-Anweisung wird korrekt ausgeführt, da das Feld `abtnr` den Wert 1 besitzt.

```

MariaDB [uebungen]> INSERT INTO v_ma_abt1
->     (id, name, vname, abtnr) VALUES(103, 'Funke', 'Franziska', 2);
ERROR 1369 (44000): CHECK OPTION failed 'uebungen' . 'v_ma_abt1'
MariaDB [uebungen]>
MariaDB [uebungen]> INSERT INTO v_ma_abt1
->     (id, name, vname, abtnr) VALUES(103, 'Funke', 'Franziska', 1);
Query OK, 1 row affected (0.00 sec)

```

Fehlermeldung beim Einfügen eines Datensatzes, der die WHERE-Klausel nicht erfüllt, über eine Sicht

Die Überprüfung der Daten wird auch beim Löschen von Datensätzen mit der `DELETE`-Anweisung angewendet. Sie können daher nur die Datensätze löschen, die auch von der Sicht angezeigt werden.

Syntax

```
CREATE VIEW viewname AS SELECT ... WITH CHECK OPTION;
```

- ✓ Die Definition der Sicht erfolgt über die Anweisung `CREATE VIEW`, gefolgt vom Namen der Sicht und der `SELECT`-Anweisung, die die Sicht definiert.
- ✓ Nach der Abfrage wird die Klausel `WITH CHECK OPTION` angegeben, damit eine Überprüfung der Datensätze, die geändert, eingefügt oder gelöscht werden sollen, mit der `WHERE`-Klausel der Abfrage erfolgt.

11.5 Übung

Sichten erstellen

Übungsdatei: --

Ergebnisdateien: [Uebung1.sql](#), [Uebung2.sql](#)

1. Wechseln Sie zur Datenbank *Uebungen*.

Erstellen Sie basierend auf der Tabelle `t_ma` eine Sicht `v_ma_berlin`, die die ID-Nummer, den Familiennamen, den Vornamen, die Abteilung, die Straße, die Hausnummer, den Ort und die Postleitzahl aller Mitarbeiter aus Berlin enthält.

Rufen Sie mit einer Abfrage alle Datensätze der Sicht ab. Sortieren Sie die Ausgabe einmal aufsteigend nach Familienname und Vorname sowie ein zweites Mal nach der Abteilungsnummer.

Löschen Sie die Sicht. Eventuell müssen Sie vorher laufende Abfragen mit dem Befehl `COMMIT` bestätigen.

Erstellen Sie die Sicht neu und verwenden Sie dabei die Option zur Datenüberprüfung.

Erstellen Sie einen neuen Datensatz für die Mitarbeiterin Helene Weigelt, Südstr. 6a, 10114 Berlin. Die Mitarbeiter-ID soll 140 lauten.

Versuchen Sie, für den gerade eingefügten Datensatz den Ort von *Berlin* in *Potsdam* zu ändern. Was stellen Sie fest?

2. Wechseln Sie zur Datenbank *Uebungen*.

Erstellen Sie eine Sicht, die für alle Mitarbeiter aus Berlin die Projektnummern anzeigt, an denen die Mitarbeiter beteiligt sind. Verknüpfen Sie dazu die Tabellen `t_ma` und `t_ma_proj` mit einem Join.

Rufen Sie zuerst alle Datensätze der Sicht ab. Zeigen Sie danach nur die Datensätze an, für die die Projektnummer 2 ist.

Erstellen Sie eine `SELECT`-Anweisung, um die Anzahl der Mitarbeiter aus Berlin, die am Projekt 2 beteiligt sind, zu ermitteln.

Ermitteln Sie diesen Wert mithilfe einer Sicht und vergleichen Sie die Komplexität der Abfragen.

12

Cursor

12.1 Sequenzielles Lesen von Datensätzen

SELECT-Abfragen liefern in der Regel eine Ergebnismenge mit allen Datensätzen, auf die eine angegebene Bedingung, sofern vorhanden, zutrifft. Da sich auch alle anderen Anweisungen in SQL auf Mengen beziehen, wird SQL auch als mengenorientierte Sprache bezeichnet. Insbesondere bei der Weiterverarbeitung der Datensätze in einem eigenen Programm kann es nützlich sein, nur einen bestimmten Datensatz aus der Ergebnismenge abzurufen. Dieser wird verarbeitet und anschließend wird der nächste Datensatz aufgerufen.

Eine Möglichkeit wäre, die Bedingung der Abfrage so zu verändern, dass nur ein einziger Datensatz zurückgeliefert wird. Eine effektivere Lösung ist jedoch das Verwenden eines Cursors (engl. Cursor = Positions- bzw. Schreibmarke).

Damit Sie die Datensätze einer Abfrage Schritt für Schritt bearbeiten können, wird intern ein Datenpuffer angelegt. Der **Cursor (Zeiger)** verweist nun auf eine bestimmte Stelle im Puffer und erlaubt den Zugriff auf den entsprechenden Datensatz. Das Verarbeiten der Ergebnismenge der Abfrage erfolgt dabei wie das sequenzielle Lesen aus einer Datei. Zuerst wird die Datei geöffnet, danach wird ein Datensatz nach dem anderen gelesen. Am Ende wird die Datei geschlossen und der Zugriff beendet.

Bei der Arbeit mit einem Cursor sind folgende SQL-Befehle von Bedeutung:

DECLARE CURSOR	Definieren eines Cursors mit der zugrunde liegenden Abfrage
OPEN	Öffnen des Datenpuffers zur sequenziellen Abfrage der Datensätze
FETCH	Abrufen eines Datensatzes
CLOSE	Schließen des Datenpuffers

Mit der Anweisung **DECLARE CURSOR** erstellen Sie einen serverseitigen Cursor, der direkt über das Datenbanksystem verwaltet wird. Eine andere Möglichkeit sind sogenannte clientseitige Cursor, bei denen die Verwaltung in einer Anwendung erfolgt, die auf die Datenbank aufsetzt, z. B. ein PHP-Skript in der Web-Programmierung.

Serverseitige Cursor benötigen viel Arbeitsspeicher und Rechenzeit auf dem verarbeitenden Computer. Sie sind daher weniger für viele simultane Zugriffe geeignet. Clientseitige Cursor bieten sich für kleine Datenbestände an, wie sie bei Datenbanken im Internet häufig vorkommen.

Beachten Sie, dass das sequenzielle Lesen von Datensätzen mithilfe eines Cursors nicht direkt in einem Datenbank-Frontend angewendet werden kann. Dies ist nur innerhalb eines Anwendungsprogramms über sogenanntes eingebettetes SQL und eine interne Schnittstelle zum Datenbanksystem möglich. Solche Anwendungsprogramme können beispielsweise in C oder C++ programmiert werden.

12.2 Cursor erstellen

Mit der SQL-Anweisung `DECLARE CURSOR` wird ein Cursor definiert. Dabei können Sie eine beliebige gültige SQL-Abfrage verwenden. Mit dieser Anweisung wird nur der Zeiger auf die Ergebnismenge erzeugt, ein Datenzugriff erfolgt dabei noch nicht. Die Definition des Cursors muss vor allen anderen SQL-Anweisungen ausgeführt werden, die den Cursor verwenden sollen.

Beispiel: *CursorErzeugen.sql*

Die folgende SQL-Anweisung erzeugt einen Cursor zum sequenziellen Lesen des aktuellen Lagerbestandes. Dabei wird eine einfache `SELECT`-Abfrage verwendet, die als Ergebnismenge alle Artikel liefert, von denen mindestens ein Stück am Lager ist.

```
① DECLARE c_lager CURSOR FOR
②   SELECT * FROM t_lager WHERE stueck >= 1;
```

- ① Es wird der Cursor `c_lager` definiert. Das Präfix `c_` soll das Unterscheiden von anderen Datenbankobjekten, z. B. Tabellen, erleichtern.
- ② Nach dem Schlüsselwort `FOR` folgt die `SELECT`-Abfrage, die in diesem Fall alle Datenfelder der Tabelle `t_lager` zurückliefert. Als Bedingung wird angegeben, dass die Stückzahl mindestens 1 betragen muss.

Syntax

```
DECLARE cursorname CURSOR FOR SELECT ... ;
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort `DECLARE`. Danach folgt der gewünschte Name des Cursors.
- ✓ Nach dem Schlüsselwort `FOR` wird eine gültige `SELECT`-Anweisung angegeben, die auch Joins und Bedingungen enthalten kann.

12.3 Datenzugriff mit dem Cursor

Um Datensätze mithilfe eines Cursors zu lesen, muss der zuvor definierte Cursor explizit geöffnet werden. Dies geschieht mit der SQL-Anweisung OPEN. Mit der FETCH-Anweisung können Sie danach Datensatz für Datensatz auslesen.

Die OPEN-Anweisung führt die bei der Definition des Cursors angegebene SELECT-Abfrage aus und speichert das Ergebnis in einem Datenpuffer. Danach befindet sich der Zeiger auf dem ersten Datensatz der Ergebnismenge.

Beispiel: *CursorOpenFetch.sql*

Mit der OPEN-Anweisung wird der im vorherigen Beispiel definierte Cursor `c_lager` geöffnet und mit FETCH wird der erste Datensatz ausgelesen. Die Werte der einzelnen Datenfelder werden dabei in Variablen des Programms gespeichert.

```
① OPEN c_lager;
② FETCH c_lager INTO :id, :stueck, :preis;
```

- ① Hier wird der vorher definierte Cursor `c_lager` geöffnet.
- ② Mit der FETCH-Anweisung erfolgt das Lesen des ersten Datensatzes. Die Werte der einzelnen Datenfelder werden dabei in den angegebenen Variablen des Programms gespeichert. Nach dem Ausführen der Anweisung wird der Zeiger auf dem zweiten Datensatz der Ergebnismenge positioniert.

Nach dem Öffnen des Cursors befindet sich der Zeiger stets auf dem ersten Datensatz. Jedes Ausführen der FETCH-Anweisung liefert den aktuellen Datensatz und rückt den Zeiger auf den nächsten Datensatz. Einige Datenbanksysteme erlauben zusätzlich das Abrufen des vorherigen, des ersten, des letzten oder eines beliebigen Datensatzes. Mehr Informationen dazu finden Sie in der Dokumentation Ihres Datenbanksystems unter dem Stichwort FETCH.

Syntax

```
OPEN cursorname;
FETCH cursorname INTO|USING :hostvariable1, ...;
```

- ✓ Mit dem Schlüsselwort OPEN, gefolgt vom Namen des Cursors, wird der Zugriff auf die Ergebnismenge der dazugehörigen Abfrage ermöglicht.
- ✓ Um den nächsten Datensatz abzurufen, wird die FETCH-Anweisung verwendet. Dem Schlüsselwort FETCH folgt dabei der Name des Cursors.
- ✓ Über das Schlüsselwort INTO bzw. USING wird das Speichern der Werte der einzelnen Datenfelder in entsprechende Variablen des ausführenden Programms gesteuert. Diese Variablen müssen vorher definiert werden und einen passenden Datentyp besitzen.
- ✓ Über eine Statusvariable können Sie prüfen, ob sich der Cursor auf einem gültigen bzw. dem letzten Datensatz befindet.

12.4 Cursor schließen

Wird der Datenpuffer nicht mehr benötigt, können Sie den Cursor mit der Anweisung CLOSE wieder schließen. Mit dieser Anweisung wird der Cursor inaktiv und die Verknüpfung mit der Ergebnismenge wird aufgehoben. Alle verwendeten Systemressourcen sind danach wieder freigegeben. Sie können jetzt keine Zugriffe mehr auf die Daten vornehmen. Eine anschließende OPEN-Anweisung öffnet den Cursor wieder für einen erneuten Datenzugriff. Die Definition des Cursors, die mit der Anweisung DECLARE CURSOR erfolgte, wird beim Schließen nicht verändert.

Syntax

```
CLOSE cursorname;
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort CLOSE. Danach folgt der Name des Cursors.

13

Zugriffsrechte und Benutzer verwalten

13.1 Sicherheitskonzepte

Benutzerverwaltung in Datenbanksystemen

Datenbanksysteme speichern häufig sensible Daten, zu deren Schutz ein Sicherheitskonzept mit einer Rechteverwaltung für die Objekte der Datenbank notwendig ist. Dieses Konzept ist bei allen Datenbanksystemen im Wesentlichen ähnlich aufgebaut und unterscheidet sich nur in Details der Umsetzung.

Der wichtigste Schritt im Aufbau eines sicheren Datenbanksystems besteht im Verwalten unterschiedlicher Benutzer bzw. Benutzergruppen. Diesen Benutzern können mit den SQL-Anweisungen GRANT und REVOKE definierte Zugriffsberechtigungen zugewiesen oder entzogen werden.

Die Benutzerverwaltung übernimmt in der Regel der Datenbank- oder Systemadministrator. Er verfügt über die meisten Rechte im System und kann diese anderen Benutzern zuweisen oder entziehen. Für den Datenbankadministrator wird beim Installieren des Datenbanksystems automatisch ein Benutzerkonto erstellt. Dieser Benutzer hat einen vom verwendeten Datenbanksystem abhängigen Namen.

Datenbanksystem	Benutzername des Administrators	Standard-Passwort
MariaDB	root	keins
PostgreSQL	postgres	keins
Microsoft SQL-Server	sa	keins



Nach der Installation des Datenbanksystems sollten Sie unbedingt das Passwort des standardmäßig installierten Datenbankadministrators ändern bzw. ein Passwort vergeben, um die Sicherheit des Systems zu bewahren. Da Sie nur für bestimmte Befehle über die Administratorrechte verfügen müssen, sollten Sie (auch für sich selbst) zusätzlich einen Benutzer mit etwas eingeschränkten Rechten erstellen, den Sie als Standardbenutzer einrichten.

Bei der Eingabe des Passwortes gelten folgende Regeln:

- ✓ Das Passwort kann maximal 32 Zeichen lang sein. Es sind jedoch nur die ersten acht Zeichen relevant.
- ✓ Bei der Überprüfung des Passwortes wird, im Gegensatz zum Benutzernamen, zwischen Groß- und Kleinschreibung unterschieden.

Benutzer eines Datenbanksystems

Die Benutzer in einem Datenbanksystem sind unabhängig von den Benutzern des Betriebssystems, auf dem das Datenbanksystem installiert ist. Sie müssen für das Datenbanksystem extra angelegt werden. Die Benutzer des Datenbanksystems sind in der Hierarchie über den einzelnen Datenbanken angeordnet. Daher müssen die Benutzer auch nur einmal angelegt werden. Anschließend kann ein Benutzer dann Zugriff auf alle oder einzelne ausgewählte Datenbanken des Systems erhalten.

Grundsätzlich ist jeder Benutzer, der ein Datenbankobjekt wie z. B. eine Tabelle erstellt, automatisch auch Besitzer dieses Objektes. Alle anderen Benutzer des Systems besitzen für ein Datenbankobjekt eines anderen Benutzers nach dessen Erstellung keine Zugriffsrechte. Eine Ausnahme ist der Datenbankadministrator, der über alle Datenbankobjekte verfügen kann. Er kann auch anderen Benutzern zusätzliche Rechte einräumen.

13.2 Benutzerverwaltung unter PostgreSQL

Mit Rollen arbeiten in PostgreSQL

Sie können in PostgreSQL sowohl einzelne Benutzer als auch Benutzergruppen erstellen. Der Vorteil von Gruppen ist, dass Sie diese mit bestimmten Rechten versehen können, welche dann automatisch allen Benutzern zugewiesen werden, die dieser Gruppe angehören.

Bei PostgreSQL werden die Benutzer und Gruppen unter dem Begriff „Rollen“ zusammengefasst. Sie werden allein dadurch unterschieden, dass eine Rolle, die als Benutzer dient, sich im Datenbanksystem anmelden kann, während andere Rollen das nicht können. Dabei können Rollen sowohl individuelle Zugriffsrechte besitzen als auch Teil einer anderen Rolle sein.

