Daten verwalten – sicher und schnell mit SQL

6., überarbeitete und aktualisierte Auflage

SQL

FÜR DUMMIES®

Auf einen Blick:

- Die Grundlagen relationaler Datenbanken kennenlernen
- Datenbanken erstellen, verwalten und bearbeiten
- Werte, Wertausdrücke und Klauseln richtig einsetzen
- Benutzerrechte vergeben und Datenbanken schützen

Allen G. Taylor

SQL für Dummies – Schummelseite

Diese Schummelseiten enthalten mehrere nützliche Tabellen und Listen mit Informationen, die beim Arbeiten mit SQL wiederholt benötigt werden. An einer einzigen Stelle können Sie schnelle Antworten auf verschiedene Fragen finden, die häufig bei der SQL-Entwicklung auftauchen.

SQL-Kriterien für Normalformen

Damit Datenbanktabellen Ihre Daten zuverlässig speichern, müssen sie so entworfen werden, dass keine Änderungsanomalien auftreten können. Zu diesem Zweck müssen sie normalisiert werden. Vergleichen Sie die SQL-Kriterien in der folgenden Liste mit den Tabellen in Ihrer Datenbank, um die Stellen zu entdecken, an denen Anomalien auftreten könnten und die gegebenenfalls weiter normalisiert werden müssen.

Erste Normalform (1NF)

- ✔ Die Tabelle muss zweidimensional sein und Zeilen und Spalten enthalten.
- ✔ Jede Zeile enthält Daten, die zu einem Objekt oder einem Teil des Objekts gehören.
- ✔ Jede Spalte enthält Daten für ein einzelnes Attribut des beschriebenen Objekts.
- ✔ Jede Zelle (Schnittpunkt einer Zeile und Spalte) einer Tabelle enthält einen einzigen Wert.
- ✔ Alle Einträge in einer Spalte müssen vom selben Typ sein.
- ✔ Jede Spalte hat einen eindeutigen Namen.
- ✔ Keine zwei Zeilen sind identisch.
- ✔ Die Reihenfolge der Spalten und Zeilen spielt keine Rolle.

Zweite Normalform (2NF)

- ✔ Die Tabelle muss in der ersten Normalform (1NF) vorliegen.
- ✔ Alle Nicht-Schlüssel-Attribute (Spalten) sind vom gesamten Schlüssel abhängig.

Dritte Normalform (3NF)

- ✔ Die Tabelle muss in der zweiten Normalform (2NF) vorliegen.
- ✔ Die Tabelle enthält keine transitiven Abhängigkeiten.

Wertebereich-/Schlüssel-Normalform (DKNF, Domain Key Normal Form)

- ✔ Jede Einschränkung der Tabelle ist eine logische Folge der Definition von Schlüsseln und Wertebereichen (Domänen).

SQL-Datentypen

Genaue Zahlen

Die folgende Liste enthält alle formalen Datentypen in ISO/IEC-Standard-SQL. Zusätzlich zu diesen Typen können Sie weitere Datentypen definieren, die von diesen Typen abgeleitet sind.

```
INTEGER
SMALLINT
BIGINT
NUMERIC
DECIMAL
```

Annähernd genaue Zahlen

```
REAL
DOUBLE PRECISION
FLOAT
```

Binärstrings

```
BINARY
BINARY VARYING
BINARY LARGE OBJECT
```

Boolesche Werte

```
BOOLEAN
```

Zeichenfolgen

```
CHARACTER
CHARACTER VARYING (VARCHAR)
CHARACTER LARGE OBJECT
NATIONAL CHARACTER
NATIONAL CHARACTER VARYING
NATIONAL CHARACTER LARGE OBJECT
```

Datums- und Zeit-Werte

```
DATE
TIME WITHOUT TIMEZONE
TIMESTAMP WITHOUT TIMEZONE
TIME WITH TIMEZONE
TIMESTAMP WITH TIMEZONE
```

Intervalle

```
INTERVAL DAY
INTERVAL YEAR
```

Collection-Typen

```
ARRAY
MULTISET
```

Andere Typen

```
ROW
XML
```

SQL für Dummies – Schummelseite

SQL-Wertefunktionen

Die folgenden SQL-Wertefunktionen verändern Daten. Daten können auf vielerlei Arten verändert werden; die hier genannten Funktionen führen einige der am häufigsten benötigen Operationen aus.

Zeichenfolgenfunktionen	Beschreibung
`SUBSTRING`	Extrahiert eine Teilzeichenfolge aus einer Quellzeichenfolge
`SUBSTRING SIMILAR`	Extrahiert eine Teilzeichenfolge mithilfe eines POSIX-basierten regulären Ausdrucks aus einer Quellzeichenfolge
`SUBSTRING_REGEX`	Extrahiert das erste oder jedes Auftreten eines regulären XQuery-Ausdrucks aus einer Quellzeichenfolge und gibt den Teilstring zurück
`TRANSLATE_REGEX`	Extrahiert das erste oder jedes Auftreten eines regulären XQuery-Ausdrucks aus einer Quellzeichenfolge und ersetzt den Teilstring durch einen XQuery-Ersetzungsstring
`UPPER`	Konvertiert eine Zeichenfolge in Großbuchstaben
`LOWER`	Konvertiert eine Zeichenfolge in Kleinbuchstaben
`TRIM`	Schneidet Leerzeichen am Anfang und am Ende einer Zeichenfolge ab
`TRANSLATE`	Wandelt den Zeichensatz einer Zeichenfolge in einen anderen Zeichensatz um
`CONVERT`	Konvertiert den Zeichensatz einer Zeichenfolge in einen anderen Zeichensatz

Numerische Funktionen	Beschreibung
`POSITION`	Gibt die Startposition einer Zielzeichenfolge innerhalb einer Quellzeichenfolge zurück
`CHARACTER_LENGTH`	Gibt die Anzahl der Zeichen in einer Zeichenfolge zurück
`OCTET_LENGTH`	Gibt die Anzahl der Achtbitzeichen (= Bytes) innerhalb einer Zeichenfolge zurück
`EXTRACT`	Extrahiert ein einzelnes Feld aus einem Datumswert oder Intervall

Datums- und Zeitfunktionen	Beschreibung
`CURRENT_DATE`	Gibt das aktuelle Datum zurück
`CURRENT_TIME(p)`	Gibt die aktuelle Zeit zurück; (p) ist die Genauigkeit der Sekundenbruchteile
`CURRENT_TIMESTAMP(p)`	Gibt das aktuelle Datum und die aktuelle Zeit zurück; (p) ist die Genauigkeit der Sekundenbruchteile

Mengenfunktionen

Die SQL-Mengenfunktionen liefern schnelle Antworten auf Fragen, die die Eigenschaften der Daten insgesamt betreffen. Wie viele Zeilen sind in einer Tabelle enthalten? Was ist der größte oder kleinste Wert in einer Spalte? Solche Fragen werden mit den SQL-Mengenfunktionen beantwortet.

`COUNT`	Gibt die Anzahl der Zeilen in der durch die `WHERE`-Klausel spezifizierten Datenmenge zurück
`MAX`	Gibt den größten Wert in der durch die `WHERE`-Klausel spezifizierten Datenmenge zurück
`MIN`	Gibt den kleinsten Wert in der durch die `WHERE`-Klausel spezifizierten Datenmenge zurück
`SUM`	Addiert die Werte in der durch die `WHERE`-Klausel spezifizierten Datenmenge
`AVG`	Gibt den Durchschnittswert aller Werte in der durch die `WHERE`-Klausel spezifizierten Datenmenge zurück

Prädiktate der WHERE-Klausel

Prädikate haben einen der beiden booleschen Werte `TRUE` oder `FALSE`. Mit `WHERE`-Klauseln können Sie unerwünschte Zeilen aus dem Ergebnis einer SQL-Abfrage herausfiltern.

Vergleichsprädikate

=	Gleich
<>	Ungleich
<	Kleiner als
<=	Kleiner als oder gleich
>	Größer als
>=	Größer als oder gleich

Andere Prädikate

`ALL`
`BETWEEN`
`DISTINCT`
`EXISTS`
`IN`
`LIKE`
`MATCH`
`NOT IN`
`NOT LIKE`
`NULL`
`OVERLAPS`
`SIMILAR`
`SOME, ANY`
`UNIQUE`

SQL für Dummies

Allen G. Taylor

SQL für Dummies

Übersetzung aus dem Amerikanischen
von Reinhard Engel

6., überarbeitete und aktualisierte Auflage

WILEY

WILEY-VCH Verlag GmbH & Co. KGaA

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

6., überarbeitete und aktualisierte Auflage 2014

Printed in Germany
Gedruckt auf säurefreiem Papier

Coverfoto: © hauhu – istockphoto.com
Korrektur: Petra Heubach-Erdmann und Jürgen Erdmann, Düsseldorf
Satz: inmedialo Digital- und Printmedien UG, Plankstadt
Druck und Bindung: CPI – Ebner & Spiegel, Ulm

Print ISBN: 978-3-527-71020-1
ePDF ISBN: 978-3-527-68041-2
ePub ISBN: 978-3-527-68039-9
mobi ISBN: 978-3-527-68040-5

Über den Autor

Allen G. Taylor gilt als Veteran der Computerindustrie. Er verfügt über 30 Jahre Erfahrung und hat in dieser Zeit über dreißig Bücher veröffentlicht. Zurzeit arbeitet er als Dozent für Datenbanken, Netzwerke, Innovationen und Existenzgründung. Darüber hinaus schult er über einen Online-Provider Datenbankentwicklung. Wenn Sie mehr über Allen G. Taylor wissen möchten, sollten Sie die Website `www.DatabaseCentral.Info` besuchen.

Widmung

Dieses Buch ist Walker Taylor gewidmet, der Erstaunliches tun wird, wenn er herangewachsen ist.

Danksagung des Autors

Zunächst einmal möchte ich mich ganz herzlich bei Jim Melton, dem Herausgeber der ISO/ANSI-Spezifikationen, für seine Hilfe bedanken. Ohne seine unermüdlichen Anstrengungen wäre sowohl dieses Buch als auch SQL als internationaler Standard viel weniger wert. Auch Andrew Eisenberg hat durch seine Bücher viel zu meinen SQL-Kenntnissen beigetragen. Weiterhin möchte ich Michael Durthaler für seine hilfreichen Vorschläge zum Thema Cursors danken. Ich möchte auch meinem Projektleiter Pat O'Brien, meinem Fachlektor Mike Chapple und meiner Lektorin Kyle Looper danken, die für die Herausgabe des Buchs zuständig waren. Danke auch an meine Agentin Carole McClendon von Waterside Productions für ihre Unterstützung meiner Karriere.

Über die Übersetzer

Meinhard Schmidt, Jahrgang 1949 und glücklich verheiratet, beschäftigt sich seit mehr als 20 Jahren mit Soft- und Hardware und arbeitet hauptberuflich als IT-Trainer. Er hat acht eigene Bücher zu unterschiedlichen IT-Themen veröffentlicht (wobei der Schwerpunkt in den letzten Jahren im Bereich der relationalen Datenbanken lag) und übersetzt seit vielen Jahren IT-Fachliteratur.

Reinhard Engel ist seit 1980 in der IT-Branche tätig. Er arbeitete etwa fünfzehn Jahre als Berater, Software-Entwickler und IT-Trainer und danach als Autor (Schwerpunkte: OOP, Hypertext mit dem Microsoft Help Compiler, C++-Programmierung) und Übersetzer (Schwerpunkte: … *für-Dummies*-Bücher über IT-Themen und Wirtschaftsfachwissen; Web-Marketing; anspruchsvolle Programmiertitel).

Cartoons im Überblick

von Christian Kalkert

Seite 25

Seite 93

Seite 149

Seite 295

Seite 343

Seite 373

Internet: www.stiftundmaus.de

Inhaltsverzeichnis

Einführung

Willkommen bei der Datenbankentwicklung mit der Standard-Abfragesprache SQL. Es gibt viele verschiedene Werkzeuge für Datenbankverwaltungssysteme (DBMS, Database Management System), die auf verschiedenen Hardware-Plattformen laufen. Diese Werkzeuge unterscheiden sich zum Teil beträchtlich, aber alle ernst zu nehmenden Produkte haben eines gemeinsam: Sie unterstützen den Datenzugriff und die Datenbearbeitung mit SQL. Wenn Sie SQL kennen, können Sie relationale Datenbanken erstellen und nützliche Informationen daraus abrufen.

Über dieses Buch

Relationale Datenbankverwaltungssysteme spielen in vielen Unternehmen eine lebenswichtige Rolle. Laien sind häufig der Meinung, dass die Erstellung und die Verwaltung dieser Systeme mit sehr komplexen Aufgaben verbunden und somit das Herrschaftsgebiet von Datenbankgurus sei, die einen Grad der Erleuchtung erlangt haben, der über den des Normalsterblichen hinausgeht. Dieses Buch lüftet den Schleier dieser Datenbankgeheimnisse, und Sie werden

- ✔ die Grundlagen von Datenbanken kennenlernen
- ✔ herausfinden, wie ein DBMS aufgebaut ist
- ✔ die funktionalen Hauptkomponenten von SQL kennenlernen
- ✔ eine Datenbank erstellen
- ✔ eine Datenbank vor Schäden schützen
- ✔ mit den Daten einer Datenbank arbeiten
- ✔ gewünschte Daten aus einer Datenbank abrufen

Dieses Buch soll Ihnen helfen, mit SQL relationale Datenbanken zu erstellen und nützliche Informationen daraus zu extrahieren. SQL ist eine international standardisierte Sprache, die weltweit eingesetzt wird, um relationale Datenbanken zu erstellen und zu verwalten. Diese Auflage beschreibt die neueste Version des Standards: SQL:2011.

Ich gehe nicht näher darauf ein, wie eine Datenbank entworfen wird, da ich davon ausgehe, dass Sie bereits über einen arbeitsfähigen Datenbankentwurf verfügen, den Sie erstellt oder von jemand anderem übernommen haben. Ich zeige Ihnen, wie Sie diesen Entwurf mit SQL umsetzen können. Falls Sie der Meinung sind, dass Ihr Datenbankentwurf verbesserungsfähig sein könnte, sollten Sie ihn auf jeden Fall überarbeiten, bevor Sie versuchen, die Datenbank zu erstellen. Je früher Sie bei einem Entwicklungsprojekt Probleme erkennen und beheben, desto billiger werden die erforderlichen Korrekturen sein.

Wer sollte dieses Buch lesen?

Wenn Sie Daten in einem DBMS speichern und daraus abrufen müssen, können Sie Ihre Aufgaben viel besser erledigen, wenn Sie SQL beherrschen. Sie müssen kein Programmierer sein, um SQL zu benutzen, und Sie müssen keine anderen Programmiersprachen, wie zum Beispiel Java, C oder BASIC, kennen. Die SQL-Syntax ist an das normale Englisch angelehnt.

Wenn Sie Programmierer *sind*, können Sie SQL in Ihre Programme einbinden. SQL erweitert konventionelle Sprachen um sehr mächtige Funktionen zur Bearbeitung und zum Auslesen von Daten. In diesem Buch erfahren Sie, was Sie wissen müssen, um die vielseitigen Werkzeuge und Funktionen von SQL in Ihren Programmen nutzen zu können.

Symbole, die in diesem Buch verwendet werden

Tipps sparen Zeit und lotsen Sie um Gefahren herum.

Achten Sie auf die Informationen, die mit diesem Symbol gekennzeichnet sind – Sie könnten sie später brauchen.

Die Ratschläge unter diesem Symbol können Sie vor größerem Schaden bewahren. Missachten Sie diese Hinweise auf eigene Gefahr.

Dieses Symbol kennzeichnet technische Details, die zwar interessant, aber nicht unbedingt notwendig sind, um das gerade behandelte Thema zu verstehen.

Wie es weitergeht

Und jetzt beginnt der Spaß! Datenbanken sind das beste Werkzeug, das jemals erfunden wurde, um die Dinge im Auge zu behalten, die für Sie wichtig sind. Wenn Sie Datenbanken verstehen und mit SQL dazu bringen können, Ihre Wünsche zu erfüllen, haben Sie die Möglichkeit, ungeahnten Einfluss auf Ihre Daten auszuüben. Ihre Kollegen kommen zu Ihnen, wenn sie wichtige Informationen benötigen. Ihre Vorgesetzten suchen Ihren Rat. Praktikanten bitten Sie um ein Autogramm. Aber – und das ist am wichtigsten – Sie verstehen, wie Ihr Unternehmen wirklich funktioniert.

Teil I

Grundbegriffe

In diesem Teil ...

- ✔ Wesentliche Begriffe relationaler Datenbanken
- ✔ Grundlegende SQL-Begriffe
- ✔ Grundlegende Datenbank-Tools

Grundlagen relationaler Datenbanken

1

In diesem Kapitel

- Informationen organisieren
- Den Begriff *Datenbank* definieren
- Den Begriff *DBMS* definieren
- Datenbankmodelle im Vergleich
- Den Begriff *relationale Datenbanken* definieren
- Die Probleme beim Entwurf einer Datenbank

SQL (*Structured Query Language*) ist eine Sprache, (die *ess-ku-el* und nicht *si-quel* ausgesprochen wird und) die speziell für das Arbeiten mit Datenbanken entwickelt wurde. Man kann damit Datenbanken erstellen, neue Daten in eine Datenbank einfügen und ausgewählte Teile der Daten abrufen. SQL wurde 1970 eingeführt und ist im Laufe der Jahre gewachsen und zu einem Branchenstandard geworden. Sie wird durch einen formellen Standard definiert, der von der International Standards Organization (ISO) betreut wird.

Es gibt verschiedene Datenbankarten, die unterschiedliche konzeptionelle Modelle für die Organisation von Daten repräsentieren.

SQL wurde ursprünglich zu dem Zweck entwickelt, Daten in Datenbanken zu verwalten, die nach dem *relationalen Modell* aufgebaut sind. Der internationale SQL-Standard wurde kürzlich um einen Teil des *Objektmodells* erweitert, was zu hybriden Strukturen führt, die als objektrelationale Datenbanken bezeichnet werden. In diesem Kapitel beschreibe ich verschiedene Formen der Datenspeicherung, vergleiche das relationale Modell mit anderen wichtigen Modellen und gehe dann auf die wichtigen Merkmale relationaler Datenbanken ein.

Doch bevor ich auf SQL eingehe, möchte ich den Begriff der *Datenbank* klar definieren. Seine Bedeutung änderte sich ähnlich, wie Computer die Art und Weise veränderten, Informationen zu speichern und zu verwalten.

Die Übersicht über Dinge behalten

Heute werden Computer für viele Aufgaben benutzt, die früher mit anderen Werkzeugen erledigt wurden. Beispielsweise haben Computer Schreibmaschinen weitgehend ersetzt, um Dokumente zu erstellen und zu ändern. Sie haben elektromechanische Rechenmaschinen verdrängt. Sie haben Millionen von Papierseiten, Ordnern und Ablageschränken als Hauptmedium für die Aufbewahrung wichtiger Informationen abgelöst. Verglichen mit diesen alten Werkzeugen können Computer natürlich sehr viel mehr Aufgaben viel schneller und genauer erledigen. Diese Vorteile haben jedoch einen Nachteil: Computerbenutzer können nicht mehr direkt physisch auf ihre Daten zugreifen.

Wenn Computer gelegentlich ausfallen, fragen sich die Benutzer manchmal, ob die Computerisierung tatsächlich einen Fortschritt gebracht hat. Früher konnte ein Ordner nicht »abstürzen«, sondern höchstens zu Boden fallen. Dann sammelten Sie die Blätter einfach wieder auf und fügten sie wieder in den Ordner ein. Außer bei Erdbeben können Ablageschränke nicht »abstürzen«. Sie melden Ihnen auch niemals einen Fehler. Ein Festplattenabsturz ist dagegen etwas ganz anderes: Sie können die verloren gegangenen Bits und Bytes nicht einfach wieder »aufheben«. Mechanische, elektrische und menschliche Fehlfunktionen können dazu führen, dass Ihre Daten im »Nirgendwo« verschwinden und für immer verloren sind.

Doch wenn Sie sich mit den erforderlichen Vorsichtsmaßnahmen gegen einen zufälligen Verlust Ihrer Daten schützen, können Sie die Vorteile nutzen, die Computer in Form höherer Geschwindigkeit und Genauigkeit bieten.

Wenn Sie wichtige Daten speichern, müssen Sie Ihr Augenmerk auf die folgenden vier Bereiche lenken:

- ✔ Die Daten müssen schnell und einfach gespeichert werden, weil dieser Vorgang sehr häufig notwendig ist.
- ✔ Das Speichermedium muss zuverlässig sein. Sie wollen später keine Überraschung erleben und feststellen, dass Ihre Daten ganz oder teilweise verschwunden sind.
- ✔ Die Daten müssen schnell und einfach abgerufen werden können, und zwar unabhängig von der Anzahl der gespeicherten Einträge.
- ✔ Sie benötigen eine einfache Methode, um genau die gewünschten Daten aus der Riesenmenge der insgesamt gespeicherten Daten herauszufiltern.

Computer-Datenbanken, die dem Stand der Technik entsprechen, erfüllen diese vier Kriterien. Wenn Sie mehr als einige Dutzend Datenelemente speichern müssen, sollten Sie dafür eine Datenbank verwenden.

Was ist eine Datenbank?

In den letzten Jahren hat der Begriff *Datenbank* seine ursprüngliche Bedeutung ziemlich weitgehend eingebüßt. Für einige ist jede Sammlung von Datenelementen (Telefonbücher, Einkaufszettel, Schriftrollen und so weiter) eine Datenbank. Andere definieren den Begriff genauer.

In diesem Buch definiere ich eine *Datenbank* als eine selbstbeschreibende Sammlung integrierter Datensätze. Und diese Definition impliziert Computertechnologie zusammen mit Programmiersprachen wie SQL.

Ein *Datensatz* ist eine Repräsentation eines physischen oder konzeptionellen Objekts. Wenn Sie beispielsweise die Kunden einer Firma verwalten wollen, legen Sie für jeden Kunden einen Datensatz an. Jeder Datensatz enthält ein oder mehrere *Attribute*, wie beispielsweise den Namen, die Adresse oder die Telefonnummer. Einzelne Namen, Adressen und so weiter sind die eigentlichen Daten oder *Datenelemente*.

Eine Datenbank besteht sowohl aus Daten als auch aus Metadaten. *Metadaten* sind Daten, die die Struktur der Daten innerhalb einer Datenbank beschreiben. Wenn Sie wissen, wie Ihre Daten strukturiert sind, können Sie sie abrufen. Eine Datenbank ist *selbstbeschreibend*, weil sie eine Beschreibung ihrer eigenen Struktur enthält. Die Datenbank ist *integriert*, weil sie neben den Datenelementen auch Beziehungen zwischen den Datenelementen enthält.

Die Datenbanken speichern Metadaten in einem Bereich, der *Datenverzeichnis* (englisch *Data Dictionary*) genannt wird und der die Tabellen, Spalten, Indizes, Einschränkungen (Bedingungen) und andere Elemente beschreibt, aus denen die Datenbank besteht.

Weil ein flaches Dateisystem (das später in diesem Kapitel beschrieben wird) keine Metadaten enthält, müssen Anwendungen, die mit flachen Dateien arbeiten, das Äquivalent zu den Metadaten als Teil des Anwendungsprogramms enthalten.

Datenbankgröße und -komplexität

Datenbanken gibt es in allen möglichen Größen, angefangen bei einer einfachen Sammlung weniger Datensätze bis hin zu Millionen von Datensätzen. Die meisten Datenbanken lassen sich einer von drei Kategorien zuordnen, die von der Größe der Datenbank selbst, der Größe der Computer, auf denen sie laufen, und der Größe der Organisationen, von denen sie betrieben werden, abhängen:

- ✔ Eine *persönliche Datenbank* ist für die Benutzung durch eine einzige Person auf einem einzigen Computer bestimmt. Eine solche Datenbank ist normalerweise ziemlich einfach strukturiert und relativ klein.
- ✔ Eine *Abteilungs-* oder *Arbeitsgruppendatenbank* ist für die Benutzung durch die Mitglieder einer einzelnen Abteilung oder Arbeitsgruppe innerhalb eines Unternehmens vorgesehen. Diese Art von Datenbank ist im Allgemeinen größer als eine persönliche Datenbank und demgemäß auch komplexer, weil sie mehrere Benutzer verwalten muss, die gleichzeitig auf dieselben Daten zugreifen können.
- ✔ Eine *Unternehmensdatenbank* kann riesig sein. Unternehmensdatenbanken können den gesamten geschäftskritischen Informationsfluss großer Unternehmen modellieren.

Was ist ein Datenbankverwaltungssystem?

Gut, dass Sie das fragen. Ein *Datenbankverwaltungssystem* (*DBMS*, im Englischen heißt das *Database Management System*) ist ein Satz von Programmen, mit denen Sie Datenbanken und die dazugehörigen Anwendungen definieren, verwalten und ausführen können. Eine *verwaltete Datenbank* ist im Wesentlichen eine Struktur, um Daten zu speichern, die für Sie oder Ihr Unternehmen wichtig sind. Ein DBMS ist ein Werkzeug, mit dem Sie eine solche Struktur erstellen können, um mit den darin enthaltenen Daten zu arbeiten.

Heute gibt es viele Datenbankverwaltungssysteme auf dem Markt. Einige laufen nur auf Mainframes und einige auf PCs, Laptops oder Tablets. Die Entwicklung geht jedoch in die Richtung von Produkten, die auf mehreren Plattformen und in verschiedenen Netzwerken mit Rechnern verschiedener Klassen arbeiten können. Ein noch stärkerer Trend geht dahin, Daten in der *Cloud* zu speichern, einem Speicher, der in der Regel von großen Unternehmen wie etwa Amazon, Google oder Microsoft über das Internet zur Verfügung gestellt wird. Unternehmen können ihre Daten aber auch in einer privaten Cloud im firmeninternen Intranet speichern.

Heute ist *Cloud* das angesagte IT-Modewort, nur nicht zu scharf definiert, aber in aller Munde. Tatsächlich besteht eine Cloud aus einer Sammlung von Computer-Ressourcen, auf die per Browser über das Internet oder ein privates Intranet zugegriffen werden kann. Der Unterschied zwischen Computer-Ressourcen in der Cloud und ähnlichen Ressourcen in einem physischen Rechenzentrum besteht in der Art des Zugriffs. In der Cloud erfolgt der Zugriff per Browser, während herkömmliche Anwendungsprogramme direkt auf diese Ressourcen zugreifen.

Ein DBMS, das auf mehreren verschiedenen großen und kleinen Plattformen läuft, wird als *skalierbar* bezeichnet.

Unabhängig von der Größe des Computers, auf dem die Datenbank läuft, und unabhängig davon, ob der Rechner in ein Netzwerk eingebunden ist, bleibt der Informationsfluss zwischen der Datenbank und dem Benutzer gleich. Abbildung 1.1 zeigt, wie der Benutzer über das DBMS mit der Datenbank kommuniziert. Das Datenbankverwaltungssystem verbirgt die physischen Details der Datenbankspeicherung, weshalb die Anwendung nur die logischen Eigenschaften der Daten, nicht aber deren physische Speicherform kennen muss.

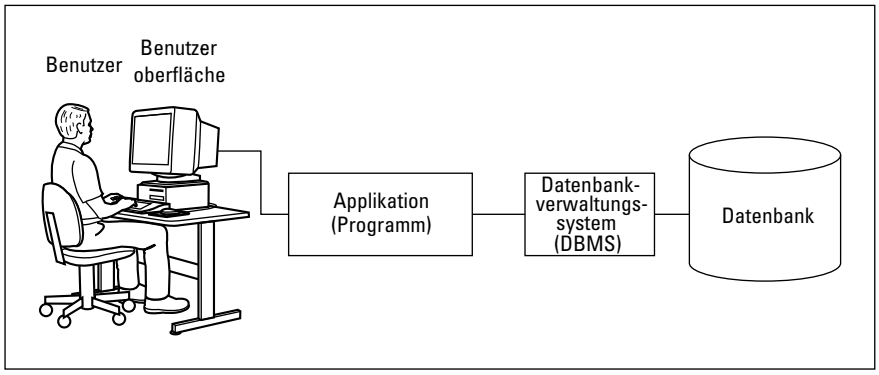


Abbildung 1.1: Blockdiagramm eines DBMS-Informationssystems

Flache Dateien

Die einfachste Sammlung von strukturierten Daten ist eine sogenannte *flache Datei*. Flache Dateien haben eine minimale Struktur. Sie bestehen einfach aus einer Sammlung von Datensätzen, die nacheinander in einem bestimmten Format in einer Liste angeordnet sind – sie enthält die Daten, die ganzen Daten und nichts als die Daten. Bezogen auf Computer sind flache Dateien sehr einfach aufgebaut. Weil die Datei keine Informationen über ihre eigene Struktur enthält (Metadaten), ist ihr Overhead minimal (Informationen in der Datei, die selbst keine Daten sind, aber Speicherplatz belegen).

Angenommen, Sie wollten die Namen und Adressen der Kunden Ihrer Firma in einem flachen Dateisystem verwalten. Das System soll ähnlich wie das folgende Beispiel strukturiert sein:

```
Klaus Klawuttke   Heimweg 15     12345 Berlin
Jerry Cotton      Gasse 77       48787 Hermannsburg
Adrian Hansen     Perlenallee 3  58789 Oberstadt
Johann Bäcker     Terrenshof 128 98755 Germsdorf
Michael Pens      Kalauerweg 1   57730 Köln
Barbara Michimoto Sandufer 9     25252 Kiel
Linda Schmidt     Vennweg 56     44134 Düsseldorf
Robert Funnell    Kölner Ring 9  24240 Lübeck
Boris Reckal      Am Hof 9       59554 Siegen
Kevin Kirenberg   Pferdeweg 7    35305 Kassel
```

Diese Datei enthält nur Daten. Jedes Feld hat eine feste Länge (das Namensfeld ist beispielsweise immer genau 18 Zeichen lang) und ein Feld wird vom anderen nicht durch Strukturen getrennt. Die Person, die die Datenbank entworfen hat, hat die Feldpositionen und Längen festgelegt. Jedes Programm, das diese Datei benutzt, muss diese Felddefinitionen »kennen«, da diese Informationen nicht in der Datenbank selbst gespeichert sind.

Wegen des geringen Overheads kann man mit flachen Dateien sehr schnell arbeiten. Nachteilig ist jedoch, dass Anwendungsprogramme zusätzliche Logik enthalten müssen, um die Daten einer flachen Datei auf Detailebene bearbeiten zu können. Die Anwendung muss genau wissen, wo und wie die Daten in der Datei gespeichert sind. Flache Dateien eignen sich für kleinere Systeme. Je größer jedoch ein System ist, desto umständlicher wird ein flaches Dateisystem.

Wenn Sie stattdessen eine Datenbank benutzen, können Sie den Doppelaufwand verringern. Anwendungen können leichter auf andere Hardware- und Betriebssystemplattformen übertragen werden und das Schreiben von Anwendungsprogrammen wird erleichtert, weil der Programmierer die Details über den Speicherort und die Speicherform der Daten nicht kennen muss.

Datenbanken vermeiden Doppelarbeit, weil das DBMS die Details der Datenbearbeitung übernimmt. Bei Anwendungen, die mit flachen Dateien arbeiten, müssen alle Einzelheiten im Anwendungscode programmiert werden. Wenn mehrere Anwendungen auf dieselben flachen Dateien zugreifen, muss jede Anwendung (redundant) den gesamten Code zur Datenbearbeitung enthalten. Bei einem DBMS ist dieser Code in der Anwendung überflüssig.

Verständlicherweise ist es nicht einfach, eine Anwendung von einer Plattform auf eine andere zu portieren, wenn sie mit flachen Dateien arbeitet und plattformspezifischen Code zur Datenmanipulation enthält, weil Sie zunächst gezwungen sind, den gesamten spezifischen Hardware-Code zu ändern. Es ist sehr viel einfacher, eine DBMS-basierte Anwendung auf eine andere Plattform zu portieren.

Datenbankmodelle

Die ersten Datenbanken, die in den 1950er Jahren eingeführt wurden, waren nach dem hierarchischen Modell strukturiert. Sie litten unter Redundanzproblemen, und die Inflexibilität ihrer Struktur erschwerte Datenbankänderungen. Bald darauf folgten Datenbanken, die nach dem Netzwerkmodell strukturiert waren und die Hauptnachteile der hierarchischen Datenbanken beseitigen sollten. Netzwerkdatenbanken verfügen über eine minimale Redundanz auf Kosten einer hohen strukturellen Komplexität.

Einige Jahre später entwickelt Dr. E.F. Codd bei IBM das relationale Modell, das eine minimale Redundanz mit einer leicht verständlichen Struktur verbindet. Die Sprache SQL wurde für das Arbeiten mit relationalen Datenbanken entwickelt. Relationale Datenbanken haben hierarchische Datenbanken und Netzwerkdatenbanken fast vollständig verdrängt.

Sei einigen Jahren drängen verstärkt sogenannte *NoSQL-Datenbanken* auf den Markt. Sie sind nicht wie relationale Datenbanken strukturiert und arbeiten nicht mit der SQL-Sprache. NoSQL-Datenbanken werden in diesem Buch nicht behandelt.

Das relationale Modell

E.F. Codd, der bei IBM arbeitete, formulierte das relationale Datenbankmodell erstmals im Jahre 1970. Es dauerte etwa ein Jahrzehnt, bis dieses Modell für Datenbankprodukte verwendet wurde. Es entbehrt nicht einer gewissen Ironie, dass nicht die Firma IBM das erste relationale DBMS lieferte. Diese Ehre kam einer kleinen, neu gegründeten Firma zu, die ihr Produkt *Oracle* nannte.

Relationale Datenbanken haben Datenbanken der früheren Modelle weitgehend verdrängt, weil man die Struktur einer relationalen Datenbank ändern kann, ohne die Anwendungen ändern zu müssen, die mit der alten Struktur gearbeitet haben. Angenommen, Sie wollten eine oder mehrere neue Spalten in eine Datenbanktabelle einfügen. Sie müssen vorher geschriebene Anwendungen, die mit dieser Tabelle arbeiten, nicht ändern, es sei denn, Sie ändern eine oder mehrere der Spalten, die in den Anwendungen verwendet werden.

Wenn Sie natürlich eine Spalte löschen, auf die eine vorhandene Anwendung Bezug nimmt (sprich: die von einer vorhandenen Anwendung referenziert wird), haben Sie Probleme, und zwar unabhängig von dem verwendeten Datenbankmodell. Eine der wirksamsten Methoden, einen Fehler in einer Datenbankanwendung auszulösen, besteht darin, Daten aus Spalten abzurufen, die in der Datenbank nicht enthalten sind.

Komponenten einer relationalen Datenbank

Die Flexibilität relationaler Datenbanken beruht darauf, dass ihre Daten in Tabellen gespeichert werden, die im Wesentlichen unabhängig voneinander sind. Sie können in einer Tabelle Daten einfügen, ändern oder löschen, ohne dadurch die Daten in den anderen Tabellen zu beeinflussen, vorausgesetzt, die betroffene Tabelle ist den anderen Tabellen nicht *übergeordnet*. (Die Beziehungen der Über- und Unterordnung zwischen Tabellen werden in Kapitel 5 erklärt.) In diesem Abschnitt zeige ich, woraus diese Tabellen bestehen und in welcher Beziehung sie zu den Komponenten einer relationalen Datenbank stehen.

Was sind Relationen?

Datenbanken bestehen aus Tabellen, die miteinander verknüpft sind. In der Datenbankterminologie werden diese Tabellen auch als *Relationen* bezeichnet. Eine relationale Datenbank besteht also aus einer oder mehreren Relationen.

Eine *Relation* ist eine zweidimensionale, aus Zeilen und Spalten bestehende Anordnung von Feldern, auch Array genannt, die nur Einträge eines Typs von Datenwerten und keine identischen Zeilen enthalten kann. Jede Zelle des Arrays darf nur einen Wert enthalten. Keine zwei Zeilen dürfen identisch sein.

Ein Beispiel: Die meisten Benutzer sind mit zweidimensionalen, aus Zeilen und Spalten bestehenden Arrays in Form von elektronischen Tabellenkalkulationen (wie zum Beispiel *Microsoft Excel*) vertraut. Die Offensivstatistik, die auf der Rückseite der Baseball-Karte jedes Baseball-Spielers der Major-League aufgedruckt ist, ist ein weiteres Beispiel für ein Datenfeld. Die Baseball-Karte enthält Spalten für das Jahr, die Mannschaft (Team), die geleisteten Spiele sowie für die verschiedenen Leistungskriterien (At Bat, Hits, Runs und so weiter), die in der Baseball-Welt eine Rolle spielen. Eine Zeile fasst die Daten eines Jahres zusammen, in dem der Spieler in der Major-League gespielt hat. Sie können diese Daten auch in einer Relation (einer Tabelle) speichern, die dieselbe Grundstruktur hat. Abbildung 1.2 zeigt eine relationale Datenbanktabelle, die die Offensivstatistik eines Spielers der Major-League enthält. In der Praxis würde eine solche Tabelle die statistischen Daten einer ganzen Mannschaft oder möglicherweise der gesamten Liga enthalten.

Spieler	Jahr	Team	Spiel	At Bat	Hits	Runs	RBI	2B	3B	HR	Walk	Steals	Bat. Avg.
Roberts	1988	Padres	5	9	3	1	0	0	0	0	1	0	.333
			117	329	99	81	25						
Roberts	1989	Padres			172	104		15	8	3	49	21	.301

Abbildung 1.2: Tabelle mit der Offensivstatistik eines Baseball-Spielers

Spalten in der Tabelle sind in dem Sinne *konsistent*, als eine Spalte in jeder Zeile dieselbe Bedeutung hat. Wenn eine Spalte in einer Zeile den Nachnamen eines Spielers enthält, muss die Spalte in allen Zeilen den Nachnamen eines Spielers enthalten. Die Reihenfolge, in der Zeilen und Spalten in einer Tabelle erscheinen, spielt dabei keine Rolle. Aus der Sicht des DBMS ist

es deshalb gleichgültig, welche Spalte an erster, welche an zweiter und welche an letzter Stelle steht. Das DBMS verarbeitet die Tabelle unabhängig davon, wie sie aufgebaut ist.

Jede Spalte einer Datenbanktabelle verkörpert ein einzelnes Attribut der Tabelle. Die Bedeutung einer Spalte ist in allen Zeilen der Tabelle gleich. Eine Tabelle kann beispielsweise die Namen, Adressen und Telefonnummern aller Kunden einer Firma enthalten. Jede Zeile in der Tabelle (auch *Datensatz* oder *Tupel* genannt) enthält die Daten eines einzelnen Kunden. Jede Spalte enthält ein einzelnes *Attribut* des Kunden, wie zum Beispiel die Kundennummer (`KundenNr`), die Firma (`Unternehmen`), die Straße (`Anschrift1`) und so weiter. Abbildung 1.3 zeigt einige Zeilen und Spalten einer solchen Tabelle.

Abbildung 1.3: Jede Tabellenzeile enthält einen Datensatz; jede Spalte enthält ein einzelnes Attribut.

Views oder Sichten

Tabellen können viele Spalten und Zeilen enthalten. Manchmal interessieren Sie sich für alle Daten und manchmal nicht. Möglicherweise möchten Sie nur einige der Spalten einer Tabelle sehen oder nur Zeilen, die eine bestimmte Bedingung erfüllen. Vielleicht möchten Sie einige Spalten einer Tabelle und einige andere Spalten einer mit ihr verknüpften Tabelle sehen. Um die Daten auszuschließen, die Sie in diesem Augenblick nicht interessieren, erstellen Sie eine sogenannte *Sicht* (englisch *View*). Eine Sicht ist eine Untermenge einer Datenbank. Diese Untermenge kann von einer Anwendung verarbeitet werden. Eine Sicht kann Ausschnitte einer oder mehrerer Tabellen enthalten.

Sichten werden manchmal auch als *virtuelle Tabellen* bezeichnet. Für eine Anwendung oder die Benutzer verhalten sich Sichten wie Tabellen. Sichten existieren jedoch nicht unabhängig von Tabellen. Mit Views können Sie Daten betrachten, aber sie gehören nicht zu den Daten.

Angenommen, Sie arbeiteten mit einer Datenbank, die die Tabellen `Kunde` und `Rechnung` enthielte. Die Tabelle `Kunde` enthielte die Spalten `KundenNr`, `Vorname`, `Nachname`, `Strasse`, `Ort`, `Land`, `Plz` und `Tel`. Die Tabelle `Rechnung` verfüge über die Spalten `RechnungNr`, `KundenNr`, `Datum`, `Umsatz`, `Bezahlt` und `Zahlungsart`.

Ein Verkaufsleiter möchte nur den Vornamen (`Vorname`), den Nachnamen (`Nachname`) und die Telefonnummer (`Tel`) eines Kunden auf dem Bildschirm sehen. Der Verkaufsleiter kann nun aus der Tabelle `Kunde` eine Sicht erstellen, die nur die drei gewünschten Spalten und nicht auch noch die im Moment überflüssigen Informationen aus anderen Spalten enthält. Abbildung 1.4 zeigt die vom Verkaufsleiter erstellte Sicht.

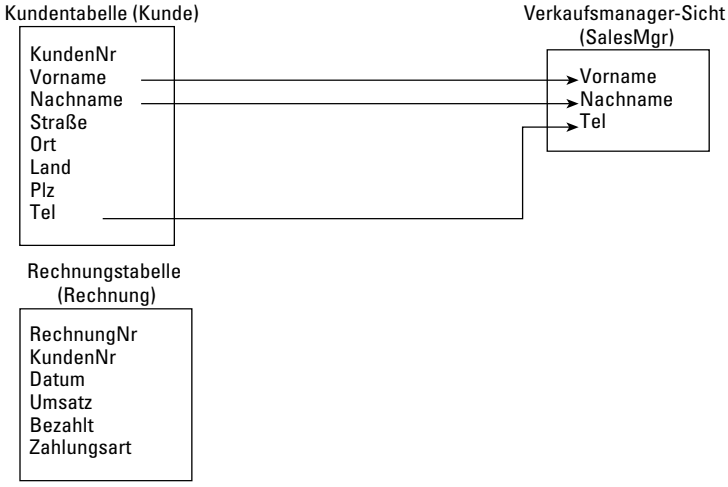


Abbildung 1.4: Die Sicht des Verkaufsmanagers wird von der Tabelle KUNDE abgeleitet.

Ein Zweigstellenleiter möchte die Namen und Telefonnummern aller Kunden sehen, deren Postleitzahl zwischen 90000 und 93999 liegt. Eine Sicht, die die von ihr abgerufenen Zeilen und angezeigten Spalten mit einer sogenannten Einschränkung, einem Filter, versieht, führt diese Aufgabe aus. Abbildung 1.5 zeigt die Datenquellen für die Spalten der Sicht des Zweigstellenleiters.

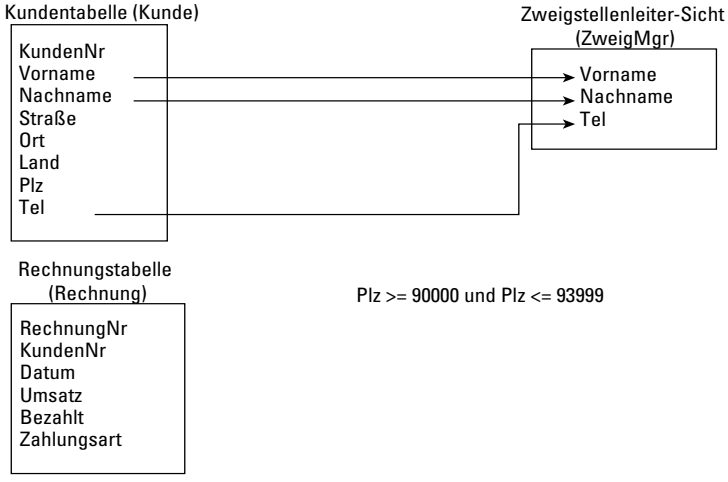


Abbildung 1.5: Die Sicht des Zweigstellenleiters enthält nur bestimmte Zeilen der Tabelle KUNDE.

Der Buchhaltungsleiter möchte die Kundennamen der Tabelle `Kunde` und die Spalten `Datum`, `Umsatz`, `Bezahlt` und `Zahlungsart` der Tabelle `Rechnung` sehen, bei denen `Bezahlt` kleiner als `Umsatz` ist. Die zuletzt genannte Bedingung ist erfüllt, wenn eine Rechnung noch nicht vollständig beglichen wurde. Für diese Sicht müssen Sie Daten aus beiden Tabellen kombinieren. Abbildung 1.6 zeigt, wie Daten der Tabellen `Kunde` und `Rechnung` in die Sicht des Buchhaltungsleiters einfließen.

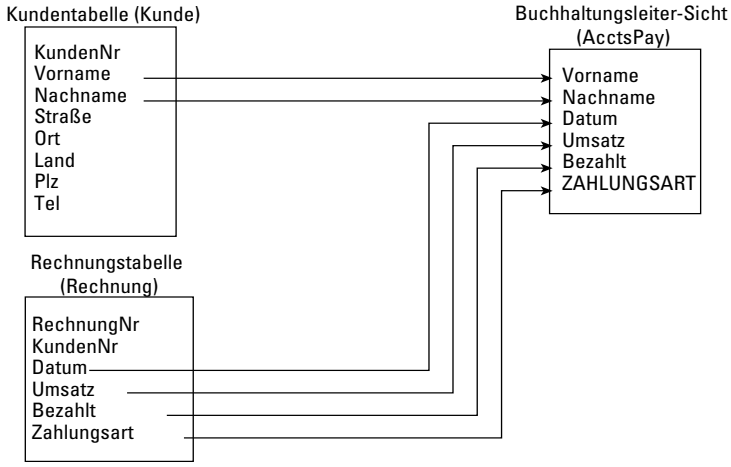


Abbildung 1.6: Die Sicht des Buchhaltungsleiters kombiniert Daten aus zwei Tabellen.

Sichten sind nützlich, weil sie es ermöglichen, Daten aus einer Datenbank zu extrahieren und zu formatieren, ohne die gespeicherten Daten wirklich zu ändern. Zugleich schützen sie auch die Daten, die Sie nicht zeigen wollen, weil diese Daten nicht in den Sichten enthalten sind. In Kapitel 6 zeige ich, wie Sie mit SQL eine Sicht erstellen.

Schemata, Domänen und Einschränkungen

Eine Datenbank ist mehr als eine Sammlung von Tabellen. Zusätzliche Strukturen, die über mehrere Schichten verteilt sind, helfen, die Integrität der Daten zu bewahren. Das *Schema* einer Datenbank stellt eine übergreifende Organisation der Tabellen dar. Die *Domäne* einer Tabellenspalte sagt Ihnen, welche Werte Sie in der Spalte speichern dürfen. Sie können *Einschränkungen* (englisch *Constraints*) für eine Datenbanktabelle definieren, um zu verhindern, dass jemand (Sie eingeschlossen) ungültige Daten in der Tabelle speichert.

Schemata

Die Struktur einer ganzen Datenbank wird als ihr *Schema* oder als *konzeptionelle Sicht* und manchmal auch als *vollständige logische Sicht* der Datenbank bezeichnet. Das Schema gehört zu den Metadaten und ist damit ein Teil der Datenbank. Die Metadaten selbst, die die Struktur der Datenbank beschreiben, werden wie normale Daten in Tabellen gespeichert. Damit sind sogar Metadaten nichts anderes als Daten, und das ist das Gute an ihnen.

Domänen

Ein *Attribut* einer Relation (das heißt die Spalte einer Tabelle) kann eine endliche Anzahl von Werten annehmen. Die Gesamtmenge dieser Werte wird als die *Domäne* des Attributs bezeichnet.

Angenommen, Sie seien ein Autohändler, der das neue Curarri GT 4000 Sportcoupé verkauft. Sie verwalten die Autos, die Sie am Lager haben, in einer Datenbanktabelle mit dem Namen `LAGER`. Eine Spalte dieser Tabelle hat den Namen `Farbe`, in der die Farbe jedes Autos gespeichert wird. Der GT 4000 wird nur in vier Farben geliefert: Lippenrot, Mitternachtsschwarz, Schneeweiß und Metallicgrau. Diese vier Farben bilden die Domäne des Attributs `Farbe`.

Einschränkungen

Einschränkungen (englisch *Constraints*) sind eine wichtige, gleichwohl oft übersehene Komponente einer Datenbank. Einschränkungen sind Regeln, die festlegen, welche Werte die Attribute einer Tabelle annehmen dürfen.

Wenn Sie Einschränkungen für eine Spalte definieren, können Sie verhindern, dass die Benutzer in diese Spalte ungültige Daten eingeben. Natürlich muss jeder Wert, der in der Domäne der Spalte gültig wäre, auch alle ihre Einschränkungen berücksichtigen. Wie ich im vorangegangenen Abschnitt erläutert habe, bildet die Menge aller Werte, die die Spalte enthalten darf, ihre Domäne. Eine Einschränkung schränkt diese Werte ein. Die Eigenschaften einer Tabellenspalte definieren zusammen mit den Einschränkungen, die auf diese Spalte angewendet werden, die Spaltendomäne.

Bei dem Beispiel des Autohändlers könnten Sie eine Einschränkung so definieren, dass die Datenbank in der Spalte `Farbe` nur die vier zulässigen Werte akzeptiert. Falls ein Benutzer dann versuchen sollte, eine andere Farbe, wie zum Beispiel `Waldgrün`, in die entsprechende Spalte einzugeben, weist das System die Eingabe zurück. Die Dateneingabe kann erst dann fortgesetzt werden, wenn der Benutzer eine gültige Farbe in das Feld `Farbe` eingibt.

Vielleicht fragen Sie sich, was passiert, wenn der Hersteller Curarri eine waldgrüne Version des GT 4000 als Sommeroption anbietet. Die Antwort impliziert eine Arbeitsplatzgarantie für Datenbankprogrammierer, denn derartige Dinge passieren immer wieder und erfordern Anpassungen der Datenbankstruktur. Dann können nur Fachleute (zu denen auch Sie gehören), die wissen, wie man die Struktur ändert, größere Katastrophen verhindern.

Das Objektmodell fordert das relationale Modell heraus

Das relationale Modell hat in vielen Anwendungsbereichen einen fantastischen Erfolg gehabt. Es ist jedoch nicht frei von Problemen. Diese Probleme wurden mit zunehmender Beliebtheit objektorientierter Programmiersprachen, zum Beispiel C++, Java und C#, immer deutlicher. Solche Sprachen können komplexere Probleme leichter handhaben als traditionelle Sprachen, weil sie über Funktionen verfügen, wie beispielsweise durch Benutzer erweiterbare Datentypen, Kapselung, Vererbung, dynamische Bindung von Methoden, komplexe und zusammengesetzte Objekte und Objektidentitäten.

Ich werde nicht alle diese Begriffe in diesem Buch erklären (obwohl Sie einigen später begegnen werden). Es reicht aus zu wissen, dass das klassische relationale Modell nicht mit allen diesen Funktionalitäten klarkommt. Deshalb werden Datenbankverwaltungssysteme entwickelt und angeboten, die auf dem Objektmodell basieren. Allerdings haben sie zurzeit nur einen relativ kleinen Marktanteil.

Das objektrelationale Modell

Datenbankentwickler streben wie fast jedermann danach, für ihre Aufgaben die beste aller Lösungen zu finden. Deshalb versuchten einige, die Vorteile eines objektorientierten Datenbanksystems mit denen des erfolgreichen relationalen Systems zu verbinden und dabei die Kompatibilität zu Letzterem zu bewahren. Das Ergebnis dieser Bemühungen ist das objektrelationale Hybridmodell. Objektrelationale Datenbankverwaltungssysteme erweitern das relationale Modell um die Fähigkeit, Daten objektorientiert zu modellieren. Der internationale SQL-Standard wurde um objektorientierte Funktionalitäten erweitert. Damit erhalten die Anbieter relationaler Datenbankverwaltungssysteme die Möglichkeit, ihre Produkte zu objektrelationalen Datenbankverwaltungssystemen auszubauen und dabei die Kompatibilität zum Standard zu wahren. Im Gegensatz zu dem SQL-92-Standard, der ein rein relationales Datenbankmodell definiert, beschreibt SQL:1999 deshalb ein objektrelationales Datenbankmodell. SQL:2003 enthält mehr objektorientierte Funktionen; und die nachfolgenden SQL-Versionen sind in dieser Richtung noch weiter fortgeschritten.

Ich beschreibe in diesem Buch den internationalen ISO-/IEC-SQL-Standard. (IEC steht für *International Electrotechnical Commission*, aber das spielt eigentlich keine Rolle. Wie viele Menschen wissen, wofür die Buchstaben der Abkürzung *LASER* stehen?) Dabei handelt es sich hauptsächlich um ein relationales Datenbankmodell. Außerdem beschreibe ich die objektorientierten Erweiterungen, die mit SQL:1999 eingeführt wurden, sowie die zusätzlichen Erweiterungen aus späteren SQL-Versionen. Die objektorientierten Funktionalitäten des neuen Standards geben Entwicklern die Möglichkeit, SQL-Datenbanken für die Lösung von Problemen einzusetzen, die für den früheren, rein relationalen Ansatz zu komplex waren. Anbieter von DBMS-Systemen bauen die objektorientierten Funktionen aus dem ISO-Standard in ihre Produkte ein. Einige dieser Funktionen werden schon seit Jahren angeboten, während andere noch nicht eingeschlossen worden sind.

Überlegungen zum Datenbankentwurf

Eine Datenbank ist eine Repräsentation einer physischen oder konzeptionellen Struktur, wie eines Unternehmens, der Montage eines Autos oder der Erfolgsstatistik aller Mannschaften der Bundesliga. Die Genauigkeit dieser Repräsentation hängt davon ab, wie detailliert Ihr Datenbankentwurf ist. Der Aufwand, den Sie in den Datenbankentwurf stecken, sollte von der Art der Informationen abhängen, die Ihnen die Datenbank liefern soll. Zu viele Details bedeuten eine Verschwendung von Arbeit, Zeit und Speicherplatz. Zu wenige Details können die Datenbank wertlos werden lassen.

SQL-Grundlagen

2

In diesem Kapitel

- SQL verstehen
- Mit falschen Vorstellungen über SQL aufräumen
- Die verschiedenen SQL-Standards betrachten
- Standard-SQL-Anweisungen und reservierte Wörter kennenlernen
- Zahlen, Zeichen, Datumsangaben, Zeiten und andere Datentypen darstellen
- Nullwerte und Einschränkungen untersuchen
- SQL in einem Client/Server-System einsetzen
- SQL im Netzwerk

SQL ist eine flexible Sprache, die Sie für verschiedene Aufgaben verwenden können. SQL ist das meistbenutzte Werkzeug für die Kommunikation mit relationalen Datenbanken. In diesem Kapitel beschreibe ich zunächst, was SQL ist und was es nicht ist – insbesondere, was SQL von anderen Computersprachen unterscheidet. Dann stelle ich die Anweisungen und Datentypen vor, die Standard-SQL unterstützt, und erläutere die Schlüsselbegriffe *Nullwerte* und *Einschränkungen*. Schließlich erhalten Sie einen Überblick darüber, wie SQL in Client/Server-Umgebungen, im Internet und im Intranet eines Unternehmens eingesetzt werden kann.

Was SQL ist und was es nicht ist

Das Erste, was Sie von SQL verstehen müssen, ist, dass es sich dabei nicht um eine *prozedurale Sprache* wie BASIC, C, C++ oder Java handelt. Um in einer dieser Sprachen ein Problem zu lösen, schreiben Sie eine Prozedur, die nacheinander eine Reihe von Operationen ausführt, bis die Aufgabe erledigt ist. Die Prozedur kann aus einer linearen Abfolge von Schritten bestehen oder Schleifen enthalten, aber auf jeden Fall legt der Programmierer die Verarbeitungsreihenfolge fest.

SQL ist dagegen *nicht prozedural*. Um ein Problem mit SQL zu lösen, teilen Sie SQL einfach mit, *was* Sie wollen (so, als wenn Sie zu Aladins Flaschengeist sprächen), anstatt dem System zu sagen, *wie* es die Aufgabe lösen soll. Das Datenbankmanagementsystem (DBMS) wählt den besten Weg aus, um die Aufgabe zu erfüllen.

Ich habe gerade erwähnt, dass SQL keine prozedurale Sprache ist. Dies ist im Wesentlichen wahr. Nun sind Millionen von Programmierern (zu denen möglicherweise auch Sie gehören) gewohnt, Probleme prozedural zu lösen. Deshalb gab es in den letzten Jahren Bestrebungen, SQL um prozedurale Funktionalitäten zu erweitern. Aus diesem Grund sind nun in SQL Be-

standteile prozeduraler Sprachen, wie `BEGIN`-Blöcke, `IF`-Anweisungen, Funktionen und Prozeduren, eingebaut worden. Deshalb können Sie auf einem Server auch Anwendungen ablegen, die von diversen Clients wiederholt verwendet werden können.

Um zu verdeutlichen, was ich mit »dem System sagen, was Sie wollen« meine, nehmen Sie an, dass Sie mit einer Mitarbeitertabelle arbeiten und alle Zeilen abrufen möchten, in denen die Mitarbeiter mit der meisten Berufserfahrung gespeichert sind, wobei jeder als »erfahren« gelten soll, der älter als 40 Jahre ist oder mehr als 60.000 Euro pro Jahr verdient. Sie können die gewünschten Daten erhalten, indem Sie die folgende Abfrage verwenden:

```
SELECT * FROM Mitarbeiter WHERE Lebensalter > 40
   OR Gehalt > 60000 ;
```

Diese Anweisung ruft alle Zeilen einer Tabelle `Mitarbeiter` ab, bei denen entweder der Wert in der Spalte `Lebensalter` größer als 40 oder der Wert in der Spalte `Gehalt` größer als 60.000 ist. In SQL müssen Sie nicht selbst festlegen, wie die Informationen abgerufen werden sollen. Stattdessen untersucht die Datenbank-Engine (die Software-Komponente, die die Interaktion und die Arbeit mit der Datenbank ermöglicht) die Datenbank und entscheidet dann, wie Ihr Auftrag am besten ausgeführt wird. Sie müssen nur die Daten angeben, die Sie abrufen möchten.

Eine *Abfrage* ist eine Frage, die Sie an die Datenbank richten. Wenn es in der Datenbank Daten gibt, die die Bedingungen (Kriterien) Ihrer Abfrage erfüllen, ruft SQL diese Daten ab.

Den aktuellen Implementierungen von SQL fehlen viele der grundlegenden Sprachkonstrukte der meisten anderen Programmiersprachen. Fast alle praktischen Anwendungen benötigen mindestens einige dieser Sprachkonstrukte; deshalb ist SQL eigentlich keine vollständige Programmiersprache, sondern (nur) eine Spezialsprache zur Datenbearbeitung. Selbst mit den SQL-Erweiterungen von 1999, 2003, 2005 und 2008 müssen Sie SQL zusammen mit einer prozeduralen Sprache wie etwa C verwenden, um eine komplette Anwendung zu erstellen.

Sie können Daten aus einer Datenbank mit zwei Methoden extrahieren:

- ✔ **Mit einer *Ad-hoc-Abfrage* von der Computerkonsole aus, bei der Sie einfach eine SQL-Anweisung eintippen und das Ergebnis vom Bildschirm ablesen.** *Konsole* ist der traditionelle Ausdruck für die Hardware, die die Aufgabe der Tastatur und des Bildschirms heutiger PCs übernimmt. Abfragen von der Computerkonsole aus liefern schnelle Antworten auf spezifische Fragen. So können Sie auch sofort Daten aus einer Datenbank abrufen, die Sie sonst nicht benötigen. Geben Sie die entsprechende SQL-Abfrage über die Tastatur ein und lesen Sie das Ergebnis innerhalb kürzester Zeit vom Bildschirm ab.
- ✔ **Mit der Ausführung eines Programms, das die Daten aus der Datenbank abruft und in einen Bericht schreibt, der entweder auf dem Bildschirm oder über einen Drucker ausgegeben wird.** Eine SQL-Abfrage direkt in ein Programm einzubauen, eignet sich für komplexe Abfragen, die Sie vielleicht wiederverwenden wollen. So müssen Sie die Abfrage nur einmal formulieren. In Kapitel 15 wird erklärt, wie Sie SQL-Code in Programme einbinden können, die in einer anderen Sprache geschrieben sind.

Ein (sehr) kurzer historischer Überblick

SQL wurde, wie die Theorie der relationalen Datenbanken, in den Forschungslaboratorien von IBM entwickelt. In den frühen 1970er-Jahren, als die IBM-Forscher anfingen, sich mit relationalen Datenbankmanagementsystemen (oder RDBMS) zu beschäftigen, entwickelten sie eine Datenuntersprache, um mit diesen Systemen arbeiten zu können. Ursprünglich nannten sie diese Untersprache *SEQUEL* (*S*tructured *E*nglish *QUE*ry *L*anguage). Doch als sie die Abfragesprache offiziell auf den Markt bringen wollten, stellten sie fest, dass »Sequel« als Markenname bereits an ein anderes Unternehmen vergeben war. Deshalb suchten die Marketing-Genies von IBM einen Namen, der sich zwar von SEQUEL unterschied, aber dennoch als zur selben Familie gehörig erkennbar war. So kamen sie auf den Namen *SQL*.

Die Syntax von SQL ist eine Art von strukturiertem Englisch, was auch der Grund für den ursprünglichen Namen war. Doch SQL ist keine strukturierte *Sprache* in dem Sinne, wie er von Informatikern verstanden wird. Deshalb ist »SQL«, entgegen der weitverbreiteten Annahme, keine Abkürzung für »Structured Query Language« (»Strukturierte Abfragesprache«), sondern einfach eine Folge von drei Buchstaben, die für nichts stehen, ähnlich wie der Name der C-Sprache für nichts steht.

Schon bevor IBM 1981 das RDBMS *SQL/DS* einführte, war das Unternehmen für seine Arbeit an relationalen Datenbanken und SQL in der Branche bekannt. Damals hatte bereits eine andere Firma namens *Relational Software Inc.* (heute *Oracle Corporation*) ihr erstes RDBMS auf den Markt gebracht. Diese ersten Produkte wurden sofort zum Standard für eine neue Klasse von Datenbankverwaltungssystemen. Sie enthielten SQL, das zum De-facto-Standard für Datenuntersprachen wurde. Anbieter anderer relationaler Datenbankverwaltungssysteme brachten eigene Versionen von SQL heraus. Diese anderen Implementierungen enthielten üblicherweise alle Kernfunktionen der IBM-Produkte sowie Erweiterungen, die die speziellen Stärken des jeweiligen RDBMS nutzten. Deshalb war die plattformübergreifende Kompatibilität nur schwach ausgeprägt.

Eine *Implementierung* ist ein bestimmtes RDBMS, das auf einer bestimmten Hardware-Plattform läuft.

Deshalb kam es bald zu Bestrebungen, einen allgemein anerkannten SQL-Standard zu schaffen. 1986 veröffentlichte das ANSI (American National Standards Institute) einen formellen Standard mit dem Namen *SQL-86*. 1989 aktualisierte das ANSI diesen Standard unter dem Namen *SQL-89* und noch einmal 1992 (*SQL-92*). Die DBMS-Anbieter bemühten sich, neue Versionen ihrer Produkte an diesen Standard anzunähern, wodurch wir dem Ideal einer echten SQL-Portabilität viel näher kamen.

Die aktuelle vollständige Version des SQL-Standards ist *SQL:2011* (ISO/IEC 9075-X:2011). Ich beschreibe in diesem Buch SQL, wie es durch SQL:2011 definiert wird. Alle Implementierungen von SQL weichen mehr oder weniger von diesem Standard ab. Weil der volle Standard umfangreich ist, wird er von den meisten Implementierungen nicht vollständig unterstützt. Dennoch arbeiten die DBMS-Anbieter daran, die zentrale Teilmenge der SQL-Standardsprache zu unterstützen. Der komplette ISO/IEC-Standard kann unter `webstore.ansi.org` käuflich erworben werden, aber das ist wahrscheinlich nur erforderlich, wenn Sie ein eigenes ISO/IEC-SQL-Standard-Datenbankverwaltungssystem entwickeln wollen. Der Standard ist in einer *sehr* technischen Sprache geschrieben und eigentlich nur für Computer-Wissenschaftler verständlich.

SQL-Anweisungen

Die SQL-Befehlssprache besteht aus einer begrenzten Anzahl von Anweisungen, die drei mögliche Funktionen zur Datenverwaltung durchführen: Einige dienen der Datendefinition, andere der Datenbearbeitung und wieder andere der Datenkontrolle. Die Anweisungen zur Datendefinition und Datenbearbeitung werden in den Kapiteln 4 bis 13 behandelt und die Anweisungen zur Datenkontrolle in den Kapiteln 14 und 15.

Um mit dem Standard SQL:2011 konform zu sein, muss eine Implementierung alle Kernfunktionen enthalten. Darüber hinaus kann sie Erweiterungen des Kerns anbieten (die ebenfalls in der SQL:2011-Spezifikation beschrieben werden). Tabelle 2.1 enthält eine Liste der Kernfunktionen sowie die erweiterten Anweisungen von SQL:2011. Die Liste ist recht umfangreich. Programmierer, die gerne neue Funktionen ausprobieren, werden ihre Freude daran haben.

Reservierte Wörter

Zusätzlich zu den Anweisungen gibt es eine Reihe weiterer Wörter, die innerhalb von SQL eine besondere Bedeutung haben. Diese Wörter und die Anweisungen sind für spezielle Zwecke reserviert und dürfen nicht für Variablennamen oder auf andere Weise zweckentfremdet benutzt werden. Es ist leicht einzusehen, warum Tabellen, Spalten und Variablen keine Namen haben sollten, die zu den reservierten Wörtern gehören. Der Verwirrung wären Tür und Tor geöffnet, wenn Anweisungen wie die folgende zulässig wären:

```
SELECT SELECT FROM SELECT WHERE SELECT = WHERE ;
```

Eine vollständige Liste der reservierten SQL-Wörter finden Sie im Anhang A.

ADD	DEALLOCATE PREPARE	FREE LOCATOR
ALLOCATE CURSOR	DECLARE	GET DESCRIPTOR
ALLOCATE DESCRIPTOR	DECLARE LOCAL TEMPORARY TABLE	GET DIAGNOSTICS
ALTER DOMAIN	DELETE	GRANT PRIVILEGE
ALTER ROUTINE	DESCRIBE INPUT	GRANT ROLE
ALTER SEQUENCE GENERATOR	DESCRIBE OUTPUT	HOLD LOCATOR
ALTER TABLE	DISCONNECT	INSERT
ALTER TRANSFORM	DROP	MERGE
ALTER TYPE	DROP ASSERTION	OPEN
CALL	DROP ATTRIBUTE	PREPARE
CLOSE	DROP CAST	RELEASE SAVEPOINT
COMMIT	DROP CHARACTER SET	RETURN
CONNECT	DROP COLLATION	REVOKE
CREATE	DROP COLUMN	ROLLBACK
CREATE ASSERTION	DROP CONSTRAINT	SAVEPOINT
CREATE CAST	DROP DEFAULT	SELECT
CREATE CHARACTER SET	DROP DOMAIN	SET CATALOG
CREATE COLLATION	DROP METHOD	SET CONNECTION
CREATE DOMAIN	DROP ORDERING	SET CONSTRAINTS
CREATE FUNCTION	DROP ROLE	SET DESCRIPTOR
CREATE METHOD	DROP ROUTINE	SET NAMES
CREATE ORDERING	DROP SCHEMA	SET PATH
CREATE PROCEDURE	DROP SCOPE	SET ROLE
CREATE ROLE	DROP SEQUENCE	SET SCHEMA
CREATE SCHEMA	DROP TABLE	SET SESSION AUTHORIZATION
CREATE SEQUENCE	DROP TRANSFORM	SET SESSION CHARACTERISTICS
CREATE TABLE	DROP TRANSLATION	SET SESSION COLLATION
CREATE TRANSFORM	DROP TRIGGER	SET TIME ZONE
CREATE TRANSLATION	DROP TYPE	SET TRANSACTION
CREATE TRIGGER	DROP VIEW	SET TRANSFORM GROUP
CREATE TYPE	EXECUTE IMMEDIATE	START TRANSACTION
CREATE VIEW	FETCH	UPDATE
DEALLOCATE DESCRIPTOR		

Tabelle 2.1: SQL:2011-Anweisungen

Datentypen

Je nach ihrer Herkunft unterstützen verschiedene SQL-Implementierungen eine Vielzahl von Datentypen. Die SQL-Spezifikation erkennt jedoch nur sieben vordefinierte allgemeine Typen an:

- ✔ Numerics (Zahlen)
- ✔ Binary (Binärdaten)
- ✔ Strings (Zeichenketten)
- ✔ Booleans (Boolesche Werte)
- ✔ Datetimes (Datum-/Zeit-Werte)
- ✔ Intervals (Intervalle)
- ✔ XML

Innerhalb dieser allgemeinen Typen kann es mehrere Untertypen geben (genaue Zahlen, annähernd genaue Zahlen, Zeichenketten, Bitfolgen, große Objekt-Zeichenketten). Zusätzlich zu den integrierten, vordefinierten Typen unterstützt SQL auch Auflistungen, zusammengesetzte Typen und benutzerdefinierte Typen, auf die ich später in diesem Kapitel eingehe.

Wenn Sie mit einer SQL-Implementierung arbeiten, die Datentypen unterstützt, die nicht der SQL-Spezifikation entsprechen, können Sie die Portierbarkeit Ihrer Datenbank verbessern, indem Sie auf diese hier nicht beschriebenen Datentypen verzichten. Bevor Sie sich dazu benutzerdefinierte Datentypen erstellen und verwenden, sollten Sie prüfen, ob jedes DBMS, das Sie als Ziel möglicher Portierungen ins Auge gefasst haben, solche Datentypen unterstützt.

Genaue Zahlen

Wie Sie vielleicht bereits am Namen erraten haben, können Sie mit den Datentypen der Gruppe *genaue Zahlen* numerische Werte genau ausdrücken. Fünf Datentypen zählen zu dieser Kategorie:

- ✔ `INTEGER`
- ✔ `SMALLINT`
- ✔ `BIGINT`
- ✔ `NUMERIC`
- ✔ `DECIMAL`

Der Datentyp INTEGER

Der Datentyp `INTEGER` (Ganzzahl) hat keine Dezimalstellen. Seine Genauigkeit hängt von der SQL-Implementierung ab, daher kann der Datenbankentwickler die Genauigkeit nicht spezifizieren.

Die *Genauigkeit* einer Zahl ist die maximale Anzahl von Ziffern, aus denen diese Zahl bestehen kann.

Der Datentyp SMALLINT

Der Datentyp `SMALLINT` (kleine Ganzzahl) dient ebenfalls zur Speicherung von Ganzzahlen. Seine Genauigkeit darf in ein und derselben Implementierung nicht größer sein als die von `INTEGER`. Bei vielen Implementierungen sind `SMALLINT` und `INTEGER` gleich.

Wenn Sie in einer Tabellenspalte Ganzzahlen speichern wollen und wissen, dass deren Wertebereich nicht über die Genauigkeit von `SMALLINT`-Daten hinausgeht, sollten Sie der Spalte den Datentyp `SMALLINT` statt `INTEGER` zuweisen. Durch diese Zuweisung kann Ihr DBMS möglicherweise Speicherplatz sparen.

Der Datentyp BIGINT

Auch der Datentyp `BIGINT` (große Ganzzahl) dient zur Speicherung von Ganzzahlen. Seine Genauigkeit ist mindestens so groß wie die von `INTEGER`, kann aber auch größer sein. Die exakte Genauigkeit des Datentyps `BIGINT` ist von der jeweiligen Implementierung abhängig.

Der Datentyp NUMERIC

Der Datentyp `NUMERIC` dient zur Speicherung von Zahlen, die aus einer ganzzahligen Komponente und Nachkommastellen bestehen. Sie können die Genauigkeit und die Skalierung von `NUMERIC`-Daten vorgeben. (Sie erinnern sich? Die Genauigkeit ist die maximale Anzahl möglicher Ziffern.)

Die *Skalierung* einer Zahl ist die Anzahl der Ziffern nach dem Komma. Sie darf nicht negativ und nicht größer als die Genauigkeit der Zahl sein.

Wenn Sie den Datentyp `NUMERIC` verwenden, liefert Ihre SQL-Implementierung exakt die von Ihnen geforderte Genauigkeit und Skalierung. Sie können `NUMERIC` einfach mit der Standardgenauigkeit und -skalierung verwenden oder Sie setzen `NUMERIC(g)` ein, um die Genauigkeit festzulegen und die Standardskalierung zu übernehmen. Sie können auch mit `NUMERIC(g,s)` arbeiten und damit sowohl die Genauigkeit als auch die Skalierung festlegen. Die Parameter `g` und `s` sind Platzhalter, die Sie in der Datendeklaration durch Zahlenwerte ersetzen müssen.

Angenommen, die Standardgenauigkeit des Datentyps `NUMERIC` betrüge bei Ihrer SQL-Implementierung zwölf Stellen und die Standardskalierung sechs Stellen. Wenn Sie einer Datenbankspalte den Datentyp `NUMERIC` zuweisen, kann die Spalte Zahlen bis zu einer Größe von 999.999,999999 speichern. Wenn Sie dagegen einer Spalte den Datentyp `NUMERIC(10)` zuweisen, kann diese Spalte nur Zahlen bis zu einem Maximalwert von 9.999,999999 speichern. Der Parameter `(10)` legt die maximale Anzahl an Ziffern fest, die in der Zahl zulässig sind.

Wenn Sie einer Spalte den Datentyp `NUMERIC(10,2)` zuweisen, kann diese Spalte Zahlen bis zu einem Maximalwert von 99.999.999,99 speichern. In diesem Fall haben Sie ebenfalls insgesamt zehn Stellen, aber nur zwei Stellen hinter dem Komma.

Der Datentyp `NUMERIC` dient zum Speichern von Werten, wie etwa 595,72. Dieser Wert hat eine Genauigkeit von fünf Stellen (die Gesamtzahl der Ziffern) und eine Skalierung von zwei (die Anzahl der Ziffern nach dem Komma). Der passende Datentyp für Zahlen dieser Art ist `NUMERIC(5,2)`.

Der Datentyp DECIMAL

Der Datentyp `DECIMAL` ähnelt `NUMERIC`. Er kann Nachkommastellen haben, und Sie sind in der Lage, seine Genauigkeit und Skalierung festlegen. Der Unterschied besteht darin, dass die Genauigkeit, die Ihre Implementierung unterstützt, größer sein kann als die Genauigkeit, die Sie angeben. Falls dies der Fall ist, arbeitet die Implementierung mit der größeren Genauigkeit. Falls Sie keine Genauigkeit oder Skalierung angeben, arbeitet die Implementierung wie beim Datentyp `NUMERIC` mit Standardwerten.

Wenn Sie einem Element den Datentyp `NUMERIC(5,2)` zuweisen, kann es nur Zahlen bis zu einem absoluten Wert von 999,99 speichern. Wenn Sie einem Element den Datentyp `DECIMAL(5,2)` zuweisen, können Sie ebenfalls Zahlen bis zu einer absoluten Größe von 999,99 speichern. Wenn die Implementierung jedoch größere Zahlen zulässt, wird das DBMS auch solche Werte nicht zurückweisen, die größer als 999,99 sind.

Benutzen Sie die Datentypen `NUMERIC` oder `DECIMAL`, wenn Ihre Daten Nachkommastellen enthalten. Verwenden Sie `INTEGER`, `SMALLINT` oder `BIGINT` für ganzzahlige Daten. Arbeiten Sie mit dem Datentyp `NUMERIC`, wenn Ihre Anwendung möglichst anwendungsunabhängig sein soll, weil bei diesem Datentyp der Wertebereich, den Sie beispielsweise als `NUMERIC(5,2)` festlegen, auf allen Systemen gleich ist.

Annähernd genaue Zahlen

Einige Größen weisen ein derart breites Spektrum möglicher Werte (Größenordnungen) auf, dass sie ein Computer mit seiner begrenzten Registergröße nicht genau darstellen kann. (*Register* haben heute üblicherweise Größen von 32, 64 oder 128 Bits.) Normalerweise ist in diesen Fällen eine absolute Genauigkeit auch nicht notwendig, da man hier in der Regel mit einer guten Annäherung leben kann. SQL kennt drei Datentypen, um mit annähernd genauen Zahlen zu arbeiten: `REAL`, `DOUBLE`, `PRECISION`.

Der Datentyp REAL

Der Datentyp `REAL` dient dazu, Fließkommazahlen mit einfacher Genauigkeit zu speichern, wobei die Genauigkeit von der Implementierung abhängt. Im Allgemeinen bestimmt Ihre Hardware die Genauigkeit. So liefert beispielsweise ein 64-Bit-Rechner eine höhere Genauigkeit als ein 32-Bit-Rechner.

Eine *Fließkommazahl* ist eine Zahl, die ein Dezimalkomma enthält. Das Dezimalkomma »fließt« oder erscheint innerhalb der Zahl an verschiedenen Stellen, und zwar abhängig vom Wert der Zahl. 3,1 oder 3,14 oder 3,14159 sind Beispiele verschiedener Fließkommazahlen.

Der Datentyp DOUBLE PRECISION

Der Datentyp `DOUBLE PRECISION` dient dazu, Fließkommazahlen mit doppelter Genauigkeit zu speichern, wobei die Genauigkeit von der Implementierung abhängt. Die Bedeutung des Wortes `DOUBLE` hängt ebenfalls von der Implementierung ab. Berechnungen mit doppelter Genauigkeit werden hauptsächlich bei wissenschaftlichen Anwendungen benötigt. Verschiedene wissenschaftliche Disziplinen benötigen verschiedene Bereiche von Genauigkeit. Einige SQL-Implementierungen sind speziell auf bestimmte Zielgruppen zugeschnitten.

Bei einigen Systemen hat der Datentyp `DOUBLE PRECISION` sowohl bei der Mantisse als auch beim Exponenten genau die doppelte Kapazität wie der Datentyp `REAL`. (Zur Erinnerung: Man kann jede Zahl durch eine *Mantisse* darstellen, die mit einer Zehnerpotenz [10 hoch Exponent] multipliziert wird. Beispielsweise können Sie die Zahl 6.626 in der wissenschaftlichen Notation auch als 6,626E3 darstellen, wobei die Zahl 6,626 die Mantisse ist, die Sie mit 10^3 multiplizieren. 3 ist dabei der Exponent.)

Bei Zahlen, die bei dieser Schreibweise dicht an eins liegen (wie 6.626 oder selbst noch 6.626.000), bringt Ihnen einer der Datentypen zur Darstellung annähernd genauer Zahlen keinen Vorteil. Die Datentypen zur Darstellung genauer Zahlen funktionieren genauso gut und sind natürlich genau. Sind Zahlen jedoch sehr viel kleiner oder sehr viel größer als eins, wie etwa 6,626E-34 (eine sehr kleine Zahl), müssen Sie einen Datentyp zur Darstellung annähernd genauer Zahlen verwenden, weil diese Zahlen mit den Datentypen für genaue Zahlen nicht dargestellt werden können. Bei manchen Systemen hat der Datentyp `DOUBLE PRECISION` etwas mehr als die doppelte Mantissenkapazität und etwas weniger als die doppelte Exponentenkapazität als der Datentyp `REAL`. Bei wieder anderen Systemen hat der Datentyp `DOUBLE PRECISION` die doppelte Mantissenkapazität, aber dieselbe Exponentenkapazität wie der Datentyp `REAL`. In diesem Fall wird die Genauigkeit verdoppelt, während der Wertebereich gleich bleibt.

Die SQL-Spezifikation versucht nicht, die Bedeutung von `DOUBLE PRECISION` festzulegen. Die Spezifikation verlangt nur, dass die Genauigkeit einer Zahl vom Typ `DOUBLE PRECISION` größer ist als die Genauigkeit einer Zahl vom Typ `REAL`. Diese Einschränkung ist recht schwach, aber angesichts der großen Unterschiede in der Hardware wahrscheinlich die bestmögliche.

Der Datentyp FLOAT

Der Datentyp `FLOAT` ist besonders dann nützlich, wenn die Möglichkeit besteht, dass Ihre Datenbank eines Tages auf eine andere Hardware-Plattform portiert werden soll, die eine andere Registergröße als Ihr Rechner hat. Mit dem Datentyp `FLOAT` können Sie die Genauigkeit festlegen – etwa mit `FLOAT(5)`. Wenn Ihre Hardware die vorgegebene Genauigkeit mit ihrer Fließkommaeinheit unterstützt, verwendet Ihr System für Berechnungen einfache Genauigkeit. Wenn die vorgegebene Genauigkeit Berechnungen mit doppelter Genauigkeit verlangt, wird diese auch bei Berechnungen verwendet.

Wenn Sie den Datentyp FLOAT statt REAL oder DOUBLE PRECISION benutzen, wird das Übertragen Ihrer Datenbanken auf eine andere Hardware einfacher, weil Sie beim Datentyp FLOAT die Genauigkeit festlegen können. Die Genauigkeit von Zahlen vom Typ REAL und DOUBLE PRECISION ist dagegen hardwareabhängig.

Wenn Sie nicht sicher sind, ob Sie einen Datentyp für genaue Zahlen (NUMERIC/DECIMAL) oder einen Datentyp für annähernd genaue Zahlen (FLOAT/REAL) benutzen sollen, verwenden Sie einen Datentyp für genaue Zahlen. Dieser Datentyp beansprucht die Systemressourcen weniger und liefert Ihnen natürlich genaue (statt annähernd genaue) Ergebnisse. Falls der Bereich der möglichen Werte Ihrer Daten so groß sein sollte, dass Sie einen Datentyp zur Darstellung annähernd genauer Zahlen benötigen, können Sie dies wahrscheinlich bereits im Vorfeld herausfinden.

Zeichenketten

Datenbanken speichern viele Typen von Daten, einschließlich Grafiken, Töne und Animationen. Ich vermute, dass demnächst auch Gerüche an der Reihe sind. Können Sie sich ein dreidimensionales, 1920-x-1080-24-Bit-Farbbild eines großen Stücks Pepperoni-Pizza auf Ihrem Bildschirm vorstellen, während Ihre Supermultimediakarte gleichzeitig ein Duftmuster aus *DiFilippis Pizza-Grotte* abspielt? Eine solche Konfiguration könnte ziemlich frustrierend sein, es sei denn, Sie könnten es sich leisten, gleichzeitig auch Daten vom Datentyp »Geschmack« in Ihrem System zu speichern. Leider werden Sie wohl noch lange warten müssen, bis Gerüche und Geschmäcke zu den standardmäßigen SQL-Datentypen zählen. Derzeit werden – natürlich nach den numerischen Datentypen – am häufigsten Zeichenketten zur Beschreibung von Daten verwendet.

Es gibt drei Haupttypen von Zeichenketten:

- ✔ Zeichenketten fester Länge (CHARACTER oder CHAR)
- ✔ Zeichenketten variabler Länge (CHARACTER VARYING oder VARCHAR)
- ✔ Große Zeichenkettenobjekte (CHARACTER LARGE OBJECT oder CLOB)

Außerdem gibt es drei Varianten dieser Typen:

- ✔ NATIONAL CHARACTER
- ✔ NATIONAL CHARACTER VARYING
- ✔ NATIONAL CHARACTER LARGE OBJECT

Details folgen.

Der Datentyp CHARACTER

Wenn Sie einer Spalte diesen Datentyp zuweisen, können Sie die Anzahl der möglichen Zeichen in der Spalte mit der Syntax CHARACTER(x) festlegen, wobei x die Anzahl der Zeichen angibt. Wenn Sie beispielsweise einer Spalte den Datentyp CHARACTER(16) zuweisen, können die Daten in dieser Spalte maximal 16 Zeichen lang sein. Wenn Sie das Argument x nicht verwenden, das heißt, wenn Sie keinen Wert anstelle von x angeben, setzt SQL die Feldlänge

standardmäßig auf ein Zeichen. Wenn Sie Daten in ein CHARACTER-Feld eingeben, die kürzer als die festgelegte Länge des Feldes sind, füllt SQL die fehlenden Stellen mit Leerzeichen auf.

Der Datentyp CHARACTER VARYING

Der Datentyp CHARACTER VARYING ist nützlich, wenn die Einträge einer Spalte unterschiedlich lang sein und von SQL nicht mit Leerzeichen aufgefüllt werden sollen. Mit diesem Datentyp können Sie genau die Anzahl an Zeichen speichern, die der Benutzer eingibt. Für diesen Datentyp gibt es keinen Standardwert. Der Datentyp wird mit der Syntax CHARACTER VARYING(x) oder VARCHAR(x) festgelegt, wobei x die maximale Anzahl der zulässigen Zeichen angibt.

Der Datentyp CHARACTER LARGE OBJECT

Der Datentyp CHARACTER LARGE OBJECT (CLOB) wurde mit SQL:1999 eingeführt. Wie der Name vermuten lässt, dient er dazu, Zeichenketten zu speichern, die für den CHARACTER-Typ zu groß sind. CLOBs verhalten sich ähnlich wie normale Zeichenketten, sind aber einigen Bedingungen unterworfen.