Neue Benutzer anlegen

Bei der Installation von PostgreSQL wird automatisch ein mit allen Rechten ausgestatteter Benutzer angelegt. In der Regel heißt dieser Benutzer *postgres*. Um weitere Benutzer zu erzeugen, müssen Sie sich zunächst mit diesem Benutzer anmelden. Anschließend können Sie mithilfe der Anweisung `CREATE ROLE` und der Option `LOGIN` einen Benutzer anlegen und, wenn gewünscht, einige Benutzerattribute festlegen.

Beispiel: BenutzerAnlegenPostgreSQL.sql

Im folgenden Beispiel sollen die Benutzer *pscherhorn* und *jbossert* erstellt werden. Beide Benutzer sollen ein Passwort erhalten. Der Benutzer *pscherhorn* soll alle Rechte erhalten, also ein sogenannter Superuser sein. Der Benutzer *jbossert* darf Datenbanken erstellen. Zusätzlich sollen die Benutzer *fmueller* und *pfunke* erstellt werden, die zunächst keine Rechte haben.

```
① CREATE ROLE fmueller LOGIN; CREATE ROLE pfunke LOGIN;
② CREATE ROLE pscherhorn LOGIN PASSWORD 'passwort' SUPERUSER;
③ CREATE USER jbossert PASSWORD 'passwort' CREATEDB;
```

- ① Die Benutzer *fmueller* und *pfunke* werden ohne weitere Optionen angelegt.
- ② Der Benutzer *pscherhorn* wird angelegt. Mit der Anweisung **PASSWORD** geben Sie dem Benutzer ein Passwort mit. Die zusätzliche Anweisung **SUPERUSER** bewirkt, dass dieser Benutzer alle Zugriffsrechte erhält.
- ③ Hier wird der Benutzer *jbossert* erstellt. Auch diesem Benutzer wird ein Passwort zugewiesen und durch die Anweisung **CREATEDB** das Recht erteilt, Datenbanken zu erstellen. Da versäumt wurde, den Zusatz **LOGIN** einzugeben, wird sich der Benutzer nicht im Datenbanksystem anmelden können.

Möchten Sie in dem Namen Sonderzeichen oder Großbuchstaben verwenden, müssen Sie den Namen in Anführungszeichen setzen.

Syntax zum Anlegen neuer Benutzer

Die Anweisung **CREATE ROLE** ist sehr umfangreich. Hier ein Beispiel der wichtigsten Optionen:

```
CREATE ROLE benutzername LOGIN [[WITH]
    | ENCRYPTED | UNENCRYPTED] PASSWORD 'passwort'
    | SUPERUSER | NOSUPERUSER
    | CREATEROLE | NOCREATEROLE
    | CREATEDB | NOCREATEDB
    | INHERIT | NOINHERIT
    | IN ROLE rollenname(n) [, ...]
| VALID UNTIL 'datum');
```

- ✓ Neue Benutzer legen Sie mit der Anweisung **CREATE ROLE** gefolgt von einem eindeutigen Benutzernamen und der Option **LOGIN** an.

! Wenn Sie die Anweisung **LOGIN** nicht hinzufügen, erstellen Sie eine Rolle, die lediglich als Gruppe dient. Mit dieser Rolle können Sie sich nicht im Datenbanksystem anmelden.

Sie haben die Möglichkeit, weitere Attribute beim Anlegen des Benutzers festzulegen, z. B.:

[ENCRYPTED UNENCRYPTED] PASSWORD 'passwort'	Der Benutzer erhält ein Passwort. Bei Angabe der Option ENCRYPTED wird das Passwort verschlüsselt. UNENCRYPTED ist der Vorgabewert.
SUPERUSER NOSUPERUSER	Ist der Benutzer Superuser, erhält er alle Zugriffsrechte. NOSUPERUSER ist der Vorgabewert.

CREATEROLE NOCREATEROLE	Mit der Option CREATEROLE erhält der Benutzer das Recht, Rollen zu erstellen.
CREATEDB NOCREATEDB	Mit der Option CREATEDB erhält der Benutzer das Recht, Datenbanken zu erstellen.
INHERIT NOINHERIT	Hier können Sie festlegen, ob der Benutzer automatisch alle Rechte der Rollen erben soll, denen er angehört.
IN ROLE rollename (n)	Ordnet den Benutzer einer oder mehreren Rollen zu
VALID UNTIL 'datum'	Beschränkt die Gültigkeit eines Passworts auf ein bestimmtes Datum
CONNECTION LIMIT limitangabe	Mit der Anweisung CONNECTION LIMIT können Sie ein Verbindungslimit eingeben, das die Anzahl der Datenbankverbindungen des Benutzers beschränkt.

Alternativ zu dem Befehl CREATE ROLE *name* LOGIN kann auch noch der Befehl CREATE USER verwendet werden. Diese Anweisung wurde in älteren Versionen von PostgreSQL verwendet.

Benutzerattribute anzeigen

Um sich einen Überblick über die Rollen im Datenbanksystem zu verschaffen, können Sie in PostgreSQL alle Rollen und deren Attribute mit dem Befehl "\du" anzeigen. In der Ansicht werden die Rollennamen aufgelistet sowie einige andere Informationen angegeben, z. B. ob die Rolle Superuser ist und ob sie Datenbanken erstellen darf.

Liste der Rollen Attribute		
Rollenname		Mitglied von
fmueller	DB erzeugen	{}
jbossert	Superuser, Rolle erzeugen, DB erzeugen, Replikation, Bypass RLS	{}
postgres	Superuser	{}
pscherhorn		{}

Rollen und deren Attribute anzeigen

Benutzerattribute ändern

Möchten Sie die Benutzer bzw. die Rollen bearbeiten, können Sie mit der Anweisung ALTER ROLE arbeiten. Bei dieser Anweisung können Sie u. a. die gleichen Optionen eingeben wie bei der Anweisung CREATE ROLE. Zusätzlich haben Sie z. B. die Möglichkeit, mit dem Befehl RENAME TO eine Rolle umzubenennen.

Beispiel: BenutzerAendernPostgreSQL.sql

①	ALTER ROLE hschneider CREATEDB;
②	ALTER USER jbossert LOGIN;
③	ALTER ROLE pscherhorn RENAME TO phscherhorn;

- ① Die Rolle *hschneider* kann nun Datenbanken erstellen.
- ② Die Rolle *jbossert* kann sich nun im Datenbanksystem einloggen.
- ③ Die Rolle *pscherhorn* wird umbenannt in *phscherhorn*.

- ✓ Sie können nicht die Rolle ändern, mit der Sie aktuell eingeloggt sind.
- ✓ Wenn Sie eine Rolle umbenennen, die ein Passwort hat, wird dieses gelöscht.

Möchten Sie eine Rolle löschen, verwenden Sie die Anweisung `DROP ROLE rollename`.

13.3 Benutzerverwaltung unter MariaDB

Neue Benutzer anlegen

Seit der Version 10.4.13 verwaltet MariaDB die Benutzernamen in der Tabelle mit dem Namen `global_priv` in der Datenbank `mysql`. Die bis zu dieser Version genutzte Tabelle `user` existiert als Sicht, welche die Tabelle `global_priv` referenziert, weiter. Um einen neuen Benutzer anzulegen, wird ein neuer Datensatz in dieser Tabelle erstellt. Um dies zu erreichen, bietet MariaDB verschiedene Wege an:

- ✓ die Verwendung der speziellen Befehle `CREATE USER` und `GRANT`, die den Server veranlassen, die Tabelle `user` zu bearbeiten;
- ✓ das direkte Einfügen der Datensätze mittels der SQL-Anweisung `INSERT` in die Tabelle `user`.

Die Verwendung der speziellen Befehle ist vorzuziehen, da diese klarer und weniger fehleranfällig sind.

Beim Anlegen des Benutzers müssen Sie neben Benutzernamen zusätzlich Angaben darüber machen, von welchen Computern (Hosts) im Netzwerk der Zugriff erfolgen darf. Die Vergabe eines Passwortes ist nicht zwingend erforderlich, sollte aber aus Gründen der Datensicherheit immer erfolgen. Bei der Verwendung von `GRANT` ist die Angabe eines Passwortes mittels der Option `IDENTIFIED BY` in Abhängigkeit von den Servereinstellungen unter Umständen jedoch notwendig.

Zum Einrichten eines neuen Benutzers benötigen Sie umfassende Rechte. Auf die Datenbank `mysql` haben nur Benutzer mit allen Rechten Zugriff, in der Regel daher nur der Datenbankadministrator.

Beispiel: *BenutzerAnlegenMariaDB.sql*

Im folgenden Beispiel sollen die Benutzer `fmueller` und `pfunke` erstellt werden. Der Benutzer `pfunke` darf nur von dem Computer aus auf die Datenbank zugreifen, auf dem sie installiert ist. Für den Benutzer `fmueller` ist dagegen Zugriff von beliebigen Computern des Netzwerks erlaubt.

```
① CREATE USER 'fmueller'@'%' IDENTIFIED BY 'geheim';
② CREATE USER 'pfunke'@'localhost' IDENTIFIED BY 'geheim';
```

- ① Hier wird der Benutzer `fmueller` erstellt. Dem Datenfeld `host` wird dabei der Platzhalter `%` zugewiesen. Dieser bewirkt, dass der Zugriff von beliebigen Computern im Netzwerk erfolgen kann. Das Passwort wird per Standard verschlüsselt in der Datenbank gespeichert. Dadurch ist das Passwort bei `SELECT`-Abfragen der Tabelle `user` auch für den Administrator nicht lesbar.

- ② Dem Benutzer *pfunke* sind nur Zugriffe vom lokalen Computer aus erlaubt. Deshalb wird dem Feld *host* der Wert *localhost* zugewiesen.

Syntax zum Anlegen neuer Benutzer

```
CREATE USER 'user'@'host' IDENTIFIED BY 'password' [PASSWORD EXPIRE INTERVAL X DAY | PASSWORD EXPIRE NEVER];
```

- ✓ Neue Benutzer legen Sie mit der CREATE USER-Anweisung an.
- ✓ Den Benutzernamen und den Wert für das Datenfeld *host* geben Sie in Apostrophe eingeschlossen und getrennt durch das Zeichen '@' an. Der Hostname bezeichnet den Netzwerknamen des Computers, von dem aus der Zugriff erfolgen soll. Wird dieser nicht angegeben, wird '%' verwendet.
- ✓ Das Passwort für den Benutzer folgt ebenfalls in Apostrophe eingeschlossen nach den Schlüsselwörtern IDENTIFIED BY. Die Verschlüsselung des Passwortes in der Datenbank erfolgt automatisch.
- ✓ Seit der Version 10.4.3 kann über die Option PASSWORD EXPIRE INTERVAL X DAY bzw. PASSWORD EXPIRE NEVER die Dauer der Gültigkeit des Passworts bestimmt werden. Damit wird der für das per Systemvariabler gesetzte Wert für den zu definierende Benutzer überschrieben.

Beispiele für die Angabe des Hostnamens

Wert	Erklärung
%	Alle Computer im Netzwerk
localhost	Lokaler Computer, auf dem der MariaDB-Server installiert ist
db.testserver.de	Computer mit dem Domainnamen db.testserver.de
%.testserver.de	Computer, deren Domain auf testserver.de endet
server	Computer mit dem Netzwerknamen server

Die neuen Benutzer verfügen standardmäßig über keine Rechte. Diese können Sie nachträglich mit der Anweisung GRANT definieren.

Benutzer verwalten

Um Benutzerdaten unter MariaDB anzuzeigen, können Sie die Tabelle *mysql.user* mit einem SELECT-Befehl abfragen, beispielsweise in der Form

```
SELECT user, password FROM mysql.user;
```

Zum Löschen eines Benutzers nutzen Sie den Befehl

```
DROP USER 'user'
```

Zum Ändern vergebener Benutzerrechte empfiehlt es sich, die Befehle GRANT und REVOKE zu verwenden.

Die direkte Bearbeitung der Tabelle `mysql.user` mit dem Befehl

```
UPDATE mysql.user SET datenfeld = wert, ...;
```

ist ebenfalls möglich.

13.4 Zugriffsrechte an Benutzer vergeben

Zugriffsrechte mit **GRANT** definieren

Standardmäßig kann außer dem Datenbankadministrator nur derjenige Benutzer ein Datenbankobjekt bearbeiten, der es auch erstellt hat. Um zusätzliche Zugriffsrechte zu definieren, wird die SQL-Anweisung **GRANT** verwendet.

Mit der Anweisung **GRANT** vergeben Sie Rechte für verschiedene SQL-Anweisungen, die sich auf ein ausgewähltes oder alle Datenbankobjekte beziehen. So können Sie beispielsweise einem Benutzer schreibenden Zugriff auf eine bestimmte Tabelle ermöglichen und einen lesenden Zugriff auf alle Tabellen und Datenbanken. Standardmäßig können Benutzerrechte nur vom Datenbankadministrator vergeben werden.

Beispiel: *Grant.sql*

Im folgenden Beispiel wird davon ausgegangen, dass die Benutzer *fmueller* und *pfunke* angelegt wurden. Dem Benutzer *fmueller* sollen alle Rechte für die Tabelle *t_ma* gewährt werden. Zusätzlich soll er Leserechte und Aktualisierungsrechte für die Tabelle *t_proj* erhalten. Der Benutzer *pfunke* soll für die Tabelle *t_ma* nur Leserechte erhalten.

①	<code>GRANT ALL ON t_ma TO fmueller;</code>
②	<code>GRANT SELECT, UPDATE ON t_proj TO fmueller;</code>
③	<code>GRANT SELECT ON t_ma TO 'pfunke'@'localhost';</code>

- ① Hier werden mithilfe des Schlüsselworts **ALL** alle Rechte für die Tabelle *t_ma* an den Benutzer *fmueller* vergeben.
- ② Für den gleichen Benutzer wird für die Tabelle *t_proj* das Ausführen der **SELECT**- und **UPDATE**-Anweisung gestattet.
- ③ Der Benutzer *pfunke* erhält an dieser Stelle Leserechte für die Tabelle *t_ma*.

Syntax der **GRANT**-Anweisung

<code>GRANT rechteliste ON datenbankobjekt TO benutzername;</code>
--

- ✓ Die Anweisung beginnt mit dem Schlüsselwort **GRANT**.
- ✓ Danach folgt die Angabe des zu vergebenden Rechts. Sollen mehrere Rechte vergeben werden, müssen diese mit Kommata getrennt werden.
- ✓ Nach dem Schlüsselwort **ON** folgt der Name des Datenbankobjekts, für das die Rechte gelten sollen.
- ✓ Am Ende folgt nach dem Schlüsselwort **TO** der Benutzername.

Möglichkeiten für das Vergeben von Rechten

Folgende Rechte können mit der GRANT-Anweisung z. B. vergeben werden:

ALL [PRIVILEGES]	Gewährt alle Rechte für das entsprechende Datenbankobjekt
DELETE	Recht zum Löschen von Datensätzen, Ausführen der DELETE-Anweisung
EXECUTE	Ausführungsrecht für Stored Procedures
INSERT	Recht zum Einfügen neuer Datensätze, Ausführen der INSERT-Anweisung
SELECT	Leserecht, Recht zum Ausführen der SELECT-Anweisung
UPDATE	Recht zum Ändern von Datensätzen, Ausführen der UPDATE-Anweisung

Je nach Datenbanksystem sind viele zusätzliche Rechte möglich. Mehr Informationen erhalten Sie im Hilfesystem des jeweiligen Datenbanksystems.

Für Datenbankobjekte Rechte vergeben

Benutzerrechte können entweder für alle oder für bestimmte Datenbankobjekte vergeben werden, dabei haben Sie z. B. folgende Möglichkeiten:

GRANT rechte ON *.* TO ...	Die betreffenden Rechte gelten für alle Datenbanken und Tabellen.
GRANT rechte ON datenbank.* TO ...	Die Rechte gelten für alle Tabellen der angegebenen Datenbank.
GRANT rechte ON datenbank.tabelle TO ...	Die Rechte gelten für die angegebene Tabelle der angegebenen Datenbank.
GRANT rechte ON * TO ...	Die Rechte gelten für alle Tabellen der aktuellen Datenbank.
GRANT rechte ON tabelle TO ...	Die Rechte gelten für die angegebene Tabelle der aktuellen Datenbank.

Ist die Datenbank des Objekts, dessen Rechte geändert werden sollen, nicht die aktuelle, müssen Sie den Namen der Datenbank, gefolgt von einem Punkt, dem Namen des Objekts voranstellen.

! Wurde der Benutzername, den Sie in der GRANT-Anweisung angeben, vorher nicht angelegt, so wird er durch diese Anweisung definiert. Achten Sie aus diesem Grund auf die richtige Schreibweise des Benutzernamens, da Sie sonst einen neuen Benutzer anlegen und nicht die Rechte des vorhandenen Nutzers ändern.

Privilegien an andere Benutzer weitergeben

Im Datenbanksystem ist der Objekteigentümer derjenige, der das alleinige Recht zur Vergabe von Privilegien hat. Dieses Recht kann er jedoch in der GRANT-Anweisung mithilfe der Option WITH GRANT OPTION an andere weitergeben.

Beispiel: *WithGrantOption.sql*

Der Benutzer *fmueller* soll alle Rechte für die Tabelle *t_ma* besitzen und zusätzlich selber in der Lage sein, Privilegien für diese Tabelle an andere Benutzer zu vergeben. Der Benutzer *pfunke* soll das Privileg erhalten, Daten der Tabelle *t_proj* auszuwählen, und er soll in der Lage sein, das Privileg auch an andere Benutzer der Tabelle weiterzugeben.

```
① GRANT ALL ON t_ma TO fmueller WITH GRANT OPTION;
② GRANT SELECT ON t_proj TO pfunke@localhost WITH GRANT OPTION;
```

- ① Mit der GRANT-Anweisung und der Option ALL werden dem Benutzer an dieser Stelle alle Privilegien für die Tabelle *t_ma* erteilt. Durch die Option WITH GRANT OPTION wird ihm das Recht erteilt, diese Privilegien für die betreffende Tabelle auch an andere zu vergeben.
- ② Mit der GRANT-Anweisung und der Option SELECT wird dem Benutzer an dieser Stelle das Privileg der Auswahl für die Tabelle *t_proj* erteilt. Durch die Option WITH GRANT OPTION wird ihm das Recht erteilt, dieses Privileg für die betreffende Tabelle auch an andere zu vergeben.

Benutzerrechte einsehen

Um die gewährten Benutzerrechte für den aktuell angemeldeten Benutzer unter MariaDB anzuzeigen, können Sie diese mit dem Befehl SHOW GRANTS FOR CURRENT_USER (Kurzform: SHOW GRANTS) abfragen.

```
MariaDB [uebungen]> SHOW GRANTS FOR CURRENT_USER ;
+-----+
| Grants for root@localhost
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY PASSWORD '*81F5E21E35407D884A6CD4A731AEBFB6AF209E1B' WITH GRANT OPTION
| GRANT PROXY ON '%' TO 'root'@'localhost' WITH GRANT OPTION
+-----+
2 rows in set (0.00 sec)
```

Für einen bestimmten Benutzer verwenden Sie entsprechend dessen Namen in der FOR-Klausel, zum Beispiel:

```
SHOW GRANTS FOR fmueller;
```

```
MariaDB [uebungen]> SHOW GRANTS FOR fmueller;
+-----+
| Grants for fmueller@%
+-----+
| GRANT USAGE ON *.* TO 'fmueller'@'%' IDENTIFIED BY PASSWORD '*462366917EEDD1970A48E87D8EF59EB67D2CA26F'
| GRANT SELECT, UPDATE ON `uebungen`.* TO 'fmueller'@'%'
| GRANT ALL PRIVILEGES ON `uebungen`.* TO 'fmueller'@'%' WITH GRANT OPTION
+-----+
3 rows in set (0.00 sec)
```

In PostgreSQL erhalten Sie über den Befehl \dp eine Übersicht der Benutzern gewährten Rechte.

13.5 Benutzern die Zugriffsrechte entziehen

Zugriffsrechte mit REVOKE löschen

Mit der GRANT-Anweisung werden Benutzerrechte in SQL vergeben. Um den Benutzern diese Rechte wieder zu entziehen, verwenden Sie die REVOKE-Anweisung.

Falls ein Benutzer bis auf ein Recht alle anderen Zugriffsrechte erhalten soll, kann es effektiver sein, wenn Sie ihm zuerst mit GRANT ALL alle Rechte gewähren. Mit der REVOKE-Anweisung können Sie danach einzelne Rechte wieder entziehen.

Beispiel: *Revoke.sql*

Dem Benutzer *fmueluer* sollen alle eventuell gewährten Rechte an der Tabelle *t_proj* wieder entzogen werden. Für die Tabelle *t_ma* besitzt der gleiche Benutzer alle Rechte. Damit er jedoch keine Datensätze löschen kann, wird dieses Recht wieder entzogen.

```
① REVOKE ALL ON t_proj FROM fmueluer;  
② REVOKE DELETE ON t_ma FROM fmueluer;
```

- ① Mit der REVOKE-Anweisung und der Option ALL werden dem Benutzer an dieser Stelle alle Rechte für die Tabelle *t_proj* entzogen.
- ② Hier wird dem gleichen Benutzer explizit das Recht zum Löschen von Datensätzen in der Tabelle *t_ma* entzogen.

Wurde einem Benutzer ein Recht von mehreren Instanzen gewährt, behält er das Recht so lange, bis alle Instanzen es ihm entzogen haben.

Syntax der REVOKE-Anweisung

```
REVOKE rechteliste ON datenbankobjekt FROM benutzer;
```

- ✓ Die Anweisung wird mit dem Schlüsselwort REVOKE eingeleitet. Danach folgt das zu entziehende Benutzerrecht.
- ✓ Sollen mehrere Rechte entzogen werden, müssen diese mit Komma getrennt werden. Mit dem Schlüsselwort ALL werden alle Rechte entzogen.
- ✓ Nach dem Schlüsselwort ON wird der Name des Datenbankobjektes angegeben, für das die Rechte entzogen werden sollen.
- ✓ Danach folgen das Schlüsselwort FROM und der Benutzername.

Um Benutzerrechte für einzelne Datenfelder zu entziehen, können Sie nach der Angabe des Benutzerrechts in runden Klammern die Namen der gewünschten Datenfelder angeben. Die Vorgehensweise ist dabei analog zur GRANT-Anweisung.

Die GRANT-Option wieder entziehen

Möchten Sie einem Benutzer ein Privileg entziehen, das er auch an andere Benutzer weitergegeben hat, müssen Sie darauf achten, dass den anderen Benutzern dieses Recht auch entzogen wird. Bei MariaDB funktioniert das automatisch. Bei PostgreSQL müssen Sie die Option CASCADE hinzufügen.

Beispiel: *RevokeGrantOption.sql*

Dem Benutzer *fmueller* soll das Recht, Privilegien für die Tabelle *t_proj* weiterzugeben, wieder entzogen werden.

```
① REVOKE GRANT OPTION FOR ALL ON t_proj FROM fmueller CASCADE;
```

- ① Mit der REVOKE-Anweisung und der Option GRANT OPTION FOR ALL werden dem Benutzer an dieser Stelle alle Rechte für die Tabelle *t_proj* entzogen. Mit der Option CASCADE wird auch allen Benutzern, die dieses Privileg von *fmueller* erhalten haben, das Privileg entzogen.

13.6 Übung

Benutzer in einer Datenbank anlegen

Übungsdatei: --

Ergebnisdatei: *Uebung.SQL*

1. Legen Sie in Ihrer Datenbank einen neuen Benutzer mit dem Namen *umeyer* und einem beliebigen Passwort an.
2. Gewähren Sie diesem Benutzer für die Tabelle *t_ma* Rechte zum Lesen und Aktualisieren von Datensätzen.
3. Melden Sie sich an der Datenbank mit dem neuen Benutzernamen an.
4. Versuchen Sie, einen Datensatz in der Tabelle *t_ma* zu löschen.
5. Entziehen Sie dem Benutzer alle Rechte für die Tabelle *t_ma*.
6. Gewähren Sie dem Benutzer nun alle Rechte für die Tabelle *t_proj*, Leserechte für die Tabelle *t_ma* und das Recht zum Ändern der Abteilungsnummer in der Tabelle *t_ma*.
7. Entziehen Sie dem Benutzer das Recht zum Löschen von Datensätzen in der Tabelle *t_proj*. Prüfen Sie danach, was jetzt mit den anderen Rechten des Benutzers an der Tabelle geschieht.

14

Transaktionsverwaltung

14.1 Konsistente Datenbestände und Transaktionen

Datenbanksysteme erlauben den gleichzeitigen Datenzugriff mehrerer Benutzer. Bei lesendem Zugriff auf die Daten ist dies kein Problem. Wenn jedoch mehrere Benutzer zur selben Zeit die gleichen Datensätze bearbeiten, muss das Datenbanksystem für die Konsistenz der Daten sorgen.

Darüber hinaus muss die Datenkonsistenz auch bei Fehlern gewährleistet werden, die z. B. durch Programmfehler oder Hardwareausfälle auftreten können. Das Datenbanksystem muss daher in der Lage sein, die betroffenen Daten in jedem Fall in den letzten konsistenten Zustand zu überführen. Zu diesem Zweck verwenden die meisten Systeme die sogenannte Transaktionsverwaltung.

Transaktionen

Transaktionen sind eine Gruppe von logisch zusammenhängenden Datenbankoperationen (SQL-Anweisungen), die nur **gemeinsam** ausgeführt werden sollen. Kann auf Grund eines Fehlers, einer Zugriffsverletzung oder Ähnlichem eine der Operationen nicht ausgeführt werden, dann wird keine der Operationen in der Transaktion ausgeführt und der Datenbestand wird in den Ausgangszustand versetzt.

Das Datenbanksystem garantiert bei der Ausführung einer Transaktion die Einhaltung der vier grundlegenden Eigenschaften, die auch als ACID-Eigenschaften bezeichnet werden:

Atomicity – Atomarität 'Alles oder Nichts'	Die Transaktion, die aus einer oder mehreren Operationen bestehen kann, wird entweder vollständig ausgeführt oder gar nicht. Tritt während der Transaktion ein Fehler auf (z. B. Programmfehler, Hardwarefehler oder Betriebssystemabsturz), werden alle bisherigen Operationen der Transaktion rückgängig gemacht.
---	---

Consistency – Konsistenz	Ist die Transaktion abgeschlossen, befindet sich die Datenbank in einem konsistenten Zustand. Dazu tragen die definierten Integritätsbedingungen (Wertebereiche, Schlüsseleigenschaften, Fremdschlüssel usw.) bei, welche die logische Konsistenz sichern. Verletzungen dieser Integritätsbedingungen führen zum Zurücksetzen der Transaktion. Während einer Transaktion können aber zwischenzeitlich inkonsistente Zustände auftreten, die mit einer weiteren Operation wiederhergestellt werden. Solche „verzögerten Integritätsbedingungen“ treten beispielsweise bei der Umbuchung zwischen Konten auf. Sie werden meist am Ende der Transaktion überprüft.
Isolation – Isolation	Transaktionen laufen isoliert ab, mehrere gleichzeitig ablaufende Transaktionen stören und beeinträchtigen sich nicht gegenseitig. Das wird seitens des DBMS durch geeignete Synchronisationsmaßnahmen erreicht. Dies ist eine wichtige Voraussetzung zur Sicherung der Konsistenz der Datenbank.
Durability – Dauerhaftigkeit	Das Ergebnis einer erfolgreichen Transaktion ist dauerhaft (persistent). Das bedeutet, dass vom DBMS sichergestellt wird, dass auch beim Auftreten von Fehlern bei der Übertragung der Transaktionsergebnisse in die Datenbank die Änderungen vollständig durchgeführt werden.

Isolationsebenen (Transaction Isolation Level)

Durch den Transaction Isolation Level wird der Grad der Parallelität von Transaktionen gesteuert. Je höher Sie den Level wählen, desto sicherer können Benutzer sein, dass sie bei einer Leseaktion die aktuellen Daten erhalten, und umso weniger Probleme bezüglich der Konsistenz können auftreten. Je höher Sie aber die Isolationsebenen wählen, desto geringer ist der Durchsatz bei einer großen Zahl gleichzeitiger Zugriffe, was durch die notwendigen zeitweiligen Sperrungen der Daten hervorgerufen wird. Im SQL2-Standard werden vier Transaction Isolation Levels definiert.

Isolationsebene 0: READ UNCOMMITTED	Auf Daten, die von einer Transaktion geschrieben werden, kann mittels einer anderen Transaktion lesend zugegriffen werden. Dabei können zum Teil neue und veraltete Daten gelesen werden = unsauberes Lesen (dirty read). Von der Transaktion werden keine Sperren auf die Datensätze gesetzt.
Isolationsebene 1: READ COMMITTED	Datensätze, die von der Transaktion modifiziert werden, sind gesperrt, Datensätze, die nur gelesen wurden, aber nicht. Dadurch können die Datensätze, die nur gelesen wurden, zwischenzeitlich geändert werden. Das Problem des inkonsistenten Lesens und das sogenannte Phantome-Problem können auftreten.
Isolationsebene 2: REPEATABLE READ	Die Transaktion sperrt alle Datensätze, die von ihr gelesen und geschrieben werden. Nur das Phantome-Problem kann hier noch auftreten.
Isolationsebene 3: SERIALIZABLE	Damit wird sichergestellt, dass in einer Transaktion gelesene Daten bis zum Ende der Transaktion gültig sind. Das Lesen von sogenannten Phantomen wird verhindert. Diese Ebene bietet volle Serialisierbarkeit.

Phantome sind ausgewählte Daten, die einmal von einer Transaktion A gelesen wurden und anschließend durch Schreib-, Änderungs- oder Lösch-Operationen in einer anderen Transaktion B geändert werden. Die gleiche Leseoperation der ersten Transaktion A würde danach eine andere Auswahl von Daten bringen. Transaktion A arbeitet mit veralteten Daten. Beispielsweise werden in Transaktion A alle Artikel ausgewählt, von denen weniger als 1 Stück im Lager ist. In Transaktion B werden 2 dieser Artikel gelöscht. Bei der gleichen Anfrage würde Transaktion A nun 2 Datensätze weniger erhalten. Sie arbeitet nun mit Daten, die eigentlich nicht mehr vorhanden sind.

Über die Anweisung zum Start der Transaktion können das Verhalten und die Sicherheit jeder einzelnen Transaktion eingestellt bzw. beeinflusst werden. Die dafür zur Verfügung stehenden verschiedenen Kombinationen an Einstellungen sind vielfältig. Es werden aber nicht von jedem DBS alle möglichen Einstellungen und Isolationslevels unterstützt.

Beispiel für eine Transaktion: *Transaktionen.sql*

Für eine Aktion sollen alle Artikel, von denen mehr als 200 Stück am Lager sind, um 5 % verbilligt angeboten werden. Die Artikel, von denen zwischen 100 und 200 Stück vorrätig sind, werden um 2 % billiger.