Ein CLOB darf nicht in den Prädikaten PRIMARY KEY, FOREIGN KEY oder UNIQUE verwendet werden. Außerdem darf es nur in Vergleichsoperationen verwendet werden, die die Gleichheit oder Ungleichheit testen. Wegen ihrer besonderen Größe werden CLOBs von den meisten Anwendungen nicht in eine oder aus einer Datenbank übertragen. Stattdessen wird ein spezieller clientseitiger Typ, ein sogenannter *CLOB-Locator*, verwendet, um CLOB-Daten zu bearbeiten. Ein großes Zeichenkettenobjekt wird durch den Wert eines Parameters identifiziert.

Ein *Prädikat* ist ein Ausdruck, der logisch entweder wahr oder falsch ist.

Die Datentypen NATIONAL CHARACTER, NATIONAL CHARACTER VARYING und NATIONAL CHARACTER LARGE OBJECT

Manche Sprachen benutzen Zeichen, die in anderen Sprachen nicht bekannt sind. Beispielsweise gibt es im Deutschen Umlaute, die man im Englischen nicht kennt. Einige Sprachen, wie etwa Russisch, haben einen Zeichensatz, der sich erheblich von den Zeichensätzen des Deutschen oder des Englischen unterscheidet. Wenn Sie beispielsweise Englisch als Standardzeichensatz Ihres Systems festlegen, können Sie mit den Datentypen NATIONAL CHARACTER, NATIONAL CHARACTER VARYING und NATIONAL CHARACTER LARGE OBJECT einen alternativen Zeichensatz definieren. Diese Datentypen funktionieren dann wie die Datentypen CHARACTER, CHARACTER VARYING und CHARACTER LARGE OBJECT, wobei der Zeichensatz, den Sie festlegen, vom Standardzeichensatz abweicht.

Sie können den Zeichensatz wie eine Tabellenspalte definieren. Jede Spalte kann einen anderen Zeichensatz benutzen. Im folgenden Beispiel wird eine Tabelle angelegt, die mit mehreren Zeichensätzen arbeitet:

```
CREATE TABLE XLATE (
   Sprache1 CHARACTER(40),
   Sprache2 CHARACTER VARYING(40) CHARACTER SET GREEK,
   Sprache3 NATIONAL CHARACTER(40),
   Sprache4 CHARACTER(40) CHARACTER SET KANJI
   ) ;
```

Die Spalte `Sprache1` enthält Zeichen im Standardzeichensatz der Implementierung. Die Spalte `Sprache3` enthält Zeichen im nationalen Zeichensatz der Implementierung. Die Spalte `Sprache2` enthält griechische Zeichen und die Spalte `Sprache4` enthält Kanji-Zeichen. Nach langer Abwesenheit sind jetzt auch asiatische Zeichensätze, etwa Kanji, in vielen DBMS-Produkten enthalten.

Binäre Zeichenketten

Die `BINARY`-Datentypen wurden in SQL:2008 eingeführt. Wenn man bedenkt, welche zentrale Rolle Binärdaten seit dem Atanasoff-Berry-Computer aus den 1930ern für Digitalcomputer gespielt haben, scheint ihre Anerkennung in SQL ein wenig spät zu erfolgen. (Aber vermutlich besser spät als nie.) Es gibt drei verschiedene Binärtypen: `BINARY`, `BINARY VARYING` und `BINARY LARGE OBJECT`.

Der Datentyp BINARY

Wenn Sie den Datentyp einer Spalte als `BINARY` definieren, können Sie die Anzahl der Bytes (Oktette) angeben, die in der Spalte gespeichert werden können. Die Syntax lautet `BINARY (x)`, wobei `x` die Anzahl der Bytes angibt. Definieren Sie den Datentyp einer Spalte etwa als `BINARY (16)`, muss die binäre Zeichenkette 16 Bytes lang sein. `BINARY`-Daten müssen, beginnend mit Byte eins, als Bytes eingegeben werden.

Der Datentyp BINARY VARYING

Verwenden Sie den Datentyp `BINARY VARYING` oder `VARBINARY`, wenn die Länge einer binären Zeichenkette variabel ist. Die Syntax lautet `BINARY VARYING (x)` oder `VARBINARY (x)`, wobei `x` die maximal zugelassene Anzahl von Bytes angibt. Die minimale Länge der Zeichenkette ist null und die maximale Länge ist `x`.

Der Datentyp BINARY LARGE OBJECT

Der Datentyp `BINARY LARGE OBJECT` (`BLOB`) wird für riesige binäre Zeichenketten verwendet, die für den `BINARY`-Typ zu groß sind. Beispiele für riesige binäre Zeichenketten sind Grafiken oder Musikdateien. `BLOB`s verhalten sich ähnlich wie normale binäre Zeichenketten, aber SQL schränkt ihre Manipulationsmöglichkeiten ein.

Zum einen können Sie etwa einen `BLOB` nicht in einem `PRIMARY KEY`, `FOREIGN KEY` oder `UNIQUE`-Prädikat verwenden. Zum anderen sind die Vergleichsmöglichkeiten bei `BLOB`s eingeschränkt: Nur ihre Gleichheit und Ungleichheit kann getestet werden. `BLOB`s sind groß; deshalb werden sie von Anwendungen im Allgemeinen nicht aus der Datenbank abgerufen.

Stattdessen verwenden Anwendungen einen bestimmten clientseitigen Datentyp, einen sogenannten *BLOB Locator*, um die BLOB-Daten zu manipulieren. Der Locator ist ein Parameter, dessen Wert ein BINARY LARGE OBJECT identifiziert.

Boolesche Werte

Der Datentyp BOOLEAN umfasst die Wahrheitswerte TRUE, FALSE und UNKNOWN *(wahr*, *falsch* und *unbekannt)*. Wenn der boolesche Wert TRUE oder FALSE mit dem Wert NULL oder dem Wahrheitswert UNKNOWN verglichen wird, ist das Ergebnis der Wert UNKNOWN.

Datums- und Zeitwerte

Der SQL-Standard kennt fünf Datentypen, die mit Datums- und Zeitangaben arbeiten. Diese Datentypen werden als *Datetime-Datentypen* oder kurz *Datetimes* bezeichnet. Es gibt beträchtliche Überschneidungen zwischen diesen Datentypen, weshalb einige Implementierungen nicht alle fünf Varianten unterstützen.

Wenn Sie versuchen, Datenbanken auf Implementierungen zu übertragen, die nicht alle fünf Datentypen für Datums- und Zeitangaben voll unterstützen, kann es Probleme geben. Stellen Sie in diesem Fall fest, wie die Quell- und die Zielimplementierung Datumsangaben und Zeiten darstellen.

Der Datentyp DATE

Der Datentyp DATE speichert das Jahr, den Monat und den Tag eines Datums in genau dieser Reihenfolge. Das Jahr wird vierstellig, der Monat und die Tage werden jeweils zweistellig gespeichert. Ein Wert vom Typ DATE kann jedes Datum vom Jahr 0001 bis zum Jahr 9999 aufnehmen. Der Datentyp DATE ist zehn Stellen lang, zum Beispiel 1957-08-14.

Der Datentyp TIME WITHOUT TIME ZONE

Der Datentyp TIME WITHOUT TIME ZONE speichert die Zeit in Form von Stunden, Minuten und Sekunden. Die Stunden und Minuten werden jeweils zweistellig gespeichert. Die Sekunden sind standardmäßig zweistellig, können aber optional zusätzlich einen Nachkommateil enthalten, zum Beispiel 09:32:58.436.

Die Genauigkeit des Nachkommastellenanteils ist von der Implementierung abhängig, beträgt aber wenigstens sechs Ziffern. Ein Wert vom Typ TIME WITHOUT TIME ZONE ist acht Stellen lang (einschließlich der Doppelpunkte), wenn er keinen Nachkommateil enthält. Andernfalls ist er neun Stellen plus die Anzahl der Stellen für den Nachkommateil lang. Die neunte Stelle enthält das Dezimalzeichen (in der amerikanischen SQL-Schreibweise ist dies immer ein Punkt). Der Datentyp TIME WITHOUT TIME ZONE wird entweder mit TIME (Standardwert ohne Nachkommastellen) oder mit TIME WITHOUT TIME ZONE (p) festgelegt, wobei p für die Gesamtzahl der Nachkommastellen steht. Das Beispiel im vorangegangenen Absatz hat den Datentyp TIME WITHOUT TIME ZONE (3).

Der Datentyp TIMESTAMP WITHOUT TIME ZONE

Der Datentyp `TIMESTAMP WITHOUT TIME ZONE` enthält sowohl Datums- als auch Zeitangaben. Die Längen und die Bedingungen für die Werte der Komponenten des Datentyps `TIMESTAMP WITHOUT TIME ZONE` sind dieselben wie bei den Datentypen `DATE` und `TIME WITHOUT TIME ZONE`, wobei eine Ausnahme gilt: Die Standardlänge des Nachkommateils der Zeitkomponente eines Wertes vom Typ `TIMESTAMP WITHOUT TIME ZONE` beträgt sechs statt null Ziffern.

Wenn der Wert keine Nachkommastellen hat, ist der Typ `TIMESTAMP WITHOUT TIME ZONE` 19 Stellen lang – zehn Stellen für das Datum, eine Leerstelle als Trennzeichen und acht Stellen für die Zeit, und zwar in dieser Reihenfolge. Wenn Nachkommastellen (der Standardwert beträgt sechs Stellen) vorhanden sind, beträgt die Länge 20 Stellen plus die Anzahl der Nachkommastellen. Die zwanzigste Stelle enthält das Dezimalzeichen (in der amerikanischen Schreibweise ist dies immer der Punkt). Der Datentyp `TIMESTAMP WITHOUT TIME ZONE` wird mit `TIMESTAMP WITHOUT TIME ZONE` oder `TIMESTAMP WITHOUT TIME ZONE (p)` festgelegt, wobei p für die Anzahl der Nachkommastellen steht. Der Wert von p darf nicht negativ sein. Seine Maximalgröße ist von der Implementierung abhängig.

Der Datentyp TIME WITH TIME ZONE

Der Datentyp `TIME WITH TIME ZONE` entspricht genau dem Datentyp `TIME WITHOUT TIME ZONE`, außer dass er zusätzlich Informationen über die Abweichung von der *koordinierten Weltzeit* (UTC, Universal Time Coordinated, dem Nachfolger der Greenwich Mean Time oder GMT) enthält. Die Abweichung kann einen beliebigen Wert von -12:59 bis +13:00 betragen. Diese zusätzliche Information belegt sechs weitere Stellen. Erst wird die Zeit angegeben, dann folgen ein Trennungsstrich, ein Plus- oder ein Minuszeichen und die Abweichung in Stunden (zwei Stellen) und Minuten (zwei Stellen), die durch einen Doppelpunkt voneinander getrennt werden. Daten vom Typ `TIME WITH TIME ZONE` sind ohne Nachkommateil (Standard) 14 Stellen lang. Mit Nachkommateil beträgt die Länge 15 Stellen plus die Anzahl der Stellen des Nachkommastellenanteils.

Der Datentyp TIMESTAMP WITH TIME ZONE

Der Datentyp `TIMESTAMP WITH TIME ZONE` entspricht dem Datentyp `TIMESTAMP WITHOUT TIME ZONE`, außer dass er zusätzlich Informationen über die Abweichung von der koordinierten Weltzeit enthält. Diese zusätzliche Information belegt sechs weitere Stellen (siehe den vorangegangenen Abschnitt über die Form der Zeitzoneninformation). Daten vom Typ `TIMESTAMP WITH TIME ZONE` sind ohne Nachkommateil 25 Stellen lang. Mit Nachkommateil beträgt die Länge 26 Stellen plus die Anzahl der Stellen des Bruchteils (standardmäßig gibt es sechs Stellen nach dem Dezimalpunkt).

Intervalle

Intervalldatentypen sind verwandt mit den Datums- und Zeittypen. Ein *Intervall* ist die Differenz zwischen zwei Datums- und Zeitwerten. In vielen Anwendungen, die mit Datums- und/oder Zeitangaben zu tun haben, wird häufig das Intervall zwischen zwei Datumsangaben oder zwei Zeiten benötigt.

SQL unterscheidet zwei verschiedene Intervalltypen: das Intervall *year-month* und das Intervall *day-time*. Das Intervall year-month gibt Aufschluss über die Anzahl der Jahre und Monate zwischen zwei Datumsangaben. Das Intervall *day-time* gibt die Anzahl der Tage, Stunden, Minuten und Sekunden zwischen zwei Zeitpunkten innerhalb eines Monats an. In einer Berechnung dürfen nicht beide Intervalltypen gleichzeitig verwendet werden, weil die Länge der Monate variiert (28, 29, 30 oder 31 Tage).

Der Datentyp XML

XML ist eine Abkürzung für *eXtensible Markup Language* (dt. *Erweiterbare Auszeichnungssprache*), die einen Satz von Regeln für die Auszeichnung von Daten definiert. Die Auszeichnungsstruktur der Daten beschreibt deren Bedeutung. XML ermöglicht einen plattformübergreifenden Datenaustausch.

XML-Daten haben eine Baumstruktur. Das bedeutet, dass ein Wurzelknoten Unterknoten besitzt, die wiederum über eigene Unterknoten verfügen können. Der Datentyp XML wurde bereits mit SQL:2003 eingeführt, dann mit SQL/XML:2005 und SQL:2008 erweitert. SQL:2005 führte fünf parametrisierte Untertypen ein, wobei der ursprüngliche einfache XML-Typ beibehalten wurde. XML-Werte können in Instanzen von zwei oder mehr Typen existieren, weil einige der Untertypen wiederum Untertypen von Untertypen sind. Vielleicht sollte ich sie Unter-Untertypen oder Unter-Unter-Untertypen nennen. Glücklicherweise hat SQL:2008 eine Standardmethode eingeführt, um Untertypen zu referenzieren.

Die primären Modifizierer des XML-Typs sind `SEQUENCE`, `CONTENT` und `DOCUMENT`. Die sekundären Modifizierer sind `UNTYPED`, `ANY` oder `XMLSCHEMA`. Abbildung 2.1 zeigt die Baumähnliche Struktur der hierarchischen Beziehungen zwischen den Untertypen.

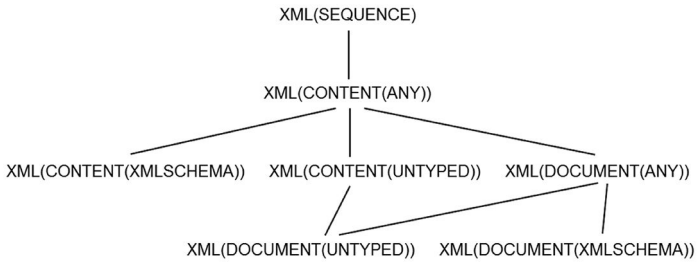


Abbildung 2.1: Beziehungen der XML-Untertypen

Sie finden hier eine Übersicht der XML-Typen, die Sie kennen sollten. Eine ausführlichere Beschreibung dieser Typen finden Sie in Kapitel 18. Ich habe die Liste so geordnet, dass ich mit den grundlegenden Typen anfange und mit dem kompliziertesten aufhöre.

- ✔ `XML(SEQUENCE)`: Bei jedem Wert in XML handelt es sich entweder um einen Nullwert oder um eine XQuery-Folge. Auf diese Art und Weise ist jeder XML-Wert eine Instanz des Typs `XML(SEQUENCE)`. Bei *XQuery* handelt es sich um eine Abfragesprache, die speziell dafür entwickelt worden ist, Informationen aus XML-Daten zu extrahieren. Dieser Datentyp ist der grundlegendste von allen XML-Datentypen.

XML(SEQUENCE) ist auch der XML-Datentyp, der die wenigsten Bedingungen kennt. Er ist in der Lage, auch Werte zu akzeptieren, die nicht unbedingt gut geformt sein müssen. Die übrigen XML-Typen verzeihen derartige Fehler nicht.

- ✔ XML(ANY CONTENT): Dieser Datentyp ist etwas einschränkender als XML(SEQUENCE). Bei jedem XML-Wert, der entweder ein Nullwert oder ein XQuery-Dokumentenknoten (oder ein Unterknoten dieses Knotens) ist, handelt es sich um eine Instanz dieses Datentyps. Jede Instanz von XML(ANY CONTENT) ist gleichzeitig eine Instanz von XML(SEQUENCE). Auch XML-Werte vom Typ XML(ANY CONTENT) müssen nicht wohlgeformt sein. Werte dieser Art können das Zwischenergebnis einer Abfrage sein und später in wohlgeformte Werte umgewandelt werden.
- ✔ XML(CONTENT (UNTYPED)): Dieser Datentyp ist restriktiver als XML(ANY CONTENT). Somit ist jeder Wert dieses Typs eine Instanz von XML(ANY CONTENT) und von XML(SEQUENCE). Jeder XML-Wert, der entweder der Nullwert oder ein Nicht-Nullwert vom Typ XML(CONTENT(ANY)) ist, ist ein XQuery-Dokument-Knoten D, sodass für jeden XQuery-Element-Knoten, der in dem XQuery-Baum T enthalten ist, der seine Wurzel in D hat, gilt:

  Die Typ-Name-Property ist xdt:untyped.

  Die **nilled**-Property ist False.

  Für jeden XQuery-Attribut-Knoten in T ist die Typ-Property xdt:untypedAtomic.

  Für jeden XQuery-Attribut-Knoten in T ist die Typ-Property ein Wert vom Typ-Namen XML(CONTENT(UNTYPED)).

- ✔ XML(CONTENT(XMLSCHEMA)): Dies ist neben XML(CONTENT(UNTYPED)) der zweite Untertyp von XML(CONTENT(ANY)). Als solcher ist er auch ein Untertyp von XML(SEQUENCE). Jeder XML-Wert ist entweder ein Nullwert oder ein Nicht-Nullwert vom Typ XML(CONTENT (ANY)) und auch ein XQuery-Dokument-Knoten D, sodass für jeden XQuery-Element-Knoten, der in dem XQuery-Baum T enthalten ist, der seine Wurzel in D hat, gilt:

  Ist gültig entsprechend des XML-Schemas S oder

  ist gültig entsprechend eines XML-Namensraums N in einem XML-Schema S oder

  ist gültig entsprechend einer globalen Element-Deklaration-Schema-Komponente E in einem XML-Schema S,

  ist ein Wert vom Typ XML(CONTENT(XMLSCHEMA)), dessen Typ-Deskriptor folgende Komponenten umfasst: den registrierten XML-Schema-Deskriptor von S und, wenn N spezifiziert ist, den XML-Namensraum-URI von E und den XML-NCName von E.

- ✔ XML(DOCUMENT(ANY)): Hierbei handelt es sich um einen weiteren Untertyp von XML(CONTENT(ANY)) und der zusätzlichen Bedingung, dass Instanzen von XML(DOCUMENT (ANY)) Dokumentenknoten sind, die genau einen XQuery-Element-Knoten, null oder mehr XQuery-Kommentar-Knoten und null oder mehr XQuery-Verarbeitungsanweisungen-Knoten haben.

- ✔ `XML(DOCUMENT(UNTYPED))`: Jeder Wert, der entweder ein Nullwert oder ein Nicht-Nullwert vom Typ `XML(CONTENT(UNTYPED))` ist, der ein XQuery-Dokument-Knoten ist, dessen `children`-Property genau einen XQuery-Element-Knoten, null oder mehr XQuery-Kommentar-Knoten und null oder mehr XQuery-Verarbeitungsanweisungen-Knoten hat, ist ein Wert vom Typ `XML(DOCUMENT(UNTYPED))`. Alle Instanzen von `XML(DOCUMENT (UNTYPED))` sind auch Instanzen von `XML(UNTYPED CONTENT)`. Darüber hinaus sind alle Instanzen von `XML(UNTYPED DOCUMENT)` auch Instanzen von `XML(DOCUMENT (ANY))`. `XML(UNTYPED DOCUMENT)` hat von allen Untertypen die meisten Bedingungen, weil er die Bedingungen aller anderen Untertypen teilt. Jedes Dokument, das den Anforderungen eines `XML(UNTYPED DOCUMENT)` entspricht, ist gleichzeitig eine Instanz aller anderen Untertypen.

Der Datentyp ROW

Der Datentyp `ROW` wurde mit SQL:1999 eingeführt. Er ist nicht leicht zu verstehen und als Anfänger oder fortgeschrittener Anfänger der SQL-Programmierung sollten Sie ihn nicht unbedingt verwenden. Schließlich kamen die Anwendungsentwickler zwischen 1986 und 1999 auch ohne diesen Datentyp ziemlich gut zurecht.

`ROW` verfügt über die bemerkenswerte Eigenschaft, dass er die Regeln der Normalisierung verletzt, die E.F. Codd in den frühen Tagen der relationalen Datenbanken definiert hat. In Kapitel 5 werden Sie mehr über diese Regeln erfahren. Eine wesentliche Eigenschaft der ersten Normalform besteht darin, dass ein Feld in einer Tabellenzeile nicht mehrere Werte enthalten darf. Ein Feld darf einen und nur einen Wert enthalten. Mit dem `ROW`-Datentyp können Sie jedoch festlegen, dass eine komplette Datenzeile in einem einzelnen Feld einer einzelnen Tabellenzeile enthalten sein darf – anders ausgedrückt: Eine Zeile kann in eine andere Zeile eingebettet werden.

Die *Normalformen*, die zum ersten Mal von Dr. Codd aufgestellt worden sind, legen Merkmale relationaler Datenbanken fest. Das Aufnehmen des Datentyps `ROW` in den SQL-Standard war der erste Versuch, die Limitierungen von SQL im reinen relationalen Modell aufzubrechen.

Schauen Sie sich als Beispiel die folgende SQL-Anweisung an, die einen `ROW`-Typ für die Adressdaten einer Person definiert:

```
CREATE ROW TYPE AdressTyp (
   Strasse      CHARACTER VARYING(25)
   Ort          CHARACTER VARYING(20)
   Bundesland   CHARACTER(3)
   Postleitzahl CHARACTER VARYING(5)
   ) ;
```

Nachdem dieser ROW-Typ definiert wurde, können Sie ihn in einer Tabellendefinition verwenden:

```
CREATE TABLE Kunden (
   KundenID      INTEGER      PRIMARY KEY,
   Nachname      CHARACTER VARYING(25),
   Vorname       CHARACTER VARYING(20),
   Adresse       AdressTyp
   Telefon       CHARACTER VARYING(15)
   ) ;
```

Der Vorteil besteht darin, dass Sie, wenn Sie Adressdaten für mehrere Kategorien verwalten (wie beispielsweise Kunden, Lieferanten, Mitarbeiter und Anteilseigner), die Einzelheiten der Adressenspezifikation nur einmal definieren müssen – in der Definition des ROW-Typs.

Datentypen für Auflistungen

Nachdem SQL mit SQL:1999 aus der relationalen Zwangsjacke ausbrach, wurden Datentypen möglich, die die erste Normalform verletzen. Ein Feld konnte nun eine ganze Sammlung von Objekten (eine Objektauflistung) anstatt wie bisher nur ein Objekt enthalten. Der Datentyp ARRAY wurde mit SQL:1999 und der Datentyp MULTISET mit SQL:2003 eingeführt.

Zwei Auflistungen können nur dann miteinander verglichen werden, wenn sie beide vom selben Typ sind (ARRAY oder MULTISET) und wenn die Datentypen der in den Auflistungen enthaltenen Elemente ebenfalls vergleichbar sind. Da Arrays eine festgelegte Elementreihenfolge besitzen, ist ein Vergleich zwischen den sich deckenden Elementen zweier Arrays möglich. Auflistungen vom Typ MULTISET besitzen keine festgelegte Elementreihenfolge, können aber dennoch miteinander verglichen werden, wenn (a) für jedes MULTISET eine Datenauflistung besteht und (b) diese Auflistungen gepaart werden können.

Der Datentyp ARRAY

Der Datentyp ARRAY verstößt gegen die Regeln der ersten Normalform (1NF), allerdings anders als der ROW-Typ. Der ARRAY-Typ für Auflistungen ist nicht – wie beispielsweise bei den Datentypen CHARACTER oder NUMERIC – ein spezifischer Typ. Mit einem ARRAY-Typ können Sie nur in einem Feld einer Tabelle mehrere Werte eines anderen Datentyps speichern. Angenommen, Ihr Unternehmen möchte einen Kunden jederzeit – geschäftlich, privat oder unterwegs – erreichen. Deshalb wollen Sie mehrere Telefonnummern für den Kunden speichern und deklarieren zu diesem Zweck das Attribut Telefon wie folgt als Array:

```
CREATE TABLE KUNDEN (
   KundenID      INTEGER      PRIMARY KEY,
   Nachname      CHARACTER VARYING(25),
   Vorname       CHARACTER VARYING(20),
   Adresse       AdressTyp
   Telefon       CHARACTER VARYING(15) ARRAY[3]
   ) ;
```

Die Notation `ARRAY[3]` ermöglicht es Ihnen, bis zu drei Telefonnummern in der Tabelle `Kunden` zu speichern. Die drei Telefonnummern sind ein Beispiel für eine sogenannte *Wiederholungsgruppe* (englisch *Repeating Group*). In der klassischen Theorie der relationalen Datenbanken sind Wiederholungsgruppen verboten. Dies ist ein Beispiel für mehrere Fälle, in denen SQL:1999 gegen die Regeln verstößt. Als Dr. Codd die Regeln der Normalisierung erstmals formulierte, verzichtete er zugunsten der Datenintegrität auf eine gewisse funktionale Flexibilität. SQL:1999 führte einen Teil dieser funktionalen Flexibilität zuungunsten einer höheren Komplexität der Datenstrukturen wieder ein.

Diese höhere Komplexität der Strukturen kann eine Gefahr für die Integrität Ihrer Daten bedeuten, wenn Sie sich nicht über alle Auswirkungen der Aktionen im Klaren sind, die Sie mit der Datenbank ausführen. Jedes Element eines Arrays wird mit *genau einer* Ordnungszahl, einer eindeutigen Position innerhalb des Arrays verknüpft.

Ein *Array* ist eine geordnete Sammlung von Werten. Die *Kardinalität* eines Arrays ist die Anzahl der in ihm enthaltenen Elemente. Ein SQL-Array kann eine beliebige Kardinalität von null bis zu einer deklarierten maximalen Anzahl von Elementen (einschließlich) haben. Dies bedeutet, dass die Kardinalität einer Spalte vom Typ `Array` von einer Zeile zur nächsten variieren kann. Ein Array kann atomar null sein; dann wäre seine Kardinalität ebenfalls null. Ein Null-Array ist nicht dasselbe wie ein leeres Array, dessen Kardinalität null beträgt. Ein Array, das nur Null-Elemente enthält, hat eine Kardinalität größer als null. So hat etwa ein Array mit fünf Null-Elementen eine Kardinalität von fünf.

Wenn ein Array eine Kardinalität hat, die unter dem deklarierten Maximum liegt, werden seine nicht belegten Zellen als nicht-existent behandelt. Es wird nicht angenommen, dass sie Null-Werte enthielten, sondern sie sind einfach nicht vorhanden.

Sie können auf einzelne Elemente eines Arrays zugreifen, indem Sie ihren Index in eckigen Klammern angeben. Haben Sie etwa ein Array namens `Telefon`, dann referenziert `Telefon [3]` das dritte Element in dem `Telefon`-Array.

Seit SQL:1999 ist es möglich, die Kardinalität eines Arrays mit der Funktion `CARDINALITY` abzurufen. Neu in SQL:2011 ist die Möglichkeit, die maximale Kardinalität eines Arrays mit der Funktion `ARRAY_MAX_CARDINALITY` abzurufen. Dies ist sehr nützlich, weil Sie damit Allzweckroutinen schreiben können, die auf Arrays mit unterschiedlichen maximalen Kardinalitäten anwendbar sind. Routinen mit fest einprogrammierten maximalen Kardinalitäten sind nur auf Arrays mit einer vorgegebenen maximalen Kardinalität anwendbar und müssten für Arrays mit einer anderen maximalen Kardinalität umgeschrieben werden.

Auch wenn SQL:1999 den Datentyp `ARRAY` und die Möglichkeit einführte, einzelne Elemente in einem Array zu adressieren, bot es keine Möglichkeit, Elemente aus einem Array zu löschen. Dieses Versehen wurde in SQL:2011 mit der Funktion `TRIM_ARRAY` korrigiert, mit der Sie Elemente vom Ende eines Arrays löschen können.

Der Datentyp MULTISET

Ein *Multiset* ist eine ungeordnete Auflistung. Sie können bestimmte Elemente eines Multisets nicht referenzieren, da sie innerhalb der Auflistung keine Ordnungszahl besitzen.

REF-Typen

Die REF-Typen gehören nicht zum SQL-Kern. Dies bedeutet, dass Datenbankverwaltungssysteme mit dem SQL-Standard konform sein können, ohne REF-Typen zu implementieren. Bei dem REF-Typ handelt es sich nicht – wie beispielsweise bei den Datentypen CHARACTER oder NUMERIC – um einen spezifischen Typ, sondern um einen Zeiger auf ein Datenelement, einen ROW-Typ oder einen abstrakten Datentyp, der sich in einer Tabelle (auf einer Site) befindet. Durch das Dereferenzieren des Zeigers können Sie den Wert abrufen, der unter dem angegebenen Ziel gespeichert ist.

Sie sollten sich keine Sorgen machen, falls Sie dieser Typ verwirrt – Sie sind mit Ihrer Verwirrung in guter Gesellschaft. Wenn Sie mit REF-Typen arbeiten wollen, müssen Sie die Prinzipien der objektorientierten Programmierung (OOP) beherrschen. In diesem Buch vermeide ich es, zu tief in die dunklen Zonen der OOP einzudringen. Da dieser Datentyp nicht zum SQL-Kern gehört, könnte es für Sie tatsächlich besser sein, ihn nicht zu benutzen. Wenn Sie eine maximale Übertragbarkeit Ihrer Daten über mehrere DBMS-Plattformen hinweg anstreben, sollten Sie sich auf die zentralen SQL-Komponenten beschränken.

Benutzerdefinierte Typen

Eine weitere Datentypkategorie, die mit SQL:1999 eingeführt wurde und aus der Welt der objektorientierten Programmierung stammt, enthält die *benutzerdefinierten Typen* (User Defined Types = UDTs). Als SQL-Programmierer sind Sie nicht mehr auf die Datentypen beschränkt, die in der SQL-Spezifikation festgelegt sind, sondern Sie können eigene Datentypen definieren, indem Sie die Prinzipien der abstrakten Datentypen (Abstract Data Types = ADTs) anwenden, die in objektorientierten Sprachen wie C++ zu finden sind.

Einer der größten Vorteile der UDTs besteht darin, dass Sie damit die Unterschiede zwischen SQL und der Hostsprache überbrücken können, in die SQL eingebettet ist. Es war häufig ein Problem, dass die vordefinierten Datentypen von SQL nicht zu den Datentypen der Hostsprache passten, in die die SQL-Anweisungen eingebettet waren. Mit UDTs kann ein Datenbankprogrammierer jetzt Datentypen definieren, die denen der Hostsprache entsprechen.

Ein UDT verfügt über Attribute und Methoden, die in ihm gekapselt sind. Die Außenwelt sieht die Definitionen der Attribute sowie die Ergebnisse der Methoden, aber die speziellen Implementierungen der Methoden bleiben ihr verborgen. Der Zugriff auf die Attribute und Methoden eines UDT kann weiter eingeschränkt werden, indem diese als *public* (öffentlich), *private* (privat) oder *protected* (geschützt) deklariert werden:

- ✔ *Public* Attribute oder Methoden sind allen Benutzern eines UDT zugänglich.
- ✔ *Private* Attribute oder Methoden sind nur für den UDT selbst zugänglich.
- ✔ *Protected* Attribute oder Methoden sind nur dem UDT selbst oder seinen Untertypen zugänglich.

Damit verhält sich ein UDT in SQL etwa wie eine Klasse in einer objektorientierten Programmiersprache. Es gibt zwei Arten von benutzerdefinierten Datentypen: eindeutige Typen und strukturierte Typen.

Eindeutige Typen (DISTINCT)

DISTINCT-*Typen* sind die einfachere Form benutzerdefinierter Datentypen. Das Besondere an einem DISTINCT-Typ ist, dass er nur einen einzelnen Datentyp darstellt. Er wird aus einem der vordefinierten Datentypen konstruiert, die als *Quelltypen* bezeichnet werden. Mehrere eindeutige Typen, die auf demselben Quelltyp beruhen, unterscheiden sich voneinander und sind deshalb nicht miteinander vergleichbar. Sie können beispielsweise mit eindeutigen Typen verschiedene Währungen unterscheiden. Betrachten Sie die folgende Datentypdefinition:

```
CREATE DISTINCT TYPE USdollar AS DECIMAL (9,2) ;
```

Damit wird ein neuer Datentyp für US-Dollar erstellt, der auf dem vordefinierten Datentyp DECIMAL basiert. Auf dieselbe Weise können Sie einen weiteren spezifischen Typ erstellen:

```
CREATE DISTINCT TYPE Euro AS DECIMAL (9,2) ;
```

Jetzt können Sie Tabellen erstellen, die diese neuen Datentypen verwenden:

```
CREATE TABLE USRechnung (
   RechID      INTEGER      PRIMARY KEY,
   KundenID    INTEGER,
   MitarbID    INTEGER,
   Netto       USdollar,
   Steuer      USdollar,
   Versand     USdollar,
   Brutto      USdollar
   ) ;

CREATE TABLE EuroRechnung (
   RechID      INTEGER      PRIMARY KEY,
   KundenID    INTEGER,
   MitarbID    INTEGER,
   Netto       Euro,
   Steuer      Euro,
   Versand     Euro,
   Brutto      Euro
   ) ;
```

Sowohl der Typ USdollar als auch der Typ Euro basieren auf dem Datentyp DECIMAL, aber die Instanzen des einen Typs können nicht direkt mit Instanzen des anderen Typs oder mit Instanzen des Typs DECIMAL verglichen werden. Nun ist es in SQL – wie in der Praxis – möglich, US-Dollar in Euro umzuwandeln, aber dazu ist eine spezielle Operation (CAST) erforderlich. Danach sind dann Vergleiche zulässig.

Strukturierte Typen

Die zweite Form benutzerdefinierter Datentypen, der strukturierte Typ, basiert nicht auf einem einzelnen vordefinierten Quelltyp, sondern wird als Liste von Attributdefinitionen und Methoden ausgedrückt.

Konstruktoren

Wenn Sie einen strukturierten UDT erstellen, erzeugt das Datenbankverwaltungssystem automatisch eine Konstruktorfunktion für diesen Datentyp. Diese Funktion erhält denselben Namen wie der UDT. Die Aufgabe des Konstruktors besteht darin, die Attribute des UDTs zu initialisieren und auf ihre Standardwerte zu setzen.

Wandler und Beobachter

Wenn Sie einen strukturierten UDT erstellen, erzeugt das Datenbankverwaltungssystem automatisch eine Wandler- und eine Beobachtungsfunktion. Wenn eine Wandlerfunktion (englisch *Mutator*) aufgerufen wird, ändert sie einen Attributwert des strukturierten Typs. Eine Beobachtungsfunktion (englisch *Observer*) ist das Gegenteil vom Wandler. Ihre Aufgabe besteht darin, den Attributwert eines strukturierten Typs abzurufen. Sie können Beobachtungsfunktionen in `SELECT`-Anweisungen einbinden, um Werte aus einer Datenbank abzurufen.

Unter- und übergeordnete Typen

Zwischen zwei strukturierten Typen können hierarchische Beziehungen bestehen. Ein Datentyp namens `MusikCDudt` kann beispielsweise über einen untergeordneten Typ mit der Bezeichnung `RockCDudt` und einen weiteren untergeordneten Typ namens `KlassikCDudt` verfügen. `MusikCDudt` ist dann der übergeordnete Typ dieser beiden Untertypen. `RockCDudt` ist ein *direkter Untertyp* von `MusikCDudt`, wenn `MusikCDudt` über keinen Untertyp verfügt, der `RockCDudt` übergeordnet ist. Wenn `RockCDudt` einen Untertyp namens `HeavyMetalCDudt` besitzt, ist dieser ebenfalls ein Untertyp von `MusikCDudt`, aber kein direkter.

Ein strukturierter Datentyp, dem kein anderer Typ übergeordnet ist, wird als *maximal übergeordneter Typ* bezeichnet, während ein strukturierter Datentyp ohne Untertypen ein *Leaf-Untertyp* ist (*Leaf* ist das englische Wort für Blatt, womit das Blatt eines Logikbaums gemeint ist, also das letzte Element in einer hierarchischen Baumstruktur).

Beispiel für einen strukturierten Datentyp

Sie können strukturierte UDTs wie folgt erzeugen:

```
/* Einen UDT namens MusikCDudt erstellen */
CREATE TYPE MusikCDudt AS
/* Attribute festlegen */
Titel                 CHAR(40),
Kosten                DECIMAL(9,2),
Preisvorstellung      DECIMAL(9,2)
/* Untertypen zulassen */
NOT FINAL ;

CREATE TYPE RockCDudt UNDER MusikCDudt NOT FINAL ;
```

Der Untertyp RockCDudt erbt die Attribute seines übergeordneten Typs MusikCDudt.

```
CREATE TYPE HeavyMetalCDudt UNDER RockCDudt FINAL ;
```

Nachdem Sie nun alle erforderlichen Typen besitzen, können Sie Tabellen erstellen, die diese Datentypen benutzen. Dazu ein Beispiel:

```
CREATE TABLE MetalSKU (
    Album       HeavyMetalCDudt,
    SKU         INTEGER) ;
```

Sie können der Tabelle jetzt Zeilen hinzufügen:

```
BEGIN
   /* Eine temporäre Variable namens a deklarieren */
   DECLARE a = HeavyMetalCDudt ;
   /* Die Konstruktorfunktion ausführen */
   SET a = HeavyMetalCDudt() ;
   /* Die erste Wandlerfunktion ausführen */
   SET a = a.Titel('Edward the Great') ;
   /* Die zweite Wandlerfunktion ausführen */
   SET a = a.Kosten(7.50) ;
   /* Die dritte Wandlerfunktion ausführen */
   SET a = a.Preisvorstellung(15.99) ;
   INSERT INTO METALSKU VALUES (a, 31415926) ;
END
```

Benutzerdefinierte Typen, die von Collection-Typen abgeleitet sind

In dem vorangegangenen Abschnitt *Eindeutige Typen (DISTINCT)* zeige ich, wie Sie einen benutzerdefinierten Typ von einem vordefinierten Typ ableiten können. Als Beispiel definiere ich dort den Type USDollar vom Typ DECIMAL ab. Diese Funktionalität wurde in SQL:1999 eingeführt. SQL:2011 erweitert sie so, dass Sie benutzerdefinierte Typen auch von Collection-Typen ableiten können. Dies ermöglicht es Entwicklern, Methoden für ein Array insgesamt und nicht nur für die einzelnen Elemente des Arrays zu definieren, wie dies in SQL:1999 erlaubt war.

Zusammenfassung der Datentypen

Tabelle 2.2 enthält eine Übersicht über die verschiedenen Datentypen und zeigt Muster für jeden Datentyp.

Datentyp	Beispielwert
CHARACTER (20)	'Amateurfunk' '
VARCHAR (20)	'Amateurfunk'
CLOB (1000000)	'Diese Zeichenkette ist eine Million Zeichen lang . . .'
SMALLINT, BIGINT oder INTEGER	7500
NUMERIC oder DECIMAL	3425.432
REAL, FLOAT oder DOUBLE PRECISION	6.626E-34
BINARY (1)	'01100011'
VARBINARY (4)	'011000111100011011100110'
BLOB (1000000)	'1001001110101011010101010101. . .'
BOOLEAN	'TRUE'
DATE	DATE '1957-08-14'
TIME (2) WITHOUT TIME ZONE (Das Argument gibt die Anzahl der Stellen des Nachkommateils an.)	TIME '12:46:02.43' WITHOUT TIME ZONE
TIME (3) WITH TIME ZONE	TIME '12:46:02.432-08:00' WITH TIME ZONE
TIMESTAMP WITHOUT TIME ZONE (0)	TIMESTAMP '1957-08-14 12:46:02' WITHOUT TIME ZONE
TIMESTAMP WITH TIME ZONE (0)	TIMESTAMP '1957-08-14 12:46:02-08:00' WITH TIME ZONE
INTERVAL DAY	INTERVAL '4' DAY
XML(SEQUENCE)	<Client>Vince Tenetria</Client>
ROW	ROW (Street VARCHAR (25), City VARCHAR (20), State CHAR (2), PostalCode VARCHAR (9))
ARRAY	INTEGER ARRAY [15]
MULTISET	Auf diesen Datentyp passt kein Muster.
REF	Kein Typ, sondern ein Zeiger
USER DEFINED TYPE	Währungstyp, der auf DECIMAL basiert

Tabelle 2.2: Datentypen

Denken Sie daran, dass Ihre spezielle Implementierung von SQL möglicherweise nicht alle Datentypen unterstützt, die ich in diesem Abschnitt beschrieben habe. Außerdem könnte sie Datentypen unterstützen, die nicht dem Standard entsprechen und die ich hier nicht beschrieben habe.

Nullwerte

Wenn ein Datenbankfeld ein Datenelement enthält, besitzt das Feld einen speziellen Wert. Wenn ein Feld kein Datenelement enthält, sagt man, dass es einen *Nullwert* enthält.

- ✔ Bei einem numerischen Feld ist ein Nullwert nicht dasselbe wie der Wert null.
- ✔ Bei einem Zeichenfeld ist ein Nullwert nicht dasselbe wie ein Leerzeichen.

Sowohl die numerische Null als auch das Leerzeichen sind bestimmte Werte. Ein Nullwert drückt dagegen aus, dass der Wert eines Feldes nicht definiert ist – sein Wert ist unbekannt.