Diese Operationen sind entweder für alle oder für keinen Artikel auszuführen. Eine Wiederholung der Operation würde auch bereits geänderte Preise nochmals ändern.

```
① SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
② UPDATE t_lager SET preis = preis * 0.95 WHERE stueck > 200;
   UPDATE t_lager SET preis = preis * 0.98
      WHERE stueck > 100 and stueck <= 200;
③ COMMIT;
```

- ① Mit dieser Anweisung wird eine neue Transaktion gestartet. Die Option ISOLATION LEVEL REPEATABLE READ steuert das Verhalten des Datenbanksystems während der Transaktion. In diesem Fall wird das Isolationslevel 2 gesetzt.
- ② Hier erfolgen die beiden UPDATE-Anweisungen zum Ändern der Preise in der Tabelle *t_lager*.
- ③ Mit der Anweisung COMMIT wird die begonnene Transaktion bestätigt und die Änderungen werden endgültig ausgeführt.

14.2 Transaktionen erstellen

Transaktionen können mit der START-TRANSACTION- oder der SET-TRANSACTION-Anweisung gestartet werden. Bei der zweiten Version haben Sie die Möglichkeit, mit verschiedenen Klauseln zu steuern, wie sich das DBMS während der Transaktion in Bezug auf Zugriffe anderer Transaktionen auf die gleichen Tabellen verhalten soll. Zusätzlich ist es möglich, nur lesende Zugriffe oder lesende und schreibende Zugriffe für die Anweisungen der Transaktion zuzulassen.

Sie können bei MariaDB und PostgreSQL für das Starten einer Standard-Transaktion die Anweisung BEGIN ohne weitere Parameter verwenden.

```
BEGIN;
```

Syntax der SET-TRANSACTION-Anweisung

```
SET TRANSACTION [transaktionsname] [READ WRITE|READ ONLY]
[WAIT|NO WAIT] [ISOLATION LEVEL SERIALIZABLE| REPEATABLE READ |
READ COMMITTED | READ      UNCOMMITTED];
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern SET TRANSACTION. Danach kann ein Transaktionsname angegeben werden.
- ✓ Über die Schlüsselwörter READ WRITE oder READ ONLY können Sie steuern, für welche Zugriffe die Sperrung der Tabellen für die Operationen während der Transaktion erfolgen soll (nicht bei MariaDB).
- ✓ Dann kann über die Klausel WAIT bzw. NO WAIT das Warteverhalten der Transaktion in Bezug auf andere Prozesse beeinflusst werden (nicht bei MariaDB und PostgreSQL).
- ✓ Optional kann durch die Klausel ISOLATION LEVEL, gefolgt von den entsprechenden Schlüsselwörtern, angegeben werden, welche Isolationsebene für die Transaktion eingestellt werden soll.

Zugriffsart für Tabellen festlegen

Mit den Optionen READ WRITE und READ ONLY können Sie festlegen, ob während der Transaktion alle Zugriffsarten oder nur lesende Zugriffe auf die Daten ausgeführt werden sollen.

READ WRITE	Die Operationen der Transaktion können Daten aus den betreffenden Tabellen lesen und Datensätze in den Tabellen ändern, löschen und neu einfügen.
READ ONLY	Die Operationen der Transaktion dürfen nur Daten aus den Tabellen mit der SELECT-Anweisung lesen.

Die READ-WRITE-Option ist die Standardeinstellung. Diese Angabe kann daher auch entfallen.

Wenn Sie eine Transaktion für schreibende Zugriffe sperren, erzeugt das Ausführen einer INSERT-, UPDATE- oder DELETE-Anweisung eine Fehlermeldung. Wird eine solche Transaktion beispielsweise in einem Anwendungsprogramm eingesetzt, so können Sie damit das unerlaubte oder unbewusste Ändern von Daten durch den Benutzer des Programms unterbinden.

Das Datenbanksystem MariaDB unterstützt die Optionen READ WRITE und READ ONLY nicht. Dafür kann durch die Anweisung LOCK TABLES das Lese- bzw. Schreibrecht für den aktuellen Thread eingeräumt werden. Beispielsweise kann auf die Tabelle *t_lager* nur noch lesend zugegriffen werden, wenn sie durch die Anweisung

```
LOCK TABLE t_lager READ;
```

gesperrt wird. Durch den Aufruf von UNLOCK TABLES wird diese Sperrung wieder aufgehoben.

Warten auf andere Prozesse

Mit den Optionen `WAIT` und `NO WAIT` steuern Sie das Verhalten der Transaktionen, wenn andere Transaktionen gerade auf die gleichen Daten zugreifen.

<code>WAIT</code>	Wird festgestellt, dass eine andere Transaktion auf die gleichen Tabellen zugreift, wird mit den Operationen der Transaktion gewartet, bis alle anderen Zugriffe beendet wurden. Dadurch werden Zugriffskonflikte vermieden.
<code>NO WAIT</code>	Beim gleichzeitigen Zugriff anderer Transaktionen auf die an der Transaktion beteiligten Tabellen wird sofort eine Fehlermeldung erzeugt und die Transaktion wird abgebrochen.

Die Option `WAIT` ist die Standardeinstellung. Sie kann daher auch entfallen.

Die Option `NO WAIT` ist sinnvoll, wenn Sie z. B. bei Lesezugriffen auf eine Tabelle sofort erfahren möchten, dass die Abfrage möglicherweise ungültige und nicht mehr aktuelle Daten liefert.

Unter MariaDB und PostgreSQL sind die Optionen `WAIT` und `NO WAIT` nicht verfügbar.

Isolation Level

Mit der Option `ISOLATION LEVEL` können Sie beim Erstellen einer Transaktion steuern, wie diese von anderen Transaktionen isoliert werden soll. Die Isolation der Daten bei konkurrierenden Transaktionen auf den gleichen Datenbestand ist ein wichtiges Merkmal vieler relationaler Datenbanksysteme.

Bei der Isolation der Daten entsteht das Problem, dass einerseits der parallele Zugriff mehrerer Benutzer unbedingt erwünscht ist, andererseits jedoch die Konsistenz der Daten durch isolierte Transaktionen bewahrt werden soll. Durch eine zu starke Isolation verliert ein Datenbanksystem seine Tauglichkeit für die Nutzung durch sehr viele parallele Benutzerzugriffe. Die meisten Datenbanksysteme verfügen daher über unterschiedliche Implementierungen der Klausel `ISOLATION LEVEL`. Mehr Informationen darüber finden Sie im Hilfesystem Ihrer Datenbank.

Im Folgenden sind einige Klauseln aufgeführt. Diese können in verschiedenen DBS mit unterschiedlichen Schlüsselwörtern definiert sein.

<code>ISOLATION LEVEL READ UNCOMMITTED</code> <i>oder</i> <code>ISOLATION LEVEL SNAPSHOT</code>	Zu Beginn der Transaktion werden die aktuellen Daten kopiert; es wird ein Schnappschuss davon erzeugt. Dieser Schnappschuss dient als Grundlage für die Operationen der Transaktion. Änderungen, die durch andere gleichzeitig laufende Transaktionen durchgeführt werden, sind im Datenbestand dieser Transaktion nicht sichtbar (Isolationsebene 0).
---	--

ISOLATION LEVEL READ COMMITTED	Bei Angabe dieser Klausel können im Verlauf der Transaktion auch Änderungen durch andere Transaktionen stattfinden, wenn diese mit COMMIT abgeschlossen wurden. Für die Transaktion sind diese Änderungen sichtbar (Isolationsebene 1).
ISOLATION LEVEL REPEATABLE READ oder ISOLATION LEVEL SNAPSHOT TABLE STABILITY	Bei diesem Isolation Level wird sichergestellt, dass die an der Transaktion beteiligten Tabellen für Schreibzugriffe anderer Transaktionen gesperrt sind. Andere Transaktionen können aber trotzdem daraus lesen (Isolationsebene 2).
ISOLATION LEVEL SERIALIZABLE	Dieser Isolation Level gewährt die volle Serialisierbarkeit (Isolationsebene 3).

In einigen SQL-Dialektken wird für die Wahl des ISOLATION LEVEL nur die entsprechende Nummer angegeben.

Meist ist der niedrigste Isolation Level die Standardeinstellung. Sie ist für viele Anwendungen mit wenigen gleichzeitigen Benutzern ausreichend.

Mehr Aufwand bei der Isolation der Daten müssen Sie betreiben, wenn Sie ein Datenbanksystem mit vielen Tausend gleichzeitig arbeitenden Benutzern betreiben. Dann leidet die Performance der Datenbank unter einem ungeeigneten Isolation Level.

Besonderheiten bei MariaDB

Um Transaktionen verwenden zu können, müssen Sie in MariaDB den Tabellentyp InnoDB verwenden. Dieser ist mit Ausnahme der Systemtabellen der Standardtabellentyp.

Welche Tabellentypen von dem aktuell gestarteten MariaDB-Server unterstützt werden, erfahren Sie durch Aufruf der SQL-Anweisung SHOW ENGINES in der MariaDB-Konsole.

Engine	Support	Comment	Transactions	XA	Savepoints
CSV	YES	Stores tables as CSV files	NO	NO	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
Aria	YES	Crash-safe tables with MyISAM heritage. Used for internal temporary tables and privilege tables	NO	NO	NO
MyISAM	YES	Non-transactional engine with good performance and small data footprint	NO	NO	NO
SEQUENCE	YES	Generated tables filled with sequential values	YES	NO	YES
DEFAULT	YES	Supports transactions, row-level locking, foreign keys and encryption for tables	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO

Der MariaDB-Tabellentyp MyISAM unterstützt keine Transaktionen, da dieser Tabellentyp kein Sperren auf Datenebene unterstützt.

Um Transaktionen unter MariaDB benutzen zu können, stellen Sie sicher, dass bei Installation des MariaDB-Servers die Unterstützung des InnoDB-Tabellentyps nicht durch den Eintrag skip-innodb ausgeschaltet wurde bzw. unter Linux das configure-Skript mit dem zusätzlichen Parameter --with-innodb ausgeführt wurde.

Wenn Sie eine ältere Version des Ursprungssystems von MariaDB MySQL verwenden sollten, können Sie explizit Tabellen vom Typ InnoDB erstellen.

- ▶ Erstellen Sie in einer Datenbank eine Tabelle vom Typ InnoDB.

```
CREATE TABLE tabellenname (
    feldname_1 datentyp
    ...
    feldname_n datentyp(länge),
    PRIMARY KEY (feldname) TYPE=InnoDB;
```

oder Haben Sie nach einem System-Update noch Tabellen in Verwendung, die mit einer älteren Version erstellt wurden, wandeln Sie eine vorhandene MyISAM-Tabelle mit der SQL-Anweisung ALTER TABLE in eine Tabelle vom Typ InnoDB um.

```
ALTER TABLE tabellenname TYPE=INNODB;
```

! Wandeln Sie keine MariaDB-Systemtabellen (MyISAM-Tabellentyp) in den Tabellentyp InnoDB um, da sonst Ihr MariaDB-System nicht mehr gestartet werden kann.

14.3 Transaktionen abschließen

Mit der Anweisung COMMIT schließen Sie eine Transaktion ab. Alle Änderungen an den Daten werden daraufhin endgültig durchgeführt.

- ✓ Alle Änderungen werden fest in die Datenbank geschrieben.
- ✓ Die neuen Daten werden für andere Prozesse sichtbar gemacht.
- ✓ Alle offenen Cursors werden geschlossen.

Auch wenn in der Transaktion nur SELECT-Abfragen verwendet wurden, sollten Sie die Transaktion mit COMMIT beenden, da eventuelle andere Prozesse auf den Zugriff warten.

Syntax der COMMIT-Anweisung

```
SET TRANSACTION ...;
...
COMMIT [TRANSACTION transaktionsname];
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort COMMIT.
- ✓ Einige Datenbanksysteme erwarten danach das Schlüsselwort TRANSACTION.
- ✓ Wurde beim Erstellen einer Transaktion ein Name angegeben, so kann gezielt diese Transaktion beendet werden. Standardmäßig wird die aktuelle Transaktion abgeschlossen.

14.4 Transaktionen zurücksetzen

Transaktionen erlauben das Zurücknehmen aller Änderungen an den Daten bis zum Zustand vor der Transaktion. Dies erreichen Sie mit der ROLLBACK-Anweisung. Die Anweisung nimmt stets alle Anweisungen in ihrer Gesamtheit zurück. Es ist nicht möglich, einzelne Änderungen zu verwerfen. Wie mit der COMMIT-Anweisung wird auch mit ROLLBACK die Transaktion beendet.

Falls das Datenbanksystem aufgrund eines Fehlers abstürzt, wird beim Neustart in der Regel automatisch ein ROLLBACK ausgeführt und alle noch offenen Transaktionen werden verworfen.

Beispiel: *Rollback.sql*

Es sollen alle Artikel aus dem Datenbestand gelöscht werden, die nicht mehr am Lager sind. Dazu will ein Mitarbeiter aus der Tabelle `t_lager` alle Artikel löschen, deren Stückzahl kleiner ist als 1. Durch einen Eingabefehler löscht er jedoch alle Artikel mit einer Stückzahl größer als 1. Mit der Anweisung ROLLBACK wird die Änderung rückgängig gemacht.

```

① BEGIN;
② SELECT COUNT(id) FROM t_lager WHERE stueck < 1;
③ DELETE FROM t_lager WHERE stueck > 1;
④ ROLLBACK;
```

- ① Hier wird die Transaktion gestartet.
- ② Die Abfrage ermittelt, wie viele Artikel gelöscht werden müssen.
- ③ Mit der DELETE-Anweisung sollen die betreffenden Artikel gelöscht werden. Durch einen Eingabefehler wird aus der korrekten Bedingung `stueck < 1` jedoch fälschlicherweise `stueck > 1`. Dadurch werden alle Daten der gelagerten Artikel gelöscht.
- ④ Die Anweisung ROLLBACK macht das Löschen der Artikel wieder rückgängig, indem alle Änderungen der Anweisungen innerhalb der Transaktion zurückgesetzt werden.

In einer realen Anwendung mit grafischer Benutzeroberfläche können durch die Schaltflächen *OK* und *Abbrechen* die entsprechenden Aktionen bestätigt (*OK* – COMMIT) oder rückgängig (*Abbrechen* – ROLLBACK) gemacht werden.