In einigen Situationen kann ein Feld einen Nullwert haben. Die folgende Liste beschreibt einige dieser Situationen und nennt entsprechende Beispiele:

- ✔ **Der Wert existiert, aber Sie kennen ihn noch nicht.** Sie setzen in der Zeile `TOP` der Tabelle `Quark` die Spalte `Masse` auf NULL, weil die Masse des Top-Quarks noch nicht korrekt bestimmt wurde.
- ✔ **Der Wert existiert noch nicht.** Sie setzen die Spalte `GESAMTUMSATZ` in der Zeile `SQL für Dummies` der Tabelle `BUECHER` auf NULL, weil die Umsatzzahlen des ersten Quartals noch nicht bekannt sind.
- ✔ **Das Feld ist auf die spezielle Zeile nicht anwendbar.** Sie setzen in der Zeile `C-3PO` der Tabelle `Angestellte` die Spalte `Geschlecht` auf NULL, weil `C-3PO` ein geschlechtsloser Androide ist (was Sie natürlich wussten).
- ✔ **Der Wert liegt außerhalb des Wertebereichs.** Sie setzen in der Zeile `Michael Schumacher` der Tabelle `Angestellte` die Spalte `Gehalt` auf NULL, weil Sie der Spalte `Gehalt` den Datentyp `NUMERIC(8,2)` zugewiesen haben und Schumachers Vertrag einiges mehr als 999.999,99 Euro vorsieht (was Sie natürlich auch wussten).

Es gibt verschiedene Gründe dafür, dass ein Feld einen Nullwert hat. Ziehen Sie keine vorschnellen Schlüsse darüber, was ein spezieller Nullwert bedeutet.

Einschränkungen

Einschränkungen (englisch *Constraints*) sind Begrenzungen, die Sie für die Dateneingabe festlegen. Wenn Sie beispielsweise bereits wissen, dass die Einträge in einer bestimmten numerischen Spalte innerhalb eines Wertebereichs liegen müssen, können Sie mit einer Einschränkung verhindern, dass jemand falsche Werte, also solche, die außerhalb dieses Wertebereichs liegen, in diese Spalte eingibt.

Früher definierte die Anwendung, die die Datenbank benutzte, die erforderlichen Einschränkungen für diese Datenbank. Bei den neueren DBMS-Produkten ist es jedoch möglich, Ein-

schränkungen direkt in der Datenbank zu definieren. Dieser Ansatz bietet mehrere Vorteile. Falls mehrere Anwendungen dieselbe Datenbank verwenden, müssen Sie die Einschränkung nur einmal und nicht für jede Anwendung formulieren. Außerdem ist es einfacher, Einschränkungen auf der Datenbankebene zu definieren, als sie einer Anwendung hinzuzufügen. In vielen Fällen genügt eine einfache Klausel in Ihrer CREATE-Anweisung.

Einschränkungen und Zusicherungen (englisch *assertion*; das sind Einschränkungen, die auf mehr als eine Tabelle angewendet werden) behandele ich detailliert in Kapitel 5.

SQL in einem Client/Server-System benutzen

SQL ist eine Datenuntersprache, die auf eigenständigen oder Mehrbenutzersystemen eingesetzt wird. SQL eignet sich besonders für den Einsatz in einem Client/Server-System. Bei einem solchen System sind mehrere Benutzer, die an sogenannten *Client-Rechnern* arbeiten, über ein lokales Netzwerk (LAN) oder einen anderen Kommunikationskanal mit einer Datenbank verbunden, die auf einem zentralen Rechner, dem sogenannten *Server*, liegt. Die Anwendung auf dem Client enthält SQL-Anweisungen zur Datenbearbeitung.

Der Teil des DBMS, der sich auf dem Client befindet, sendet diese Anweisungen über die Kommunikationskanäle, die den Client mit dem Server verbinden, an den Server. Auf dem Server interpretiert die Server-Komponente des DBMS die SQL-Anweisungen, führt sie aus und sendet das Ergebnis über die Kommunikationskanäle zurück an den Client. Sie können auf dem Client sehr komplexe Operationen in SQL kodieren und diese Operationen auf dem Server dekodieren und ausführen lassen. Mit dieser Konfiguration wird die Bandbreite des Kommunikationskanals am besten genutzt.

Wenn Sie auf einem Client/Server-System Daten mit SQL abrufen, werden nur die gewünschten Daten über den Kommunikationskanal vom Server zum Client übertragen. Im Gegensatz dazu werden bei einem einfachen System zur gemeinsamen Nutzung von Ressourcen, dessen Server nur über eine minimale Intelligenz verfügt, riesige Datenblöcke über den Kanal übertragen, um Ihnen die kleine Menge von Daten zu liefern, die Sie benötigen. Diese Art massiver Datenübertragung kann natürlich eine Operation beträchtlich verlangsamen. Die Client/Server-Architektur ergänzt die Eigenschaften von SQL und sorgt bei kleinen, mittleren und großen Netzwerken dafür, dass ein gutes Leistungsverhalten zu vertretbaren Kosten möglich wird.

Der Server

Der Server tritt erst in Aktion, wenn er die Anfrage eines Clients erhält, sonst wartet er nur. Wenn mehrere Clients gleichzeitig seinen Dienst beanspruchen wollen, muss der Server jedoch schnell reagieren können. Server unterscheiden sich von Client-Rechnern im Allgemeinen dadurch, dass sie über sehr große, schnelle Plattenspeicher verfügen. Server sind im Hinblick auf einen schnellen Datenzugriff und den Abruf von Daten optimiert. Weil sie häufig Anfragen mehrerer Clients gleichzeitig bearbeiten müssen, brauchen sie einen schnellen Prozessor oder sogar mehrere Prozessoren.

Was ist der Server?

Der *Server* (die in diesem Buch verwendete Kurzform für *Datenbank-Server*) ist die Komponente eines Client/Server-Systems, auf dem die Datenbank liegt. Der Server speichert außerdem den Serverteil eines Datenbankverwaltungssystems. Dieser Teil des DBMS interpretiert Anweisungen, die von den Clients kommen, und übersetzt sie in Datenbankoperationen. Außerdem formatiert die Serversoftware die Ergebnisse von Datenabfragen und sendet sie an den anfragenden Client zurück.

Was macht der Server?

Die Aufgabe des Servers ist relativ einfach und unkompliziert. Ein Server braucht nur die Anweisungen, die er über das Netzwerk von den Clients erhält, zu lesen, zu interpretieren und auszuführen. Diese Anweisungen sind in einer von mehreren Datenuntersprachen formuliert.

Eine Datenuntersprache ist keine vollständige Programmiersprache – sie führt nur einen Teil einer Sprache aus. Eine Datenuntersprache beschäftigt sich nur mit der Handhabung von Daten. Sie enthält Operationen für das Einfügen, Ändern, Löschen und Auswählen von Daten, aber keine Strukturen zur Steuerung des Programmablaufs, wie zum Beispiel `DO`-Schleifen, keine lokalen Variablen, keine Funktionen oder Prozeduren und keine Anweisungen für die Druckerausgabe. SQL ist heute die gebräuchlichste Datenuntersprache und hat sich zum Industriestandard entwickelt. SQL hat proprietäre Datenuntersprachen auf Rechnern aller Leistungsklassen abgelöst. Mit SQL:1999 hat SQL viele Funktionen erhalten, die den traditionellen Datenuntersprachen fehlen. Dennoch ist es immer noch keine vollständige Allzweck-Programmiersprache und muss deshalb mit einer Hostsprache kombiniert werden, wenn Sie eine Datenbankanwendung erstellen möchten.

Der Client

Der *Client*-Teil eines Client/Server-Systems besteht aus einer Hardware- und einer Software-Komponente. Die Hardware besteht aus dem Client-Computer und seiner Schnittstelle zum lokalen Netzwerk. Diese Hardware kann der Hardware des Servers sehr ähnlich sein oder ihr sogar entsprechen. Die Software ist die Komponente, die den Client vom Server unterscheidet.

Was ist der Client?

Die Hauptaufgabe des Clients besteht darin, eine Benutzeroberfläche zur Verfügung zu stellen. Aus der Sicht des Benutzers *ist* der Client-Rechner der Computer und die Benutzeroberfläche *ist* die Anwendung. Der Benutzer erkennt möglicherweise gar nicht, dass seine Arbeit auch mit einem Server zu tun hat. Der Server ist normalerweise nicht sichtbar – und häufig sogar in einem anderen Raum untergebracht. Abgesehen von der Benutzeroberfläche enthält der Client auch das Anwendungsprogramm und den clientseitigen Teil des DBMS. Das Anwendungsprogramm führt spezielle Aufgaben aus, zum Beispiel die Erfassung von offenen Rechnungen oder Bestellungen. Der Client-Teil des DBMS führt die Anweisungen des Anwendungsprogramms aus und tauscht Daten und SQL-Anweisungen zur Datenbearbeitung mit dem Server-Teil des DBMS aus.

Was macht der Client?

Der Client-Teil eines DBMS zeigt Informationen auf dem Bildschirm an und reagiert auf die Eingaben des Benutzers, die per Tastatur, Maus oder über ein anderes Eingabegerät erfolgen. Der Client kann auch die Daten verarbeiten, die über eine Telekommunikationsverbindung oder von anderen Arbeitsstationen in das Netzwerk gesendet werden. Der Client-Teil des DBMS ist für die anwendungsspezifischen Aufgaben zuständig. Für einen Entwickler ist der Client-Teil eines DBMS der interessante Teil. Der Server-Teil bedient nur auf mechanische und immer gleiche Weise die Anfragen des Clients.

SQL mit dem Internet oder einem Intranet benutzen

Der Betrieb von Datenbanken im Internet und in Intranets unterscheidet sich grundsätzlich von der Arbeit mit einem traditionellen Client/Server-System. Der Unterschied besteht hauptsächlich auf der Client-Seite. Bei einem traditionellen Client/Server-System befindet sich ein großer Teil der Funktionalität des DBMS auf dem Client-Rechner. Bei internetbasierten Datenbanksystemen befindet sich das gesamte DBMS oder der größte Teil davon auf dem Server. Der Client ist häufig nicht mehr als ein Webbrowser, der möglicherweise noch über eine Erweiterung in Form eines Netscape-Plug-ins oder eines ActiveX-Steuerelements verfügt. Deshalb verlagert sich der konzeptionelle Massenschwerpunkt des Systems auf den Server. Diese Verlagerung bietet mehrere Vorteile:

- ✔ Der Client-Teil des Systems (Browser) ist preiswert oder sogar kostenlos.
- ✔ Es gibt eine standardisierte Benutzeroberfläche.
- ✔ Der Client ist einfach zu verwalten.
- ✔ Es gibt eine standardisierte Client/Server-Beziehung.
- ✔ Die üblichen Mittel zur Anzeige von Multimediadaten können eingesetzt werden.

Die Hauptnachteile von Datenbankmanipulationen über das Internet betreffen die Sicherheit und die Datenintegrität:

- ✔ Um die Daten vor unerwünschtem Zugriff und Manipulationen zu schützen, müssen sowohl der Webserver als auch der clientseitige Browser leistungsstarke Verschlüsselungsmechanismen unterstützen.
- ✔ Ein Browser kann keine angemessenen Gültigkeitsprüfungen bei der Dateneingabe durchführen.
- ✔ Datenbanktabellen, die sich auf verschiedenen Servern befinden, werden möglicherweise nicht ordnungsgemäß synchronisiert.

Clientseitige und serverseitige Erweiterungen, die diese Nachteile ausgleichen, machen das Internet zu einer Umgebung, in der Datenbankanwendungen produktiv eingesetzt werden können. Die Architektur von Intranets ähnelt der des Internets, aber die Fragen der Sicherheit stehen hier nicht so sehr im Vordergrund. Weil ein Unternehmen bei einem Intranet die Kontrolle über alle Client-Rechner, den Server und das Netzwerk hat, sind Intranets weniger durch Einbrüche böswilliger Hacker gefährdet. In Intranets ist das Problem eher bei Dateneingabefehlern und einer nicht ordnungsgemäßen Synchronisierung von Datenbanken zu suchen.

Die Komponenten von SQL

3

In diesem Kapitel

- Datenbanken erstellen
- Daten verändern
- Datenbanken schützen

SQL ist eine Spezialsprache, die für die Erstellung und Verwaltung von Daten in relationalen Datenbanken entwickelt worden ist. Obwohl alle Anbieter relationaler Datenbankverwaltungssysteme ihre eigenen Implementierungen von SQL verwenden, definiert ein ISO/ANSI-Standard (2011 überarbeitet), was SQL ist. Alle Implementierungen unterscheiden sich mehr oder weniger von diesem Standard. Eine enge Anlehnung an den Standard ist besonders für diejenigen Anwender wichtig, die ihre Datenbanken (und die zugehörigen Anwendungen) auf mehr als einer Plattform einsetzen wollen.

Obwohl SQL keine Allzweck-Programmiersprache ist, enthält es einige beeindruckende Werkzeuge. Drei Untersprachen der Sprache bieten Ihnen alles, was Sie benötigen, um relationale Datenbanken zu erstellen, zu verwalten, zu sichern und zu schützen.

- ✔ **DDL (Data Definition Language, Datendefinitionssprache):** Dies ist der Teil von SQL, mit dem Sie Datenbanken erstellen, die Struktur von Datenbanken ändern und Datenbanken löschen, die Sie nicht mehr benötigen.
- ✔ **DML (Data Manipulation Language, Datenbearbeitungssprache):** Sie dient der Datenbankverwaltung. Mit diesem leistungsfähigen Hilfsmittel können Sie genau angeben, was mit den Daten in Ihrer Datenbank geschehen soll – Sie können Daten eingeben, ändern oder abrufen.
- ✔ **DCL (Data Control Language, Datenkontrollsprache):** Sie schützt Ihre Datenbanken vor Beschädigung. Wenn Sie DCL korrekt einsetzen, ist für die Sicherheit Ihrer Datenbank gesorgt. Der Umfang der Schutzmaßnahmen, die DCL bereitstellt, ist von Ihrer Implementierung abhängig. Wenn Ihre Implementierung nicht genügend Schutzmöglichkeiten bietet, müssen Sie den benötigten Schutz in Ihre Anwendungsprogramme einbauen.

Dieses Kapitel führt Sie in DDL, DML und DCL ein.

Data Definition Language

DDL, die Datendefinitionssprache, ist der Teil von SQL, mit dem Sie die Grundkomponenten einer relationalen Datenbank erstellen, ändern oder löschen. Zu den Grundkomponenten gehören Tabellen, Sichten, Schemata, Kataloge, Cluster und möglicherweise eine Reihe anderer Dinge. In diesem Abschnitt behandle ich die Hierarchie, die die Beziehungen zwischen diesen

Komponenten regelt. Ich beschreibe außerdem die Anweisungen, die auf diese Komponenten angewendet werden können.

In Kapitel 1 habe ich Tabellen und Schemata erwähnt und darauf hingewiesen, dass ein Schema eine allgemeine Struktur ist, die Tabellen enthält. Tabellen und Schemata sind zwei Elemente der sogenannten *Containment-Hierarchie* (wörtlich *Enthaltenseinshierarchie*) einer relationalen Datenbank. Sie können diese Hierarchie folgendermaßen gliedern:

- ✔ Tabellen enthalten Spalten und Zeilen.
- ✔ Schemata enthalten Tabellen und Sichten.
- ✔ Kataloge enthalten Schemata.

Die Datenbank selbst enthält Kataloge. Eine Datenbank, die über mehrere Rechner verteilt ist, wird bisweilen auch als *Cluster* oder *Datenbankcluster* bezeichnet. Cluster werden etwas später in diesem Kapitel im Abschnitt über Kataloge noch einmal erwähnt.

Wenn »Mach' es einfach!« kein guter Rat ist

Angenommen, Sie wollten eine Datenbank für Ihr Unternehmen erstellen. Erfüllt von dem Gedanken, eine nützliche, wertvolle und vollkommen korrekte Struktur zu entwerfen, die für die Zukunft Ihrer Firma von großer Bedeutung ist, setzen Sie sich direkt an Ihren Computer und beginnen damit, `CREATE`-Anweisungen in SQL einzugeben. Richtig?

Nein, falsch! Ganz falsch! Dieses Vorgehen ist ein sicherer Weg in die Katastrophe. Viele Datenbankprojekte laufen von Anfang an schief, weil die Kombination aus Aufregung und Enthusiasmus eine sorgfältige Planung in den Hintergrund gedrängt hat. Selbst wenn Sie eine klare Vorstellung von der Struktur Ihrer Datenbank besitzen, *schreiben Sie alles auf einem Stück Papier nieder*, bevor Sie die Tastatur berühren.

Datenbankentwicklung ähnelt dem Schach. Mitten in einem spannenden Schachspiel könnten Sie etwas entdecken, was wie ein guter Zug aussieht. Der Druck auf Sie, diesen Zug zu machen, kann Sie fast überwältigen. Dennoch besteht die Gefahr, dass Sie etwas übersehen haben. Großmeister geben Anfängern gerne den nicht ganz ernst gemeinten Rat, sich auf ihre Hände zu setzen. Das Sitzen auf den Händen hindert Sie daran, einen voreiligen Zug zu machen. Also: Setzen Sie sich auf Ihre Hände. Wenn Sie sich die Zeit nehmen, die Positionen der Figuren ein wenig länger zu studieren, finden Sie vielleicht einen besseren Zug – vielleicht sehen Sie aber auch einen brillanten Gegenzug, den Ihr Gegner unternehmen könnte.

Sich in die Datenbankentwicklung zu stürzen, ohne vorher die Datenbankstruktur gründlich zu durchdenken, führt bestenfalls zu suboptimalen Lösungen. Im schlimmsten Fall ist das Ergebnis ein Desaster, eine Einladung, die Daten zu beschädigen. Nehmen Sie also Papier und Bleistift und planen Sie Ihre Datenbank. Hilfe, zu entscheiden, was Ihr Plan enthalten sollte, finden Sie in meinem Buch *Datenbankentwicklung für Dummies*, in dem der Planungsprozess ausführlich beschrieben wird.

Beachten Sie bei der Planung Ihrer Datenbank folgende Punkte:

- ✔ Identifizieren Sie alle Tabellen.
- ✔ Definieren Sie alle Spalten aller Tabellen.
- ✔ Weisen Sie jeder Tabelle einen *Primärschlüssel* zu, um Eindeutigkeit zu gewährleisten (siehe Kapitel 4 und 5).
- ✔ Sorgen Sie dafür, dass jede Tabelle der Datenbank mindestens eine Spalte mit (wenigstens) einer anderen Tabelle gemein hat. Diese gemeinsamen Spalten dienen als logische Verknüpfungen, mit denen Sie Daten in einer Tabelle mit denen einer anderen verknüpfen können.
- ✔ Bringen Sie jede Tabelle in die *dritte Normalform* (3NF), um Anomalien beim Einfügen, Löschen oder Aktualisieren von Daten zu verhindern (siehe Kapitel 5).

Nachdem Sie die Datenbank auf Papier entworfen und den Entwurf auf seine Brauchbarkeit geprüft haben, können Sie ihn mit SQL-`CREATE`-Anweisungen auf den Computer übertragen. Doch wahrscheinlich verwenden Sie eher die grafische Benutzerschnittstelle (GUI) Ihres DBMS, um die Komponenten Ihres Entwurfs zu erstellen. Wenn Sie ein GUI verwenden, werden Ihre Eingaben von Ihrem DBMS im Hintergrund in SQL umgewandelt.

Tabellen erstellen

Eine Datenbanktabelle ist eine zweidimensionale Anordnung aus Zeilen und Spalten. Tabellen werden mit der SQL-Anweisung `CREATE TABLE` erstellt. In dieser Anweisung legen Sie den Namen und den Datentyp jeder Spalte fest.

Wenn Sie eine Tabelle erstellt haben, können Sie Daten darin speichern. (Diese Funktion gehört zur DML und nicht zur DDL.) Wenn sich die Anforderungen ändern, können Sie die Struktur einer Tabelle mit der Anweisung `ALTER TABLE` ändern. Wenn eine Tabelle nicht mehr benötigt wird, können Sie sie mit der Anweisung `DROP` löschen. Die verschiedenen Formen der Anweisungen `CREATE`, `ALTER` und `DROP` bilden die DDL von SQL.

Angenommen, Sie seien Datenbankdesigner und wollten verhindern, dass Ihre Datenbanktabellen nach und nach durch Eingabe fehlerhafter Daten unbrauchbar werden. Deshalb wollen Sie die Datenbank der besten Normalform entsprechend strukturieren, um die Integrität der Daten zu gewährleisten.

Normalisierung ist eine Methode, um Datenbanktabellen so zu strukturieren, dass durch Änderungen der Daten keine Anomalien in der Datenbank entstehen. Die Spalten jeder Tabelle, die Sie erstellen, entsprechen Attributen, die eng miteinander verknüpft sind.

Angenommen, Sie möchten eine Kundentabelle erstellen. Die Tabelle heißt `Kunde` und hat die Attribute `Kunde.KundenID`, `Kunde.Vorname`, `Kunde.Nachname`, `Kunde.Straße`, `Kunde.Ort`, `Kunde.Land`, `Kunde.Plz` und `Kunde.Tel`. Diese Attribute sind enger mit der Kunden-Entität (A.d.Ü.: *Entity*, deutsch *Entität*, ist das, was durch eine Tabelle erfasst oder modelliert wird) verbunden als mit den anderen Entities in den Tabellen Ihrer Datenbank.

Diese Attribute enthalten alle relativ dauerhaften Kundendaten, die Ihr Unternehmen speichert.

Die meisten Datenbankverwaltungssysteme stellen heute grafische Werkzeuge zur Verfügung, um solche Tabellen zu erstellen. Sie können Tabellen jedoch auch mit einer SQL-Anweisung anlegen. Das folgende Beispiel zeigt die Anweisung, mit der Sie die Tabelle `Kunde` definieren:

```
CREATE TABLE Kunde (
   KundenID     INTEGER          NOT NULL,
   Vorname      CHARACTER(15),
   Nachname     CHARACTER(20)    NOT NULL,
   Straße       CHARACTER(25),
   Plz          INTEGER,
   Ort          CHARACTER(20),
   Land         CHARACTER(2),
   Tel          CHARACTER(13) ) ;
```

Für jede Spalte legen Sie den Spaltennamen (beispielsweise `KundenID`), den Datentyp (beispielsweise `INTEGER`) und möglicherweise eine oder mehrere Einschränkungen (beispielsweise `NOT NULL`) fest.

Abbildung 3.1 zeigt einen Teil der Tabelle `Kunde` mit einigen Beispieldaten.

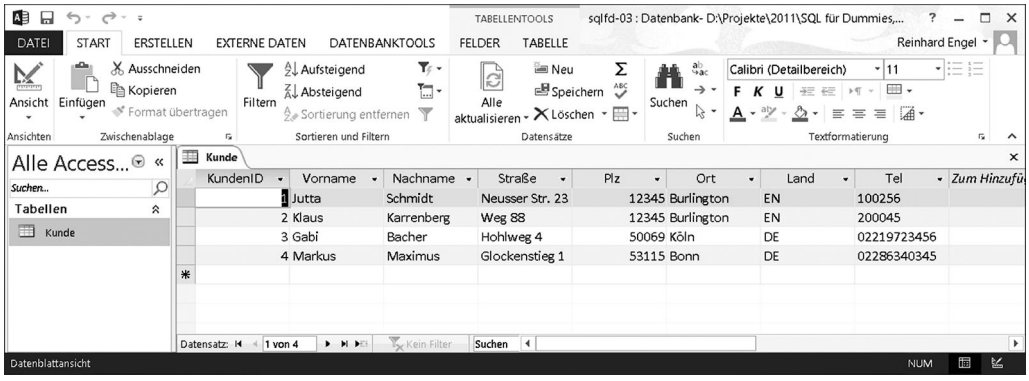


Abbildung 3.1: Die Tabelle `Kunde` mit einigen Beispieldaten

Wenn Ihre SQL-Implementierung nicht die neueste Version von ISO/IEC-Standard-SQL vollständig unterstützt, kann die Syntax, die Sie benötigen, von der hier gezeigten abweichen. Lesen Sie die Dokumentation zu Ihrem Datenbankverwaltungssystem, um die erforderlichen Informationen zu erhalten.

Sichten

Manchmal wollen Sie einige spezielle Informationen aus der Tabelle Kunde abrufen und benötigen dabei nicht alle, sondern nur ganz bestimmte Spalten und Zeilen. Um dies zu erreichen, verwenden Sie eine Sicht (englisch *View*).

Eine *Sicht* ist eine virtuelle Tabelle. Bei den meisten Implementierungen hat eine Sicht keine eigenständige physische Existenz. Die Definition der Sicht existiert nur in den Metadaten der Datenbank, aber die tatsächlichen Daten stammen aus der Tabelle oder den Tabellen, von der oder denen Sie die Sicht ableiten. Die Daten der Sicht werden physisch nirgendwo auf dem Online-Plattenspeicher dupliziert. Einige Sichten bilden spezielle Spalten und Zeilen einer einzigen Tabelle ab. Andere, die sogenannten *Mehrtabellensichten*, beziehen ihre Daten aus zwei oder mehr Tabellen.

Sichten aus einzelnen Tabellen

Manchmal ist die Antwort auf eine Frage in den Daten einer einzigen Tabelle Ihrer Datenbank enthalten. Wenn alle gesuchten Informationen in einer einzigen Tabelle vorhanden sind, können Sie eine Sicht auf der Basis dieser einzelnen Tabelle erstellen. Angenommen, Sie möchten die Namen und Telefonnummern aller Kunden sehen, die in *Deutschland* leben. Sie können dann eine Sicht aus der Tabelle Kunde erstellen, die nur die gewünschten Daten enthält. Die folgende SQL-Anweisung erstellt diese Sicht:

```
CREATE VIEW DeKunde AS
   SELECT Kunde.Vorname,
          Kunde.Nachname,
          Kunde.Tel
      FROM Kunde
      WHERE Kunde.Land = 'DE';
```

Abbildung 3.2 zeigt, wie die Sicht aus der Tabelle KUNDE abgeleitet wird.

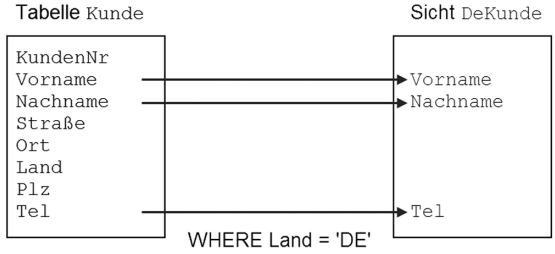


Abbildung 3.2: Die Sicht DeKunde wird aus der Tabelle Kunde abgeleitet.

Dieser Code ist korrekt, aber etwas wortreich. Sie können dasselbe Ergebnis mit einem geringeren Tippaufwand erreichen, wenn Ihre SQL-Implementierung davon ausgeht, dass alle Tabellenbeziehungen dieselben wie in der FROM-Klausel sind. Wenn Ihr System von dieser vernünftigen Standardannahme ausgeht, können Sie die Anweisung folgendermaßen verkürzen:

```
CREATE VIEW DeKunde AS
   SELECT Vorname, Nachname, Tel
      FROM Kunde
      WHERE Land = 'DE';
```

Die zweite Version ist leichter zu schreiben und zu lesen, aber sie ist anfälliger für Störungen durch ALTER TABLE-Anweisungen. Eine solche Störung stellt für diesen einfachen Fall ohne JOIN-Klausel kein Problem dar. Doch Sichten mit JOIN-Klauseln sind viel robuster, wenn sie vollständige Spaltennamen (inklusive des Tabellennamens) benutzen. Ich behandle JOIN-Klauseln in Kapitel 11.

Die Mehrtabellensicht

In der Regel benötigen Sie Daten aus zwei oder mehr Tabellen, um Ihre Fragen beantworten zu können. Angenommen, Sie arbeiten für ein Sportartikelgeschäft und möchten eine Liste aller Kunden erstellen, die seit der Eröffnung des Geschäfts im letzten Jahr eine Skiausrüstung gekauft haben. Sie haben vor, diesen Kunden ein Werbeschreiben zu senden. Dazu benötigen Sie Daten aus den Tabellen Kunde, Produkt, Rechnung und RechnungPosition. Sie können eine Mehrtabellensicht erstellen, die die benötigten Daten anzeigt, und Sie können diese Sicht immer wieder verwenden. Jedes Mal, wenn Sie die Sicht benutzen, zeigt sie alle Änderungen an, die seit ihrer letzten Benutzung an den zugrunde liegenden Tabellen vorgenommen worden sind.

Die Datenbank des Sportartikelgeschäfts enthält vier Tabellen: Kunde, Produkt, Rechnung und RechnungPosition. Die Tabellen haben die folgenden Strukturen:

Tabelle	Spalte	Datentyp	Einschränkung
Kunde	KundenID	INTEGER	NOT NULL
	Vorname	CHARACTER(15)	
	Nachname	CHARACTER(20)	NOT NULL
	Straße	CHARACTER(25)	
	Ort	CHARACTER(20)	
	Land	CHARACTER(2)	
	Plz	INTEGER	
	Tel	CHARACTER(13)	

Tabelle	Spalte	Datentyp	Einschränkung
Produkt	ProduktID	INTEGER	NOT NULL
	Name	CHARACTER(25)	
	Beschreibung	CHARACTER(30)	
	Kategorie	CHARACTER(15)	
	LieferantID	INTEGER	
	Lieferantname	CHARACTER(30)	
Rechnung	RechnungNr	INTEGER	NOT NULL
	KundenID	INTEGER	
	Rechnungdatum	DATE	
	Umsatz	NUMERIC(9,2)	
	Bezahlt	NUMERIC(9,2)	
	Zahlung	CHARACTER(10)	
RechnungPosition	PosNr	INTEGER	NOT NULL
	RechnungNr	INTEGER	
	ProduktID	INTEGER	
	Menge	INTEGER	
	Verkaufspreis	NUMERIC(9,2)	

Tabelle 3.1: Die Datenbanktabellen des Sportartikelgeschäfts

Einige Spalten in Tabelle 3.1 enthalten die Einschränkung `NOT NULL`. Diese Spalten gehören entweder zum Primärschlüssel der jeweiligen Tabelle oder es sind Spalten, die Ihrer Meinung nach aus dem einen oder anderen Grund einen Wert enthalten müssen. Der Primärschlüssel einer Tabelle muss jede Zeile eindeutig identifizieren. Er darf deshalb in keiner Zeile einen Nullwert enthalten. (Schlüssel werden detailliert in Kapitel 5 behandelt.)

Die Tabellen sind durch die gemeinsamen Spalten miteinander verknüpft. Die folgende Liste verdeutlicht diese Beziehungen (siehe auch Abbildung 3.3).

- ✔ Die Tabelle `Kunde` steht zur Tabelle `Rechnung` in einer *Eins-zu-viele-Beziehung* (1:n-Beziehung). Ein Kunde kann mehrere Käufe tätigen und dadurch mehrere Rechnungen verursachen. Jede Rechnung bezieht sich jedoch nur auf genau einen Kunden.
- ✔ Die Tabelle `Rechnung` steht zur Tabelle `RechnungPosition` in einer 1:n-Beziehung. Eine Rechnung kann aus mehreren Positionen bestehen, aber jede Position erscheint nur auf einer Rechnung.
- ✔ Die Tabelle `Produkt` steht zu der Tabelle `RechnungPosition` ebenfalls in einer 1:n-Beziehung. Ein Produkt kann in mehreren Positionen und auf mehreren Rechnungen stehen. Aber jede Position enthält genau ein Produkt.

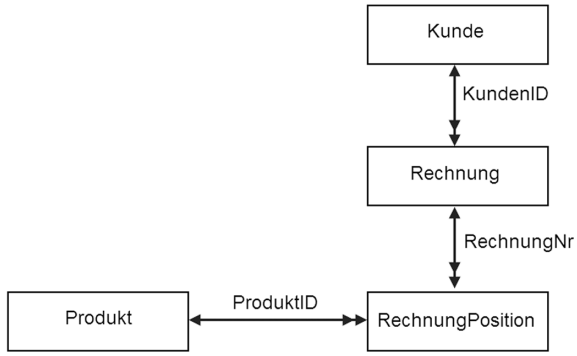


Abbildung 3.3: Die Datenbankstruktur des Sportartikelgeschäfts

Die Tabelle `Kunde` ist mit der Tabelle `Rechnung` durch die gemeinsame Spalte `KundenID` verknüpft. Die Tabelle `Rechnung` ist mit der Tabelle `RechnungPosition` durch die gemeinsame Spalte `RechnungNr` verknüpft. Die Tabelle `Produkt` ist mit der Tabelle `RechnungPosition` durch die gemeinsame Spalte `ProduktID` verknüpft. Diese Verknüpfungen sind es, die diese Datenbank zu einer *relationalen* Datenbank machen.

Um Informationen über die Kunden zu bekommen, die eine Skiausrüstung gekauft haben, benötigen Sie die Felder `Vorname`, `Nachname`, `Straße`, `Ort`, `Land` und `Plz` aus der Tabelle `Kunde`, `Kategorie` aus der Tabelle `Produkt`, `RechnungNr` aus der Tabelle `Rechnung` und `PosNr` aus der Tabelle `RechnungPosition`. Sie könnten die gewünschte Sicht schrittweise mit der folgenden Anweisung erstellen:

```
CREATE VIEW SkiKunde1 AS
    SELECT Vorname,
        Nachname,
        Straße,
        PLZ,
        Ort,
        Land,
        RechnungNr
    FROM Kunde JOIN Rechnung
    USING (KundenID) ;
CREATE VIEW SkiKunde2 AS
    SELECT Vorname,
        Nachname,
        Straße,
        Plz,
        Ort,
        Land,
        ProduktID
    FROM SkiKunde1 JOIN RechnungPosition
    USING (RechnungNr) ;
```

```
CREATE VIEW SkiKunde3 AS
    SELECT Vorname,
        Nachname,
        Straße,
        Plz,
        Ort,
        Land,
        Kategorie
    FROM SkiKunde2 JOIN Produkt
    USING (ProduktID) ;
CREATE VIEW SkiKunde AS
    SELECT DISTINCT Vorname,
        Nachname,
        Straße,
        Plz
        Ort,
        Land,
    FROM SkiKunde3
    WHERE Kategorie = 'Ski' ;
```

Diese `CREATE VIEW`-Anweisungen kombinieren Daten aus mehreren Tabellen mit dem `JOIN`-Operator. Abbildung 3.4 veranschaulicht diesen Prozess.

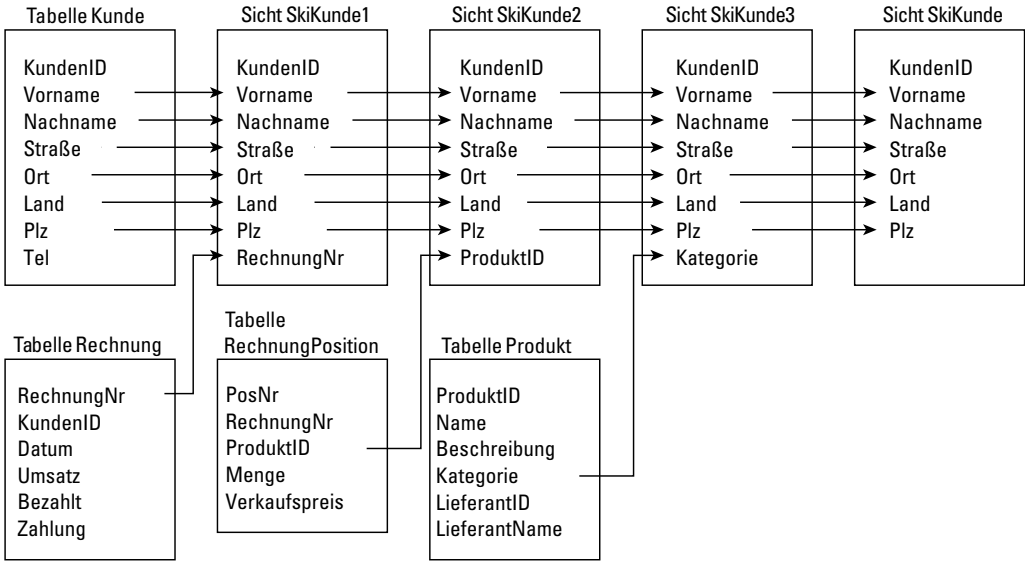


Abbildung 3.4: Eine Mehrtabellensicht mit `JOIN` erstellen

Nachfolgend finden Sie eine Erläuterung der vier CREATE VIEW-Anweisungen:

- ✔ Die erste Anweisung kombiniert Spalten der Tabelle Kunde mit einer Spalte der Tabelle Rechnung, um die Sicht SkiKunde1 zu erstellen.
- ✔ Die zweite Anweisung kombiniert SkiKunde1 mit einer Spalte der Tabelle Rechnung Position, um die Sicht SkiKunde2 zu erstellen.
- ✔ Die dritte Anweisung kombiniert SkiKunde2 mit einer Spalte der Tabelle Produkt, um die Sicht SkiKunde3 zu erstellen.
- ✔ Die vierte Anweisung filtert alle Zeilen, bei denen die Spalte Kategorie den Wert »Ski« enthält. Das Endergebnis ist die Sicht SkiKunde, die die Namen und Adressen aller Kunden enthält, die wenigstens ein Produkt aus der Kategorie »Ski« gekauft haben.

  Das Schlüsselwort DISTINCT in der SELECT-Klausel der vierten CREATE VIEW-Anweisung sorgt dafür, dass Sie nur einen einzigen Eintrag für jeden Kunden erhalten, selbst wenn einige von ihnen mehrfach Skizubehör gekauft haben. (JOINs werden in Kapitel 10 behandelt.)

Es ist möglich, Mehrtabellensichten mit einer einzigen SQL-Anweisung zu erstellen. Doch wenn Sie bereits eine oder alle der voranstehenden Anweisungen für komplex halten sollten, stellen Sie sich vor, wie komplex eine einzelne Anweisung wäre, die alle diese Funktionen ausführte. Ich ziehe Einfachheit der Komplexität vor. Deshalb wähle ich, wann immer das möglich ist, die einfachste Methode, eine Funktion auszuführen, selbst wenn es nicht die »effizienteste« sein sollte.

Tabellen in Schemata zusammenfassen

Eine Tabelle besteht aus Zeilen und Spalten und speichert normalerweise eine spezielle Entity, zum Beispiel Kunden, Produkte oder Rechnungen. In der Praxis benötigen Sie im Allgemeinen Daten über mehrere zusammengehörige Entities. Organisatorisch fassen Sie die Tabellen, denen Sie diese Entities zuordnen, in einem sogenannten logischen Schema zusammen. Ein *logisches Schema* ist die organisatorische Struktur zusammengehöriger Tabellen.

Eine Datenbank verfügt auch über ein *physisches Schema*. Das physische Schema legt fest, wie die Daten und die zugehörigen Komponenten (beispielsweise die Indizes) physisch auf den Speichergeräten des Systems angeordnet sind. Im Folgenden ist mit Schema einer Datenbank das logische Schema gemeint.

Wenn Sie mit einem System arbeiten, das mehrere nicht zusammengehörige Projekte verwaltet, können Sie einem Schema alle zusammengehörigen Tabellen zuordnen. Die anderen Tabellengruppen können Sie ebenfalls in eigenen Schemata zusammenfassen.

Achten Sie darauf, dass Sie Schemata auch benennen, um zu verhindern, dass gleichnamige Tabellen verschiedener Projekte aus Versehen verwechselt werden. Jedes Projekt hat sein eigenes Schema, das sich durch seinen Namen von den anderen Schemata unterscheidet. Häufig werden jedoch Tabellennamen (wie etwa `Kunde`, `Produkt` und so weiter) in verschiedenen Projekten wiederholt verwendet. Falls die Gefahr besteht, dass ein Name nicht eindeutig ist, sollten Sie den Namen des betreffenden Schemas vor Ihren Tabellennamen setzen (zum Beispiel `SchemaName.TabelleName`). Wenn Sie den Tabellennamen nicht auf diese Weise eindeutig bestimmen, ordnet SQL die Tabelle dem Standardschema zu.

Ordnung durch Kataloge

Bei wirklich großen Datenbanksystemen reichen oft mehrere Schemata nicht aus. In einer großen, verteilten Datenbankumgebung mit vielen Benutzern werden manchmal auch die Namen von Schemata mehrfach verwendet. Um die Probleme, die aus einer solchen Situation entstehen können, zu vermeiden, fügt SQL der Komponentenhierarchie eine weitere Ebene hinzu: den Katalog. Ein *Katalog* ist eine benannte Sammlung von Schemata.

Sie können einen Tabellennamen nicht nur mit dem Namen eines Schemas, sondern zusätzlich auch mit einem Katalognamen eindeutig bestimmen, um zu verhindern, dass die Tabelle mit einer gleichnamigen anderen Tabelle in einem gleichnamigen anderen Schema verwechselt wird. Der eindeutige Katalogname wird folgendermaßen angegeben:

```
KatalogName.SchemaName.TabelleName
```

Cluster bilden die höchste Stufe in der Komponentenhierarchie einer Datenbank. Es kommt jedoch selten vor, dass ein System die komplette Hierarchie umfasst. In der Regel geht eine solche Hierarchie nicht über Kataloge hinaus. Ein Katalog enthält Schemata und ein Schema enthält Tabellen und Sichten. Tabellen und Sichten enthalten Spalten und Zeilen.

Der Katalog enthält auch das sogenannte *Informationsschema*. Das Informationsschema enthält die Systemtabellen. Die Systemtabellen enthalten die Metadaten, die mit den anderen Schemata verknüpft sind. In Kapitel 1 habe ich eine Datenbank als selbstbeschreibende Sammlung integrierter Datensätze bezeichnet. Die Metadaten in den Systemtabellen bilden die Informationen, die die Datenbank selbstbeschreibend machen.

Da die Kataloge durch ihren Namen identifiziert werden, kann eine Datenbank mehrere Kataloge enthalten. Jeder Katalog kann mehrere Schemata enthalten, und jedes Schema kann mehrere Tabellen umfassen. Und natürlich kann jede Tabelle mehrere Spalten und Zeilen enthalten. Diese hierarchischen Beziehungen werden in Abbildung 3.5 verdeutlicht.

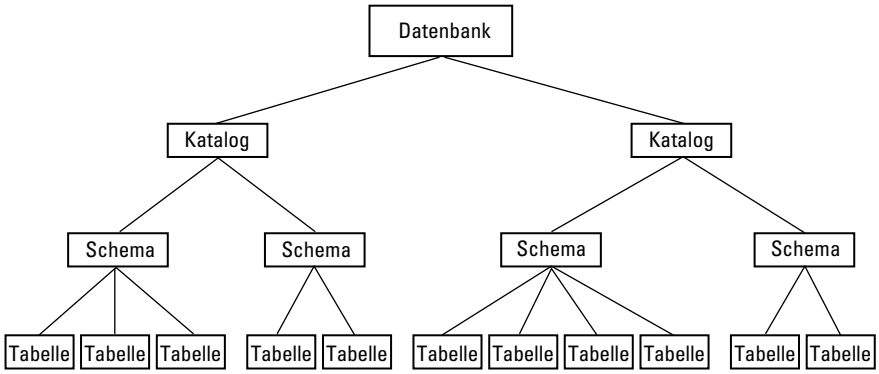


Abbildung 3.5: Die hierarchische Struktur einer typischen SQL-Datenbank

Die DDL-Anweisungen kennenlernen

Die DDL (Data Definition Language) von SQL beschäftigt sich mit der Struktur einer Datenbank, während die Datenbearbeitungssprache DML (die weiter unten in diesem Kapitel im Abschnitt *Data Manipulation Language* beschrieben wird) mit den Daten innerhalb dieser Struktur arbeitet. DDL besteht aus den folgenden drei Anweisungen:

- ✔ `CREATE`. Sie nutzen die verschiedenen Formen dieser Anweisung, um die grundlegenden Strukturen einer Datenbank zu erstellen.
- ✔ `ALTER`. Mit dieser Anweisung ändern Sie die von Ihnen erzeugten Strukturen.
- ✔ `DROP`. Wenn Sie diese Anweisung auf eine Tabelle anwenden, werden sowohl die Daten als auch die Struktur der Tabelle gelöscht.