Syntax der ROLLBACK-Anweisung

```

SET TRANSACTION . . . ;
. . .
ROLLBACK [TRANSACTION transaktionsname];
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort ROLLBACK.
- ✓ Falls beim Erstellen einer Transaktion ein Name angegeben wurde, kann mithilfe des Schlüsselworts TRANSACTION, gefolgt vom gewünschten Namen, eine Transaktion gezielt zurückgenommen werden.

14.5 Übung

Mit Transaktionen arbeiten

Übungsdatei: --

Ergebnisdateien: *Uebung1-Client1.sql*,
Uebung1-Client2.sql, *Uebung2.sql*

1. Starten Sie zweimal den Datenbank-Client psql (PostgreSQL) bzw. mysql (MariaDB). Wechseln Sie jeweils zur Datenbank *Uebungen*. Starten Sie in beiden Clients eine neue Transaktion. Führen Sie in beiden Clients eine SELECT-Abfrage aus, um alle Datensätze der Tabelle *t_lager* zu ermitteln. Vergleichen Sie die beiden Ergebnisse. Fügen Sie nun in einem Client in die Tabelle *t_lager* einen neuen Artikel mit der ID-Nummer 55, der Stückzahl 30, dem Preis 25 und der Bezeichnung 'A5 Ordner' ein. Führen Sie in beiden Clients wiederum die Abfrage für alle Datensätze der Tabelle aus. Was stellen Sie fest? Schließen Sie die erste Transaktion nach dem Einfügen der Daten ab und wiederholen Sie den vorherigen Schritt. Vergleichen Sie die Ergebnisse. Wird der neue Datensatz in dem anderen Client noch nicht angezeigt, führen Sie dort ebenfalls ein COMMIT aus und wiederholen Sie die Anzeige der Tabelle.
2. Starten Sie in der Datenbank *Uebungen* eine neue Transaktion. Löschen Sie alle Mitarbeiter, die aus Berlin kommen, aus der Tabelle *t_ma*. Führen Sie eine SELECT-Abfrage auf diese Tabelle aus, um zu prüfen, ob die Datensätze gelöscht wurden. Verwerfen Sie die Änderungen der Transaktion. Führen Sie die SELECT-Abfrage erneut aus und überprüfen Sie das Ergebnis.

15

Stored Procedures

15.1 Programmabläufe speichern

Beim Einsatz von Datenbanksystemen spielt der Datenbankserver eine herausragende Rolle. Er arbeitet am zentralen Speicherort der Daten, verwaltet die Daten dort und liefert auf Anfragen die entsprechenden Ergebnisse. Durch diese zentrale Datenhaltung wird ein kontrollierter Zugang zu den Daten gewährleistet.

Damit der Server seiner Aufgabe als zentrale Verarbeitungseinheit gerecht wird, sollten alle aufwendigen Arbeiten mit den Daten direkt auf dem Server durchgeführt werden. Die Clients erhalten nur die Ergebnisdaten und werden dadurch entlastet, dass sie keine Berechnungen durchführen müssen.

Für diesen Zweck ist es notwendig, dass ganze Abläufe von SQL-Anweisungen auf dem Datenbankserver ausgeführt werden und auch die Bearbeitung möglicher Zwischenschritte hier erfolgt. In SQL können Sie dafür sogenannte Prozeduren oder Stored Procedures (gespeicherte Prozeduren) einsetzen, mit denen Programmabläufe vereinfacht und automatisiert werden können.

In der Praxis werden die Daten oft nicht nur auf einem Datenbankserver gehalten, sondern z. B. durch Replikation (Daten in mehreren Datenbanken auf dem gleichen Stand halten) und Shadowing (Kopie der Datenbank) auf mehreren Servern verteilt. Außerdem können Unternehmensdaten auf mehrere Server verteilt sein (z. B. Filialen einer Sparkasse), sodass es keinen gemeinsamen Datenbestand gibt.

Prozeduren auf dem Server speichern

Stored Procedures sind auf dem Server gespeicherte **Prozeduren** (Unterprogramme), in denen neben den üblichen SQL-Anweisungen auch zusätzliche Befehle zur Ablaufsteuerung und Auswertung von Bedingungen zur Verfügung stehen. Sie sind damit den Makros mancher Anwendungsprogramme vergleichbar, mit denen Sie Programmabläufe automatisieren können.

Beispiel

Bei der Arbeit mit einem Datenbanksystem werden meist verschiedene Daten abgefragt und verarbeitet. Bei der Verarbeitung werden aber oft die gleichen Arbeitsabläufe, jedoch mit unterschiedlichen Werten und/ oder Parametern ausgeführt. Diese Arbeitsabläufe können in Stored Procedures zusammengefasst werden. Bei einer Banküberweisung werden z. B. der Betrag und die Informationen zu den beiden Konten benötigt. Sie können diese vom Client (Bankautomat) an eine Stored Procedure auf dem Server übergeben. Diese führt die notwendigen Operationen aus und liefert als Rückgabewert eine Information zum Erfolg oder Misserfolg der Überweisung.

Höhere Verarbeitungsgeschwindigkeit

Ein wichtiger Vorteil der Stored Procedures liegt in einem Geschwindigkeitsgewinn bei der Ausführung umfangreicher Datenbearbeitungen. Dieser liegt vor allem darin begründet, dass die Prozeduren auf dem Server ausgeführt werden und daher direkten Zugriff auf die gespeicherten Daten erhalten. Ein oft sehr zeitaufwendiger Transport der Daten über ein Netzwerk entfällt. Je langsamer das Netzwerk ist und je größer der Datenbestand, umso deutlicher wird dabei der Gewinn an Geschwindigkeit. Die Verringerung der Datenmenge bei der Datenübertragung über das Netzwerk kann bei vielen aktiven Client-Anwendungen sehr bedeutsam sein.

Bessere Systemsicherheit

Neben der höheren Geschwindigkeit spricht auch eine bessere Sicherheit des Datenbanksystems für den Einsatz von Stored Procedures. Mit ihrer Hilfe ist es möglich, den Anwender einer Datenbank von vielen systemrelevanten Daten abzuschirmen. Er erfährt nicht die Namen der Tabellen und deren Struktur und er erhält keinen Zugriff auf möglicherweise sensible Daten, die bei der Verarbeitung durch gespeicherte Prozeduren verwendet werden.

Thin Clients

Um Updates und Programmänderungen zu vereinfachen, wird die Funktionalität der Anwendungen vom Client mehr auf den Server verlagert. Die Thin Clients (thin = dünn) besitzen dann nur wenig Funktionalität, z. B. eine Benutzeroberfläche zur Darstellung der Resultate. Auf dem Server werden alle gemeinsam genutzten Funktionen, die sogenannte Geschäftslogik (Business Logic), bereitgestellt. Sind nun beispielsweise Updates dieser Funktionen notwendig, müssen diese nur zentral auf dem Server aktualisiert werden (bei gleichbleibenden Schnittstellen), anstatt auf vielen Tausend Clients.

Durch die Kapselung von SQL-Anweisungen und Verarbeitungsschritten in einzelne Prozeduren wird zusätzlich die Wartung der Programmcodes vereinfacht. Für den Anwender sind nur der Aufruf einer Stored Procedure und deren Rückgabewerte von Bedeutung. Die eigentlichen Anweisungen können unabhängig davon optimiert und verändert werden.

Ausführbare und selektierbare Prozeduren

Die Implementation der Stored Procedures ist bei jedem Datenbanksystem verschieden. Manche Datenbanksysteme unterstützen neben den ausführbaren Prozeduren als Besonderheit selektierbare Prozeduren.

Ausführbare Prozeduren (CALL-Prozedur)	Sie werden aufgerufen, um Änderungen an den Datenbeständen vorzunehmen. In der Regel erfolgt die Rückgabe eines Statuswertes, der über Erfolg oder Misserfolg der Aktion informiert. Es können auch mehrere Werte zurückgeliefert werden, um z. B. die Ergebnisse einer Berechnung zu übermitteln.
Selektierbare Prozeduren (SELECT-Prozedur)	Diese Prozeduren liefern eine Ergebnismenge als Resultat und können in einer SELECT-Abfrage wie eine Tabelle verwendet werden. Sie können auch mit Tabellen, Views oder anderen Stored Procedures verknüpft werden.

- ✓ Das Datenbanksystem PostgreSQL unterstützt keine Stored Procedures im klassischen Sinne. Es bietet jedoch eine breite Unterstützung zur Programmierung von benutzerdefinierten Funktionen. Dazu stellt PostgreSQL in der Standardimplementierung vier prozedurale Sprachen bereit: PL/pgSQL, PL/Tcl, PL/Perl und PL/Python. Sie können in diesen Sprachen mit der Anweisung CREATE FUNCTION in PostgreSQL Prozeduren entsprechend umschreiben. Weitere Informationen finden Sie im Handbuch von PostgreSQL.
- ✓ Jedes Datenbanksystem, das Stored Procedures unterstützt, besitzt eine eigene Implementierung von Stored Procedures, die sich in der Deklaration und Verwendung der Anweisungen unterscheidet. Die grundsätzliche Arbeitsweise und Verwendung ist jedoch bei allen gleich.
- ✓ Im Folgenden werden Stored Procedures mit MariaDB erstellt und verwendet.

15.2 Stored Procedures erstellen und bearbeiten

Prozedur definieren

Bevor Sie eine Stored Procedure verwenden können, muss sie in einer Datenbank definiert werden. Dies geschieht nach dem Verbinden mit der Datenbank mit der Anweisung CREATE PROCEDURE. Bei der Deklaration werden neben einem Prozedurnamen und den eigentlichen Anweisungen auch die Übergabe- und Rückgabeparameter festgelegt.

Zur besseren Unterscheidung von anderen Elementen der Datenbank kann dem Namen der Prozedur das Präfix *p_* vorangestellt werden.

Beispiel: *EinfacheProzedur.sql*

Es sollen alle Artikelnamen mit den Namen der dazugehörigen Lieferanten ausgewählt werden. Hierzu müssen die Artikelnamen aus der Tabelle *t_artikel* und die Lieferantennamen aus der Tabelle *t_liefer* ermittelt und verknüpft werden.

```

① DELIMITER //
② CREATE PROCEDURE p_artikel_liefer ()
③ BEGIN
④ SELECT t_artikel.name , t_liefer.name
  FROM t_artikel INNER JOIN t_liefer ON
  t_artikel.lieferant=t_liefer.id;
⑤ END //
⑥ DELIMITER ;
⑦ CALL p_artikel_liefer ();

```

- ① Innerhalb der Prozedur werden SQL-Anweisungen wie üblich mit einem Semikolon abgeschlossen. Damit die Eingabe an dieser Stelle aber nicht fälschlicherweise abgebrochen wird, muss mit dem Befehl **DELIMITER** ein neues Endekennzeichen für SQL-Anweisungen definiert werden. Hier ist es das Zeichen **//**.
- ② An dieser Stelle beginnt die Definition der Stored Procedure mit dem Namen **p_artikel_liefer**. Danach werden in runden Klammern die Namen der Übergabeparameter und deren Datentypen angegeben. Unter diesen Namen können die übergebenen Werte im Verlauf der Prozedur angesprochen werden. Im Beispiel sind keine Übergabeparameter notwendig.
- ③ Einzelne Blöcke werden stets durch das Schlüsselwort **BEGIN** eingeleitet und mit **END** abgeschlossen.
- ④ Hier beginnt der Anweisungsblock der Prozedur. Mithilfe einer **SELECT**-Abfrage werden die Artikel und die dazugehörigen Lieferantennamen aus den entsprechenden Tabellen ausgewählt.
- ⑤ Die Deklaration der Prozedur endet mit **END**. Die Anweisung muss entsprechend der Neudefinition des Endekennzeichens (①) mit dem Zeichen **//** abgeschlossen werden.
- ⑥ Nun kann als Endekennzeichen wieder das Semikolon definiert werden. Achten Sie dabei auf das Leerzeichen zwischen dem Befehl **DELIMITER** und dem Semikolon.
- ⑦ Mit der Anweisung **CALL** wird die Prozedur ausgeführt.

Syntax einfacher Prozeduren

```

CREATE PROCEDURE prozedurname ([ IN | OUT | INOUT] parameter1
datentyp1, ...)
  BEGIN
    anweisungsblock
  END;

```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern **CREATE PROCEDURE**. Danach folgt der Prozedurname.
- ✓ In runden Klammern werden die einzelnen Übergabeparameter der Prozedur angegeben. Bei mehreren Parametern müssen diese durch Komma getrennt werden. Für jeden Parameter muss der gewünschte Datentyp angegeben werden.
- ✓ Der Vorgabewert für Parameter ist der Eingabeparameter **IN**. Das bedeutet, die Daten können von der Prozedur nur ausgelesen werden. Der Ausgabeparameter **OUT** überibt Werte an die **CALL**-Anweisung. Der **INOUT**-Parameter kann beides. Um den Vorgabewert zu ändern, setzen Sie das Schlüsselwort **OUT** oder **INOUT** vor den Parameternamen.
- ✓ In einem durch **BEGIN** und **END** eingeschlossenen Anweisungsblock folgt die Liste der Anweisungen, die von der Prozedur ausgeführt werden sollen.

Endekennzeichen festlegen

Die einzelnen Anweisungen der Stored Procedure müssen, wie alle SQL-Anweisungen, mit einem Semikolon als Endekennzeichen abgeschlossen werden. Erfolgt die Eingabe der Prozedur unter MariaDB mit einem Standard-Client, wird das Semikolon jedoch fälschlich als Ende der gesamten Prozedurdefinition interpretiert. Um dies zu vermeiden, können Sie mit der Anweisung `DELIMITER endekennzeichen` temporär ein anderes Zeichen als Endekennzeichen (als sogenannten Terminator) festlegen.

Verwenden Sie als Endekennzeichen ein Zeichen, das im Normalfall in keiner SQL-Anweisung vorkommt.

Damit das Semikolon wieder als Endekennzeichen interpretiert wird, müssen Sie die Anweisung `DELIMITER ' ; '` nach der Definition der Stored Procedure erneut aufrufen.

Syntax

```
DELIMITER endekennzeichen
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort `DELIMITER`. Danach folgt das gewünschte Zeichen, das ab diesem Zeitpunkt als Endekennzeichen verwendet werden soll.
- ✓ Beim Beenden der Anweisung muss stets das aktuelle Endekennzeichen verwendet werden, standardmäßig das Semikolon.

Anweisungsblöcke definieren

Bei der Definition der Stored Procedure wird mit den Anweisungen `BEGIN` und `END` ein Anweisungsblock definiert. Die Anweisungen werden auch im Verlauf der Prozedur benötigt, um zusammengehörige Anweisungen zu einem Block zusammenzufassen.

- ✓ In allen Anweisungen, die der Programmsteuerung dienen, beispielsweise einer Alternative (`IF... THEN... ELSE...`), können eine oder mehrere SQL-Anweisungen ausgeführt werden. Sind mehrere Anweisungen erforderlich, müssen diese durch die Schlüsselwörter `BEGIN` und `END` eingeschlossen werden. Es wird dann von einem Anweisungsblock gesprochen.
- ✓ Anweisungsblöcke können beliebig ineinander verschachtelt werden.

Syntax

```
BEGIN
    anweisung1;
    ...
END
```

- ✓ Ein Anweisungsblock wird mit dem Schlüsselwort `BEGIN` eingeleitet und mit `END` abgeschlossen. Dazwischen können beliebige Anweisungen angegeben werden.

Vorhandene Prozeduren anzeigen

Mit der Anweisung SHOW PROCEDURE STATUS erhalten Sie eine Auflistung der vorhandenen Stored Procedures der Datenbank in Tabellenform. Da die Tabellenform für diese Anzeige etwas unübersichtlich ist, können Sie auch die Listenform wählen, indem Sie SHOW PROCEDURE STATUS \G; eingeben. Zusätzlich können Sie die Anzeige durch eine WHERE-Klausel auf eine Datenbank beschränken.

```
MariaDB [uebungen]> SHOW PROCEDURE STATUS WHERE db="uebungen" \G;
***** 1. row *****
      Db: uebungen
      Name: p_artikel_liefer
      Type: PROCEDURE
      Definer: root@localhost
     Modified: 2021-06-10 17:16:32
      Created: 2021-06-10 17:16:32
Security_type: DEFINER
      Comment:
character_set_client: latin1
collation_connection: latin1_swedish_ci
Database Collation: latin1_swedish_ci
1 row in set (0.044 sec)
```

Anzeige der definierten Stored Procedures in Listenform

Syntax

```
SHOW PROCEDURE STATUS;
```

- ✓ Die Anweisung besteht aus den Schlüsselwörtern SHOW PROCEDURE STATUS. Als Resultat liefert sie eine Liste der definierten Prozeduren.

Mit der Anweisung SHOW CREATE PROCEDURE *prozedurname* können Sie sich den Code einer bestimmten Prozedur anzeigen lassen.

Prozeduren löschen

Wenn eine Prozedur nicht wunschgemäß arbeitet oder wenn sie nicht mehr benötigt wird, können Sie die Prozedur löschen. Dies geschieht mit der Anweisung DROP PROCEDURE.

Eine Prozedur kann nur dann gelöscht werden, wenn Sie momentan nicht ausgeführt wird und andere Stored Procedures nicht darauf aufbauen.