In den folgenden Abschnitten beschreibe ich kurz die DDL-Anweisungen. Ich verwende diese Anweisungen in Beispielen der Kapitel 4 und 5.

CREATE

Sie können die SQL-Anweisung `CREATE` auf mehrere SQL-Objekte anwenden, einschließlich Schemata, Domänen, Tabellen und Sichten. Mit `CREATE SCHEMA` können Sie ein Schema erstellen, seinen Besitzer festlegen und einen Standardzeichensatz angeben. Ein Beispiel für eine solche Anweisung könnte wie folgt aussehen:

```
CREATE SCHEMA Verkauf
   AUTHORIZATION Verkaufsleiter
   DEFAULT CHARACTER SET ASCII_FULL ;
```

Mit der Anweisung `CREATE DOMAIN` können Sie Einschränkungen für die Werte einer Spalte festlegen oder eine Sortierfolge definieren. Die Einschränkungen, die Sie für eine Domäne definieren, legen fest, welche Objekte die Domäne enthalten kann. Sie können Domänen erstellen, nachdem Sie ein Schema eingerichtet haben. Ein Beispiel:

```
CREATE DOMAIN Lebensalter AS INTEGER
   CHECK (Lebensalter < 20) ;
```

(Achten Sie darauf, dass Sie bei der Abfrage nach dem Alter von Personen daran denken, dass das deutsche Wort »Alter« zu den reservierten SQL-Wörtern zählt und deshalb ein alternativer Ausdruck wie zum Beispiel »Lebensalter« verwendet werden muss.)

Mit der Anweisung `CREATE TABLE` können Sie eine Tabelle erstellen und mit der Anweisung `CREATE VIEW` können Sie eine Sicht definieren. Weiter vorn in diesem Kapitel finden Sie Beispiele für diese beiden Anweisungen. Wenn Sie mit `CREATE TABLE` eine neue Tabelle erstellen, können Sie für deren Spalten gleichzeitig Einschränkungen definieren.

Wenn Sie Einschränkungen nicht für eine einzelne Tabelle, sondern für ein gesamtes Schema definieren wollen, benutzen Sie den Befehl `CREATE ASSERTION`.

Außerdem können Sie mit den Anweisungen `CREATE CHARACTER SET`, `CREATE COLLATION` und `CREATE TRANSLATION` neue Zeichensätze, Sortierreihenfolgen oder Übersetzungstabellen definieren. Sortierreihenfolgen (englisch *Collation Sequences*) legen die Reihenfolge fest, in der Daten sortiert werden. Übersetzungstabellen (englisch *Translation Tables*) steuern die Umwandlung von Strings aus einem Zeichensatz in einen anderen. Es gibt noch einige andere Sachen (siehe Kapitel 2, Tabelle 2.1), die Sie erstellen können, auf die ich hier aber nicht eingehe.

ALTER

Wenn Sie eine Tabelle erstellt haben und feststellen, dass sie nicht mehr Ihren Anforderungen entspricht, können Sie sie mit dem Befehl `ALTER TABLE` ändern, indem Sie Spalten hinzufügen, ändern oder löschen. Außerdem können Sie mit der Anweisung `ALTER` Spalten und Domänen ändern.

DROP

Das Entfernen einer Tabelle aus einem Datenbankschema ist sehr einfach. Benutzen Sie die Anweisung `DROP TABLE <tabellenname>`. Dabei werden nicht nur alle Daten der Tabelle, sondern auch die Metadaten im Datenverzeichnis gelöscht, die die Tabelle definieren. Danach ist es so, als hätte die Tabelle nie existiert. Mit der `DROP`-Anweisung können Sie auch alle anderen Dinge löschen, die mit der `CREATE`-Anweisung erstellt wurden.

`DROP` funktioniert nicht, wenn es die referenzielle Integrität verletzen würde (siehe später in diesem Kapitel).

Data Manipulation Language

Obwohl die DDL der Teil von SQL ist, mit dem Sie Datenbankstrukturen erstellen, ändern oder löschen, hat sie mit den Daten selbst nichts zu tun. Der Umgang mit den Daten ist die Aufgabe der *Data Manipulation Language* (DML, Datenbearbeitungssprache). Einige DML-Anweisungen sehen fast wie gewöhnliche Sätze der englischen Sprache aus und sind leicht zu verstehen. Weil Sie Ihre Daten mit SQL sehr feinkörnig kontrollieren können, können andere DML-Anweisungen extrem komplex sein.

Enthält eine DML-Anweisung mehrere Ausdrücke, Klauseln, Prädikate oder Unterabfragen, ist ihre Funktion mitunter schwer zu verstehen. Haben Sie einige dieser Anweisungen studiert, könnten Sie auf die Idee kommen, sich ein leichteres Aufgabengebiet zu suchen, etwa Gehirnchirurgie oder Quantenmechanik. Solche drastischen Maßnahmen sind jedoch glücklicherweise nicht notwendig, weil Sie komplizierte SQL-Anweisungen verstehen können, indem Sie sie in ihre Grundkomponenten zerlegen und diese Stück für Stück analysieren.

Folgende DML-Anweisungen stehen zur Verfügung: `INSERT`, `UPDATE`, `DELETE` und `SELECT`. Sie können aus einer Reihe verschiedener Bestandteile, einschließlich mehrerer Klauseln, bestehen. Jede Klausel kann Ausdrücke, logische Verknüpfungen, Prädikate, Aggregatfunktionen und Unterabfragen enthalten. Mit Klauseln können Sie die Datensätze in Ihrer Datenbank sehr fein trennen, um genau die Informationen abzurufen, die Sie für einen bestimmten Zweck benötigen. In Kapitel 6 behandle ich die Funktionen der DML-Anweisungen und in den Kapiteln 7 bis 12 gehe ich näher auf die Einzelheiten dieser Anweisungen ein.

Ausdrücke

Ausdrücke (englisch *Value Expressions*) sind die Abschnitte einer Anweisung, die zwei oder mehr Werte miteinander kombinieren. Es gibt mehrere Ausdrücke, die den verschiedenen Datentypen entsprechen:

- ✔ Numerische
- ✔ String
- ✔ Datum und Zeit
- ✔ Intervalle
- ✔ Boolesche
- ✔ Benutzerdefinierte
- ✔ Zeilen
- ✔ Auflistungen

Boolesche, benutzerdefinierte und Zeilen- und Auflistungstypen wurden mit SQL:1999 eingeführt. Einige Implementierungen unterstützen diese Typen noch nicht. Wollen Sie einen dieser Datentypen benutzen, prüfen Sie, ob Ihre Implementierung sie überhaupt unterstützt.

Numerische Ausdrücke

Numerische Werte können mit den Operatoren für die Addition (+), die Subtraktion (–), die Multiplikation (*) und die Division (/) verknüpft werden. Die folgenden Zeilen sind Beispiele für numerische Ausdrücke:

```
12 – 7
15/3 – 4
6 * (8 + 2)
```

Die Werte in diesen Beispielen sind sogenannte *numerische Literale*. Diese Werte könnten auch Spaltennamen, Parameter, Host-Variablen oder Unterabfragen sein – vorausgesetzt, dass diese Spaltennamen, Parameter, Host-Variablen oder Unterabfragen einen numerischen Wert ergeben wie zum Beispiel:

```
Summe + MWSt + Versandkosten
6 * km/Stunde
:monate/12
```

Der Doppelpunkt im letzten Beispiel zeigt an, dass es sich bei dem folgenden Ausdruck (`monate`) entweder um einen Parameter oder um eine Host-Variable handelt.

Stringausdrücke

Stringausdrücke können einen *Verkettungsoperator* (| |) enthalten. Mit ihm werden zwei oder mehr Strings zu einem einzigen verknüpft, wie Tabelle 3.2 zeigt.

Ausdruck	Ergebnis
`'ehrlicher ' || 'Politiker'`	`'ehrlicher Politiker'`
`'Oxy' || 'moron'`	`'Oxymoron'`
`PLZ || ' ' || ORT`	Eine einzelne Zeichenfolge mit der Postleitzahl und dem Ort, durch ein Leerzeichen getrennt

Tabelle 3.2: Beispiele für Stringverkettungen

Einige SQL-Implementierungen benutzen den Verkettungsoperator + statt | | . Lesen Sie die Dokumentation Ihrer Implementierung.

Einige Implementierungen verfügen noch über andere Stringoperatoren als die Verkettung, aber SQL unterstützt solche Operatoren nicht.

Datum/Zeit- und Intervall-Ausdrücke

Datum/Zeit-Ausdrücke haben (Überraschung!) mit Datumsangaben und Zeiten zu tun. Daten vom Typ `DATE`, `TIME`, `TIMESTAMP` und `INTERVAL` können in Datum/Zeit-Ausdrücken verwendet werden. Das Ergebnis eines Datum/Zeit-Ausdrucks ist immer ein Ausdruck für Datums- und Zeitangaben. Sie können ein Intervall von einem Datums- und Zeitwert abziehen oder zu einem Intervall addieren und dabei auch Zeitzoneninformationen festlegen.

Ein Beispiel für einen Datum/Zeit-Ausdruck kann so aussehen:

```
Faelligkeit + INTERVAL '7' DAY
```

Eine Bibliothek könnte einen solchen Ausdruck benutzen, um die Rückgabe eines ausgeliehenen Buches anzumahnen. Ein weiteres Beispiel, das mit einer Zeit-, statt mit einer Datumsangabe arbeitet, ist:

```
TIME '18:55:48' AT LOCAL
```

Das Schlüsselwort `AT LOCAL` zeigt an, dass sich die Zeit auf die lokale Zeitzone bezieht.

Intervall-Ausdrücke berücksichtigen den Unterschied (die verstrichene Zeit) zwischen Datums- und Zeitwerten. Es gibt zwei Intervallarten: *Jahr-Monat* oder *Year-Month* und *Tag-Zeit* oder *Day-Time*. Diese beiden Arten dürfen nicht zusammen in einem Ausdruck verwendet werden.

Angenommen, ein Student gäbe ein ausgeliehenes Buch nach dem Fälligkeitsdatum zurück. Mit einem Intervallausdruck können Sie feststellen, wie viele Tage das Buch überfällig ist, um die Verzugsgebühren zu berechnen:

```
(RueckgabeDatum – FaelligkeitDatum) DAY
```

Weil ein Intervall entweder vom Typ *Jahr-Monat* oder vom Typ *Tag-Zeit* sein kann, müssen Sie angeben, welchen Typ Sie verwenden wollen (in obigem Beispiel also `DAY`).

Boolesche Ausdrücke

Ein boolescher Ausdruck testet den Wahrheitswert eines Prädikats. Ein Beispiel für einen booleschen Ausdruck kann wie folgt aussehen:

```
(Stufe = Oberstufe) IS TRUE
```

Falls dies eine Bedingung für den Abruf von Zeilen aus einer Schülertabelle wäre, würden nur Zeilen mit Schülern zurückgegeben, die in der Oberstufe sind. Um die Zeilen mit Schülern zurückzugeben, die nicht der Oberstufe angehören, können Sie den folgenden Ausdruck verwenden:

```
NOT (Stufe = Oberstufe) IS TRUE
```

Alternativ können Sie auch den folgenden Ausdruck verwenden:

```
(Stufe = Oberstufe) IS FALSE
```

Um alle Zeilen abzurufen, die in der Spalte `Stufe` einen Nullwert enthalten, können Sie diese Anweisung benutzen:

```
(Stufe = Oberstufe) IS UNKNOWN
```

Benutzerdefinierte Ausdrücke

Benutzerdefinierte Ausdrücke werden in Kapitel 2 beschrieben. Mit diesen Ausdrücken können Benutzer eigene Datentypen definieren, anstatt auf die vordefinierten Typen von SQL beschränkt zu sein. Ausdrücke, die Datenelemente solcher benutzerdefinierter Typen bearbeiten, müssen ein Element desselben Typs ergeben.

Zeilenausdrücke

Ein Zeilenausdruck legt einen Zeilenwert fest (was Sie nicht überraschen sollte). Der Zeilenwert kann aus einem Ausdruck oder aus zwei oder mehr durch Kommata voneinander getrennten Ausdrücken bestehen. Ein Beispiel:

```
('Joseph Tykociner', 'Professor Emeritus', 1918)
```

Diese Zeile einer Fakultätstabelle zeigt den Namen, den Status und das Einstellungsjahr eines Fakultätsmitglieds an.

Auflistungsausdrücke

Ein *Auflistungsausdruck* ergibt immer ein Array.

Referenzausdrücke

Ein *Referenzausdruck* ergibt einen Wert, der auf eine andere Datenbankkomponente – beispielsweise eine Tabellenspalte – verweist.

Prädikate

Prädikate sind die Form, in der logische Aussagen in SQL dargestellt werden. Der folgende Satz ist ein Beispiel für eine Aussage:

»Der Schüler ist Mitglied der Oberstufe.«

Angenommen, eine Tabelle mit Daten über Schüler enthielte eine Spalte mit dem Namen `Stufe`. Die Domäne dieser Spalte soll die Werte `Oberstufe`, `Mittelstufe`, `Unterstufe` oder `NULL` umfassen. Dann können Sie mit dem Prädikat `Stufe = 'Oberstufe'` alle Zeilen herausfiltern, für die das Prädikat falsch ist, und nur die erhalten, für die das Prädikat gilt. Ist der Wert des Prädikats in einer Zeile unbekannt (`NULL`), können Sie wählen, ob Sie die Zeile in Ihre Auswahl aufnehmen wollen oder nicht. (Schließlich könnte der Schüler zur Oberstufe gehören.) Die richtige Vorgehensweise hängt von der jeweiligen Situation ab.

`Stufe = 'Oberstufe'` ist ein Beispiel für ein *Vergleichsprädikat*. SQL kennt sechs Vergleichsoperatoren. Einfache Vergleichsprädikate benutzen einen dieser Operatoren. Tabelle 3.3 gibt einen Überblick über die Vergleichsoperatoren und zeigt entsprechende Beispiele.

Operator	Vergleich	Ausdruck
=	gleich	`Stufe = 'Oberstufe'`
<>	ungleich	`Stufe <> 'Oberstufe'`
<	kleiner als	`Stufe < 'Oberstufe'`
>	größer als	`Stufe > 'Oberstufe'`
<=	kleiner als oder gleich	`Stufe <= 'Oberstufe'`
>=	größer als oder gleich	`Stufe >= 'Oberstufe'`

Tabelle 3.3: Vergleichsoperatoren und Vergleichsprädikate

In dem vorhergehenden Beispiel sind nur die ersten beiden Einträge aus Tabelle 3.3 (`Stufe = 'Oberstufe'` und `Stufe <> 'Oberstufe'`) sinnvoll, weil `Unterstufe` größer eingestuft wird als `Oberstufe`, da `Un` in der Standardsortierfolge (aufsteigend alphabetisch) hinter `Ob` kommt. Diese Interpretation ist jedoch wahrscheinlich nicht das, was Sie wollen, da Oberstufe die älteren Schüler bezeichnet.

Logische Verknüpfungen

Mit logischen Verknüpfungen können Sie einfache Prädikate zu komplexeren zusammensetzen. Angenommen, Sie wollten alle Wunderkinder in einer Schülerdatenbank identifizieren. Damit ein Schüler als Wunderkind eingestuft wird, müssen die folgenden beiden Aussagen auf ihn zutreffen:

»Der Schüler ist Mitglied der Oberstufe.«

»Der Schüler ist jünger als 14 Jahre.«

Mit der logischen Verknüpfung `AND` können Sie ein zusammengesetztes Prädikat bilden, das die gewünschten Schülerdatensätze herausfiltert, wie das folgende Beispiel zeigt:

```
Stufe = 'Oberstufe' AND Lebensalter < 14
```

Bei einer `AND`-Verknüpfung müssen beide Teilprädikate wahr sein, damit der zusammengesetzte Ausdruck wahr ist. Bei einer `OR`-Verknüpfung reicht es aus, wenn eines der beiden Teilprädikate wahr ist, damit das zusammengesetzte Prädikat wahr ist. Der dritte logische Operator heißt `NOT`. Genau genommen verknüpft `NOT` nicht zwei Prädikate, sondern kehrt den Wahrheitswert des Prädikats um, dem es zugeordnet wird. Werfen Sie dazu einen Blick auf das folgende Beispiel:

```
NOT (Stufe = 'Oberstufe')
```

Dieser Ausdruck ist dann wahr, wenn `Stufe` ungleich `Oberstufe` ist.

Mengenfunktionen

Manchmal betreffen die Informationen, die Sie aus einer Tabelle abrufen wollen, nicht einzelne Zeilen der Tabelle, sondern ganze Gruppen (Mengen) von Zeilen. SQL definiert zu diesem Zweck fünf *Mengenfunktionen*, auch *Aggregatfunktionen* genannt. Diese Funktionen sind `COUNT`, `MAX`, `MIN`, `SUM` und `AVG`. Jede dieser Funktionen führt eine Aktion aus, die die Daten aus einer Zeilenmenge und nicht aus einer einzelnen Zeile holt.

COUNT

Die Funktion `COUNT` gibt die Anzahl der Zeilen in der angegebenen Tabelle zurück. Um die Anzahl der Wunderkinder in der Schülerdatenbank des letzten Beispiels zu ermitteln, geben Sie Folgendes ein:

```
SELECT COUNT (*)
   FROM Schueler
   WHERE Stufe = 'Oberstufe' AND Lebensalter < 14 ;
```

MAX

Die Funktion `MAX` gibt den größten Wert in der angegebenen Spalte zurück. Wenn Sie beispielsweise den ältesten Schüler Ihrer Schule finden wollen, geben Sie den folgenden Befehl ein:

```
SELECT Vorname, Nachname, Lebensalter
   FROM Schueler
   WHERE Lebensalter = (SELECT MAX(Lebensalter) FROM Schueler);
```

Diese Anweisung ermittelt alle Schüler mit dem Höchstalter. Wenn beispielsweise der älteste Schüler 20 Jahre alt ist, gibt dieser Befehl Vornamen, Nachnamen und das Alter aller 20-jährigen Schüler zurück.

Diese Abfrage arbeitet mit einer Unterabfrage. Die Unterabfrage `SELECT MAX(Lebensalter) FROM Schueler` ist in die Hauptabfrage eingebettet. Sie erfahren in Kapitel 11 mehr über Unterabfragen, die auch als *verschachtelte Abfragen* bezeichnet werden.

MIN

Die Funktion `MIN` arbeitet wie die Funktion `MAX`, gibt aber den kleinsten Wert in der angegebenen Spalte zurück. Wenn Sie beispielsweise den jüngsten Schüler Ihrer Schule finden wollen, geben Sie die folgende Abfrage ein:

```
SELECT Vorname, Nachname, Lebensalter
   FROM Schueler
   WHERE Lebensalter = (SELECT MIN(Lebensalter) FROM Schueler);
```

Diese Abfrage ermittelt alle Schüler, die so jung wie der jüngste sind.

SUM

Die Funktion SUM addiert die Werte der angegebenen Spalte. Der Datentyp der Spaltenwerte muss numerisch sein, und der Wert der Summe muss innerhalb des Wertebereichs dieses Datentyps liegen. Wenn beispielsweise die Spalte vom Typ SMALLINT ist, darf die Summe nicht größer als die Obergrenze des SMALLINT-Datentyps sein. Die Verkaufsdatenbank aus dem Beispiel weiter vorn in diesem Kapitel enthält in der Tabelle Rechnung eine Spalte, in der die Rechnungsbeträge der einzelnen Rechnungen gespeichert werden. Um den Gesamtumsatz aller Verkäufe in der Datenbank zu ermitteln, benutzen Sie die Funktion SUM folgendermaßen:

```
SELECT SUM(Umsatz) FROM Rechnung;
```

AVG

Die Funktion AVG gibt den Durchschnittswert aller Werte der angegebenen Spalte zurück. Wie bei der Funktion SUM muss der Datentyp der Spaltenwerte numerisch sein. Um den Durchschnittsumsatz aller Verkäufe zu ermitteln, benutzen Sie die Funktion AVG folgendermaßen:

```
SELECT AVG(Umsatz) FROM Rechnung
```

Denken Sie daran, dass Nullwerte keinen Zahlenwert haben, weshalb alle Zeilen, die in der Spalte Umsatz einen Nullwert enthalten, bei der Berechnung des Durchschnittsumsatzes ignoriert werden.

Unterabfragen

Wie Sie weiter vorn im Abschnitt *Mengenfunktionen* lesen können, sind *Unterabfragen* Abfragen innerhalb einer Abfrage. An jeder Stelle einer SQL-Anweisung, an der Sie einen Ausdruck einsetzen können, sind Sie auch in der Lage, eine Unterabfrage einzubauen. Unterabfragen sind ein mächtiges Werkzeug, um Daten in einer Tabelle mit Daten in einer anderen Tabelle zu verknüpfen, weil Sie damit die Abfrage einer Tabelle in die Abfrage einer anderen Tabelle *einbetten* (oder *verschachteln*) können. Durch die Verschachtelung von Unterabfragen können Sie Ihr Ergebnis aus Daten von zwei oder mehr Tabellen zusammensetzen. Wenn Sie Unterabfragen richtig einsetzen, lassen sich beliebige Daten aus einer Datenbank abrufen.

DCL (Data Control Language)

Zur Datenkontrollsprache (DCL) gehören vier Anweisungen: COMMIT, ROLLBACK, GRANT und REVOKE. Sie können damit die Datenbank vor zufälligen oder beabsichtigten Schäden bewahren.

Transaktionen

Bei Änderungen ist eine Datenbank besonders anfällig für Schäden, selbst wenn das System nur einen einzigen Benutzer hat. Ein Software- oder Hardware-Ausfall während einer Änderung kann eine Datenbank in einen Zustand irgendwo zwischen dem Ausgangszustand vor der Änderung und dem beabsichtigten Zielzustand nach der Änderung versetzen.

SQL schützt Ihre Datenbank, indem es dafür sorgt, dass Operationen, die die Datenbank verändern können, nur in sogenannten *Transaktionen* erfolgen. Während einer Transaktion speichert SQL alle Operationen, die sich auf die Daten auswirken, in einer Protokolldatei. Wird die Ausführung der Transaktion unterbrochen, bevor sie mit der Anweisung `COMMIT` beendet wird, können Sie das System mit der Anweisung `ROLLBACK` in seinen ursprünglichen Zustand zurückversetzen. Diese Anweisung arbeitet alle Operationen in der Protokolldatei in umgekehrter Reihenfolge ab, wodurch alle Aktionen der Transaktion rückgängig gemacht werden. Nach einem Rollback können Sie die Ursachen des Problems suchen und beseitigen und dann die Transaktion noch einmal ausführen.

Solange Hardware- oder Software-Probleme auftreten können, kann Ihre Datenbank beschädigt werden. Um das Risiko einer Beschädigung zu minimieren, versuchen heutige DBMS, den Zeitraum, in dem die Datenbank verwundbar ist, so kurz wie möglich zu halten, indem sie alle Operationen, die die Datenbank ändern, innerhalb von Transaktionen ausführen und dann auf einmal endgültig übergeben (englisch *to commit*). Moderne Datenbankverwaltungssysteme arbeiten zusätzlich zu Transaktionen mit Protokollen, damit Hardware-, Software- oder Bedienungsprobleme die Daten nicht beschädigen können. Nachdem eine Transaktion übergeben worden ist, sind die Daten geschützt – ausgenommen natürlich, es kommt zu katastrophalen Systemfehlern. Vor der endgültigen Übergabe der Daten können Sie eine unvollständige Transaktion jederzeit rückgängig machen, um zum Ausgangszustand zurückzukehren und die Transaktion erneut auszuführen, nachdem das Problem behoben worden ist.

Bei einem Mehrbenutzersystem kann eine Datenbank auch ohne Hardware- oder Software-Fehler beschädigt werden oder es kann zu fehlerhaften Ergebnissen kommen. Wechselbeziehungen zwischen zwei oder mehr Benutzern, die gleichzeitig auf dieselbe Tabelle zugreifen, können ernsthafte Probleme verursachen. Auch dieses Problem können Sie in SQL lösen, indem Sie Änderungen in Transaktionen kapseln.

Indem Sie alle Operationen, die sich auf die Datenbank auswirken, in Transaktionen zusammenfassen, können Sie alle Aktionen eines Benutzers von denen eines anderen isolieren. Eine solche Isolation ist wichtig, wenn Sie gewährleisten wollen, dass die Daten, die Sie aus der Datenbank erhalten, auch korrekt sind.

Vielleicht fragen Sie sich, wie die Wechselbeziehung zweier Benutzer zu falschen Daten führen kann. Nehmen Sie an, dass der erste Benutzer einen Datensatz einer Datenbanktabelle liest. Kurz danach ändert der zweite Benutzer den Wert eines numerischen Feldes desselben Datensatzes. Danach schreibt der erste Benutzer einen Wert in dieses Feld, der auf dem ursprünglich gelesenen Feldinhalt beruht. Weil der erste Benutzer nichts von der Änderung des zweiten Benutzers weiß, ist der Wert nach der Schreiboperation des ersten Benutzers falsch.

Ein weiteres Problem kann dann entstehen, wenn der erste Benutzer in einen Datensatz schreibt und der zweite Benutzer danach diesen Datensatz liest. Falls der erste Benutzer die Transaktion mit einem Rollback rückgängig macht, arbeitet Benutzer zwei jetzt mit falschen Werten, weil die gelesenen Daten nicht mit den Daten übereinstimmen, die nach dem Rollback des ersten Benutzers in der Datenbank stehen. Das entbehrt sicherlich nicht einer gewissen Komik, deutet aber auf eine grottenschlechte Datenverwaltung hin.

Benutzer und Rechte

Eine weitere große Bedrohung der Datenintegrität stellen die Benutzer selbst dar. Einige Personen dürfen überhaupt keinen Zugriff auf die Daten haben. Andere sollen nur begrenzten Zugang zu einem Teil der Daten erhalten. Wieder andere sollen unbegrenzt mit allen Daten arbeiten können. Deshalb benötigen Sie ein System, um die Benutzer zu klassifizieren und ihnen verschiedene Zugriffsrechte einzuräumen.

Der Ersteller eines Schemas legt fest, wer der Besitzer des Schemas sein soll. Als Besitzer eines Schemas können Sie anderen Benutzern Zugriffsrechte einräumen. Andere Benutzer verfügen nur über die Rechte, die Sie ihnen ausdrücklich gewähren. Sie können gewährte Rechte jederzeit wieder entziehen. Ein Benutzer muss sich authentifizieren, ehe er auf die Daten zugreifen kann, für die Sie ihm ein Zugriffsrecht eingeräumt haben. Die Authentifizierungsmethoden hängen von Ihrer Implementierung ab.

In SQL können Sie folgende Datenbankobjekte schützen:

- ✔ Tabellen
- ✔ Spalten
- ✔ Sichten
- ✔ Domänen
- ✔ Zeichensätze
- ✔ Sortierreihenfolgen
- ✔ Übersetzungstabellen

Zeichensätze, Sortierreihenfolgen und Übersetzungstabellen sind Thema von Kapitel 5.

SQL unterstützt sechs Schutzarten: *Lesen*, *Hinzufügen*, *Ändern*, *Löschen*, *Referenzieren* und *Verwenden* von Datenbanken. Darüber hinaus gibt es Schutzfunktionen, die die Ausführung externer Routinen betreffen.

Rechte werden mit der Anweisung `GRANT` eingeräumt und mit `REVOKE` entzogen. Mit DCL können Sie die Benutzung der Anweisung `SELECT` steuern und damit kontrollieren, wer welche Datenbankobjekte, zum Beispiel Tabellen, Spalten oder Sichten, einsehen darf. Indem Sie kontrollieren, wer die Anweisung `INSERT` ausführen darf, legen Sie fest, wer einer Tabelle neue Datensätze hinzufügen kann. Wenn Sie den Einsatz der Anweisung `UPDATE` auf autorisierte Benutzer beschränken, bestimmen Sie auf diese Weise die Personen, die Tabellenzeilen modifizieren dürfen. Darüber hinaus können Sie die Benutzung der Anweisung `DELETE` einschränken, um festzulegen, wem es gestattet ist, Tabellenzeilen zu löschen.

Wenn eine Tabelle als Fremdschlüssel eine Spalte enthält, die in einer anderen Tabelle der Datenbank ein Primärschlüssel ist, können Sie der ersten Tabelle eine Einschränkung hinzufügen, die auf die zweite Tabelle verweist. (Auf Fremdschlüssel gehe ich in Kapitel 5 ein.) Wenn eine Tabelle eine andere Tabelle referenziert, kann der Besitzer der ersten Tabelle Schlüsse über den Inhalt der zweiten Tabelle ziehen. Als Besitzer der zweiten Tabelle wollen Sie Schnüffeleien dieser Art vielleicht unterbinden. Mit der Anweisung `GRANT REFERENCES` können Sie dies tun. Der folgende Abschnitt beschreibt das Problem der abtrünnigen Referenzen und erläutert, wie Sie es mit dem Befehl `GRANT REFERENCES` verhindern können. Mit dem Befehl `GRANT USAGE` kontrollieren Sie, wer die Inhalte einer Domäne, von Zeichensätzen, Sortierfolgen und Übersetzungstabellen verwenden oder einsehen darf. (Sicherheitsaspekte werden in Kapitel 14 ausführlich behandelt.)

Tabelle 3.4 fasst die SQL-Befehle zusammen, mit denen Sie Rechte einräumen und entziehen können.

Schutzoperation	Befehl
Das Lesen einer Tabelle zulassen	`GRANT SELECT`
Das Lesen einer Tabelle verhindern	`REVOKE SELECT`
Das Einfügen von Zeilen in eine Tabelle zulassen	`GRANT INSERT`
Das Einfügen von Zeilen in eine Tabelle verhindern	`REVOKE INSERT`
Die Änderung von Daten in Tabellenzeilen zulassen	`GRANT UPDATE`
Die Änderung von Daten in Tabellenzeilen verhindern	`REVOKE UPDATE`
Das Löschen von Tabellenzeilen zulassen	`GRANT DELETE`
Das Löschen von Tabellenzeilen verhindern	`REVOKE DELETE`
Das Referenzieren einer Tabelle zulassen	`GRANT REFERENCES`
Das Referenzieren einer Tabelle verhindern	`REVOKE REFERENCES`
Die Benutzung einer Domäne, eines Zeichensatzes, einer Sortierfolge oder einer Übersetzungstabelle zulassen	`GRANT USAGE ON DOMAIN`, `GRANT USAGE ON CHARACTER SET`, `GRANT USAGE ON COLLATION`, `GRANT USAGE ON TRANSLATION`
Die Benutzung einer Domäne, eines Zeichensatzes, einer Sortierfolge oder einer Übersetzungstabelle verhindern	`REVOKE USAGE ON DOMAIN`, `REVOKE USAGE ON CHARACTER SET`, `REVOKE USAGE ON COLLATION`, `REVOKE USAGE ON TRANSLATION`

Tabelle 3.4: Schutzmechanismen

Sie können verschiedenen Benutzern je nach Aufgabe unterschiedliche Zugriffsrechte zuweisen. Die folgenden Anweisungen zeigen einige Beispiele für diese Möglichkeit:

```
GRANT SELECT
   ON Kunde
   TO Verkaufsleiter;
```

Dieses erste Beispiel gewährt einer Person (`Verkaufsleiter`) das Recht, die Kundentabelle (`Kunde`) einzusehen.

Das zweite Beispiel gewährt jedem Benutzer (`PUBLIC`) mit Systemzugang das Recht, die Liste mit den Verkaufspreisen (`Preisliste`) einzusehen.

```
GRANT SELECT
   ON Preisliste
   TO PUBLIC;
```

Das dritte Beispiel gewährt dem Verkaufsleiter das Recht, die Preisliste zu ändern. Mit diesem Recht kann er den Inhalt vorhandener Zeilen ändern, aber keine Zeilen hinzufügen oder löschen.

```
GRANT UPDATE
   ON Preisliste
   TO Verkaufsleiter;
```

Das vierte Beispiel gewährt dem Verkaufsleiter das Recht, Zeilen in die Preisliste einzufügen.

```
GRANT INSERT
   ON Preisliste
   TO Verkaufsleiter;
```

Das letzte Beispiel gewährt dem Verkaufsleiter das Recht, Zeilen aus der Preisliste zu löschen.

```
GRANT DELETE
   ON Preisliste
   TO Verkaufsleiter;
```

Einschränkungen der referenziellen Integrität können Ihre Daten gefährden

Vielleicht meinen Sie, dass Ihre Datenbank gut geschützt ist, wenn Sie das Lesen, Erstellen, Ändern und Löschen einer Tabelle kontrollieren. Für die *meisten* Bedrohungen trifft dies auch zu. Aber ein fähiger Hacker ist häufig in der Lage, Ihr Haus mit indirekten Methoden auszuplündern.

Eine korrekt entworfene relationale Datenbank erfüllt die Regeln der *referenziellen Integrität*, was bedeutet, dass die Daten einer Tabelle der Datenbank konsistent zu den Daten aller anderen Tabellen sind. Um für die referenzielle Integrität zu sorgen, wenden Datenbankdesigner Einschränkungen auf Tabellen an, um die Daten zu begrenzen, die in den Tabellen gespeichert werden können. Wenn Sie die referenzielle Integrität einer Datenbank nur mit Einschränkungen schützen, könnte ein Benutzer eine neue Tabelle erstellen, die eine Spalte einer vertraulichen Tabelle als Fremdschlüssel verwendet. Diese Spalte könnte dann als Verbindungsglied dienen, über das vertrauliche Informationen gestohlen werden. Ups.

Angenommen, Sie seien ein berühmter Aktienanalytiker an der Wall Street. Viele Leute glauben an die Richtigkeit Ihrer Aktienauswahl, weshalb viele Ihrer Kunden die Aktien kaufen, die Sie empfehlen, und der Wert dieser Aktien steigt. Sie speichern Ihre Analysen in einer Datenbank, die eine Tabelle mit dem Namen `VierSterne` enthält. Die Spitzenempfehlungen für Ihr nächstes Rundschreiben sind in dieser Tabelle gespeichert. Natürlich beschränken Sie den Zugriff auf `VierSterne`, damit Ihre Empfehlungen nicht in die Öffentlichkeit dringen, ehe Ihre zahlenden Abonnenten Ihr Rundschreiben erhalten haben.

Dennoch sind Sie verwundbar, wenn ein anderer Benutzer eine neue Tabelle anlegen kann, die das Feld mit den Namen der Aktien in der Tabelle `VierSterne` als Fremdschlüssel benutzt. Das folgende Beispiel zeigt die entsprechende Anweisung:

```
CREATE TABLE AktienEmpfehlung (
   Aktien CHARACTER (30) REFERENCES VierSterne
   );
```

Der Hacker kann jetzt versuchen, die Namen aller Aktien, die an der New Yorker und Frankfurter Börse gehandelt werden, in diese neue Tabelle einzufügen. Falls ein `INSERT` erfolgreich ist, weiß er, dass die betreffende Aktie in Ihrer vertraulichen Tabelle enthalten ist. Es wird nicht lange dauern, bis der Hacker die gesamte Liste der Aktien in Ihrer Tabelle ermittelt hat.

Sie können sich gegen Einbruchsversuche dieser Art schützen, indem Sie sorgfältig vermeiden, Anweisungen wie die folgende abzusetzen:

```
GRANT REFERENCES (Aktien)
   ON VierSterne
   TO Hacker;
```

Sicher übertreibe ich hier. Denn Sie würden niemals einer nicht vertrauenswürdigen Person Zugriff auf eine Tabelle mit kritischen Daten gewähren, nicht wahr? Jedenfalls nicht, falls Sie genau wissen, was Sie tun. Doch heutige Hacker sind nicht nur technisch versiert. Sie sind auch Experten des *Social Engineering*, der Kunst, Menschen zu Handlungen zu verleiten, die sie normalerweise nicht ausführen wollten. Seien Sie äußerst wachsam, wenn ein geschmeidiger Gesprächspartner das Gespräch auf Dinge lenkt, die mit vertraulichen Informationen zu tun haben.

Vermeiden Sie, Personen Rechte einzuräumen, die Ihr Vertrauen missbrauchen könnten. Ehrlichkeit und Loyalität sind natürlich keinem Menschen auf die Stirn geschrieben. Wenn Sie aber jemandem Ihr neues Auto für eine lange Reise nicht leihen würden, sollten Sie ihm vielleicht auch kein `REFERENCES`-Recht für eine wichtige Tabelle gewähren.

Das obige Beispiel zeigt, warum es sich lohnt, das `REFERENCES`-Recht sorgfältig zu kontrollieren. Es gibt noch zwei weitere Gründe dafür, mit diesem Recht vorsichtig umzugehen:

- ✔ Wenn die andere Person in der Tabelle `AktienEmpfehlung` in einer `RESTRICT`-Option eine Einschränkung angibt, können Sie aus Ihrer Tabelle keine Zeilen mehr löschen, die in der anderen Tabelle referenziert werden, weil Sie dadurch gegen eine referenzielle Einschränkung verstoßen würden.

✔ Wenn Sie Ihre Tabelle mit dem Befehl DROP löschen wollen, müssen Sie die andere Person dazu veranlassen, die Einschränkung aufzuheben (oder die – fremde – Tabelle zu löschen).

Wenn Sie jemand anderem das Recht einräumen, Integritätseinschränkungen für Ihre Tabelle zu definieren, öffnen Sie nicht nur Sicherheitslücken, sondern machen den anderen Benutzer auch zu einem Störfaktor Ihrer eigenen Arbeit.

Die Verantwortung für die Sicherheit delegieren

Wenn Sie die Sicherheit Ihres Systems gewährleisten wollen, müssen Sie strikt kontrollieren, wem Sie welche Rechte einräumen. Denken Sie dabei aber daran, dass Personen, die ihre Arbeit nicht erledigen können, weil sie nicht über die entsprechenden Zugriffsrechte verfügen, Ihnen ständig auf die Nerven gehen. Deshalb sollten Sie einen Teil Ihrer Verantwortung für die Sicherheit der Datenbank delegieren. Zu diesem Zweck gibt es in SQL die Klausel WITH GRANT OPTION. Betrachten Sie das folgende Beispiel:

```
GRANT UPDATE
   ON Preisliste
   TO Verkaufsleiter WITH GRANT OPTION
```

Diese Anweisung ähnelt der Anweisung im vorherigen GRANT UPDATE-Beispiel, bei dem dem Verkaufsleiter das Recht gewährt wird, die Preisliste zu ändern. Zusätzlich gibt sie ihm auch das Recht, seinerseits anderen Benutzern das Änderungsrecht einzuräumen. Wenn Sie diese Form der GRANT-Anweisung benutzen, müssen Sie nicht nur demjenigen trauen, dem Sie das Recht einräumen, sondern Sie müssen sich auch darauf verlassen, dass er das Recht nur an vertrauenswürdige andere Benutzer weitergibt.

Das ultimative Vertrauen zeigen und gleichzeitig die ultimative Angreifbarkeit ermöglichen, erreichen Sie mit einer Anweisung wie dieser:

```
GRANT ALL PRIVILEGES
   ON VierSterne
   TO Jutta_Schmidt WITH GRANT OPTION;
```

Gehen Sie mit Anweisungen dieser Art äußerst sorgfältig um. Wenn Sie jemandem alle Rechte zusammen mit der GRANT OPTION einräumen, sind Sie äußerst verwundbar. Benedict Arnold war einer der Generäle, denen George Washington während des amerikanischen Unabhängigkeitskriegs am meisten vertraute. Er lief zu den Briten über und wurde so zu dem meistgeschmähten Verräter in der Geschichte Amerikas. Sie wollen sicher nicht, dass Ihnen Ähnliches widerfährt.

Teil II

Datenbanken mit SQL erstellen

In diesem Teil ...

- ✔ Einfache Strukturen erstellen
- ✔ Beziehungen zwischen Tabellen definieren

Eine einfache Datenbankstruktur erstellen und verwalten

4

In diesem Kapitel

- Eine Datenbanktabelle mit RAD erstellen, ändern und löschen
- Eine Datenbanktabelle mit SQL erstellen, ändern und löschen
- Eine Datenbank auf ein anderes Datenbankverwaltungssystem übertragen

Der technische Wandel auf dem Gebiet der Computer vollzieht sich so schnell, dass es schwer ist, mitzuhalten. Zunächst wurden hoch entwickelte Programmiersprachen (Sprachen der sogenannten *dritten Generation*), zum Beispiel FORTRAN, COBOL, BASIC, Pascal und C, verwendet, um große Datenbanken zu erstellen und damit zu arbeiten. Später wurden Sprachen speziell für das Arbeiten mit Datenbanken entwickelt, wie zum Beispiel dBASE, Paradox und R:BASE (Sprachen der dreieinhalbten Generation; nur ein Scherz). Der nächste Schritt war die Entstehung von Entwicklungsumgebungen, wie etwa Access, PowerBuilder und C++-Builder (Sprachen der vierten Generation, auch *4GLs* genannt). Heute sind wir über die nummerierten Generationen hinaus und verwenden sogenannte *RAD-Werkzeuge (Rapid Application Development*, deutsch *schnelle Anwendungsentwicklung)* und IDEs (Integrated Development Environment, deutsch Integrierte Entwicklungsumgebung) wie zum Beispiel Eclipse oder Visual Studio .NET, die mit verschiedenen Sprachen (wie etwa C, C++, C#, Python, Java, Visual Basic oder PHP) eingesetzt werden können, um Anwendungskomponenten zu Produktionsanwendungen zusammenzusetzen.

Da SQL keine vollständige Sprache ist, lässt es sich nicht sauber in eine der genannten Kategorien der Sprachgenerationen einordnen. SQL ist keine IDE; und es enthält Anweisungen wie die Sprachen der dritten Generation, ist aber im Wesentlichen nicht prozedural wie die Sprachen der vierten Generation. Aber im Grunde ist es unwichtig, wie Sie SQL klassifizieren. Sie können es zusammen mit IDEs und Sprachen der dritten und vierten Generation verwenden. Sie können den SQL-Code selbst schreiben oder Objekte auf dem Bildschirm verschieben und der Entwicklungsumgebung die Aufgabe überlassen, für Sie den dazu passenden Code zu erzeugen. Die Anweisungen, die an die entfernte Datenbank gesendet werden, sind auf jeden Fall reine SQL-Befehle.

Ich zeige Ihnen in diesem Kapitel, wie Sie eine einfache Tabelle erst mit einem RAD-Werkzeug und dann mit SQL erstellen, ändern und löschen können.

Eine einfache Datenbank mit einem RAD-Werkzeug erstellen

Menschen verwalten wichtige Daten mit Datenbanken. Manchmal sind gesuchte Daten einfach, manchmal kompliziert. Ein gutes Datenbankverwaltungssystem erfüllt beide Anforderungen. Einige Systeme stellen Ihnen SQL zur Verfügung. Andere, wie die sogenannten *RAD-Werkzeuge*, bieten eine objektorientierte grafische Umgebung an. Einige Datenbankverwaltungssysteme unterstützen beide Ansätze. In den folgenden Abschnitten zeige ich Ihnen, wie Sie eine einfache, aus einer Tabelle bestehende Datenbank mit einem grafischen Datenbankentwurfswerkzeug erstellen. In meinen Beispielen verwende ich Microsoft Access, aber die Vorgehensweise ist bei anderen Windows-basierten Entwicklungsumgebungen ähnlich.

Entscheiden, was in die Datenbank gehört

Der erste Schritt besteht darin, festzulegen, welche Daten Sie speichern wollen. Angenommen, Sie hätten gerade 15 Millionen im Lotto gewonnen. (Lassen Sie uns einfach einmal träumen. In Wirklichkeit ist das genauso wahrscheinlich wie der Fall, dass Ihr Auto von einem Meteoriten zerquetscht wird.) Plötzlich tauchen aus allen Ecken Leute auf, von denen Sie seit Jahren nichts mehr gehört haben, und alte Freunde, die schon lange in Vergessenheit geraten waren. Einige bieten Ihnen todsichere Geschäftstipps an. Andere suchen Ihre Unterstützung für einen wohltätigen Zweck. Da Sie ein verantwortungsbewusster Mitbürger sind, wollen Sie Ihren neuen Wohlstand so gut wie möglich verwalten. Sie erkennen, dass einige Geschäftsgelegenheiten wahrscheinlich nicht so gut und einige Wohltätigkeitszwecke nicht so unterstützungswürdig sind. Um den Überblick zu behalten und allen Vorschlägen gerecht werden zu können, wollen Sie alle Alternativen in einer Datenbank ablegen.

Sie haben deshalb vor, folgende Daten zu speichern:

- ✔ Vorname
- ✔ Nachname
- ✔ Anschrift
- ✔ Postleitzahl
- ✔ Ort
- ✔ Bundesland
- ✔ Telefon
- ✔ Beziehung zu der Person
- ✔ Vorschlag
- ✔ Geschäft oder Wohltätigkeit

Sie entschließen sich dazu, alle Daten in einer einzigen Datenbanktabelle abzulegen. Sie brauchen nichts Kompliziertes.

Eine Datenbanktabelle erstellen

Wenn Sie Ihre Access-2013-Entwicklungsumgebung zum ersten Mal starten, werden Sie von einem Fenster ähnlich dem aus Abbildung 4.1 begrüßt. Von dort aus können Sie eine Datenbanktabelle auf mehreren Wegen erstellen.

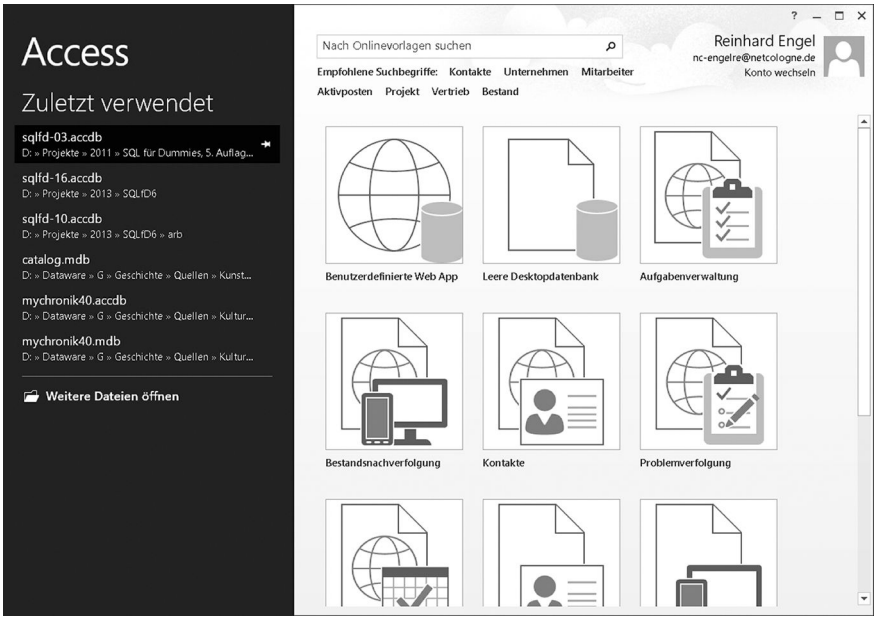


Abbildung 4.1: Der Startbildschirm von Access 2013

Eine Tabelle in der Datenblattansicht erstellen

Access 2013 wird standardmäßig in der Datenblattansicht geöffnet. Um eine Access-Datenbank in der Datenblattansicht zu erstellen, doppelklicken Sie auf die Vorlage Leere Datenbank.

Die neue Datenbank wird erstellt und eine neue Tabelle wird in der Datenblattansicht geöffnet (siehe Abbildung 4.2).

Sie können der Tabelle später einen aussagekräftigeren Namen zuweisen. Access gibt einer neuen Datenbank standardmäßig den Namen `Database1` (oder `Database31`, wenn Sie bereits 30 Datenbanken erstellt und deren Namen nicht geändert haben). Es ist besser, der Datenbank von vornherein einen aussagekräftigen Namen zu geben, um Verwirrung zu vermeiden.

Das ist die Methode, mit der eine Datenbank von Grund auf erstellt wird. Aber es gibt noch mehrere andere Methoden, um eine Access-Datenbanktabelle zu erstellen. Die nächste Methode verwendet die Entwurfsansicht.

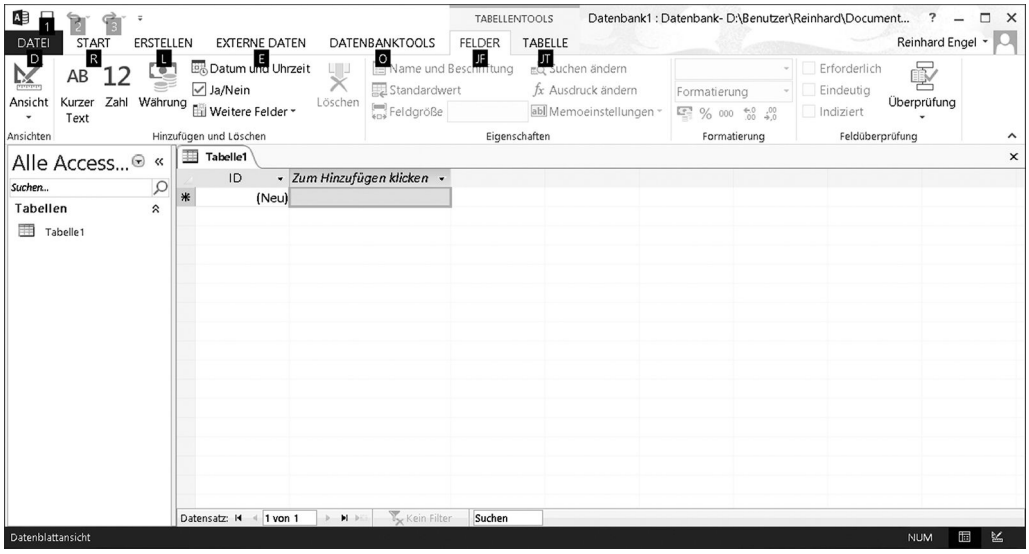


Abbildung 4.2: Die Datenblattansicht in der Access-Entwicklungsumgebung

Eine Tabelle in der Entwurfsansicht erstellen

Eine Datentabelle in der Datenblattansicht (siehe Abbildung 4.2) zu erstellen, ist ziemlich einfach. Sie geben einfach Daten ein. Dieser Ansatz ist jedoch fehleranfällig, weil man leicht Details übersehen kann. Die Entwurfsansicht ist für diesen Zweck besser geeignet. Führen Sie folgende Schritte aus:

1. **Gehen Sie von der standardmäßigen Datenblattansicht aus und klicken Sie auf das Symbol Ansicht in der oberen linken Ecke des Fensters. Wählen Sie dann in dem Dropdown-Menü den Eintrag Entwurfsansicht aus.**

   Wenn Sie auf Entwurfsansicht klicken, wird ein Dialogfeld geöffnet, das Sie auffordert, einen Tabellennamen einzugeben.

2. **Geben Sie `Angebote` ein und klicken Sie dann auf Weiter.**

   Die Entwurfsansicht wird angezeigt (siehe Abbildung 4.3).

   Das Fenster ist in funktionale Bereiche unterteilt; insbesondere zwei dienen der Erstellung von Datenbanktabellen:

   - *Die Registerkarten der Entwurfsansicht:* Ein Menü am oberen Rand des Fensters bietet die Optionen oder Registerkarten Start, Erstellen, Externe Daten, Datenbanktools und Entwurf an. Die in der Entwurfsansicht verfügbaren Tools werden direkt unter dem Menü als Symbole angezeigt. In Abbildung 4.3 sind die Registerkarte Entwurf und das Symbol Primärschlüssel hervorgehoben.

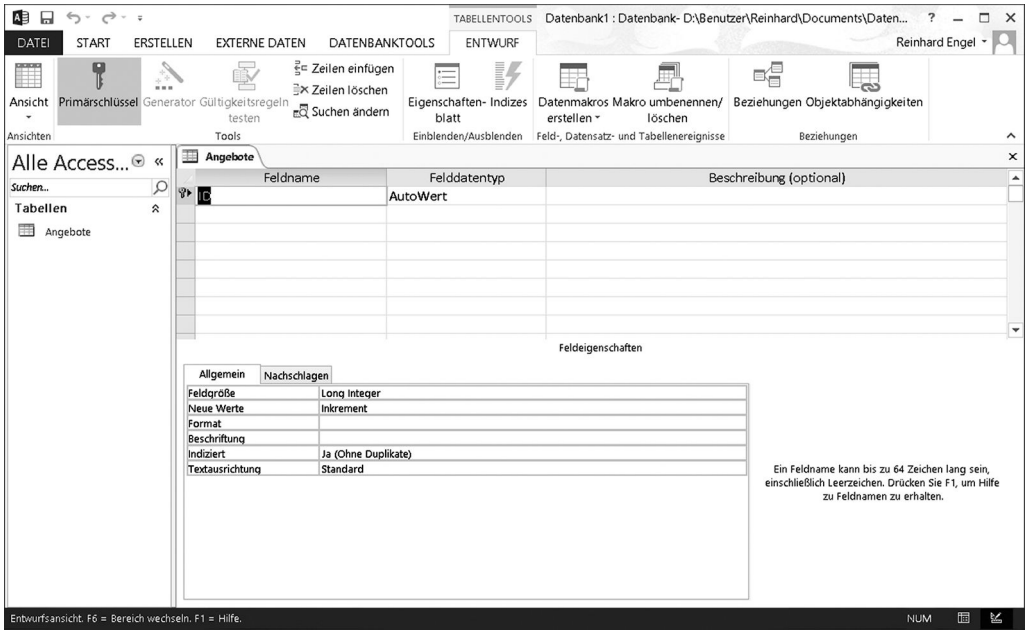


Abbildung 4.3: Anfangszustand der Entwurfsansicht

- *Der Bereich* FELDEIGENSCHAFTEN *in der Mitte:* Hier können Sie die Felder der Tabelle definieren. Der Cursor steht blinkend in der FELDNAME-Spalte der ersten Zeile. Access schlägt vor, hier den Primärschlüssel zu definieren, ihn `ID` zu nennen und ihm den Datentyp `AutoWert` zuzuweisen.

Der Datentyp `AutoWert` ist ein Access-Datentyp, kein standardmäßiger SQL-Typ. Er erhöht eine Ganzzahl in dem Feld automatisch jeweils um eins, wenn Sie eine neue Zeile in die Tabelle einfügen. Dieser Datentyp garantiert, dass das Feld, das Sie als Primärschlüssel verwenden, nicht dupliziert wird und deshalb eindeutig bleibt.

3. Ändern Sie im Bereich FELDEIGENSCHAFTEN den Feldnamen des Primärschlüssels von `ID` in `AngebotNr`.

Der vorgeschlagene Feldname für den Primärschlüssel, `ID`, ist einfach nicht besonders aussagekräftig. Wenn Sie sich angewöhnen, ihn in einen bedeutungsträchtigeren Namen umzubenennen (und/oder zusätzliche Informationen in der BESCHREIBUNG-Spalte angeben), können Sie leichter den Überblick über die Felder in Ihrer Datenbank behalten.

Abbildung 4.4 zeigt den Stand des Tabellenentwurfs zu diesem Zeitpunkt.

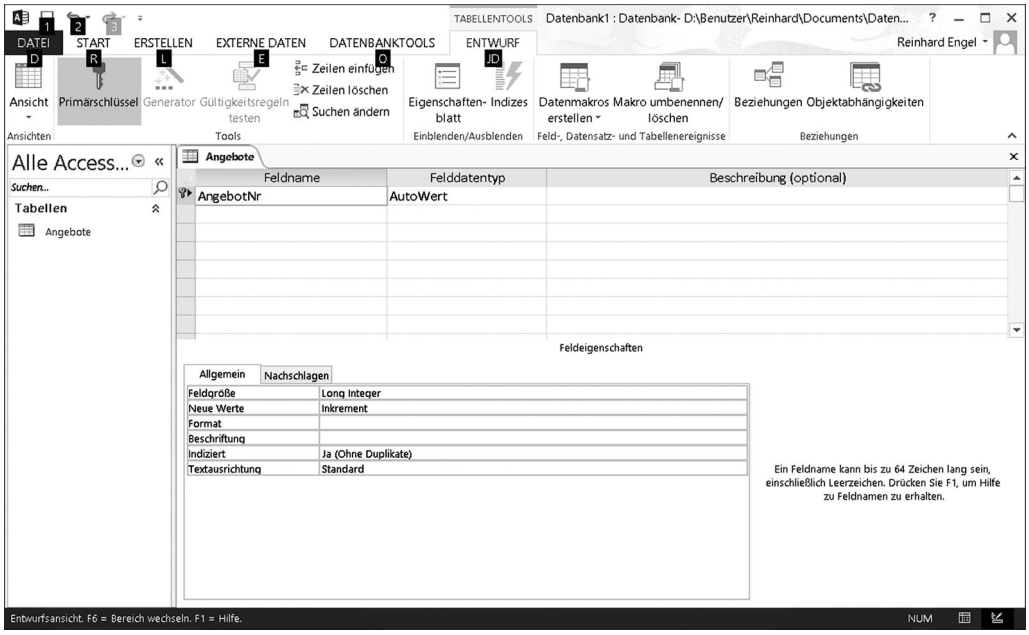


Abbildung 4.4: Einen aussagekräftigen Feldnamen für den Primärschlüssel verwenden

4. Prüfen Sie im Bereich FELDEIGENSCHAFTEN die Voreinstellungen, die Access automatisch für das `AngebotNr`-Feld vorgenommen hat.

Abbildung 4.4 zeigt die folgenden Voreinstellungen:

- Das Feld FELDGRÖSSE ist auf `Long Integer` gesetzt.
- Das Feld NEUE WERTE ist auf `Inkrement` gesetzt.
- Das Feld INDIZIERT ist auf `Ja (Ohne Duplikate)` gesetzt.
- Das Feld TEXTAUSRICHTUNG ist auf `Standard` gesetzt.

Oft sind die Voreinstellungen von Access für die anstehende Aufgabe geeignet. Ist dies nicht der Fall, können Sie die Werte durch eigene Eingaben ersetzen.

5. Spezifizieren Sie die restlichen Felder für diese Tabelle.

Abbildung 4.5 zeigt die Entwurfsansicht nach Eingabe des `Vorname`-Feldes.

Der Datentyp für `Vorname` ist `Text`, nicht `AutoWert`; deshalb gelten für dieses Feld andere Feld-Properties. Hier hat Access dem Feld die Standardgröße 255 Zeichen zugewiesen. Ich kenne nicht allzu viele Menschen, deren Vornamen so lang sind. Warum Speicherplatz verschwenden? Im nächsten Schritt wird dieser Standardwert geändert.

Abbildung 4.5: Das Fenster zur Erstellung der Tabelle nach der Definition des `Vorname`-Feldes

Hier nimmt Access standardmäßig an, dass `Vorname` kein Pflichtfeld sei. Sie könnten also einen Datensatz in die `Lotto`-Tabelle eingeben und das `Vorname`-Feld leer lassen. Damit könnten Sie also auch Personen erfassen, die nur einen Namen haben, wie etwa *Cher* oder *Bono*.

6. Ändern Sie die Feldgröße für `Vorname` in `15`.

Warum dies sinnvoll ist, wird in dem Einschub *Beim Tabellendesign vorausdenken* beschrieben.

7. Damit Sie einen Datensatz aus der Angebote-Tabelle per `Nachname` abrufen können (was Sie *wahrscheinlich* tun wollen), ändern Sie die `Indiziert`-Eigenschaft für `Nachname` in `Ja (Duplikate möglich)` (siehe Abbildung 4.6).

Die Abbildung zeigt einige Änderungen, die ich im Bereich FELDEIGENSCHAFTEN vorgenommen habe.

- Ich habe die maximale FELDGRÖSSE von `255` auf `20` reduziert, um Speicherplatz zu sparen.
- Ich habe EINGABE ERFORDERLICH auf `Ja`, LEERE ZEICHENFOLGE auf `Nein` und INDIZIERT auf `Ja (Duplikate möglich)` gesetzt. Jedes Angebot soll den Nachnamen der dafür verantwortlichen Person enthalten. Ein Name der Länge null ist nicht erlaubt; und das Feld `Nachname` wird indiziert.

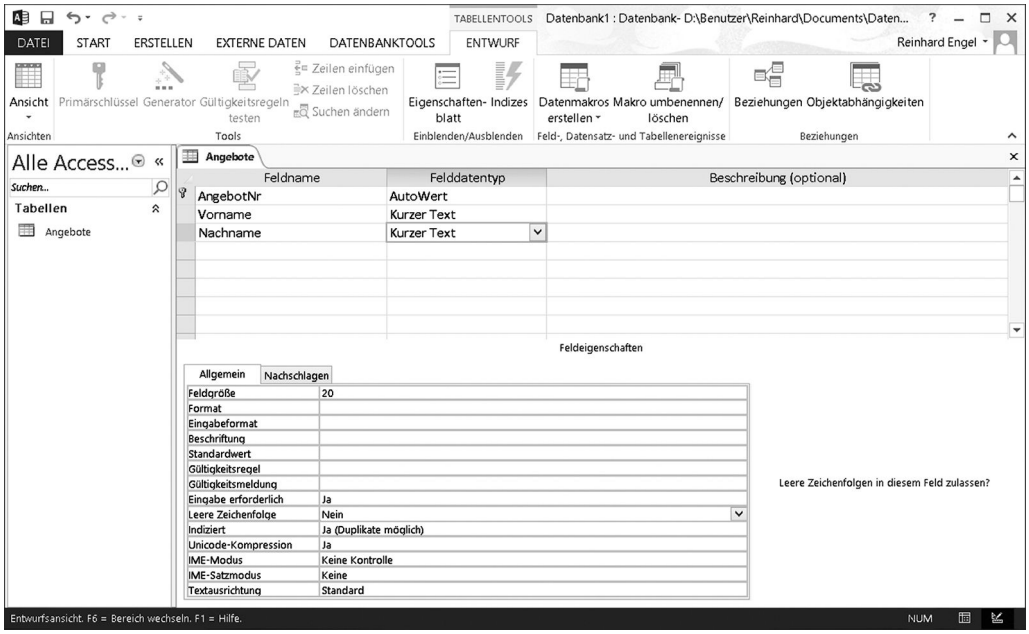


Abbildung 4.6: Das Fenster zur Erstellung der Tabelle nach der Definition des `Nachname`*-Feldes*

- Ich lasse Duplikate zu; zwei oder mehr Anbieter dürfen denselben Nachnamen haben. Dies ist bei der `Lotto`-Tabelle praktisch gegeben; denn ich erwarte Angebote von allen meinen drei Brüdern sowie von meinen Söhnen und meiner unverheirateten Tochter, von meinen Nichten und Neffen ganz zu schweigen.

Die Option `Ja (Ohne Duplikate)`, die ich *nicht* ausgewählt habe, wäre für ein Feld geeignet, das zugleich Primärschlüssel einer Tabelle wäre. Der Primärschlüssel einer Tabelle darf nie Duplikate enthalten.

8. Geben Sie die restlichen Felder ein und ändern Sie die standardmäßige Feldgröße in allen Fällen in einen geeigneten Wert.

Abbildung 4.7 zeigt das Ergebnis.

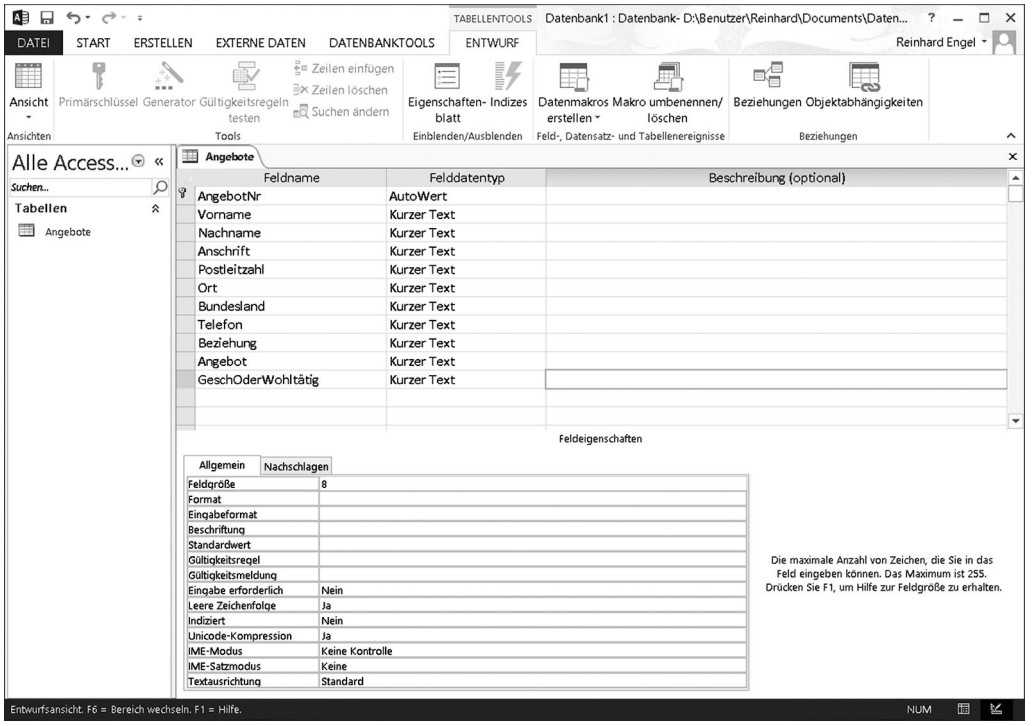


Abbildung 4.7: Das Fenster zur Erstellung der Tabelle nach der Definition aller Felder

Laut Abbildung 4.7 ist das Feld `GeschOderWohltätig` nicht indiziert. Es ist nicht sinnvoll, ein Feld zu indizieren, das nur zwei mögliche Einträge haben kann. Durch die Indizierung wird die Auswahl nicht nennenswert eingegrenzt.

Access verwendet die Bezeichnung *Feld* anstelle von *Attribut* oder *Spalte*. Die ursprünglichen Systeme des Programms waren nicht relational und verwendeten die Termini *Datei*, *Feld* und *Datensatz*, die üblicherweise für flache Dateisysteme benutzt werden.

9. Speichern Sie Ihre Tabelle, indem Sie auf das Disketten-Symbol in der oberen linken Ecke klicken.

Access speichert die Tabelle unter dem Namen, unter dem Sie sie angelegt haben, `Angebote`.

Sie sollten auch künftige Entwicklungen im Auge behalten, wenn Sie Ihre Datenbank entwickeln. So ist es etwa empfehlenswert, Ihre Arbeit bei der Entwicklung häufig zu speichern; klicken Sie einfach gelegentlich auf das Disketten-Symbol in der linken oberen Ecke. Dadurch können Sie sich viele mühsame Wiederholungen ersparen, sollte der Strom ausfallen oder das System aus einem anderen Grund abstürzen. Um künftige Administratoren und Anwender nicht zu verwirren, sollten Sie möglichst nicht denselben Namen für eine Tabelle und die Datenbank verwenden, in der sie enthalten ist.

Beim Tabellendesign vorausdenken

Wenn die Größe des Feldes `Vorname` von 255 auf 15 reduziert wird, werden *bei jedem Datensatz in der Tabelle* 240 Bytes gespart, wenn Sie ASCII(UTF-8)-Zeichen verwenden, 480 Bytes, wenn Sie UTF-16-Zeichen, oder 960 Bytes, wenn Sie UTF-32-Zeichen verwenden. Es addiert sich. Wenn Sie schon dabei sind, sollten Sie sich auch andere Voreinstellungen für einige andere Feldeigenschaften anschauen und überlegen, wie Sie sie verwenden könnten, wenn die Datenbank größer wird. Einige dieser Felder erfordern Ihre Aufmerksamkeit sofort, damit sie effizienter werden (`Vorname` ist ein praktisches Beispiel), andere werden nur in relativ seltenen Fällen benötigt.

Vielleicht ist Ihnen eine weitere Feldeigenschaft aufgefallen, die häufig auftaucht: die Indiziert-Eigenschaft. Wenn Sie nicht damit rechnen, einen Datensatz anhand eines bestimmten Feldes abzurufen, sollten Sie keine Verarbeitungskapazität damit verschwenden, das Feld zu indizieren. Denken Sie jedoch daran, dass Sie das Abrufen von Daten bei einer großen Tabelle mit zahlreichen Feldern erheblich beschleunigen können, indem Sie das Feld indizieren, mit dem Sie den abzurufenden Datensatz identifizieren. Der Teufel – oder in diesem Fall die potenzielle Performance-Steigerung – steckt in den Details beim Design Ihrer Datenbanktabellen.

Die Struktur einer Tabelle ändern

Oft müssen Datenbanktabellen nachträglich verändert werden. Wenn Sie für eine andere Person oder Firma arbeiten, kann es passieren, dass Ihr Kunde nach dem ersten Datenbankentwurf zu Ihnen kommt, um Ihnen zu sagen, dass er noch das eine oder andere zusätzliche Datenfeld benötigt. Das bedeutet, dass Sie wieder zum Entwurfsmodus zurückkehren müssen.

Wenn Sie die Datenbank für Ihre eigenen Zwecke erstellen, werden Mängel an der Struktur unweigerlich erst deutlich, *nachdem* sie fertiggestellt worden ist. So erhalten Sie möglicherweise auch Angebote aus dem Ausland und müssen deshalb eine Spalte für das Land einfügen. Oder Sie haben eine ältere Datenbank, die keine E-Mail-Adressen speichern kann. Ich zeige Ihnen, wie Sie Access benutzen, um eine Tabelle zu modifizieren. Andere RAD-Werkzeuge verfügen über vergleichbare Fähigkeiten und Funktionen.

Falls Sie einmal eine Tabelle aktualisieren müssen, sollten Sie sich auf jeden Fall die Zeit nehmen und alle Felder überdenken. Möglicherweise benötigen Sie eindeutige Angebotsnummern, um die Angebote von Personen zu unterscheiden, die denselben Namen haben. Dabei können Sie auch gleich ein zweites Adressenfeld für Leute mit komplexen Adressen sowie ein Länderfeld für Angebote aus dem Ausland einfügen.

Obwohl es ziemlich leicht ist, Datenbanktabellen zu aktualisieren, sollten Sie es möglichst vermeiden. Anwendungen, die von der alten Datenbankstruktur abhängen, werden wahrscheinlich nicht mehr funktionieren und müssen angepasst werden. Dies kann sehr aufwendig sein, wenn Sie viele solche Anwendungen haben. Versuchen Sie, mögliche künftige Erweiterungen abzuschätzen, und treffen Sie entsprechende Vorkehrungen. Einen gewissen geringen Overhead in der Datenbank mitzuschleppen, ist normalerweise besser, als zahlreiche Anwendungen zu aktualisieren, die vor mehreren Jahren geschrieben wurden. Das Wissen, wie sie funktionieren, ist wahrscheinlich längst verschwunden; und möglicherweise können sie irreparabel sein.

Führen Sie die folgenden Schritte aus, um eine neue Zeile einzufügen und die anderen gewünschten Änderungen vorzunehmen:

1. **Klicken Sie im Tabellenerstellungsfenster auf das kleine farbige Rechteck am Anfang der Postleitzahl-Zeile; klicken Sie dann auf das Symbol Zeilen einfügen.**

   Über der Position des Cursors wird eine leere Zeile eingefügt und die anderen Zeilen werden nach unten verschoben (siehe Abbildung 4.8).

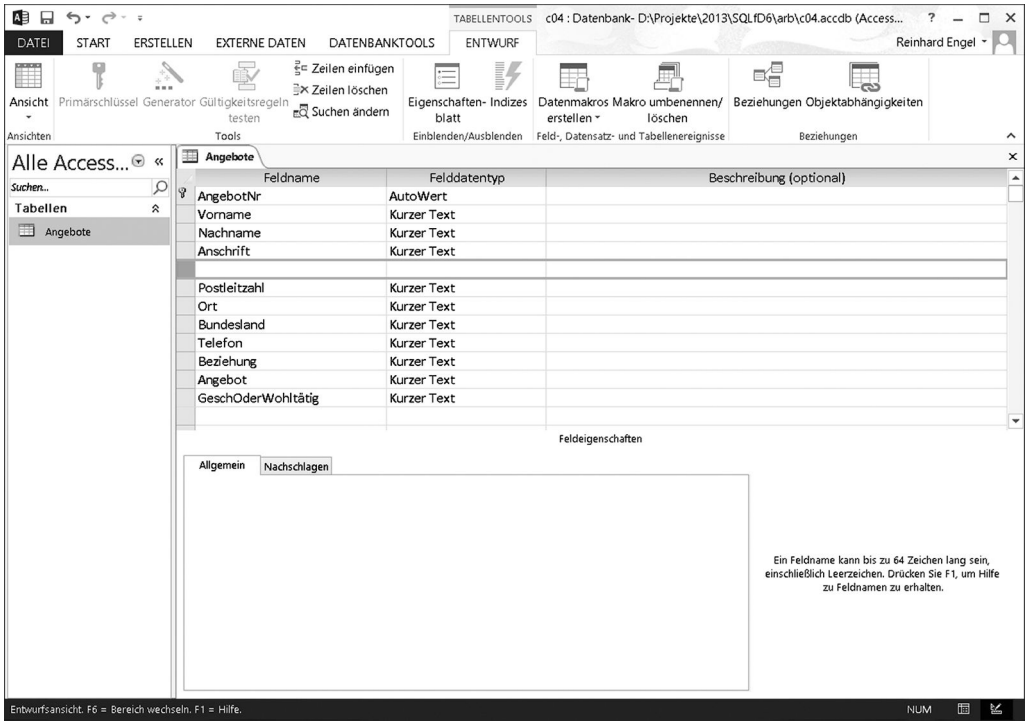


Abbildung 4.8: Eine leere Zeile wird in die Tabelle `Angebote` *eingefügt.*

2. **Geben Sie die Felder ein, die Sie in die Tabelle einfügen wollen.**

   Geben Sie so das Feld `Anschrift2` über dem Feld `Postleitzahl` ein.

3. **Nachdem Sie Ihre Änderungen beendet haben, speichern Sie die Tabelle, bevor Sie sie schließen.**

   Das Ergebnis sollte ähnlich wie Abbildung 4.9 aussehen.

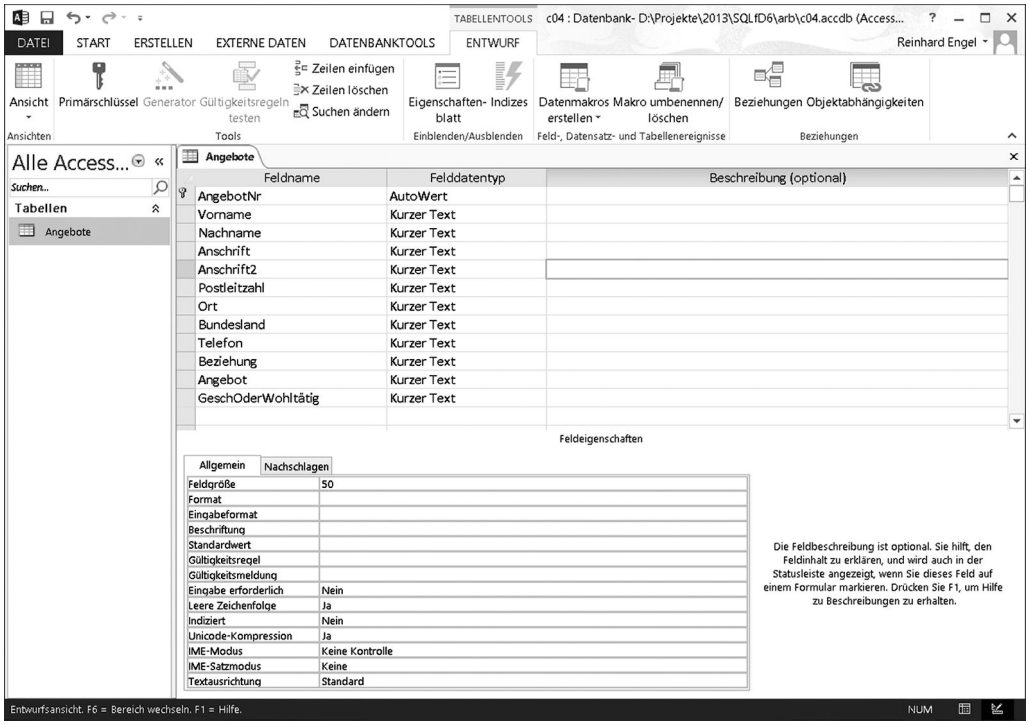


Abbildung 4.9: Die überarbeitete Tabellendefinition

Einen Index definieren

Sie benötigen in einer Datenbank eine schnelle Methode, um auf die Datensätze zugreifen zu können, die Sie interessieren. (Dies ist ganz besonders dann notwendig, wenn Sie in einer Lotterie gewinnen – die Anzahl an Investitions- und gemeinnützigen Vorschlägen geht schnell in die Tausende.) Angenommen, Sie möchten alle Vorschläge sehen, die von Personen stammen, die behaupten, Ihr Bruder zu sein. Wenn wir weiterhin annehmen, dass keiner Ihrer Brüder seinen Nachnamen geändert hat, weil er Künstler oder Schriftsteller geworden ist, können Sie diese Angebote leicht herausfiltern, indem Sie den Inhalt des Feldes `Nachname` wie folgt abfragen:

```
SELECT * FROM Angebote
   WHERE Nachname = 'Marx' ;
```

Diese Vorgehensweise funktioniert jedoch nicht mit den Vorschlägen, die von Ihren Halbbrüdern oder Schwägern stammen. Dafür müssen Sie ein anderes Feld heranziehen:

```
SELECT * FROM Angebote
   WHERE Beziehung = 'Schwager'
      ODER
      Beziehung = 'Halbbruder';
```

SQL durchsucht die Tabelle zeilenweise nach Einträgen, die die `WHERE`-Klausel erfüllen. Wenn die Tabelle `Angebote` sehr groß ist (einige Zehntausend Datensätze), müssen Sie sich auf Wartezeit einrichten. Sie können die Abfrage beschleunigen, wenn Sie für die Tabelle `Angebote` sogenannte *Indizes* definieren. (Ein *Index* ist eine Tabelle mit Zeigern. Jede Zeile in einem Index zeigt auf eine entsprechende Zeile in der Datentabelle.)

Sie können für jede Methode, mit der Sie auf die Daten zugreifen wollen, einen eigenen Index anlegen. Wenn Sie in der Tabelle Zeilen einfügen, ändern oder löschen, müssen Sie die Tabelle nicht neu sortieren. Sie müssen nur die Indizes aktualisieren. Das Aktualisieren eines Index ist sehr viel schneller als das Sortieren einer Tabelle. Nachdem Sie einen Index mit der gewünschten Sortierreihenfolge angelegt haben, können Sie damit fast augenblicklich auf die Zeilen in der Datentabelle zugreifen.

Weil das Feld `AngebotNr` eindeutig und kurz ist, bietet es die schnellste Methode, um auf einen einzelnen Datensatz zuzugreifen. Es bildet den idealen Kandidaten für einen Primärschlüssel. Und weil Primärschlüssel in der Regel den schnellsten Weg bilden, um auf Daten zuzugreifen, *sollte der Primärschlüssel einer Tabelle immer indiziert werden*. Access indiziert Primärschlüssel automatisch. Wenn Sie das Feld `AngebotNr` benutzen wollen, müssen Sie natürlich die Angebotsnummer des gewünschten Datensatzes kennen. Zusätzlich können Sie weitere Indizes definieren, die auf anderen Feldern basieren, zum Beispiel `Nachname`, `Plz` oder `Beziehung`. Wenn Sie beispielsweise in der Tabelle einen Index für die Spalte `Nachname` definieren und dann nach dem Nachnamen *Marx* suchen, finden Sie sofort alle Zeilen mit diesem Nachnamen, denn im Index sind alle *Marx*-Zeilen direkt hintereinander gespeichert. Sie können also *Chico*, *Groucho*, *Harpo*, *Zeppo* und *Karl* beinahe so schnell abrufen wie *Chico* allein.

Jeder zusätzliche Index bedeutet in Ihrem System einen zusätzlichen Overhead, der das Arbeiten etwas verlangsamt. Sie müssen diese Verlangsamung gegen den indexbedingten Gewinn an Geschwindigkeit beim Zugriff auf die Datensätze abwägen.

Hier sind einige Tipps zur Auswahl von geeigneten Indexfeldern:

- ✔ Es lohnt sich immer, Felder zu indizieren, mit denen Sie häufig auf Datensätze zugreifen, um den Zugriff zu beschleunigen.
- ✔ Felder, die Sie *nie* als Zugriffsschlüssel verwenden, sollten nicht indiziert werden. Dies wäre reine Zeit- und Ressourcenverschwendung.
- ✔ Indizes für Felder zu erstellen, die nur wenige verschiedene Werte enthalten, ist sinnlos. So trennt beispielsweise das Feld `GeschOderWohltätig` (Geschäft oder Wohltätigkeit) die gesamten Datensätze der Tabelle nur in zwei Kategorien und ist deshalb als Index wertlos.

Die Effizienz von Indizes ist von der Implementierung abhängig. Wenn Sie eine Datenbank von einer Plattform auf eine andere portieren, können Indizes, die auf dem ersten System schnell sind, auf der neuen Plattform langsamer laufen. Indizes können die Datenbank beim Suchen sogar langsamer machen, als sie es ohne Indizes wäre. Probieren Sie verschiedene Indexkombinationen aus, bis Sie das beste Gesamtverhalten für Abrufe und Aktualisierungen gefunden haben.

Um für unsere Beispieltabelle Indizes zu definieren, wählen Sie einfach in dem Bereich FELDEIGENSCHAFTEN in dem Feld INDIZIERT den Eintrag `Ja` aus.

Access erstellt automatisch einen Index für den Primärschlüssel und für das Feld `Postleitzahl`, weil dieses Feld häufig für Abfragen benutzt wird.

Die `Postleitzahl` ist kein Primärschlüssel und sie ist auch nicht notwendigerweise eindeutig. Dagegen ist die `AngebotNr` sowohl Primärschlüssel als auch eindeutig. Für `Nachname` haben wir bereits einen Index definiert. Das Feld `Beziehung` werden Sie wahrscheinlich häufiger für die Abfrage von Daten verwenden.

Nachdem Sie alle Indizes definiert haben, vergessen Sie nicht, die neue Tabellenstruktur zu speichern, bevor Sie sie schließen.

Wenn Sie ein anderes RAD-Tool als Microsoft Access verwenden, gelten die Informationen in diesem Abschnitt nicht für Sie. Doch der Prozess ist insgesamt ziemlich ähnlich.

Eine Tabelle löschen

Bis eine Tabelle genau die Struktur aufweist, die Sie sich vorstellen, entwickeln Sie in der Regel mehrere Zwischenversionen. Diese Tabellen sind später nutzlos und können unnötig Verwirrung stiften. Deshalb ist es besser, sie zu löschen, solange Sie noch wissen, welche Tabelle was enthält. Klicken Sie zu diesem Zweck mit der rechten Maustaste auf die Tabelle und wählen Sie in dem Kontextmenü den Eintrag LÖSCHEN aus (siehe Abbildung 4.10).

Sie sollten *ganz genau* wissen, was Sie tun! Wenn Sie auf LÖSCHEN klicken, wird die ganze Arbeit, die Sie in die Tabelle gesteckt haben, verschwunden sein.

Wenn Access eine Tabelle löscht, löscht es auch alle untergeordneten Tabellen (einschließlich ihrer Indizes).

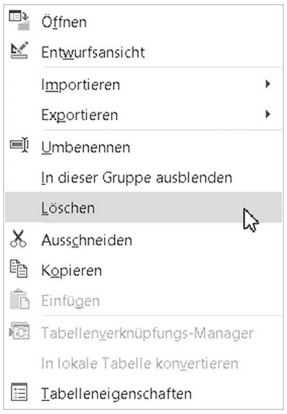


Abbildung 4.10: Sie können eine Tabelle über das Kontextmenü der Tabelle löschen.