Syntax

```
DROP PROCEDURE prozedurname;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern DROP PROCEDURE. Danach folgt der Name der zu löschen Prozedur.

Prozeduren bearbeiten

Eine umfangreiche Stored Procedure arbeitet selten von Anfang an ohne Fehler. Statt eine Prozedur im Falle eines Fehlers zu löschen und neu zu erstellen, können Sie die Prozedur mit der Anweisung ALTER PROCEDURE bearbeiten. Auf diese Weise können Sie eine Prozedur schrittweise weiterentwickeln.

Mithilfe der Anweisung ALTER PROCEDURE können Sie den Anweisungsteil der Prozedur bearbeiten, ohne Übergabe- und Rückgabeparameter zu verändern. Die Syntax entspricht dabei der Anweisung CREATE PROCEDURE.

Syntax

```
ALTER PROCEDURE prozedurname ([ IN | OUT | INOUT] parameter1
datentyp1, ...)
    BEGIN
        anweisungsblock
    END;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern ALTER PROCEDURE. Danach folgt der Prozedurname.
- ✓ In runden Klammern werden die einzelnen Übergabeparameter der Prozedur angegeben.
- ✓ In einem durch BEGIN und END eingeschlossenen Anweisungsblock folgt die Liste der Anweisungen, die von der Prozedur ausgeführt werden sollen.

Um bei umfangreicheren Prozeduren den Überblick nicht zu verlieren, ist es praktischer, diese mithilfe eines Editors zu erstellen und dann über die Zwischenablage zu kopieren.

15.3 Beispielanwendung für Stored Procedures

Eine komplexe Prozedur erstellen

In einer Stored Procedure können Sie die SQL-Anweisungen SELECT, INSERT, DELETE und UPDATE verwenden. Zusätzlich existieren noch Anweisungen zur Steuerung des Programmablaufs, die nur innerhalb einer Prozedur erlaubt sind. Auch Variablen können für den internen Gebrauch in einer Prozedur definiert werden. Hier ein Beispiel:

Beispiel: *CreateProcedure.sql*

Es soll ein neuer Mitarbeiter in die Tabelle *t_ma* eingefügt werden. Dieser Mitarbeiter wird gleichzeitig einem Projekt zugeordnet. Dieses Projekt ist entweder bereits in der Tabelle *t_proj* enthalten oder die Prozedur erstellt einen neuen Eintrag. Danach soll die Prozedur in der Tabelle *t_ma_pro* einen neuen Eintrag mit dem Mitarbeiter und dem zugehörigen Projekt erstellen.

```
① DELIMITER //
② CREATE PROCEDURE p_neuer_ma
    (name VARCHAR(50), vorname VARCHAR(50), strasse VARCHAR(150),
     hnr VARCHAR(5), plz VARCHAR(5), ort VARCHAR(50), land
     VARCHAR(4),
     projektname VARCHAR(50))
③ BEGIN
④     DECLARE mid INTEGER;
     DECLARE pid INTEGER;
```

```

⑤   SELECT MAX(id) + 1 FROM t_ma INTO mid;
⑥   INSERT INTO t_ma (id, name, vname, str, hnr, plz, ort, land)
      VALUES(mid, name, vorname, strasse, hnr, plz, ort, land);
⑦   SELECT COUNT(id) FROM t_proj WHERE t_proj.name = projektname
      INTO pid;
⑧   IF (pid = 0) THEN
      SELECT MAX(id) + 1 FROM t_proj INTO pid;
      INSERT INTO t_proj (id, name) VALUES(pid, projektname);
    ELSE
      SELECT id FROM t_proj WHERE t_proj.name = projektname
      INTO pid;
    END IF;
⑨   INSERT INTO t_ma_proj (ma_id, proj_id)
      VALUES(mid, pid);
⑩ END //
⑪ DELIMITER ;
⑫ CALL p_neuer_ma
      ('Herold', 'Frank', 'Waldstr.', '24', '10115', 'Berlin', 'D',
       Kundenumfrage');
⑬ CALL p_neuer_ma
      ('Teschau', 'Ines', '', '', '1210', 'Wien', 'AT',
       Kundenumfrage');

```

- ① Innerhalb der Prozedur werden SQL-Anweisungen wie üblich mit einem Semikolon abgeschlossen. Damit die Eingabe an dieser Stelle aber nicht fälschlicherweise abgebrochen wird, muss mit dem Befehl DELIMITER ein neues Endekennzeichen für SQL-Anweisungen definiert werden. Hier ist es das Zeichen //.
- ② An dieser Stelle beginnt die Definition der Stored Procedure mit dem Namen *p_neuer_ma*. Danach werden in runden Klammern die Namen der Übergabeparameter und deren Datentypen angegeben. Unter diesen Namen können die übergebenen Werte im Verlauf der Prozedur angesprochen werden.
- ③ Einzelne Blöcke werden stets durch das Schlüsselwort BEGIN eingeleitet und mit END abgeschlossen. Hier beginnt der Anweisungsblock der Prozedur.
- ④ Zuerst werden die lokalen Variablen *mid* und *pid* für den Gebrauch innerhalb der Prozedur definiert.
- ⑤ Mithilfe einer SELECT-Abfrage wird die größte Mitarbeiter-ID ermittelt und um den Wert 1 erhöht. Das Ergebnis wird mithilfe des Schlüsselwortes INTO in der Variablen *mid* gespeichert.
- ⑥ Die eben ermittelte Mitarbeiter-ID wird verwendet, um einen neuen Datensatz in der Tabelle *t_ma* zu erzeugen. Dabei werden zusätzlich die Übergabeparameter der Prozedur verwendet.
- ⑦ Die Abfrage ermittelt, ob es für das Projekt *projektname* einen Datensatz in der Tabelle *t_proj* gibt. Der Spaltenname *name* muss in der WHERE-Klausel mit dem Tabellennamen *t_proj* qualifiziert werden, damit ihn das System vom gleichnamigen Parameter *name* unterscheidet. Das Ergebnis der Abfrage wird in der Variablen *pid* gespeichert. Wird ein Datensatz gefunden, ist der Wert von *pid* gleich 1, sonst 0.

- ⑧ Mit einer IF-Anweisung wird an dieser Stelle überprüft, ob `pid` den Wert 0 enthält. In diesem Fall muss ein neuer Datensatz erzeugt werden. Dafür wird mit einer Abfrage eine neue Projekt-ID ermittelt und der Datensatz wird eingefügt. Existiert schon ein passender Datensatz, wird im ELSE-Zweig die entsprechende Projekt-ID in einer Abfrage ausgelesen und in der Variablen `pid` gespeichert.
- ⑨ Mit einer INSERT-Anweisung für die Tabelle `t_ma_proj` wird der Mitarbeiter mit dem entsprechenden Projekt in Bezug gesetzt. Dafür werden die entsprechenden ID-Nummern in den Variablen `pid` und `mid` verwendet.
- ⑩ Die Deklaration der Prozedur endet mit END. Die Anweisung muss entsprechend der Neudefinition des Endekennzeichens (①) mit dem Zeichen // abgeschlossen werden.
- ⑪ Nun kann als Endekennzeichen wieder das Semikolon definiert werden.
- ⑫ Mit der Anweisung CALL wird die Prozedur ausgeführt. Hier wird ein neuer Mitarbeiter eingefügt, der am Projekt *Kundenumfrage* beteiligt ist. Da dieses Projekt noch nicht existiert, wird ein neuer Datensatz in der Tabelle `t_proj` erzeugt.
- ⑬ Da das Projekt *Kundenumfrage* bereits existiert, wird kein neuer Datensatz für dieses Projekt angelegt.

Variablen in Stored Procedures

Variablen dienen zur Zwischenspeicherung von Werten, die Sie der Variablen zuweisen und später wieder auslesen können. In einer Prozedur sind verschiedene Variablentypen möglich.

! Achten Sie bei der Definition von Variablen darauf, dass der Name, den Sie für die Variable wählen, nicht schon für eine Spalte, mit der Sie arbeiten möchten, vergeben ist. Eine Namensgleichheit kann zu Fehlfunktionen führen.

Eine häufig verwendete Form von Variablen sind die lokalen Variablen (im Beispiel *Create-Procedure.sql* Schritt ④). Sie werden zu Beginn der Prozedurdefinition nach dem Schlüsselwort BEGIN mit der Anweisung DECLARE deklariert. Sie sind nur innerhalb der Prozedur gültig.

Syntax

```
DECLARE name datentyp [DEFAULT wert] ;
```

- ✓ Die Deklaration der Variablen erfolgt in der Prozedurdefinition direkt nach dem Schlüsselwort BEGIN.
- ✓ Die Anweisung beginnt mit dem Schlüsselwort DECLARE. Danach folgt der gewünschte Name der Variablen.
- ✓ Anschließend muss ein Datentyp angegeben werden, der für die zu speichernden Daten passend ist. Hier sind alle Datentypen des Datenbanksystems erlaubt. Ein Standardwert kann über die DEFAULT-Anweisung angegeben werden.
- ✓ Lokale Variable gelten immer nur innerhalb des BEGIN ... END-Blocks, in dem sie deklariert wurden.
- ✓ Variablen können auch in verschachtelten Blöcken verwendet werden.

Ergebnisse einer Abfrage mit **SELECT INTO** in Variablen speichern

Um die Ergebnisse einer Abfrage in Variablen speichern zu können, wird die SELECT-Abfrage um das Schlüsselwort **INTO** erweitert (im Beispiel *CreateProcedure.sql* Schritt ⑦). Danach können Sie für jedes Datenfeld eine Variable angeben, in der der entsprechende Wert gespeichert werden soll.

Syntax

```
SELECT ... INTO variable1, ...;
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort **SELECT**. Danach folgen die für die Abfrage benötigten Angaben.
- ✓ Am Ende der Abfrage werden das Schlüsselwort **INTO** und die Liste der Variablen angefügt. Für jedes Datenfeld der Ergebnismenge muss eine passende Variable angegeben werden. Die einzelnen Variablen werden durch Kommata getrennt.

IF-THEN-ELSE-Verzweigung

Im Ablauf einer Prozedur kann es oft notwendig sein, dass aufgrund einer Bedingung bestimmte Anweisungen ausgeführt oder auch nicht ausgeführt werden sollen (im Beispiel *CreateProcedure.sql* Schritt ⑧). Dieses Verhalten erreichen Sie mit der Alternative **IF . . . THEN . . . ELSE**. Der Anweisungsblock im **THEN**-Zweig wird immer ausgeführt, wenn die Bedingung erfüllt ist. Ist sie falsch, werden die Anweisungen im optionalen **ELSE**-Zweig ausgeführt.

Bei dieser Alternative wird stets ein Anweisungsblock ausgeführt. Es ist nicht möglich, dass beide Blöcke oder keiner von beiden ausgeführt wird.

Syntax der IF-THEN-ELSE-Anweisung

```
IF (bedingung) THEN  
    anweisungsblock1  
[ELSE  
    anweisungsblock2]  
END IF
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort **IF**. Danach folgt in runden Klammern die gewünschte Bedingung.
- ✓ Nach dem Schlüsselwort **THEN** folgt der erste Anweisungsblock. Er wird ausgeführt, wenn die Bedingung erfüllt ist.
- ✓ Optional kann mithilfe des Schlüsselworts **ELSE** ein zweiter Anweisungsblock angegeben werden, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist.
- ✓ Die Anweisung muss mit den Schlüsselwörtern **END IF** abgeschlossen werden.

Die meisten Datenbanksysteme bieten viele weitere Möglichkeiten, Prozeduren zu steuern. MariaDB unterstützt z. B. zusätzlich zu den Befehlen im Beispiel folgende Anweisungen: **CASE**, **LOOP**, **LEAVE**, **ITERATE**, **REPEAT** und **WHILE** in Prozeduren. Nähere Informationen finden Sie in den jeweiligen Handbüchern.

Prozedur ausführen

Möchten Sie eine Prozedur ausführen, verwenden Sie die Anweisung CALL (im Beispiel *CreateProcedure.sql* Schritt ⑫ und ⑬).

Syntax

```
CALL prozedurname (parameter1, ...);
```

- ✓ Die Anweisung beginnt mit dem Schlüsselwort CALL. Danach folgt der Name der auszuführenden Prozedur.
- ✓ Falls die Prozedur Übergabeparameter erwartet, müssen diese in der richtigen Reihenfolge und durch Kommata getrennt aufgeführt werden. Die runden Klammern sind dabei optional.

Selektierbare Prozedur starten

Datenbanksysteme, die selektierbare Prozeduren unterstützen, ermöglichen es, Prozeduren wie Tabellen in einer SELECT-Anweisung zu verwenden. Sie liefern eine Ergebnismenge zurück. Falls die Prozedur Übergabeparameter erwartet, müssen diese in runden Klammern angegeben werden. MariaDB unterstützt keine selektierbaren Prozeduren.

15.4 Übung

Prozeduren in MariaDB erstellen

Übungsdatei: --

Ergebnisdatei: *Uebung.sql*

1. Erstellen Sie in der Datenbank *Uebungen* eine Prozedur *p_artikel_hinzufuegen*, die dem Lagerbestand in der Tabelle *t_lager* neue Artikel hinzfügt.
2. Die Prozedur soll als Parameter die Artikelnummer, die Stückzahl und den Preis erhalten. Verwenden Sie geeignete Namen und Datentypen.
3. Deklarieren Sie eine lokale Variable *aid* mit dem Datentyp INTEGER.
4. Prüfen Sie mit einer SELECT-Abfrage, ob der einzufügende Artikel bereits vorhanden ist, und speichern Sie das Abfrageergebnis auf der Variablen *aid*.
5. Wenn der Artikel schon im Lagerbestand ist, sollen Preis und Stückzahl des vorhandenen Artikels mit den neuen Werten aktualisiert werden.
6. Handelt es sich um einen neuen Artikel, soll er dem Lagerbestand hinzugefügt werden.
7. Testen Sie die Prozedur mit geeigneten Werten.

16

Trigger

16.1 Prozeduren automatisch ausführen

Ein **Trigger** ist eine besondere Form der Stored Procedures, bei der die Prozedur nicht durch den Anwender, sondern durch das Datenbanksystem selbst gestartet wird. Auf diese Weise können Sie einen Datenbestand überwachen und automatisch Aktionen ausführen, wenn Datensätze in Tabellen eingefügt, geändert oder gelöscht werden. Trigger bestehen aus drei wesentlichen Bestandteilen:

- ✓ Name des Triggers,
- ✓ Bedingung, die besagt, wann der Trigger aktiv wird,
- ✓ Ausführungsteil, ähnlich dem der Stored Procedures.

In der Bedingung eines Triggers kann festgelegt werden, ob die Anweisungen im Ausführungsteil bei einer INSERT-, einer UPDATE- oder einer DELETE-Anweisung aktiv werden. Der Ausführungsteil umfasst in der Regel mehrere SQL-Anweisungen, bei denen, wie bei Stored Procedures auch, Variablen und Kontrollstrukturen (WHILE, IF usw.) verwendet werden können.

Ein Trigger bezieht sich stets auf eine bestimmte Tabelle oder eine bestimmte Sicht. Es ist nicht möglich, Trigger zu definieren, die gleichzeitig Änderungen in verschiedenen Tabellen überwachen. Sie können jedoch für eine Tabelle mehrere Trigger erstellen, die verschiedene Ereignisse überwachen.

Nicht alle Integritätsregeln lassen sich deklarativ in der Datendefinition der Tabelle definieren. Mithilfe von Triggern ist es möglich, Integritätsregeln prozedural zu überprüfen und somit einen weiteren Beitrag zur Sicherung der referenziellen Integrität zu leisten. Beispielsweise kann das Einfügen ungültiger Datensätze bezüglich bestimmter Werte verhindert werden.