Das gleiche Beispiel mit der DDL von SQL erstellen

Sie können alle Funktionen zur Definition einer Datenbank, die Sie mit einem RAD-Werkzeug wie Microsoft Access ausführen, auch mit SQL vollenden. Es ist natürlich nicht so glamourös, Befehle über die Tastatur einzugeben, als Menübefehle mit der Maus auszuwählen. Benutzer, die Objekte lieber optisch bearbeiten, kommen mit RAD-Werkzeugen besser zurecht. Andere ziehen es vor, Wörter zu logischen Befehlen aneinanderzureihen, und arbeiten deshalb lieber mit SQL-Befehlen.

Weil sich einige Dinge einfacher als Objektvorlagen darstellen und andere sich leichter mit SQL bewältigen lassen, ist es sinnvoll, beide Methoden zu beherrschen.

In den folgenden Abschnitten benutze ich SQL, um dieselben Operationen zum Erstellen, Ändern und Löschen der Tabelle auszuführen, für die ich im ersten Teil dieses Kapitels das RAD-Werkzeug verwendet habe.

SQL mit Microsoft Access nutzen

Access ist als ein Werkzeug zur schnellen Anwendungsentwicklung (Rapid Application Development, RAD) entworfen worden, das keine Programmierarbeiten erfordert. Sie können mit Access SQL-Anweisungen schreiben und ausführen lassen, müssen aber dafür eine »Hintertür« verwenden. Um einen einfachen Editor zu öffnen, mit dem Sie SQL-Code eingeben können, gehen Sie wie folgt vor:

1. **Öffnen Sie Ihre Datenbank und wählen Sie die Registerkarte Erstellen im Menüband am oberen Rand des Fensters aus.**
2. **Klicken Sie im Bereich Abfragen des Menübands auf das Symbol Abfrageentwurf.**

   Es erscheint das Dialogfeld Tabelle anzeigen.

3. **Markieren Sie eine beliebige Tabelle. Klicken Sie auf die Schaltfläche Hinzufügen und dann auf Schliessen.**

   Access zeigt die Abfragetools an (siehe Abbildung 4.11).

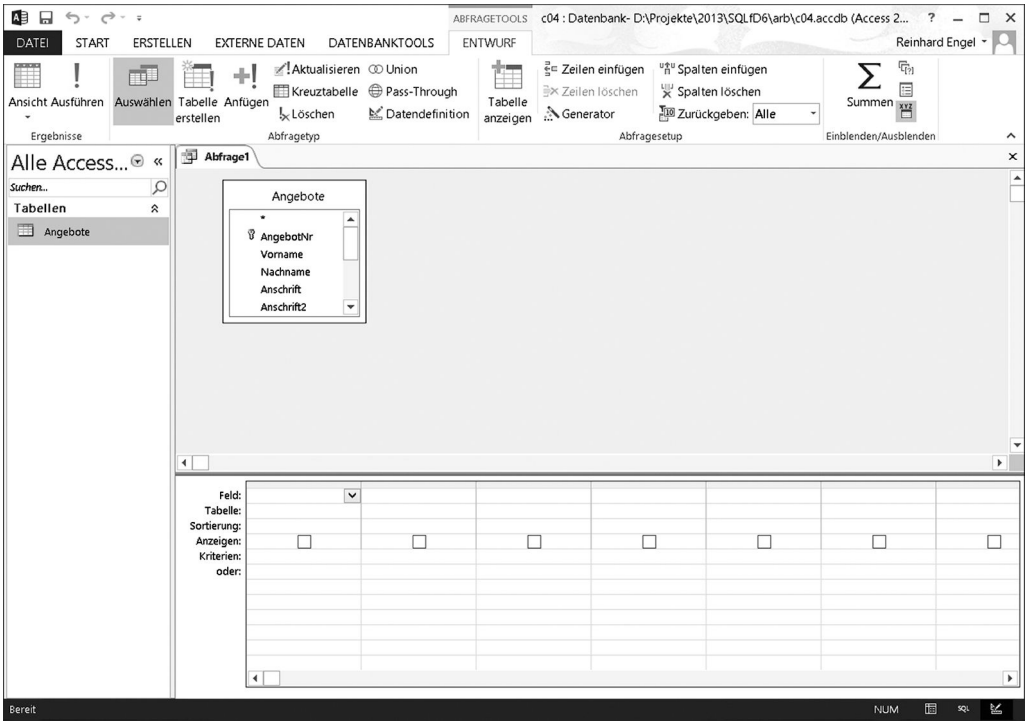


Abbildung 4.11: Die Abfragetools mit der Tabelle `Angebote`

Im mittleren Bereich des Fensters wird ein Bild der `Angebote`-Tabelle mit ihren Attributen angezeigt. Im unteren Bereich wird eine QBE-Tabelle (QBE = Query by Example, deutsch *Abfrage per Beispiel*) angezeigt. Access erwartet, dass Sie jetzt per QBE eine Abfrage eingeben. (Das *könnten* Sie tun, aber dann würden Sie nicht erfahren, wie SQL in der Access-Umgebung verwendet wird.)

4. **Klicken Sie auf den kleinen Pfeil unter dem Ansicht-Symbol am linken Rand des Menübands.**

   Es wird ein Menü mit den verschiedenen Ansichten geöffnet, die Sie im Abfragemodus öffnen können (siehe Abbildung 4.12).

   Eine dieser Sichten ist die SQL-Ansicht.

Abbildung 4.12: Datenbank-Ansichten, die im Abfragemodus zur Verfügung stehen

5. Klicken Sie auf SQL-Ansicht, löschen Sie die SELECT-Anweisung und geben Sie die SQL-Anweisung ein, die Sie ausführen lassen wollen. Dies könnte zum Beispiel folgende Anweisung sein:

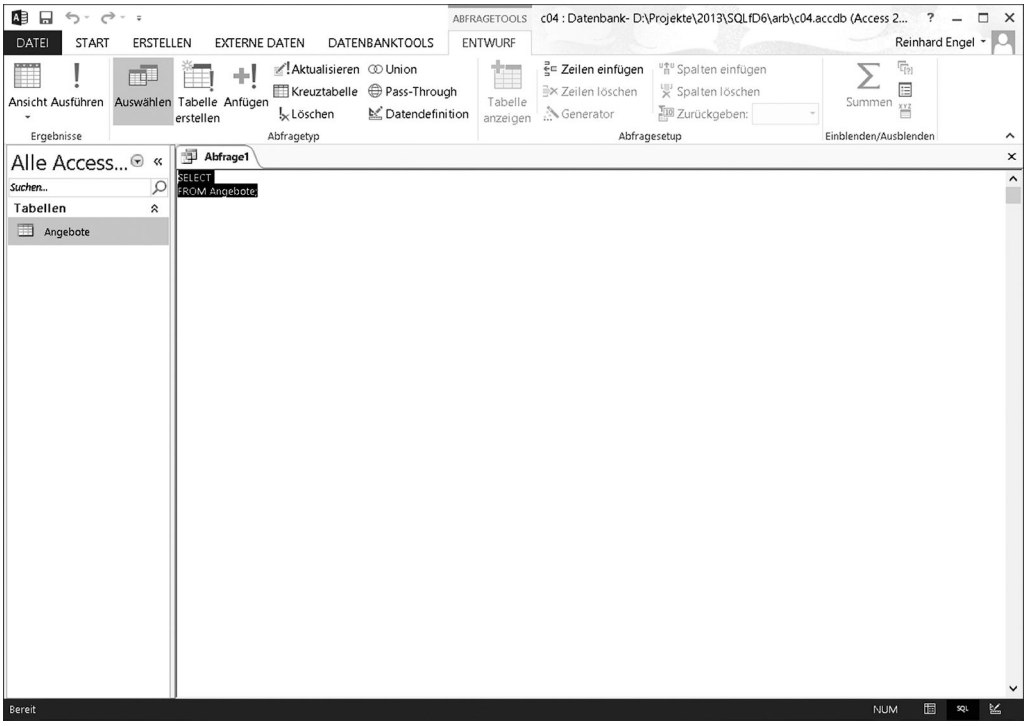


Abbildung 4.13: SQL-Ansicht im Abfragemodus

Access hat vernünftigerweise angenommen, dass Sie Daten aus der `Angebote`-Tabelle abrufen wollen. Deshalb hat es bereits den Anfang der entsprechenden Abfrage in den Arbeitsbereich eingetragen, und zwar nur den Teil, der einigermaßen sicher ist. Denn was genau Sie abfragen wollen, weiß Access natürlich nicht. Bis jetzt hat Access geschrieben:

```
SELECT (leer)
     FROM Angebote;
```

6. **Fügen Sie ein Sternchen (*) in den leeren Bereich in der ersten Zeile ein und fügen Sie eine WHERE-Klausel nach der FROM-Zeile ein.**

   Wenn Sie bereits Daten in die Angebote-Tabelle eingefügt haben, könnten Sie etwa eine Abfrage wie die folgende absetzen:

```
SELECT *
  FROM Angebote
   WHERE Nachname = 'Marx' ;
```

   Vergessen Sie nicht, die SQL-Anweisung mit einem Semikolon (;) abzuschließen. Sie müssen es von seiner Position unmittelbar hinter Angebote ans Ende der nächsten Zeile verschieben.

7. **Wenn Sie fertig sind, klicken Sie auf das Disketten-Symbol, um die Abfrage zu speichern.**

   Access möchte nun von Ihnen wissen, welchen Namen die von Ihnen erzeugte Abfrage erhalten soll.

8. **Geben Sie einen Namen ein und klicken Sie auf OK.**

Ihre Anweisung wird gespeichert und kann später als Abfrage ausgeführt werden.

Eine Tabelle erstellen

Unabhängig davon, ob Sie nun mit Access oder einem ausgewachsenen Unternehmensdatenbanksystem wie etwa Microsoft SQL Server, Oracle oder IBM DB2 arbeiten, müssen Sie dieselben Informationen angeben wie bei der Erstellung einer Tabelle mit einem RAD-Werkzeug. Der Unterschied ist der, dass Sie das RAD-Werkzeug dadurch unterstützt, dass es eine grafische Schnittstelle in Form einer Dialogbox (oder einem ähnlichen Gerüst für die Dateneingabe) hat und verhindert, dass Sie fehlerhafte Feldnamen, Typen oder Größen eingeben.

SQL hilft Ihnen so gut wie gar nicht. Sie müssen vorher wissen, was Sie tun wollen. Wenn Sie erst während des Erstellens versuchen herauszufinden, was Sie wollen, führt das in der Regel zu mehr als unbefriedigenden Ergebnissen. Sie müssen die komplette CREATE TABLE-Anweisung eingeben, ehe SQL sie sich überhaupt anschaut, geschweige denn anzeigt, ob Sie in der Anweisung einen Fehler gemacht haben.

Im ISO/IEC-Standard-SQL sieht die SQL-Anweisung zur Erstellung einer Tabelle zum Nachhalten der Angebote, die identisch ist mit der Tabelle, die ich weiter oben in diesem Abschnitt erzeugt habe, wie folgt aus:

```
CREATE TABLE AngeboteSQL (
   AngebotNr            INTEGER       PRIMARY KEY,
   Vorname              CHAR(15),
   Nachname             CHAR(20),
   Anschrift            CHAR(30),
   Plz                  CHAR(10),
   Land                 CHAR(2),
```

```
Ort                 CHAR(25),
Bundesland          CHAR(30),
Telefon             CHAR(30),
Beziehung           CHAR(30),
Angebot             CHAR(50),
GeschOderWohltätig  CHAR(1) );
```

Die Daten in der SQL-Anweisung sind im Wesentlichen dieselben, die Sie mit der grafischen Oberfläche von Access eingegeben haben. Das Schöne an SQL ist, dass die Sprache universell ist. Unabhängig davon, welches Datenbankverwaltungssystem Sie auch verwenden, es funktioniert immer die gleiche Standardsyntax.

In Access 2013 ist die Erstellung von Datenbankobjekten wie Tabellen etwas komplizierter. Man kann nicht einfach eine `CREATE`-Anweisung (wie die gerade gezeigte) in den Arbeitsbereich der SQL-Ansicht eingeben, weil diese nur als Abfragewerkzeug zur Verfügung steht; Sie müssen einige zusätzliche Aktionen ausführen, um Access darüber zu informieren, dass Sie eine Datendefinitionsabfrage und keine normale Datenabfrage ausführen wollen. Eine weitere Komplikation: Weil die Erstellung einer Tabelle die Sicherheit der Datenbank gefährden könnte, ist sie standardmäßig verboten. Sie müssen Access mitteilen, dass es sich um eine vertrauenswürdige Datenbank handelt, bevor es eine Datendefinitionsabfrage akzeptiert.

1. **Wählen Sie die Registerkarte Erstellen in dem Hauptmenü aus, um das Symbol-Menüband für die Erstellung von Objekten anzuzeigen.**

2. **Klicken Sie im Bereich Abfragen des Menübands auf das Symbol Abfrageentwurf.**

   Es erscheint das Dialogfeld Tabelle anzeigen, das zu diesem Zeitpunkt nur eine Tabelle enthält, `Angebote`.

3. **Wählen Sie Angebote aus (falls sie nicht bereits ausgewählt ist) und klicken Sie auf den Button Hinzufügen.**

   Wie Sie bereits in dem vorhergehenden Beispiel gesehen haben, wird ein Bild der `Angebote`-Tabelle mit ihren Attributen in der oberen Hälfte des Arbeitsbereichs angezeigt.

4. **Klicken Sie in dem Dialogfeld Tabelle anzeigen auf den Button Schliessen.**

5. **Klicken Sie auf den kleinen Pfeil unter dem Ansicht-Symbol am linken Rand des Menübands. Wählen Sie in dem Menü den Eintrag SQL-Ansicht aus.**

   Wie in dem vorhergehenden Beispiel hat Access »geholfen«, indem es `SELECT FROM Angebote` in den SQL-Editor eingefügt hat. Dieses Mal wollen Sie die Hilfe nicht haben.

6. Löschen Sie `SELECT FROM Angebote` und fügen Sie stattdessen die weiter vorne gezeigte Datendefinitionsabfrage wie folgt ein:

```
CREATE TABLE AngeboteSQL (
     AngebotNr             INTEGER       PRIMARY KEY,
     Vorname               CHAR(15),
     Nachname              CHAR(20),
     Anschrift             CHAR(30),
     Plz                   CHAR(10),
     Land                  CHAR(2),
     Ort                   CHAR(25),
     Bundesland            CHAR(30),
     Telefon               CHAR(30),
     Beziehung             CHAR(30),
     Angebot               CHAR(50),
     GeschOderWohltätig    CHAR(1) );
```

An diesem Punkt sollte Ihr Fenster etwa wie Abbildung 4.14 aussehen.

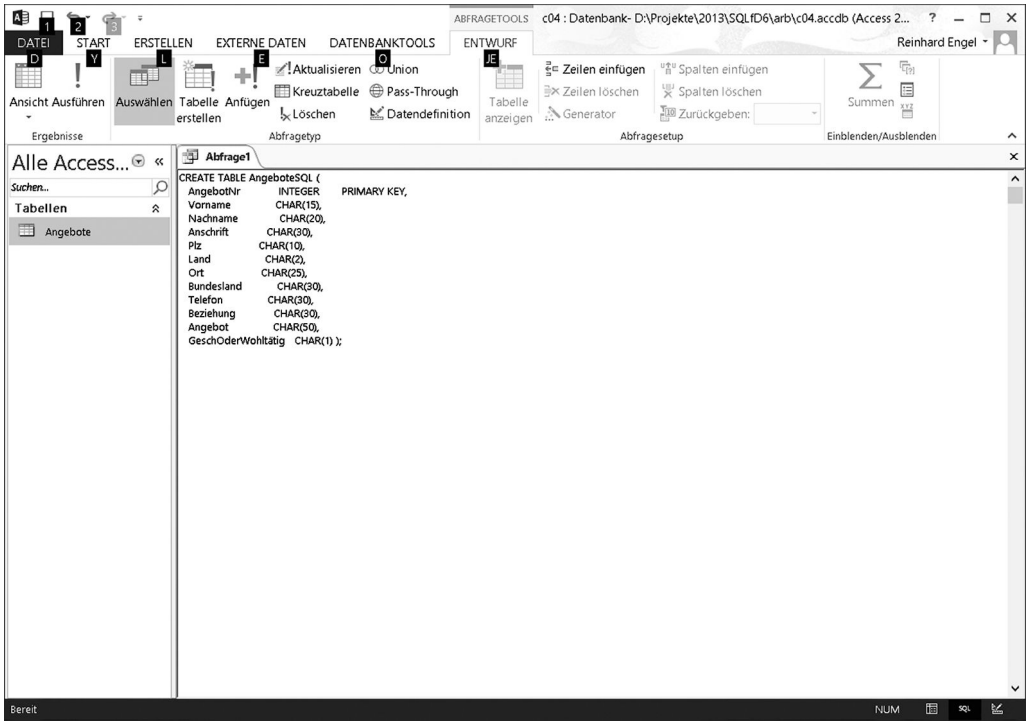


Abbildung 4.14: Datendefinitionsabfrage zur Erstellung einer Tabelle

7. Doppelklicken Sie auf das AUSFÜHREN-Symbol (das rote Ausrufezeichen) in dem Symbol-Menüband.

Dadurch wird die Abfrage ausgeführt, die die Tabelle `AngeboteSQL` erstellt (siehe Abbildung 4.15).

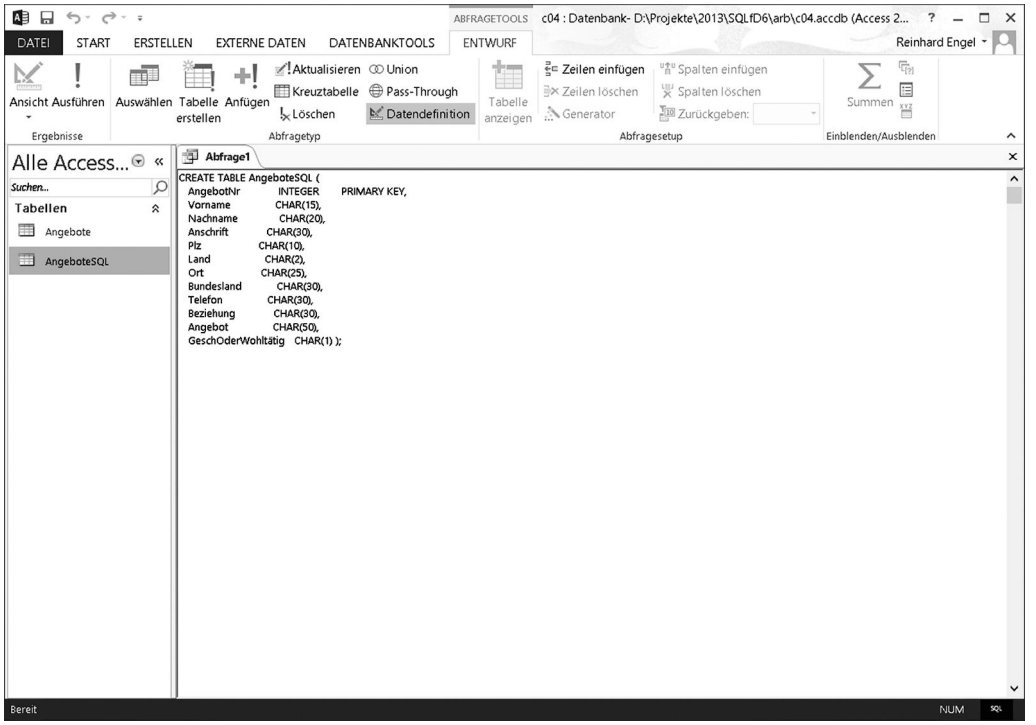


Abbildung 4.15: Links die neue `AngeboteSQL`-Tabelle

Die neue Tabelle `AngeboteSQL` sollte links unter TABELLEN angezeigt werden. In diesem Fall ist alles in Ordnung. Andernfalls müssen Sie weiterlesen.

Access 2013 gibt sich große Mühe, Sie vor böswilligen Hackern und Ihren eigenen unabsichtlichen Fehlern zu schützen, weil die Ausführung einer Datendefinitionsabfrage potenziell gefährlich ist. Access ist standardmäßig so eingestellt, dass die Abfrage nicht ausgeführt wird. Wenn Ihnen dies passiert ist, erscheint `AngeboteSQL` *nicht* unter TABELLEN, weil die Abfrage nicht ausgeführt worden ist. Stattdessen könnte die Statusleiste unter dem Symbol-Menüband eine knappe Meldung zeigen:

```
Sicherheitswarnung  Einige aktive Inhalte wurden deaktiviert.
Klicken Sie hier, um weitere Details anzuzeigen.
```

Wenn Sie diese Nachricht sehen, führen Sie die nächsten Schritte aus.

8. **Klicken Sie auf die Registerkarte Datei und wählen Sie Optionen.**

   Das Dialogfeld Access-Optionen wird geöffnet.

9. **Wählen Sie links Trust Center aus.**

10. **Klicken Sie auf den Button Einstellungen für das Trust Center.**

11. **Wählen Sie im Menü links Meldungsleiste aus und aktivieren Sie (falls erforderlich) das Optionsfeld Meldungsleiste in allen Anwendungen anzeigen, wenn aktiver Inhalt, wie zum Beispiel ActiveX-Steuerelemente, gesperrt ist.**

12. **Schließen Sie die Fenster, bis Sie zu der Stelle zurückgekehrt sind, an der Sie die Datendefinitionsabfrage ausführen können, um Ihre `Angebote`-Tabelle zu erstellen.**

13. **Führen Sie die Abfrage aus.**

Das Erlernen von SQL zahlt sich langfristig aus, weil es SQL noch lange geben wird. Die Anstrengungen, die Sie unternehmen, um ein bestimmtes Entwicklungswerkzeug zu beherrschen, zahlen sich wahrscheinlich viel weniger aus. Denn egal, wie wundervoll das neueste RAD-Werkzeug auch sein mag, es wird in drei bis fünf Jahren durch eine neuere Technologie überholt und abgelöst werden. Wenn sich innerhalb dieser Zeitspanne Ihre in das Werkzeug getätigten Investitionen bezahlt machen, wäre das großartig! Arbeiten Sie damit. Wenn Sie dies jedoch nicht schaffen, sind Sie gut damit beraten, beim Bewährten zu bleiben. Bilden Sie Ihre Mitarbeiter in SQL aus und Ihr Schulungsaufwand wird sich mit Sicherheit über eine viel längere Zeitspanne hinweg auszahlen.

Einen Index erstellen

Indizes sind ein wichtiger Bestandteil jeder relationalen Datenbank. Sie dienen als Zeiger auf die Daten in den Datenbanktabellen. Indem Sie einen Index verwenden, können Sie direkt zu einem bestimmten Datensatz gehen, ohne die Tabelle sequenziell Zeile für Zeile durchsuchen zu müssen, um den entsprechenden Datensatz zu finden. Bei großen Tabellen sind Indizes unverzichtbar. Ohne Indizes kann es vorkommen, dass Sie *Jahre* statt Sekunden auf ein Ergebnis warten müssen. (Nun ja, wahrscheinlich würden Sie nicht jahrelang warten. Einige Abfragen würden jedoch tatsächlich so lange laufen, falls Sie sie laufen ließen. Aber in der Regel ist die Computerzeit für solche Experimente zu wertvoll, weshalb Sie die Abfrage wahrscheinlich einfach abbrechen und feststellen werden, dass das Leben auch ohne dieses spezielle Ergebnis weitergeht.)

Erstaunlicherweise ist in der SQL-Spezifikation keine Anweisung definiert, um einen Index anzulegen. Die Anbieter von Datenbankverwaltungssystemen haben diese Funktion selbst implementiert. Weil diese Implementierungen aber nicht standardisiert sind, können sie sich voneinander unterscheiden. Die meisten Anbieter sorgen für eine Möglichkeit, Indizes zu erstellen, indem sie SQL um die Anweisung `CREATE INDEX` erweitern.

Aber selbst wenn zwei Anbieter dabei dieselben Wörter (`CREATE INDEX`) benutzen, kann die Anweisung unterschiedlich ausgeführt werden. Wahrscheinlich stoßen Sie auch auf einige implementierungsabhängige Klauseln. Deshalb sollten Sie Ihre DBMS-Dokumentation sorgfältig studieren, um herauszufinden, wie bei Ihrem Datenbankverwaltungssystem Indizes erstellt werden.

Die Tabellenstruktur ändern

Mit der SQL-Anweisung `ALTER TABLE` können Sie die Struktur einer vorhandenen Tabelle ändern. Das interaktive SQL auf Ihrem Client-Rechner ist nicht so bequem wie ein RAD-Werkzeug. Ein RAD-Werkzeug zeigt Ihnen die Struktur Ihrer Tabelle an, die Sie daraufhin leicht ändern können. Bei SQL müssen Sie vorher genau wissen, wie die Tabelle aufgebaut ist und was Sie ändern wollen. Die Anweisung für die entsprechende Änderung müssen Sie auf der Befehlszeilenebene eingeben. Wenn Sie jedoch die Anweisungen zur Änderung der Tabelle in ein Anwendungsprogramm einbetten, ist normalerweise das Arbeiten mit SQL der einfachste Weg.

Um ein zweites Anschriftenfeld in die Tabelle `AngeboteSQL` einzufügen, benutzen Sie folgende DDL-Anweisung:

```
ALTER TABLE AngeboteSQL
   ADD COLUMN Anschrift2 CHAR(30);
```

Sie müssen kein SQL-Guru sein, um diesen Code zu entziffern. Sogar bekennende Computerignoranten können diesen Befehl wahrscheinlich interpretieren. Die Anweisung ändert eine Tabelle mit dem Namen `AngeboteSQL`, indem sie eine Spalte in die Tabelle einfügt. Die Spalte heißt `Anschrift2`, hat den Datentyp `CHAR` und ist 30 Zeichen lang. Dieses Beispiel zeigt, wie einfach Sie die Struktur von Datenbanktabellen mit SQL-DDL-Anweisungen ändern können.

Mit dieser Standard-SQL-Anweisung sind Sie in der Lage, Spalten zu einer Tabelle hinzuzufügen. Das Löschen von Spalten geht so ähnlich, wie der folgende Code zeigt:

```
ALTER TABLE AngeboteSQL
   DROP COLUMN Anschrift2;
```

Eine Tabelle löschen

Mit der Anweisung `DROP TABLE` können Sie nicht mehr benötigte Datenbanktabellen schnell und einfach löschen:

```
DROP TABLE AngeboteSQL ;
```

Was könnte einfacher sein? Wenn Sie eine Tabelle löschen, werden auch ihre Daten und Metadaten gelöscht. Es bleibt keine Spur von der Tabelle zurück. Dies funktioniert nur nicht, wenn eine andere Tabelle in der Datenbank die Tabelle referenziert, die Sie zu löschen versuchen. Dies wird als *referenzielle Integritätseinschänkung* bezeichnet. In einem solchen Fall meldet SQL einen Fehler, anstatt die Tabelle zu löschen.

Einen Index löschen

Wenn Sie eine Tabelle mit `DROP TABLE` löschen, werden automatisch auch alle Indizes der Tabelle gelöscht. Bisweilen möchten Sie jedoch eine Tabelle beibehalten, während ein Index daraus gelöscht werden soll. Das Löschen eines einzelnen Index ist in Standard-SQL nicht vorgesehen, aber die meisten Implementierungen stellen dafür die Anweisung `DROP INDEX` zur Verfügung. Dieser Befehl ist nützlich, wenn Ihr System sehr langsam arbeitet und Sie entdecken, dass Ihre Tabellen nicht optimal indiziert sind. Ein Indexproblem zu beheben, kann das Leistungsverhalten Ihres Systems erheblich verbessern, was die Benutzer freuen wird, die bis dahin an sehr lange Antwortzeiten gewöhnt waren.

Überlegungen zur Portierbarkeit

Jede SQL-Implementierung hat ihre eigenen Erweiterungen, mit der sie Möglichkeiten zur Verfügung stellt, die in der offiziellen SQL-Spezifikation nicht vorgesehen sind. Einige dieser Funktionalitäten werden vielleicht in die nächste Version der SQL-Spezifikation aufgenommen. Andere werden wahrscheinlich für immer auf einzelne Implementierungen beschränkt bleiben.

Oft erleichtern diese Erweiterungen die Entwicklung einer Anwendung, die Ihren Bedürfnissen entspricht, und Sie geraten dadurch leicht in die Versuchung, diese speziellen Funktionalitäten zu verwenden. Häufig scheint es der beste Weg zu sein, diese Erweiterungen zu nutzen. Sie sollten sich dann aber über die Konsequenzen im Klaren sein. Falls Sie Ihre Anwendungen jemals auf eine andere SQL-Plattform übertragen wollen, müssen Sie die Abschnitte neu schreiben, in denen Sie Erweiterungen benutzt haben, die von der neuen Umgebung nicht unterstützt werden.

Je besser Sie die bestehenden Datenbankimplementierungen und ihre Erweiterungen kennen, desto fundierter werden Ihre Entscheidungen sein. Denken Sie darüber nach, wie wahrscheinlich die Möglichkeit einer solchen Migration sein wird, und denken Sie auch darüber nach, ob eine Erweiterung von vielen verschiedenen SQL-Implementierungen oder nur von Ihrer Implementierung verwendet werden kann. Auf eine Erweiterung zu verzichten, kann langfristig die bessere Lösung sein, selbst wenn Sie durch ihren Einsatz heute etwas Zeit sparen könnten. Andererseits kann es auch sein, dass Sie keinen Grund sehen, die Erweiterung nicht zu benutzen. Ihre Entscheidung.

Eine relationale Datenbank mit mehreren Tabellen erstellen

5

In diesem Kapitel

- Entscheiden, was zur Datenbank gehört
- Beziehungen zwischen Datenelementen festlegen
- Zusammengehörige Tabellen mit Schlüsseln verknüpfen
- Die Datenintegrität im Entwurf sicherstellen
- Die Datenbank normalisieren

In diesem Kapitel möchte ich mit Ihnen ein Beispiel durcharbeiten, in dem wir eine Datenbank mit mehreren Tabellen erstellen. Der erste Schritt zu einer derartigen Datenbank besteht darin, festzulegen, was zur Datenbank gehören soll und was nicht. Die nächsten Schritte bestehen darin, festzulegen, welche Beziehungen zwischen den einzelnen Elementen bestehen, um dann die Tabellen dementsprechend einzurichten. Außerdem werde ich auf das Arbeiten mit *Schlüsseln* eingehen, mit denen Sie schnell auf einzelne Datensätze und Indizes zugreifen können.

Eine Datenbank soll Ihre Daten nicht nur speichern, sondern auch vor Beschädigungen schützen. Im letzten Teil dieses Kapitels erkläre ich deshalb, wie Sie die Integrität Ihrer Daten gewährleisten können. Eine Schlüsselmethode, um dieses Ziel zu erreichen, besteht darin, die Datenbank zu normalisieren. Deshalb beschreibe ich die verschiedenen *Normalformen* und gehe auf die Probleme ein, die mit der *Normalisierung* gelöst werden.

Die Datenbank entwerfen

Um eine Datenbank zu entwerfen, müssen Sie die folgenden grundlegenden Schritte ausführen (sie werden in den nächsten Abschnitten ausführlich beschrieben):

1. **Stellen Sie fest, welche Objekte Sie in die Datenbank aufnehmen möchten.**
2. **Legen Sie fest, welche dieser Objekte Tabellen sind und welche Objekte Spalten innerhalb dieser Tabellen bilden.**
3. **Definieren Sie die Tabellen so, dass Ihre Anforderungen an die Organisation der Objekte erfüllt werden.**

   Optional können Sie für jede Tabelle eine Spalte oder eine Kombination von Spalten als Schlüssel definieren. *Schlüssel* ermöglichen es, bestimmte Zeilen einer Tabelle schnell zu lokalisieren.

Die folgenden Abschnitte beschreiben diese Schritte im Einzelnen und behandeln dabei auch technische Fragen, die beim Entwerfen einer Datenbank auftreten.

Schritt 1: Objekte definieren

Der erste Schritt beim Entwerfen einer Datenbank besteht darin festzustellen, welche Aspekte des Systems wichtig genug sind, um in das Modell aufgenommen zu werden. Behandeln Sie jeden Aspekt wie ein Objekt, und erstellen Sie eine Liste aller Objekte, die Ihnen einfallen. Versuchen Sie zu diesem Zeitpunkt noch nicht, die Beziehungen zwischen den Objekten festzulegen, sondern erstellen Sie einfach eine möglichst vollständige Liste.

Arbeiten Sie mit Leuten zusammen, die das System kennen, das Sie entwerfen wollen. Wenn mehrere Personen ihre Vorstellungen und Ideen einbringen, wird Ihre Objektliste wahrscheinlich vollständiger und genauer werden, als wenn Sie sich allein mit dem Problem beschäftigen müssen.

Wenn Ihre Liste vollständig ist, gehen Sie zum nächsten Schritt über: Legen Sie fest, welche Beziehungen es zwischen den Objekten gibt. Einige Objekte sind zentrale Dateneinheiten, die zur Bestimmung der Beziehungen unverzichtbar sind. Andere ergänzen diese Einheiten oder sind ihnen untergeordnet. Und dann gibt es noch Objekte, die sich als überflüssig erweisen und wieder gestrichen werden können.

Schritt 2: Tabellen und Spalten identifizieren

Die zentralen Einheiten, auch *Entitäten* genannt, bilden die Grundlage der Datenbanktabellen. Jede dieser Einheiten verfügt über eine Reihe von *Attributen*, die die Grundlage der Spalten einer Tabelle sind. Viele Geschäftsdatenbanken arbeiten beispielsweise mit einer Kundentabelle, um die Namen, Adressen und andere Daten der Kunden zu speichern. Jedes Attribut eines Kunden, zum Beispiel Name, Straße, Postleitzahl, Ort, Land, Telefonnummern und E-Mail-Adresse, wird zu einer Spalte in der Tabelle `Kunde`.

Wenn Sie darauf hoffen, Regeln zu finden, die dabei helfen, herauszufinden, welche Objekte Tabellen sein sollten und welche der Attribute des Systems zu welcher Tabelle gehören, vergessen Sie diesen Versuch. Manchmal macht es Sinn, ein bestimmtes Attribut der einen oder der anderen Tabelle zuzuordnen. Stellen Sie sich dann folgende Fragen:

- ✔ Welche Daten wollen Sie aus der Datenbank abrufen?
- ✔ Wie wollen Sie diese Daten nutzen?

Wenn Sie die Struktur der Datenbanktabellen entwerfen, sprechen Sie sowohl mit den späteren Benutzern der Datenbank als auch mit den Personen, die später anhand der Daten, die in der Datenbank abgelegt sind, Entscheidungen treffen. Wenn die Struktur, die Ihnen vernünftig erscheint, nicht mit der Art und Weise übereinstimmt, wie diese Personen die Daten nutzen wollen, wird Ihr System bestenfalls Frust bei seiner Nutzung auslösen – und eventuell sogar falsche Daten produzieren, was noch schlimmer wäre. Passen Sie auf, dass dies nicht geschieht! Denken Sie sorgfältig darüber nach, wie Sie Ihre Tabellen strukturieren.

Lassen Sie uns ein Beispiel betrachten, das den Denkprozess bei der Erstellung einer Datenbank mit mehreren Tabellen zeigt. Stellen Sie sich vor, dass Sie eine Firma namens *VetLab* gegründet haben, die Laboruntersuchungen für Tierärzte durchführt. Sie wollen Ihren Geschäftsbetrieb mit einer Datenbank verwalten und unter anderem Daten über folgende Bereiche speichern:

- ✔ Kunden
- ✔ Durchgeführte Tests
- ✔ Mitarbeiter
- ✔ Aufträge
- ✔ Ergebnisse

Jede dieser Dateneinheiten verfügt über Attribute. Jeder Kunde hat einen Namen, eine Adresse und andere Kontaktdaten. Jeder Test hat einen Namen und einen Standardpreis. Für die Mitarbeiter können Kontaktdaten, die jeweilige Position und das Gehalt gespeichert werden. Bei einem Auftrag müssen Sie wissen, von wem der Auftrag stammt, wann er einging und welcher Test in Auftrag gegeben wurde. Für jedes Testergebnis müssen Sie das Resultat, die zugehörige Auftragsnummer sowie ein Kennzeichen speichern, das angibt, ob das Ergebnis vorläufig oder endgültig ist.

Schritt 3: Tabellen definieren

Als Nächstes müssen Sie für jede Entität eine Tabelle und für jedes Attribut eine Spalte definieren. Tabelle 5.1 zeigt, wie die Tabellen von VetLab aussehen könnten, die ich im letzten Abschnitt vorgeschlagen habe.

Tabelle	Inhalt der Spalten
Kunde	KundenName
	Anschrift
	Anschrift2
	Ort
	Land
	Postleitzahl
	Telefon
	Fax
	Kontaktperson
Test	Testname
	Gebühr

Tabelle	Inhalt der Spalten
Mitarbeiter	Mitarbeitername
	Anschrift
	Anschrift2
	Ort
	Bundesland
	Postleitzahl
	Telefon privat
	Durchwahl
	Einstelldatum
	Position
	Stundenlohn/Gehalt/Provision
Auftrag	Auftragsnummer
	KundenName
	Bestellter Test
	Zuständiger Verkäufer
	Auftragsdatum
Ergebnis	Ergebnisnummer
	Auftragsnummer
	Ergebnis
	Ergebnisdatum
	Vorläufig/Endgültig

Tabelle 5.1: VetLab-Tabellen

Sie können diese Tabellen entweder mit einem RAD-Werkzeug erstellen oder die Datendefinitionssprache (DDL) von SQL einsetzen, wie der folgende Code zeigt:

```
CREATE TABLE Kunde (
     KundenName          CHARACTER(30)     NOT NULL,
     Anschrift1          CHARACTER(30),
     Anschrift2          CHARACTER(30),
     Ort                 CHARACTER(25),
     Land                CHARACTER(2),
     Postleitzahl        CHARACTER(10),
     Telefon             CHARACTER(13),
     Fax                 CHARACTER(13),
     Kontaktperson       CHARACTER(30) ) ;
```

```
CREATE TABLE Test (
     Testname           CHARACTER(30)      NOT NULL,
     Gebühr             CHARACTER(30) ) ;

CREATE TABLE Mitarbeiter (
     Mitarbeitername    CHARACTER(30)      NOT NULL,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefonprivat      CHARACTER(13),
     Durchwahl          CHARACTER(4),
     Einstelldatum      DATE,
     Position           CHARACTER(10),
     StuGehProv         CHARACTER(1) ) ;

CREATE TABLE Auftrag (
     Auftragsnummer     INTEGER           NOT NULL,
     KundenName         CHARACTER(30),
     BestellterTest     CHARACTER(30),
     Verkäufer          CHARACTER(30),
     Auftragsdatum      DATE ) ;

CREATE TABLE Ergebnis (
     Ergebnisnummer     INTEGER            NOT NULL,
     Auftragsnummer     INTEGER,
     Ergebnistext       CHARACTER(50),
     Ergebnisdatum      DATE,
     VorlEndg           CHARACTER(1) ) ;
```

Diese Tabellen sind miteinander durch die Attribute (Spalten) verbunden, die sie gemeinsam haben:

- ✔ Die Tabelle `Kunde` ist mit der Tabelle `Auftrag` durch die Spalte `KundenName` verbunden.
- ✔ Die Tabelle `Test` ist mit der Tabelle `Auftrag` durch die Spalte `Testname` (`Bestellter Test`) verbunden.
- ✔ Die Tabelle `Mitarbeiter` ist mit der Tabelle `Auftrag` durch die Spalte `Mitarbeiter name` (`Verkäufer`) verbunden.
- ✔ Die Tabelle `Ergebnis` ist mit der Tabelle `Auftrag` durch die Spalte `Auftragsnummer` verbunden.

Damit eine Tabelle zu einem festen Bestandteil einer relationalen Datenbank wird, muss sie mit wenigstens einer anderen Tabelle der Datenbank durch eine gemeinsame Spalte verbunden werden. Abbildung 5.1 zeigt die Beziehungen zwischen den Tabellen.

Die Pfeile in Abbildung 5.1 zeigen vier Eins-zu-viele-Beziehungen (auch *1:n-Beziehungen* genannt) zwischen den Tabellen. Der einfache Pfeil verweist auf die »Eins«-Seite der Beziehung, der Doppelpfeil auf die »Viele«-Seite.

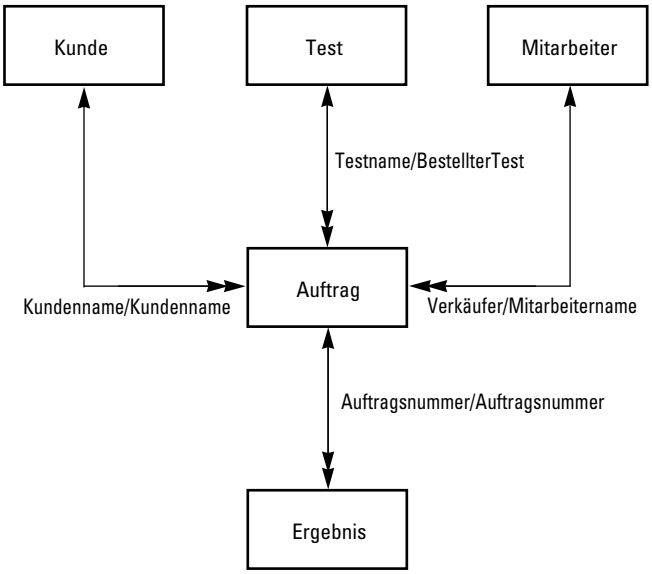


Abbildung 5.1: Die Tabellen und Verknüpfungen der VetLab-Datenbank

- ✔ Ein Kunde kann viele Aufträge platzieren, aber jeder Auftrag stammt von genau einem Kunden.
- ✔ Jeder Test kann auf vielen Aufträgen erscheinen, aber jeder Auftrag betrifft genau einen Test.
- ✔ Jeder Auftrag wird von genau einem Mitarbeiter (oder Verkäufer) entgegengenommen, aber jeder Verkäufer kann mehrere Aufträge entgegennehmen.
- ✔ Jeder Auftrag kann mehrere vorläufige Testergebnisse und ein Endergebnis produzieren, aber jedes Ergebnis gehört zu genau einem Auftrag.

Die Abbildung zeigt, dass die Attribute, die die Tabellen miteinander verknüpfen, unterschiedliche Namen haben können. In beiden Tabellen müssen jedoch ihre Datentypen übereinstimmen. An diesem Punkt habe ich keine Integritätseinschränkungen hinzugefügt, um Sie nicht mit zu vielen neuen Konzepten auf einmal zu bedrängen. Die referenzielle Integrität wird später in diesem Kapitel behandelt.