Zusätzlich zum SQL-Standard kann unter MariaDB und PostgreSQL festgelegt werden, ob der Trigger vor oder nach dem auslösenden Ereignis ausgeführt wird.

16.2 Trigger erstellen

Trigger mit MariaDB erstellen

Trigger werden mit der Anweisung CREATE TRIGGER erstellt. Die Syntax ist dabei dem Definieren einer Stored Procedure ähnlich.

Beispiel MariaDB: *CreateTrigger.sql*

Mithilfe eines Triggers soll die Arbeit mit einem großen Datenbestand beschleunigt werden.

Die Anwender benötigen häufig die Summe der Stückzahlen und Preise aller Artikel des Lagerbestandes in der Tabelle *t_lager*. Die Werte berechnen Sie mit der Aggregatfunktion SUMME. Bei jedem Aufruf der Funktion werden alle Datensätze durchlaufen und die Werte zeitaufwendig und rechenintensiv addiert. Mit einem Trigger kann die Summe automatisch bei jedem Einfügen eines Datensatzes berechnet und in der zusätzlichen Tabelle *t_summe_lager* gespeichert werden. Die Summe ist dort stets aktuell und schnell abrufbar.

```
① CREATE TABLE t_summe_lager
    (summe_stueck INTEGER,
     summe_preis FLOAT);
② DELIMITER //
③ CREATE TRIGGER trig_summe_artikel AFTER INSERT
④     ON t_lager FOR EACH ROW BEGIN
⑤         DECLARE summe_stueck INTEGER;
⑥         DECLARE summe_preis FLOAT;
⑦         SELECT SUM(stueck), SUM(preis) FROM t_lager
⑧             INTO summe_stueck, summe_preis;
⑨         DELETE FROM t_summe_lager;
⑩         INSERT INTO t_summe_lager
⑪             VALUES(summe_stueck, summe_preis);
⑫     END //
⑬     DELIMITER ;
⑭     SELECT * FROM t_summe_lager;
⑮     INSERT INTO t_lager (id, stueck, preis)
⑯         VALUES(56,10, 200);
⑰     SELECT * FROM t_summe_lager;
```

- ① Zuerst wird eine neue Tabelle *t_summe_lager* erstellt, die für jede der benötigten Summen ein Datenfeld erhält.
- ② Wie bei Stored Procedures muss auch bei der Definition eines Triggers durch `DELIMITER` ein Endekennzeichen benannt werden, da sonst die Eingabe des Triggers im Datenbank-Client nicht möglich ist.
- ③ Hier beginnt die Definition des Triggers *trig_summe_artikel* mit der Anweisung `CREATE TRIGGER`. Die Angabe der Schlüsselwörter `AFTER INSERT` bewirkt, dass der Trigger stets nach dem Einfügen eines neuen Datensatzes aktiv wird.

- ④ Der Trigger soll die Tabelle *t_lager* überwachen. Dies wird mit dem Schlüsselwort ON erreicht. Ab dem Schlüsselwort BEGIN erfolgt die Angabe des Anweisungsteils. Dieser entspricht dem einer Stored Procedure.
- ⑤ Zuerst werden dazu zwei lokale Variablen deklariert.
- ⑥ Die Summe der Datenfelder wird mit einer Anweisung SELECT INTO ermittelt und in den Variablen gespeichert.
- ⑦ Danach wird der möglicherweise vorhandene alte Datensatz der Tabelle *t_summe_lager* gelöscht.
- ⑧ Die neu berechneten Summen werden mittels der Variablen in die Tabelle *t_summe_lager* eingefügt.
- ⑨ Die Deklaration der Prozedur endet mit END. Die Anweisung muss entsprechend der Neudefinition des Endekennzeichens (②) mit dem Zeichen // abgeschlossen werden.
- ⑩ Nun kann als Endekennzeichen wieder das Semikolon definiert werden.
- ⑪ Zur Überprüfung wird der Inhalt der Tabelle *t_summe_lager* abgefragt, danach wird ein neuer Artikel eingefügt und damit die Arbeit des Triggers überprüft.

```
MariaDB [uebungen]> SELECT * FROM t_summe_lager;
+-----+-----+
| summe_stueck | summe_preis |
+-----+-----+
| 1558 | 374.073 |
+-----+-----+
1 row in set (0.00 sec)

MariaDB [uebungen]> INSERT INTO t_lager (id, stueck, preis)
-> VALUES(57, 15, 220);
Query OK, 1 row affected (0.01 sec)

MariaDB [uebungen]> SELECT * FROM t_summe_lager;
+-----+-----+
| summe_stueck | summe_preis |
+-----+-----+
| 1573 | 594.073 |
+-----+-----+
1 row in set (0.00 sec)
```

Der Trigger hat die neuen Summen berechnet und eingefügt

Syntax

```
CREATE TRIGGER triggername { BEFORE | AFTER } {INSERT|UPDATE|DELETE}
    ON tabellenname FOR EACH ROW BEGIN
        anweisungsblock
    END;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern CREATE TRIGGER. Danach folgt der Name des Triggers.
- ✓ Eines der Schlüsselwörter BEFORE oder AFTER wird verwendet, um festzulegen, ob der Trigger vor (BEFORE) oder nach (AFTER) der angegebenen Aktion aktiv werden soll.
- ✓ Mit einem der Schlüsselwörter INSERT, UPDATE oder DELETE wird die Art der Datenmodifikation benannt, bei der der Anweisungsteil des Triggers ausgeführt werden soll.
- ✓ Mit dem Schlüsselwort ON wird der Name der Tabelle bzw. der Sicht angegeben, für die der Trigger gelten soll.
- ✓ Die Schlüsselwörter FOR EACH ROW bedeuten, dass der Trigger nicht nur ein einziges Mal auf die gesamte Tabelle angewendet wird, sondern jeden einzelnen Datensatz anspricht, der von der Anweisung betroffen ist.
- ✓ Nach dem Schlüsselwort BEGIN werden, falls benötigt, lokale Variablen deklariert.
- ✓ Zwischen BEGIN und END steht der eigentliche Anweisungsteil des Triggers.
- ✓ Bei der Eingabe im Datenbank-Client muss die Anweisung mit einem Endekennzeichen, standardmäßig dem Semikolon, beendet werden.

Bei SQL dürfen keine zwei Trigger einer Tabelle dieselbe Aktionszeit und dasselbe Trigger-Ereignis haben. Zum Beispiel können Sie keine zwei BEFORE UPDATE-Trigger für eine Tabelle definieren. Sie können jedoch einen BEFORE UPDATE- und einen BEFORE INSERT-Trigger oder einen BEFORE UPDATE- und einen AFTER UPDATE-Trigger definieren.

Trigger mit PostgreSQL erstellen

Auch PostgreSQL unterstützt Trigger. Dabei ist jedoch zu beachten, dass hier kein einfacher Anweisungsblock übergeben werden kann, sondern eine benutzerdefinierte Funktion mit dem Rückgabewert *trigger* angegeben werden muss, die bereits zuvor programmiert wurde.

Seit der Version 9.3 unterstützt PostgreSQL sogenannte Event Trigger, die in Abhängigkeit von DDL-Befehlen (CREATE, ALTER, DROP) ausgeführt werden. Diese Funktionalität ist eine PostgreSQL-Besonderheit, welche nicht auf dem SQL-Standard basiert.

Syntax

```
CREATE TRIGGER triggername { BEFORE | AFTER } { INSERT|UPDATE|DELETE }
    ON tabellenname [ OF spaltenname ] FOR EACH ROW
    WHEN bedingung
    EXECUTE PROCEDURE funktion ( [ argument [, ...] ] )
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern CREATE TRIGGER. Danach folgt der Name des Triggers.
- ✓ Eines der Schlüsselwörter BEFORE oder AFTER wird verwendet, um festzulegen, ob der Trigger vor (BEFORE) oder nach (AFTER) der angegebenen Aktion aktiv werden soll.
- ✓ Mit einem der Schlüsselwörter INSERT, UPDATE oder DELETE wird die Art der Datenmodifikation benannt, bei der der Anweisungsteil des Triggers ausgeführt werden soll.
- ✓ Mit dem Schlüsselwort ON wird der Name der Tabelle bzw. der Sicht angegeben, für die der Trigger gelten soll.
- ✓ Optional kann eine Spalte angegeben werden. Ist dies der Fall, wird der Trigger nur ausgeführt, wenn der Wert in der betreffenden Spalte aktualisiert wird.
- ✓ Die Schlüsselwörter FOR EACH ROW bedeuten, dass der Trigger nicht nur ein einziges Mal auf die gesamte Tabelle angewendet wird, sondern jeden einzelnen Datensatz anspricht, der von der Anweisung betroffen ist.
- ✓ Es kann eine zusätzliche Bedingung angegeben werden, um die Ausführungshäufigkeit des Triggers weiter einzuschränken. Nur wenn die Bedingung erfüllt ist, wird der Trigger ausgeführt.
- ✓ Nach dem Schlüsselwort EXECUTE PROCEDURE und dem betreffenden Funktionsnamen wird eine Funktion aufgerufen.
- ✓ Die Argumente, die der Funktion übergeben werden sollen, werden anschließend in Klammern angegeben. Dabei hängt die Art und Weise, wie die Triggerargumente in der Funktion ermittelt werden können, von der jeweiligen Implementierungssprache der Funktion ab.

Informationen zur Programmierung von Funktionen mit PostgreSQL finden Sie in dem betreffenden Handbuch.

Bisherige und neue Datenfeldinhalte in Triggern auslesen

Beim Einfügen neuer Werte oder dem Aktualisieren bestehender Werte im Anweisungsteil eines Triggers kann es notwendig sein, auf den neuen oder den alten Inhalt eines bestimmten Datenfelds zuzugreifen. Dafür können Sie unter MariaDB und PostgreSQL die Transitionsvariablen NEW bzw. OLD in Kombination mit einem Datenfeldnamen verwenden.

NEW.datenfieldname	Liefert den neuen oder zu ändernden Wert eines Datenfeldes in einem Trigger für die INSERT- oder UPDATE-Aktion
OLD.datenfieldname	Liefert den bisherigen Wert eines Datenfeldes in einem Trigger für die UPDATE- und DELETE-Aktion

Beispiel für MariaDB: *CreateTriggerNew.sql*

Beim Einfügen von Datensätzen in die Tabelle *t_liefer* muss eine eindeutige Zahl als Lieferantennummer angegeben werden. Mithilfe eines Triggers soll erreicht werden, dass eine neue Nummer erzeugt wird, falls bei einer INSERT-Anweisung keine Angabe erfolgt.

```

DELIMITER //
① CREATE TRIGGER trig_liefer_nr BEFORE INSERT
    ON t_liefer FOR EACH ROW BEGIN
        DECLARE lid INTEGER;
        ② IF (NEW.id = 0) THEN
            ③ SELECT MAX(id)+1 FROM t_liefer INTO lid;
            ④ SET NEW.id = lid;
        END IF;
        END //
DELIMITER ;
⑤ INSERT INTO t_liefer (name, str, hnr, plz, ort)
    VALUES ("Walter", "Hauptstr.", "6", "60320", "Frankfurt");
SELECT * FROM t_liefer;
```

- ① Der Trigger wird mit der CREATE TRIGGER-Anweisung für die Tabelle *t_liefer* erstellt. Da das Feld *id* nicht den Wert NULL enthalten soll, muss die Überprüfung durch den Trigger noch vor dem tatsächlichen Einfügen erfolgen. Dies wird durch die Angabe der Schlüsselwörter BEFORE INSERT erreicht.
- ② In einer IF THEN-Anweisung erfolgt mithilfe der Transitionsvariable NEW die Prüfung, ob für das Feld *id* eine Eingabe erfolgt ist. Ist das nicht der Fall, wird der folgende Anweisungsblock ausgeführt.
- ③ In einer SELECT-Anweisung wird der maximale Wert für das Feld *id* ermittelt, um den Wert 1 erhöht und in der Variablen *lid* gespeichert.
- ④ Über die Transitionsvariable NEW wird dem Feld *id* der Wert der Variablen *lid* zugewiesen.
- ⑤ Um den Trigger zu testen, wird ein Datensatz ohne Eintrag für die *id* eingegeben. Anschließend wird geprüft, ob die fehlende *id* ergänzt wurde.

16.3 Trigger bearbeiten und löschen

Trigger ändern

Mit der Anweisung `ALTER TRIGGER` können Sie in einigen Datenbanksystemen einen bestehenden Trigger bearbeiten. Dabei ist es möglich, einen neuen Anweisungsblock anzugeben, die Positionsnummer oder die auslösende Aktion zu ändern. Sie können Kopf- und Anweisungsblock des Triggers gleichzeitig oder einzeln bearbeiten.

- ✓ Mit der `ALTER TRIGGER`-Anweisung können Sie nicht die Tabelle tauschen, an die der Trigger gebunden ist. Dies ist nur möglich, indem Sie den Trigger löschen und vollständig neu erstellen.
- ✓ Um einen Trigger zu bearbeiten, müssen Sie der Besitzer des Triggers sein, ihn erstellt haben oder Datenbankadministrator sein.
- ✓ MariaDB unterstützt diese Anweisung nicht. Wenn Sie einen Trigger in MariaDB ändern wollen, müssen Sie ihn löschen und neu erstellen.

Syntax MariaDB

```
ALTER TRIGGER triggername [{BEFORE | AFTER} { INSERT | UPDATE | DELETE }]
```

Die anschließende Syntax richtet sich nach dem jeweiligen Datenbanksystem.

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern `ALTER TRIGGER`, gefolgt vom Namen des zu bearbeitenden Triggers.
- ✓ Optional kann danach die auslösende Aktion des Triggers neu angegeben werden. Dafür wird eines der Schlüsselwörter `BEFORE` oder `AFTER` verwendet, um den Zeitpunkt festzulegen, und eine der Angaben `INSERT`, `UPDATE` oder `DELETE` für die Art der zu überwachenden Datenmodifikation.
- ✓ Nach dem Schlüsselwort `BEGIN` folgt optional ein neuer Anweisungsblock.

Verwenden Sie zur Definition eines Triggers einen separaten Editor, besitzen Sie mehr Möglichkeiten zur Texteingabe und Änderung. Nach der Fertigstellung der Triggerdefinition kopieren Sie diese in das entsprechende Programm, um den Trigger zu erstellen. Für geringe Änderungen ist die Anweisung `ALTER TRIGGER` geeignet.

Bei PostgreSQL können Sie zusätzlich zu den beschriebenen Änderungen mit der Anweisung `RENAME TO` auch den Namen des Triggers ändern.

Syntax PostgreSQL

```
ALTER TRIGGER triggername ON table RENAME TO neuename
```

Trigger löschen

Falls ein Trigger nicht mehr benötigt wird oder Sie ihn vollständig neu erstellen wollen, können Sie ihn mit der DROP TRIGGER-Anweisung aus dem System löschen.

Um einen Trigger zu löschen, müssen Sie der Besitzer des Triggers sein, ihn erstellt haben oder der Datenbankadministrator sein.

Syntax MariaDB

```
DROP TRIGGER [tabellenname] triggername ;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern DROP TRIGGER. Optional kann der Name der Tabelle genannt werden, für die der Trigger erstellt wurde. Danach folgt der Name des zu löschen Triggers.

Syntax PostgreSQL

```
DROP TRIGGER triggername ON tabellenname [ CASCADE | RESTRICT ] ;
```

- ✓ Die Anweisung beginnt mit den Schlüsselwörtern DROP TRIGGER, gefolgt von dem Namen des Triggers.
- ✓ Bei PostgreSQL ist die Angabe des Tabellennamens, für die der Trigger erstellt wurde, verpflichtend. Er wird eingeleitet durch das Schlüsselwort ON.
- ✓ Mit der Angabe CASCADE werden alle Objekte, die von dem betreffenden Trigger abhängen, gelöscht. Wenn Sie verhindern möchten, dass der Trigger gelöscht wird, solange noch Objekte von dem Trigger abhängen, geben Sie die Option RESTRICT an.

16.4 Übung

Trigger erstellen

Übungsdatei: --

Ergebnisdatei: *Uebung.sql*

1. Erstellen Sie in der Datenbank *Uebungen* einen neuen Trigger *trig_summe_artikel_update*.
2. Der Trigger soll die Summe der Stückzahlen und Preise aller Artikel am Lager berechnen, wenn ein Daten-satz in der Tabelle *t_lager* aktualisiert wird.
3. Berechnen Sie die Summen mithilfe einer Aggregatfunktion.
4. Speichern Sie die Werte in der Tabelle *t_summe_lager*. Löschen Sie gegebenenfalls alte Einträge in der Tabelle.
5. Prüfen Sie die Funktionsfähigkeit, indem Sie einige Stückzahlen und Preise in der Tabelle verändern.
6. Berechnen Sie die Summen mit einer SELECT-Abfrage und vergleichen Sie das Ergebnis mit den Werten in der Tabelle *t_summe_lager*.
7. Erstellen Sie den gleichen Trigger für die DELETE-Aktion.

Schulversion

1

1:1-Beziehung	33
1:1-Beziehungen	48
1:c-Beziehung	36
1:m-Beziehung	36
1:mc-Beziehungen	36
1:n-Beziehung	33
1:n-Beziehungen	48

3

3-Ebenen-Modell	15
-----------------	----

A

Abfragen verbinden	151
Abfragen, einfache	92
Abfragen, vordefinierte	156
Abstraktionskonzepte	29
Aggregatfunktionen	129
Aggregatfunktionen, Bedingungen	132
Aggregatfunktionen, Filter	132
Aggregation	29, 34
ALTER DOMAIN	85
ALTER PROCEDURE-Anweisung	193
ALTER TABLE	87
ALTER TRIGGER-Anweisung	203
Änderungen zurücknehmen	185
Anomalien	52
ANSI-SQL-Standard	129
Anweisungsblock	191
Anzahl Datensätze	104
Architekturtypen, parallele DBs	24
Assoziation	29, 32
Attribute	31, 32, 45
Attribute umbenennen	61, 62
AUTO_INCREMENT	75

B

Bedingungen	106
Benutzer anlegen	171
Benutzer verwalten	170, 172
Benutzer, MariaDB	171
Benutzer, PostgreSQL	168
Benutzerrechte einsehen	175
Benutzerverwaltung	167
Berechnungen ausführen	105
Bereichsprüfung	106, 108
BETWEEN-Operator	108
Beziehungen	32
Beziehungen zwischen Relationen	10

Beziehungsmenge	32
Beziehungstypen	33
Bottom-up	37
Boyce-Codd-Normalform	58

C

CHECK	82
Chen-Notation	30
Client/Server-Konzept	23
CLOSE-Anweisung	166
COMMIT-Anweisung	184
CONSTRAINT-Anweisung	81
Constraints	81
COUNT()	130
CREATE DATABASE	69
Create Domain	83
CREATE INDEX	125
CREATE PROCEDURE-Anweisung	189
CREATE TABLE	74
CREATE TRIGGER	199
CREATE VIEW	157
Cursor	163
Cursor erstellen	164
Cursor schließen	166
Cursor, Datenzugriff	165
Cursor, serverseitiger	163

D

Data Dictionary	19
Daten abfragen	100
Daten aktualisieren	94
Daten einfügen	90
Daten gruppieren	111
Daten löschen	97
Datenbank	7
Datenbank auswählen, MariaDB	72
Datenbank erstellen	68
Datenbank erstellen, MariaDB	69
Datenbank erstellen, PostgreSQL	70
Datenbank löschen	73
Datenbank wechseln, PostgreSQL	72
Datenbank, dokumentorientierte	11
Datenbank, Eigenschaften	7
Datenbank, Graphen	14
Datenbank, hierarchische	8
Datenbank, Key/Value	14
Datenbank, NoSQL	14
Datenbank, objektorientierte	12
Datenbank, objektrelationale	13
Datenbank, relationale	10
Datenbank, spaltenorientierte	11
Datenbankadministrator	173
Datenbanken anzeigen	71
Datenbankentwurf	26, 27
Datenbank-Lebenszyklus	26
Datenbankmanagementsystem	7, 17
Datenbanknamen, MariaDB	70
Datenbanknamen, PostgreSQL	71
Datenbankobjekte, GRANT-Anweisung	174
Datenbanksysteme	7
Datenbanktypen	8
Datenbestände	138
Datenlexikon	19
Datenmanipulationssprache	44
Datenmodell, relationales	44, 138
Datenmodell, textuelle Beschreibung	40
Datenpuffer	163
Datenredundanz	6
Datensatz einfügen	90
Datensätze	45
Datensätze sequentiell lesen	163
Datensätze, Anzahl beschränken	104
Datensätze, doppelte	104
Datensätze, mehrere einfügen	92
Datenschutzprobleme	6
Datensicherung	18
Datentypen, Datumswerte	78
Datentypen, Festkommazahlen	78
Datentypen, Fließkommazahlen	77
Datentypen, numerische	77
Datentypen, Texte	79
Datenunabhängigkeit	6, 15
Datenunabhängigkeit, logische	15
Datenunabhängigkeit, physische	15
Datumswerte	78
DBMS, Komponenten	19
DBS	7
DBS, paralle	24
DBS, verteilte	21
DBS, zentralisierte	20
DECLARE CURSOR	164
DEFAULT	76
DELETE-Anweisung	97
DELIMITER-Anweisung	191
Differenz	61
DISTINCT	102, 104
Dokumentorientierte Datenbank	11
Domänen	31, 83
Domänen ändern	84
Domänen definieren	83
Domänen löschen	85
Domänen verwenden	84
DROP DATABASE	73
DROP TABLE	89
DROP VIEW	159

DROP PROCEDURE-Anweisung	192	Hierarchische Datenbanken	8	Mengenoperationen	138
Durchschnitt	61	Homogene Verteilung	22	Mengenschreibweise	45
E					
EERM	30	Identifikation	29	MIN()	130
Elementprüfung	106, 108	Identifizierendes Attribut	32	Min-Max-Notation	36
Endekennzeichen	191	IF EXISTS	73	m:n-Beziehungen	49
Entität	30	IF NOT EXISTS	70	Mustervergleich	106, 109
Entitätsmenge	30, 48	IF-THEN-ELSE-Verzweigung	196	N	
Entities	30	Index	47, 115	n:m-Beziehung	33
Entity-Relationship-Modell	30	Indizes	115, 116	Natural Join	62, 63
Entity-Set	30	Indizes erstellen	125	Natürlicher Verbund	62, 63
Entity-Typ	30	Indizes löschen	127	Netzwerkdatenbanken	9
Entwurfsphasen	28	Indizes, Richtlinien	126	Normalform, Boyce-Codd	58
Equi-Join	63, 141	Inkonsistenzen	6	Normalform, dritte	56
ER-Modell	30	Inner-Join, mehrere Tabellen	147	Normalform, erste	54
ER-Modell, Darstellung	35	IN-Operator	108	Normalform, fünfte	58
Ersatznamen	157	INSERT-INTO-Anweisung	90	Normalform, vierte	58
Ersatznamen, Tabellen	144	Integrität	18	Normalform, zweite	55
Erweitertes Entity-Relationship-Modell (EERM)	30	Integrität, referenzielle	121	Normalformen	53
Existenzbedingungen	106	Integritätsbedingungen	18	Normalisierung	51
EXISTS	154	Interne Ebene	16	Normalisierungsprozess	53
Exportschemen	23	Internes Schema	17	NoSQL Datenbank	14
Externe Ebene	16	Is-a-Beziehung	30, 34, 50	NOT VALID	88
F					
Felder	45	ISOLATION LEVEL	182	Notation, Chen-	37
Festkommazahlen	78	K	O		
FETCH-Anweisung	165	Kardinalität	33	Objektorientierte Datenbank	12
FILTER	132	Key/Value-Datenbank	14	Objektrelationale	
Fließkommazahlen	77	Klassifikation	29	Datenbanksysteme	13
Fremdschlüssel	47, 116	Klausel HAVING	132	ON-Klausel	146
Fremdschlüsselverletzungen	123	Klausel WHERE	132	OPEN-Anweisung	165
Full-Join	141	Konsistenz	178	Operatoren, logische	106, 111
Funktionale Abhängigkeit	53	Konzeptionelle Ebene	16	Outer-Join	63, 149
Funktionen	129	Konzeptioneller Entwurf	27	OWNER	70
Funktionen verwenden	133	Konzeptionelles Schema	16	P	
Funktionen, mathematische	134	Krähenfuss-Notation	37	Paralle DBS	24
Funktionen, Zeichenketten	136	Künstliches Attribut	38	Part-of-Beziehung	30, 34
G					
Generalisierung	29, 34	LIKE-Operator	109	PASSWORD	70
GRANT CREATE TABLE	74	LIMIT	104	Physische Datenbankarchitektur	20
GRANT-Anweisung	173	Logbuch	18, 19	Physische Struktur	26
Graphen-Datenbanken	14	Logische Gesamtsicht	16	Physischer Entwurf	27
GROUP BY-Anweisung	111	Logische Operatoren	111	Planungsprozess	26
Gültigkeitsprüfung	81	Logische Struktur	26	Platzhalterzeichen	109
H					
HAVING-Klausel	132	Logischer Entwurf	27	Primärschlüssel	32, 47, 115
Heterogene Verteilung	23	M	Projektion		
Martin-Notation	37	Martin-Notation	37	PRIMARY KEY	117
Mehrbenutzerbetrieb, eingeschränkter	6	M	Projektion		
		Mehrbenutzerbetrieb,		Prozeduren	187

Prozeduren anzeigen	192	Shared-Memory-Architektur	24	Tupel	32, 44, 45
Prozeduren		Shared-Nothing-Architektur	25	Typumwandlung	80
automatisch ausführen	198	SHOW DATABASES	71		
Prozeduren bearbeiten	192	SHOW PROCEDURES-			
Prozeduren definieren	189	Anweisung	192	U	
Prozeduren löschen	192	SHOW TABLES	86	UML	35, 37
Prozeduren, selektierbare	197	Sicherheitskonzepte	167	Unified Modelling Language (UML)	37
Prozedursprache	193	Sicht	15	UNIQUE	119
Prozesse, warten auf	182	Sichten	156	Unterabfragen	153
<i>psql</i>	70	Spalten	44	Unterprogramme	187
		Spalten benennen	102	UPDATE-Anweisung	94
		Spaltenorientierte Datenbank	11	USE	72
R		Spezialisierung	34		
Rechte vergeben	174	SQL	8	V	
Recordsets	9	SQL-Server	24	VALIDATE CONSTRAINT	88
Recovery	18	Stored Procedures	187, 198	Variablen verwenden	195
Redundanzen	6	Stored Procedures definieren	189	Verbund	62
Rekursive Beziehungen	34	Subselect	153	Vereinigung	61
Relation	44	Symmetrischer Outer-Join	63	Vergleichsoperatoren	106, 107
Relationale Datenbank	10	Synchronisation	18	Verknüpfen, Tabellen	141
Relationales Datenmodell	44	Systemadministrator	167	Verknüpfung, Möglichkeiten	138
Relationen	10	Systemsicherheit	188	Verteilte DBS	21
Relationenalgebra	60			Views	156
Relationenkalkül	64	T		Views erstellen	157
Relationenschemata	45	Tabellen	10, 44	Views löschen	159
Relationships	30	Tabellen ändern	86	Views, Abfragen für	157
Reorganisation	27	Tabellen anzeigen	86	Views, Daten ändern	160
Repositories	19	Tabellen erstellen	74	Views, Datenfelder benennen	158
Retrieval	59	Tabellen löschen	88	Views, Einfügedaten überprüfen	161
REVOKE-Anweisung	176	Tabellen sperren	181	Views, mehrere Tabellen	158
ROLLBACK-Anweisung	185	Tabellen, gefilterte	156		
		Tabellen, Struktur	45	W	
		Tabellenstruktur anzeigen	88	Wasserfallmodell	29
S		Theta-Join	63, 143	WHERE-Bedingung	106
Schlüssel	46, 115	Top-Down	37	WHERE-Klausel	132
Schlüsselfelder	57, 117	Transaktionen	178		
Schlüsselkandidaten	46	Transaktionen abschließen	184	Z	
Sekundärindizes	47	Transaktionen erstellen	180	Zeilen	44
Sekundärschlüssel	47, 116	Transformation, ER-Modell in relationales Modell	48	Zentralisierte DBS	20
Sekundärschlüssel löschen	120	Transformationsregeln	16	Zugriffrechte entziehen	176
SELECT-Anweisung	100	Transitive Abhängigkeit	53, 56	Zugriffsrechte	173, 176
Selektion	60	Trigger	198		
Self-Join	63, 151	Trigger ändern	203		
Semi-Join	63	Trigger erstellen	199		
Sets	9	Trigger löschen	204		
Shared-Disk-Architektur	25				
Shared-Everything-Architektur	24				

Impressum

Matchcode: SQL_2021

Autor: Elmar Fuchs

Produziert im HERDT-Digitaldruck

1. Ausgabe, Juli 2021

HERDT-Verlag für Bildungsmedien GmbH
Am Kümmerling 21–25
55294 Bodenheim
Internet: www.herdt.com
E-Mail: info@herdt.com

© HERDT-Verlag für Bildungsmedien GmbH, Bodenheim

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlags reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit großer Sorgfalt erstellt und geprüft. Trotzdem können Fehler nicht vollkommen ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Wenn nicht explizit an anderer Stelle des Werkes aufgeführt, liegen die Copyrights an allen Screenshots beim HERDT-Verlag. Sollte es trotz intensiver Recherche nicht gelungen sein, alle weiteren Rechteinhaber der verwendeten Quellen und Abbildungen zu finden, bitten wir um kurze Nachricht an die Redaktion.

Die in diesem Buch und in den abgebildeten bzw. zum Download angebotenen Dateien genannten Personen und Organisationen, Adress- und Telekommunikationsangaben, Bankverbindungen etc. sind frei erfunden. Eventuelle Übereinstimmungen oder Ähnlichkeiten sind unbeabsichtigt und rein zufällig.

Die Bildungsmedien des HERDT-Verlags enthalten Verweise auf Webseiten Dritter. Diese Webseiten unterliegen der Haftung der jeweiligen Betreiber, wir haben keinerlei Einfluss auf die Gestaltung und die Inhalte dieser Webseiten. Bei der Bucherstellung haben wir die fremden Inhalte daraufhin überprüft, ob etwaige Rechtsverstöße bestehen. Zu diesem Zeitpunkt waren keine Rechtsverstöße ersichtlich. Wir werden bei Kenntnis von Rechtsverstößen jedoch umgehend die entsprechenden Internetadressen aus dem Buch entfernen.

Die in den Bildungsmedien des HERDT-Verlags vorhandenen Internetadressen, Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen waren zum Zeitpunkt der Erstellung der jeweiligen Produkte aktuell und gültig. Sollten Sie die Webseiten nicht mehr unter den angegebenen Adressen finden, sind diese eventuell inzwischen komplett aus dem Internet genommen worden oder unter einer neuen Adresse zu finden. Sollten im vorliegenden Produkt vorhandene Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen nicht mehr der beschriebenen Software entsprechen, hat der Hersteller der jeweiligen Software nach Drucklegung Änderungen vorgenommen oder vorhandene Funktionen geändert oder entfernt.