Domänen, Zeichensätze, Sortierfolgen und Übersetzungstabellen

Obwohl Tabellen die wichtigsten Komponenten einer Datenbank sind, spielen auch noch andere Elemente eine nicht unbedeutende Rolle. In Kapitel 1 habe ich die *Domäne* einer Tabellenspalte als die Menge aller Werte definiert, die die Spalte annehmen kann. Ein wichtiger Teil des Datenbankentwurfs ist die Definition klar festgelegter Domänen mithilfe von Einschränkungen.

Relationale Datenbanken werden nicht nur in Deutschland, sondern auch in anderen Ländern mit anderen Sprachen und anderen Zeichensätzen verwendet. Selbst wenn Ihre Daten nicht in einer fremden Sprache gespeichert werden, gibt es Anwendungsmöglichkeiten, die spezielle Zeichensätze erfordern. SQL gibt Ihnen die Möglichkeit, den Zeichensatz festzulegen, mit dem Sie arbeiten wollen. Sie können sogar für jede Tabellenspalte einen anderen Zeichensatz benutzen. Diese Art von Flexibilität ist im Allgemeinen in anderen Sprachen nicht verfügbar, sondern nur bei SQL vorhanden.

Eine *Sortierreihenfolge* (englisch *Collation*) ist ein Satz von Regeln, die festlegen, wie Zeichenfolgen, die in einem bestimmten Zeichensatz vorliegen, miteinander verglichen werden. Jeder Zeichensatz verfügt über eine Standardsortierreihenfolge. In der Standardsortierreihenfolge des ASCII-Zeichensatzes kommt *A* vor *B* und *B* vor *C*. Deshalb wird bei einem Vergleich *A* kleiner als *B* und *C* größer als *B* eingestuft. In SQL können Sie für einen Zeichensatz verschiedene Sortierfolgen definieren. Auch diese Flexibilität gibt es im Allgemeinen nur in SQL.

Manchmal müssen Sie die Daten in einer Datenbank von einem Zeichensatz in einen anderen übersetzen – beispielsweise wenn die Daten in einem deutschen Zeichensatz gespeichert sind, aber Ihr Drucker die deutschen Umlaute nicht unterstützt, weil diese nicht zum ASCII-Zeichensatz gehören. Zur Lösung dieses Problems sind in SQL sogenannte *Übersetzungstabellen* (englisch *Translations*) vorgesehen. Eine Übersetzungstabelle dient zur Übersetzung von Zeichenfolgen aus einem Zeichensatz in einen anderen. Eine Übersetzungstabelle kann beispielsweise ein Zeichen in zwei Zeichen übersetzen, zum Beispiel das deutsche »ü« in die ASCII-Zeichen »ue«. Sie kann auch dazu dienen, Kleinbuchstaben in Großbuchstaben umzuwandeln und umgekehrt oder ein Alphabet in ein anderes zu übersetzen, zum Beispiel Hebräisch in ASCII.

Schlüssel für den schnellen Zugriff

Eine goldene Regel für das Entwerfen von Datenbanken ist, dass Sie darauf achten, dass sich in einer Datenbanktabelle alle Tabellenzeilen voneinander unterscheiden lassen – jede Zeile sollte eindeutig sein. Manchmal wollen Sie Daten aus Ihrer Datenbank für einen speziellen Zweck, beispielsweise für eine statistische Analyse, entnehmen und in einer separaten Tabelle speichern. Dabei können Tabellenzeilen entstehen, die nicht eindeutig sind. Für einen solchen einmaligen Gebrauch spielt diese Duplizierung von Daten keine Rolle. Tabellen, die laufend benutzt werden, sollten jedoch keine doppelten Zeilen enthalten.

Ein *Schlüssel* ist ein Attribut oder eine Kombination von Attributen, das oder die eine Zeile in einer Tabelle eindeutig identifizieren. Um auf eine bestimmte Zeile in einer Datenbank zugreifen zu können, benötigen Sie ein Mittel, um diese Zeile von allen anderen Zeilen zu unterscheiden. Ein Schlüssel stellt diesen Zugriffsmechanismus zur Verfügung, da er eindeutig sein muss.

Ein Schlüssel darf niemals einen Nullwert enthalten, weil andernfalls zwei Zeilen mit einem Nullwert nicht zu unterscheiden sind.

In unserem VetLab-Beispiel können Sie geeignete Spalten als Schlüssel definieren. In der Kundentabelle eignet sich die Spalte `KundenName` als Schlüssel. Damit sind Sie in der Lage, alle Kunden voneinander zu unterscheiden. Deshalb muss in dieser Spalte ein Wert eingetragen werden. `Testname` und `Mitarbeitername` eignen sich als Schlüssel für die Tabellen `Test` beziehungsweise `Mitarbeiter`, `Auftragsnummer` und `Ergebnisnummer` kommen für die Tabellen `AUFTRAG` beziehungsweise `ERGEBNIS` infrage. Achten Sie darauf, dass der Eintrag in jeder Zeile einen eindeutigen Wert hat.

Es gibt zwei Arten von Schlüsseln: *Primärschlüssel* und *Fremdschlüssel*. Bei den Schlüsseln, über die ich im vorangegangenen Absatz geschrieben habe, handelt es sich um Primärschlüssel. Primärschlüssel garantieren die Eindeutigkeit. Ich gehe in den nächsten Abschnitten ausführlicher auf beide Schlüsselarten ein.

Primärschlüssel

Bei einem Primärschlüssel handelt es sich um eine Spalte einer Tabelle, deren Werte die Zeilen der Tabelle eindeutig identifizieren. Sie können den Primärschlüssel einer Tabelle bei ihrem Erstellen definieren. Im folgenden Beispiel für die VetLab-Datenbank sollte eine Beispielspalte ausreichen (wobei ich voraussetze, dass alle VetLab-Kunden andere Namen haben):

```
CREATE TABLE Kunde (
     KundenName         CHARACTER(30)    PRIMARY KEY,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefon            CHARACTER(13),
     Fax                CHARACTER(13),
     Kontaktperson      CHARACTER(30)
     ) ;
```

Hier ersetzt die Einschränkung `PRIMARY KEY` die Einschränkung `NOT NULL`, die wir in der früheren Definition der Tabelle `Kunde` verwendet haben. Die Einschränkung `PRIMARY KEY` schließt die Einschränkung `NOT NULL` ein, weil Primärschlüssel keine Nullwerte enthalten dürfen.

Obwohl es die meisten Datenbankverwaltungssysteme zulassen, eine Tabelle ohne Primärschlüssel zu erstellen, sollte jede Tabelle in einer Datenbank über einen solchen Schlüssel verfügen. Ersetzen Sie mit diesem Wissen im Hinterkopf in allen Tabellen eine Einschränkung vom Typ `NOT NULL` durch einen `PRIMARY KEY`, wie im folgenden Beispiel gezeigt wird:

```
CREATE TABLE Test (
     Testname           CHARACTER(30)     PRIMARY KEY,
     Gebühr             CHARACTER(30)
     ) ;
```

Manchmal kann eine einzelne Tabellenspalte keine Eindeutigkeit erzeugen. In solchen Fällen können Sie einen *zusammengesetzten Schlüssel* benutzen. Ein zusammengesetzter Schlüssel besteht aus einer Kombination von Spalten, die zusammen für die Eindeutigkeit der Zeile sorgen. Nehmen Sie an, einige VetLab-Kunden unterhalten Filialen in mehreren Städten. In diesem Fall reicht `KundenName` nicht aus, um die verschiedenen Zweigniederlassungen des Kunden voneinander zu unterscheiden. Um dieses Problem zu lösen, können Sie folgendermaßen einen zusammengesetzten Schlüssel definieren:

```
CREATE TABLE Kunde (
     KundenName        CHARACTER(30)     NOT NULL,
     Anschrift1        CHARACTER(30),
     Anschrift2        CHARACTER(30),
     Ort               CHARACTER(25)     NOT NULL,
     Land              CHARACTER(2),
     Postleitzahl      CHARACTER(10),
     Telefon           CHARACTER(13),
     Fax               CHARACTER(13),
     Kontaktperson     CHARACTER(30),
     CONSTRAINT Niederlassung PRIMARY KEY
          (KundenName, Ort)
     ) ;
```

Als Alternative zur Verwendung eines zusammengesetzten Schlüssels, um einen Datensatz eindeutig zu identifizieren, können Sie auch einen Schlüssel automatisch von Ihrem DBMS zuweisen lassen. So schlägt etwa Access in einer neuen Tabelle automatisch ein erstes Feld mit dem Namen `ID` und dem Typ `AutoWert` vor. Ein solcher Schlüssel hat keine eigene Bedeutung, sondern dient nur als eindeutiger Bezeichner eines Datensatzes.

Fremdschlüssel

Ein *Fremdschlüssel* ist eine Spalte oder eine Gruppe von Spalten einer Tabelle, die Bezug zu einem Primärschlüssel in einer anderen Tabelle der Datenbank haben. Ein Fremdschlüssel muss nicht eindeutig sein, aber er muss die Spalten einer Tabelle, auf die er verweist, eindeutig identifizieren.

Wenn die Spalte `KundenName` der Primärschlüssel der Kundentabelle ist, muss jede Zeile in der Kundentabelle in der Spalte `KundenName` einen eindeutigen Namen enthalten. `Kunden Name` ist in der Tabelle `Auftrag` ein Fremdschlüssel. Dieser Fremdschlüssel entspricht dem Primärschlüssel der Kundentabelle, aber in der Tabelle `AUFTRAG` muss dieser Schlüssel nicht eindeutig sein. Tatsächlich hoffen Sie sogar darauf, dass der Fremdschlüssel nicht eindeutig ist. Falls Sie von jedem Kunden nur einen einzigen Auftrag bekämen, müssten Sie bald Konkurs anmelden. Sie hoffen also, dass es zu jeder einzelnen Zeile der Kundentabelle viele Zeilen in der Tabelle `AUFTRAG` gibt, denn das bedeutet, dass fast alle Kunden Stammkunden sind.

Das folgende Beispiel zeigt, wie Sie bei der Definition der Tabelle AUFTRAG einen Fremdschlüssel anlegen können:

```
CREATE TABLE Auftrag (
     Auftragsnummer    INTEGER            PRIMARY KEY,
     KundenName        CHARACTER(30),
     BestellterTest    CHARACTER(30),
     Verkäufer         CHARACTER(30),
     Auftragsdatum     DATE,
     CONSTRAINT NameFK FOREIGN KEY (KundenName)
          REFERENCES Kunde (KundenName),
     CONSTRAINT TestFK FOREIGN KEY (BestellterTest)
          REFERENCES Test (Testname),
     CONSTRAINT VerkaufFK FOREIGN KEY (Verkäufer)
          REFERENCES Mitarbeiter (Mitarbeitername)
     ) ;
```

In diesem Beispiel verknüpfen die Fremdschlüssel in der Tabelle `Auftrag` diese Tabelle mit den Primärschlüsseln der Tabellen `Kunde`, `Test` und `Mitarbeiter`.

Mit Indizes arbeiten

Die SQL-Spezifikation sagt nichts über Indizes, aber diese Auslassung bedeutet nicht, dass Indizes selten benutzt werden oder optionale Komponenten eines Datenbanksystems sind. Jede SQL-Implementierung unterstützt Indizes, aber es gibt keine allgemeine Übereinkunft darüber, wie sie implementiert werden sollen. In Kapitel 4 habe ich Ihnen gezeigt, wie Sie mit dem RAD-Werkzeug Microsoft Access einen Index anlegen können. Sie sollten die Dokumentation Ihres DBMS heranziehen, um herauszufinden, wie Indizes auf Ihrem System erstellt werden.

Was ist eigentlich ein Index?

Die Daten werden in einer Tabelle im Allgemeinen in der Reihenfolge gespeichert, in der sie angelegt werden. Diese Reihenfolge stimmt selten mit der Reihenfolge überein, in der die Daten später verarbeitet werden sollen. Angenommen, Sie möchten die Kunden Ihrer Kundentabelle in alphabetischer Folge verarbeiten. Dazu muss der Computer zunächst die gesamte Tabelle nach dem Namen sortieren. Diese Sortierung kostet Zeit. Je größer die Tabelle ist, desto länger dauert die Sortierung. Was passiert bei einer Tabelle mit 100.000 Zeilen? Was bei einer Tabelle mit einer Million Zeilen? Bei manchen Anwendungen sind diese Größenordnungen durchaus normal. In extremen Fällen muss auch der beste Sortieralgorithmus mehrere Millionen Vergleiche durchführen und Tabellenzeilen austauschen, um die Tabelle in der gewünschten Reihenfolge anzuordnen. Selbst mit einem sehr schnellen Computer könnte dies sehr viel Zeit erfordern.

Indizes lösen dieses Problem und sparen Zeit. Ein *Index* ist eine Hilfstabelle zur Unterstützung der Datentabelle. Zu jeder Zeile in der Datentabelle gibt es eine entsprechende Zeile in der Indextabelle. In der Indextabelle sind die Zeilen jedoch anders sortiert.

Tabelle 5.2 zeigt eine kleine Beispieltabelle.

KundenName	Anschrift1	Anschrift2	Ort	Bundesland
Beates Tierklinik	Nussweg 5		Hummelsheim	NRW
Antons Veterinär-medizin	Baumstraße 3		Abenhausen	Sachsen-Anhalt
Veterinärmediziner Weber	Blumengasse 10	Appartement 230	Attenkirchen	Rheinl.-Pfalz
Dieter Doktor	Buschterrasse 50		Neustadt	Hessen
Tiermedizin Thomas	Veterinäramt	Waldweg 17	Grißdorf	Bayern
Doggenärztin Dorothe	Pflanzenweg 2		Kobenhausen	Meck.-Vorp.
Jürgens Pferdepraxis	Sonnengasse 70		Losdorf	Niedersachsen
Walters Wurmfarm	Mondstraße 513		Superstadt	Baden-Württemb.

Tabelle 5.2: Kundentabelle

Die Zeilen sind nicht alphabetisch nach `KundenName` sortiert. Eigentlich sind sie überhaupt nicht sinnvoll sortiert. Die Zeilen entsprechen einfach der Reihenfolge, in der die Daten eingegeben wurden.

Ein Index dieser Tabelle könnte wie Tabelle 5.3 aussehen.

KundenName	Zeiger auf Datentabelle
Antons Veterinärmedizin	2
Beates Tierklinik	1
Dieter Doktor	4
Doggenärztin Dorothe	6
Jürgens Pferdepraxis	7
Tiermedizin Thomas	5
Veterinärmediziner Weber	3
Walters Wurmfarm	8

Tabelle 5.3: KundenNamenindex für die Kundentabelle

Der Index enthält das Feld, das sortiert werden soll (in diesem Fall `KundenName`) und einen Zeiger auf die Datentabelle. Der Zeiger einer Indexzeile zeigt auf die zugehörige Zeile in der Datentabelle.

Wozu ist ein Index gut?

Wenn Sie eine Tabelle in der alphabetischen Reihenfolge der KundenNamen verarbeiten möchten und über einen Index in dieser Reihenfolge verfügen, können Sie die Verarbeitung

fast so schnell ausführen, als wäre die Tabelle selbst nach KundenNamen alphabetisch sortiert. Sie können einen Index sequenziell abarbeiten und dabei mit dem Zeiger direkt zu dem entsprechenden Datensatz in der Datentabelle springen.

Wenn Sie mit einem Index arbeiten, ist die Zeit für die Verarbeitung der Tabellendaten proportional zu *N*, wobei *N* die Anzahl der Datensätze in der Tabelle ist. Wenn Sie ohne Index arbeiten, ist die Verarbeitungszeit für dieselbe Operation proportional zu *N lg N*, wobei *lg N* der Logarithmus von *N* zur Basis 2 ist. Bei kleinen Tabellen ist der Unterschied unbedeutend, aber bei großen Tabellen ist er erheblich. Bei großen Tabellen sind manche Operationen ohne die Unterstützung von Indizes praktisch nicht durchführbar.

Nehmen wir an, Sie haben eine Tabelle mit 1.000.000 Datensätzen ($N = 1.000.000$) und die Verarbeitung jedes Datensatzes dauert eine Millisekunde (ein Tausendstel einer Sekunde). Wenn Sie mit einem Index arbeiten, dauert die Verarbeitung der gesamten Tabelle nur 1.000 Sekunden – weniger als 17 Minuten. Ohne Index müssen Sie die Tabelle jedoch etwa $1.000.000 \times 20$ Male durchlaufen, um dasselbe Ergebnis zu erzielen. Dieser Prozess würde 20.000 Sekunden dauern – mehr als 5,5 Stunden. So viel kann ein Index bewirken.

Einen Index verwalten

Ein vorhandener Index muss verwaltet werden. Glücklicherweise übernimmt Ihr DBMS diese Aufgabe für Sie, indem es den Index automatisch immer dann aktualisiert, wenn Sie die entsprechenden Datentabellen verändert haben. Dieser Prozess benötigt etwas mehr Zeit als eine Aktualisierung ohne Index, aber dieser Mehraufwand ist mehr als gerechtfertigt. Nachdem Sie einen Index erstellt haben, der von Ihrem DBMS verwaltet wird, steht er immer zur Verfügung, um den Zugriff auf Ihre Daten zu beschleunigen.

Der beste Zeitpunkt, einen Index anzulegen, ist der Zeitpunkt, wenn Sie die entsprechende Datentabelle erstellen. Dann brauchen Sie ihn nicht nachträglich zu erstellen, was – je nach vorhandener Datenmenge – recht langwierig sein kann. Versuchen Sie möglichst, alle häufig verwendeten Formen des Datenzugriffs vorherzusehen, und erstellen Sie *für jede Möglichkeit* einen entsprechenden Index.

Einige DBMS-Produkte ermöglichen es Ihnen, die Verwaltung der Indizes zu deaktivieren. Bei einigen zeitkritischen Echtzeitanwendungen kann dies sinnvoll sein, wenn die Änderung der Indizes zu viel kostbare Zeit verschlingt. Bei einer solchen Konfiguration müssen Sie die Indizes in einer separaten Operation aktualisieren, wenn es – datentechnisch gesehen – einmal ruhig zugeht.

Sie sollten nur solche Indizes anlegen, die Sie tatsächlich für den Abruf von Daten benötigen. Die Indexverwaltung bürdet dem Computer eine zusätzliche Last beim Einfügen, Ändern oder Löschen von Zeilen auf, was sich negativ auf sein Leistungsverhalten auswirkt. Um ein optimales Leistungsverhalten zu erzielen, sollten Sie deshalb nur die Indizes anlegen, die Sie wirklich brauchen, und nur wirklich große Tabellen indizieren. Andernfalls können sich Indizes sogar negativ auf das Leistungsverhalten auswirken.

Wenn Sie bestimmte Abfragen seltener ausführen, zum Beispiel Monats- oder Quartalsberichte, und die Daten dafür in einer ungewöhnlichen Sortierfolge benötigen, sollten Sie den Index dafür erst unmittelbar vor der Anfertigung des Berichts anlegen. Erstellen Sie den Bericht und löschen Sie den Index danach wieder, damit die Datenbank in den langen Zeiten zwischen den Berichten nicht mit der Verwaltung des Index belastet wird.

Die Datenintegrität bewahren

Eine Datenbank ist nur dann nützlich, wenn Sie zweifelsfrei davon ausgehen können, dass Ihre Daten korrekt sind. Falsche Daten in Medizin-, Flug- oder Raumfahrtdatenbanken können sogar dazu führen, dass Menschen sterben. Falsche Daten in anderen Anwendungen haben möglicherweise nicht so ernste Konsequenzen, können aber beträchtlichen Schaden hervorrufen. Der Datenbankdesigner muss daher alles ihm Mögliche unternehmen, damit keine falschen Daten in die Datenbank gelangen können. Die *Integrität einer Datenbank* zu verwalten, bedeutet, dafür zu sorgen, dass alle Daten, die in ein Datenbanksystem eingegeben werden, den Einschränkungen entsprechen, die für das System eingerichtet worden sind. Wenn beispielsweise ein Datenfeld vom Typ `DATE` ist, sollte das DBMS jede Eingabe zurückweisen, die kein gültiges Datum ist.

Einige Probleme können nicht auf der Ebene der Datenbank gelöst werden. Der Anwendungsprogrammierer muss diese Probleme abfangen, ehe sie die Datenbank beschädigen können. Jeder, der mit der Datenbank arbeitet, muss die Gefahren kennen, die die Datenintegrität bedrohen, und entsprechende Abwehrmaßnahmen treffen.

Bei der Integrität von Datenbanken unterscheidet man verschiedene Arten – und eine Vielzahl von Problemen, die eine Vorliebe für Datenintegrität haben. In den folgenden Abschnitten beschreibe ich drei Arten der Integrität: *Entitätenintegrität*, *Domänenintegrität* und *referenzielle Integrität* – und die möglichen Probleme, mit denen Sie konfrontiert werden können.

Integrität von Entitäten

Jede Tabelle einer Datenbank entspricht einer Entität in der »wirklichen« Welt. Diese Dateneinheit kann physisch oder konzeptionell existieren, aber in gewisser Weise ist diese Existenz unabhängig von der Datenbank. Eine Tabelle weist eine *Entitätenintegrität* auf, wenn sie vollständig mit der Entität übereinstimmt, die sie abbildet. Um Entitätenintegrität zu gewährleisten, muss eine Tabelle über einen Primärschlüssel verfügen. Ein Primärschlüssel identifiziert jede Zeile in der Tabelle eindeutig. Ohne Primärschlüssel können Sie nicht sicher sein, ob die Zeile, die Sie erhalten, auch die ist, die Sie benötigen.

Um die Entitätenintegrität zu gewährleisten, müssen Sie festlegen, dass die Spalte oder die Spaltengruppe, die den Primärschlüssel bildet, `NOT NULL` ist, also keine Nullwerte enthält. Außerdem muss der Primärschlüssel die Einschränkung `UNIQUE` erfüllen. Bei einigen SQL-Implementierungen können Sie eine solche Einschränkung in die Tabellendefinition einfügen. Bei anderen Implementierungen müssen Sie die Einschränkung nachträglich zuweisen, nachdem Sie festgelegt haben, wie Daten in die Tabelle eingefügt, geändert oder aus der Tabelle gelöscht werden sollen.

Die beste Methode, um dafür zu sorgen, dass Ihr Primärschlüssel sowohl `NOT NULL` als auch `UNIQUE` ist, besteht darin, die Einschränkung `PRIMARY KEY` bei der Tabellendefinition festzulegen, wie es dieses Beispiel zeigt:

```
CREATE TABLE Kunde (
     KundenName         CHARACTER(30)     PRIMARY KEY,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefon            CHARACTER(13),
     Fax                CHARACTER(13),
     Kontaktperson      CHARACTER(30)
     ) ;
```

Eine Alternative besteht darin, `NOT NULL` in Verbindung mit `UNIQUE` anzugeben, wie das nächste Beispiel zeigt:

```
CREATE TABLE Kunde (
     KundenName         CHARACTER(30)      NOT NULL,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefon            CHARACTER(13),
     Fax                CHARACTER(13),
     Kontaktperson      CHARACTER(30),
     UNIQUE (KundenName) ) ;
```

Integrität von Domänen

Normalerweise können Sie nicht garantieren, dass ein bestimmtes Datenelement in einer Datenbank korrekt ist, aber Sie *können* wenigstens dafür sorgen, dass es einen gültigen Wert hat. Viele Datenelemente können nur eine begrenzte Anzahl möglicher Werte annehmen. Falls Sie einen ungültigen Wert eingeben, muss dies ein Fehler sein. So bestehen beispielsweise die USA aus 50 Bundesstaaten. Jeder dieser Bundesstaaten wird durch einen Code aus zwei Zeichen gekennzeichnet, den die US-amerikanische Post erkennt. Wenn Ihre Datenbank über eine Spalte `Bundesstaat` verfügt, können Sie für diese Spalte *Domänenintegrität* erzwingen, indem Sie fordern, dass jeder Eintrag in dieser Spalte aus einem der zulässigen Codes besteht. Falls jemand einen ungültigen Code eingibt, verstößt er gegen die Domänenintegrität. Wenn Sie Eingaben daraufhin überprüfen, ob sie die Domänenintegrität erfüllen, können Sie falsche Eingaben zurückweisen oder akzeptieren.

Die Frage der Domänenintegrität stellt sich, wenn Sie mit INSERT oder UPDATE neue Daten in eine Tabelle einfügen. Sie können mit der Anweisung CREATE DOMAIN die Domäne einer Spalte festlegen, ehe Sie die Spalte mit CREATE TABLE anlegen, wie dieses Beispiel zeigt:

```
CREATE DOMAIN LigaDom CHAR(6)
   CHECK (Bundesliga IN ('Erste', 'Zweite'));
CREATE TABLE Mannschaft (
   Mannschaftsname CHARACTER(20) NOT NULL,
   Bundesliga      LigaDom       NOT NULL
   ) ;
```

Die Domäne der Spalte Bundesliga enthält nur zwei gültige Werte: Erste und Zweite. Ihr DBMS wird eine Eingabe oder eine Aktualisierung der Tabelle Mannschaft nicht zulassen, wenn in der Spalte Bundesliga der Zeile, die Sie hinzufügen, nicht Erste oder Zweite steht.

Referenzielle Integrität

Selbst wenn jede einzelne Tabelle Ihres Systems sowohl über Entitäten- als auch über Domänenintegrität verfügt, können Probleme auftreten, wenn die Daten der miteinander verknüpften Tabellen inkonsistent sind. Bei den meisten korrekt entworfenen Datenbanken enthält jede Tabelle wenigstens eine Spalte, die eine Spalte in einer anderen Tabelle der Datenbank referenziert. Diese Referenzen sind für die Aufrechterhaltung der übergreifenden Datenbankintegrität wichtig. Sie machen aber auch sogenannte *Änderungsanomalien* möglich. *Änderungsanomalien* sind Probleme, die durch das Ändern von Daten in einer Zeile einer Datenbanktabelle entstehen können.

Probleme zwischen über- und untergeordneten Tabellen

Die Beziehungen zwischen Tabellen sind im Allgemeinen nicht bidirektional. Normalerweise hängt eine Tabelle von einer anderen ab. Nehmen Sie an, Sie haben eine Datenbank mit einer Tabelle Kunde und einer Tabelle Auftrag. Sie können natürlich einen Kunden in die Kundentabelle eintragen, ehe er einen Auftrag platziert. Aber Sie können keinen Auftrag in die Tabelle Auftrag eintragen, wenn es in der Kundentabelle keinen Kunden gibt, der diesen Auftrag erteilt. Die Tabelle Auftrag ist somit von der Kundentabelle abhängig. Diese Art von Beziehung wird häufig als *Eltern/Kind-Beziehung* (englisch *parent-child relationship*) bezeichnet, wobei Kunde die Eltern-Tabelle und Auftrag die Kind-Tabelle ist. Die Kind-Tabelle ist von der Eltern-Tabelle abhängig.

Im Allgemeinen erscheint der Primärschlüssel der Eltern-Tabelle als Spalte (oder Gruppe von Spalten) in der Kind-Tabelle und ist dort ein Fremdschlüssel. Ein Fremdschlüssel muss nicht eindeutig sein.

Änderungsanomalien können auf mehrere Arten entstehen. Beispielsweise zieht ein Kunde weg und Sie wollen ihn aus Ihrer Datenbank löschen. Wenn der Kunde bereits einige Aufträge erteilt hat, die in der Tabelle Auftrag gespeichert sind, kann das Löschen des Kunden in der Tabelle Kunde ein Problem darstellen. Danach hätten Sie Datensätze in der Kind-Tabelle

Auftrag, für die es in der Eltern-Tabelle Kunde keinen entsprechenden Datensatz mehr gibt. Ähnliche Probleme treten auf, wenn Sie einen Datensatz in eine Kind-Tabelle einfügen, ohne entsprechende Daten in der Eltern-Tabelle zu ergänzen.

Die Fremdschlüssel in allen Kind-Tabellen müssen alle Änderungen an den mit ihnen korrespondierenden Primärschlüsseln einer Eltern-Tabelle widerspiegeln, weil es andernfalls zu einer Änderungsanomalie kommt.

Kaskadierende Löschungen – vorsichtig verwenden

Sie können die meisten Probleme mit der referenziellen Integrität vermeiden, wenn Sie beim Ändern von Daten sorgfältig vorgehen. In einigen Fällen müssen Sie Daten *kaskadierend* löschen, also von der Eltern-Tabelle bis hin zu den Kind-Tabellen. Wenn Sie eine Zeile aus der Eltern-Tabelle löschen, müssen Sie auch alle Zeilen in den Kind-Tabellen löschen, deren Fremdschlüssel mit dem Primärschlüssel der gelöschten Zeile in der Eltern-Tabelle übereinstimmt. Betrachten Sie dazu das folgende Beispiel:

```
CREATE TABLE Kunde (
     KundenName         CHARACTER(30)     PRIMARY KEY,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25)     NOT NULL,
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefon            CHARACTER(13),
     Fax                CHARACTER(13),
     Kontaktperson      CHARACTER(30)
     ) ;

CREATE TABLE Test (
     Testname           CHARACTER(30)     PRIMARY KEY,
     Gebühr             CHARACTER(30)
     ) ;

CREATE TABLE Mitarbeiter (
     Mitarbeitername    CHARACTER(30)     PRIMARY KEY,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     TelefonPrivat      CHARACTER(13),
     Durchwahl          CHARACTER(4),
     Einstelldatum      DATE,
     Position           CHARACTER (10),
     StuGehProv         CHARACTER (1)
     ) ;
```

```
CREATE TABLE Auftrag (
     Auftragsnummer    INTEGER            PRIMARY KEY,
     KundenName        CHARACTER(30),
     BestellterTest    CHARACTER(30),
     Verkäufer         CHARACTER(30),
     Auftragsdatum     DATE,
     CONSTRAINT NameFK FOREIGN KEY (KundenName)
          REFERENCES Kunde (KundenName)
               ON DELETE CASCADE,
     CONSTRAINT TestFK FOREIGN KEY (BestellterTest)
          REFERENCES Test (Testname)
               ON DELETE CASCADE,
     CONSTRAINT VerkaufFK FOREIGN KEY (Verkäufer)
          REFERENCES Mitarbeiter (Mitarbeitername)
               ON DELETE CASCADE
     ) ;
```

Die Einschränkung `NameFK` legt `KundenName` als Fremdschlüssel fest, der die Spalte `Kunden Name` in der Kundentabelle referenziert. Wenn Sie eine Zeile in der Kundentabelle löschen, löschen Sie damit auch automatisch alle Zeilen in der Tabelle `Auftrag`, die in der Spalte `KundenName` denselben Wert enthalten wie die Spalte `KundenName` in der Kundentabelle. Der Löschvorgang kaskadiert von der Tabelle `Kunde` hinunter zu der Tabelle `Auftrag`. Dasselbe gilt für die Fremdschlüssel in der Tabelle `Auftrag`, die sich auf die Primärschlüssel der Tabellen `Test` beziehungsweise `Mitarbeiter` beziehen.

Alternative Methoden, um Aktualisierungsanomalien zu vermeiden

Statt den Löschvorgang zu kaskadieren, können Sie die Fremdschlüssel in den Kind-Tabellen auch auf einen Nullwert setzen. Betrachten Sie folgende Variante des vorangegangenen Beispiels:

```
CREATE TABLE Auftrag (
     Auftragsnummer    INTEGER            PRIMARY KEY,
     KundenName        CHARACTER(30),
     BestellterTest    CHARACTER(30),
     Verkäufer         CHARACTER(30),
     Auftragsdatum     DATE,
     CONSTRAINT NameFK FOREIGN KEY (KundenName)
          REFERENCES Kunde (KundenName),
     CONSTRAINT TestFK FOREIGN KEY (BestellterTest)
          REFERENCES Test (Testname),
     CONSTRAINT VerkaufFK FOREIGN KEY (Verkäufer)
          REFERENCES Mitarbeiter (Mitarbeitername)
               ON DELETE SET NULL
     ) ;
```

Die Einschränkung `VerkaufFK` legt die Spalte `Verkäufer` als Fremdschlüssel fest, der die Spalte `Mitarbeitername` in der Tabelle `Mitarbeiter` referenziert. Wenn ein Verkäufer die Firma verlässt, löschen Sie seine Zeile in der Tabelle `Mitarbeiter`. Seine Kunden werden irgendwann auf neue Verkäufer verteilt, aber im Moment führt das Löschen des Namens aus der Tabelle `Mitarbeiter` dazu, dass bei allen Aufträgen des Verkäufers in der Tabelle `Auftrag` in der Spalte `Verkäufer` ein Nullwert eingetragen wird.

Sie können inkonsistente Daten auch dadurch von Ihrer Datenbank fernhalten, dass Sie eine der folgenden Methoden einsetzen:

- ✔ **Lassen Sie das Einfügen neuer Datensätze in eine Kind-Tabelle nur dann zu, wenn eine entsprechende Zeile in ihrer Eltern-Tabelle existiert.** Sorgen Sie dafür, dass in einer Kind-Tabelle nur dann Zeilen eingefügt werden können, wenn dazu eine entsprechende Zeile in einer Eltern-Tabelle existiert. Sie verhindern damit sogenannte *verwaiste Zeilen*. Damit schützen Sie die Konsistenz Ihrer Datenbank.
- ✔ **Untersagen Sie Änderungen an Primärschlüsseln.** Wenn Sie dies erreichen, brauchen Sie sich wegen der Änderung der davon abhängigen Fremdschlüssel in anderen Tabellen keine Gedanken zu machen.

Und gerade als Sie dachten, alles wäre sicher ...

Die einzige Sache, auf der Sie bei einer Datenbank zählen können, ist (wie im Leben) der Wandel. Nicht wahr? Sie legen eine Datenbank an – komplett mit Tabellen, Einschränkungen und Datenzeilen über Datenzeilen. Und dann verlangt das Management, dass die Struktur geändert werden muss. Wie fügen Sie einer Tabelle, die schon einige Zeit existiert, eine Spalte hinzu? Wie löschen Sie eine Spalte, die Sie nicht mehr benötigen? Hier heißt die Hilfe SQL.

Einer Tabelle eine Spalte hinzufügen

Angenommen, Ihr Unternehmen wolle eine Richtlinie einführen, nach der für jeden Mitarbeiter am Geburtstag eine Party ausgerichtet werden solle. Damit der Partykoordinator genügend Vorlaufzeit für die Planung der Partys bekommt, müssen Sie eine Spalte für die Geburtsdaten in die Tabelle `Mitarbeiter` einfügen. Kein Problem für Sie. Verwenden Sie einfach die Anweisung `ALTER TABLE`:

```
ALTER TABLE Mitarbeiter
   ADD COLUMN Geburtstag DATE ;
```

Jetzt müssen Sie nur noch die Geburtstage in die einzelnen Zeilen eingeben, dann kann die Party losgehen.

Eine Spalte löschen

Angenommen, Ihr Unternehmen geriete aufgrund der schlechten Wirtschaftslage in eine finanzielle Schieflage und wäre nicht mehr in der Lage, diese aufwendigen Geburtstagspartys zu sponsern. Selbst wenn es der Wirtschaft schlecht geht, steigen die Honorare von DJs stän-

dig. Keine Partys heißt aber auch, dass die Spalte mit den Geburtstagen nicht mehr benötigt wird. Auch diese Situation können Sie mit der Anweisung `ALTER TABLE` meistern:

```
ALTER TABLE Mitarbeiter
   DROP COLUMN Geburtstag ;
```

Okay, solange es ging, hat es Spaß gemacht.

Potenzielle Problembereiche

Die Datenintegrität ist von vielen Seiten bedroht. Einige dieser Problembereiche existieren nur bei Datenbanken mit vielen Tabellen, während andere sogar Datenbanken betreffen, die nur eine einzige Tabelle haben. Sie sollten die Problembereiche kennen und die davon ausgehenden Gefahren minimieren.

Ungültige Eingabedaten

Die Quelldokumente oder Datendateien, aus denen Sie die Daten für Ihre Datenbank beziehen, können ungültige Daten enthalten. Diese Daten können unerwünschte oder ehemals korrekte und nun beschädigte Daten sein. Indem Sie den *Wertebereich prüfen*, können Sie feststellen, ob die Daten die Domänenintegrität erfüllen. Damit können Sie einige, aber sicher nicht alle Probleme abfangen. Spaltenwerte, die innerhalb des gültigen Wertebereichs liegen, werden akzeptiert, können aber dennoch falsch sein.

Benutzerfehler

Ihre Quelldaten können korrekt sein, aber die für die Erfassung verantwortlichen Benutzer geben die Daten falsch ein. Diese Art von Fehler gehört ebenfalls zur Kategorie der zuvor beschriebenen ungültigen Eingabedaten, nur dass hier eine andere Fehlerquelle vorliegt. Damit können einige der für den vorherigen Abschnitt gültigen Lösungen auch für dieses Problem benutzt werden. Eine Prüfung des Wertebereichs hilft zwar, sie ist aber eben nicht idiotensicher. Eine zweite Möglichkeit besteht darin, dieselben Daten von einer zweiten Person prüfen zu lassen. Dieser Ansatz ist teuer, weil Sie für die unabhängige Prüfung doppelt so viel Personal und Zeit brauchen. Aber in den Fällen, in denen die Datenintegrität von entscheidender Bedeutung ist, sind der zusätzliche Aufwand und die dadurch gestiegenen Kosten sicherlich gerechtfertigt.

Mechanische Fehler

Wenn mechanische Fehler auftreten, zum Beispiel ein Platten-Crash, können die Daten in den Tabellen zerstört werden. Dagegen können Sie sich durch regelmäßige Datensicherungen schützen.

Böswilligkeit (Sabotage)

Ziehen Sie die Möglichkeit in Betracht, dass jemand Ihre Daten böswillig und absichtlich beschädigen will. Ihre erste Schutzmaßnahme besteht darin, den Zugriff auf die Datenbank auf bestimmte Benutzer zu beschränken und diesen nur den Zugriff auf die Daten zu gewähren,

die sie für ihre Arbeit benötigen. Zweitens müssen Sie regelmäßige Datensicherungen vornehmen, die Sie an einem sicheren Ort aufbewahren. Außerdem sollten Sie die Sicherheitsmerkmale Ihres Systems regelmäßig überprüfen. Hier ein wenig paranoid zu sein, schadet nicht.

Datenredundanz

Die *Datenredundanz* ist beim hierarchischen Datenbankmodell ein großes Problem, das allerdings auch bei relationalen Datenbanken auftreten kann. Redundante Daten verschwenden Speicherplatz und verlangsamen die Verarbeitung. Darüber hinaus können sie zu ernsten Schäden an der Datenbank führen. Wenn Sie dieselben Daten in zwei verschiedenen Tabellen einer Datenbank speichern, kann ein Element geändert werden, während das entsprechende Element in der anderen Tabelle unverändert bleibt. Damit entsteht eine Diskrepanz zwischen den Daten und Sie werden mit ziemlicher Sicherheit vor dem Problem stehen, dass Sie dann schnell nicht mehr sicher sind, welches die richtige Version der Daten ist. Deshalb sollten Sie Datenredundanz so weit wie möglich vermeiden.

Obwohl eine gewisse Redundanz notwendig ist, weil Primärschlüssel einer Tabelle als Fremdschlüssel in anderen Tabellen dienen, sollte Ihre Datenbank darüber hinaus möglichst keine redundanten Daten enthalten.

Nachdem Sie die meisten Redundanzen aus Ihrem Datenbankentwurf entfernt haben, stellen Sie möglicherweise fest, dass das Leistungsverhalten jetzt nicht mehr akzeptabel ist. Deshalb werden Redundanzen oft gezielt benutzt, um die Verarbeitungsgeschwindigkeit zu verbessern. Im VetLab-Beispiel weiter vorne in diesem Kapitel enthält die Tabelle `Auftrag` nur den Namen des Kunden, um die Herkunft eines Auftrags zu identifizieren. Wenn Sie einen Auftrag bearbeiten, müssen Sie die Tabelle `Auftrag` mit der Tabelle `Kunde` verknüpfen, um die Adresse des Kunden zu ermitteln. Falls dadurch das Drucken der Aufträge zu lange dauert, können Sie die Kundenadresse redundant in der Tabelle `Auftrag` speichern. Diese Redundanz beschleunigt zwar den Druck der Aufträge, verlangsamt und verkompliziert dafür aber das Ändern der Kundenadresse.

Gewöhnlich wird eine Datenbank zunächst mit möglichst wenig Redundanz und einem hohen Grad an Normalisierung entworfen. Dann werden, wenn wichtige Anwendungen zu langsam laufen, gezielt Redundanzen hinzugefügt und Tabellen wieder entnormalisiert. Die Redundanzen, die Sie wieder in das System einführen, müssen einem bestimmten Zweck dienen. Weil Sie sowohl die Redundanzen als auch die damit möglicherweise verbundenen Probleme genau kennen, können Sie entsprechende Vorsichtsmaßnahmen treffen, um das Risiko so klein wie möglich zu halten. Weitere Daten zu diesem Thema finden Sie im späteren Abschnitt *Die Datenbank normalisieren*.

Die Kapazität des Datenbankverwaltungssystems überschreiten

Ein Datenbanksystem kann über Jahre hinweg problemlos arbeiten, und dann treten plötzlich Fehler auf, die zunehmend ernster werden. Dies kann ein Zeichen dafür sein, dass eine der Kapazitätsgrenzen des Systems erreicht ist. Dabei kann es sich um die höchstzulässige Anzahl von Zeilen handeln, über die eine Tabelle verfügen darf. Es gibt auch Beschränkungen für die Anzahl der Spalten, Einschränkungen und anderer Elemente. Vergleichen Sie die aktuelle Größe Ihrer Datenbank und deren Inhalt mit den Spezifikationen Ihres Datenbankverwaltungssystems. Wenn Sie an eine der möglichen Grenzen stoßen, sollten Sie darüber nachdenken, ein anderes System mit einer größeren Kapazität zu verwenden. Sie können aber auch ältere Daten, die nicht mehr von Belang sind, archivieren und anschließend aus der Datenbank löschen.

Einschränkungen

Weiter vorne in diesem Kapitel habe ich Einschränkungen als Mechanismen beschrieben, mit denen Sie sicherstellen können, dass die Daten, die Sie in eine Tabellenspalte eingeben, innerhalb der Domäne dieser Spalte liegen. Eine *Einschränkung* (englisch *constraint*) ist eine Anwendungsregel, die für die Daten definiert und von dem DBMS erzwungen wird. Nachdem Sie eine Datenbank angelegt haben, können Sie Einschränkungen (zum Beispiel `NOT NULL`) in die Tabellendefinition einfügen. Das DBMS sorgt dafür, dass Sie keine Transaktionen ausführen können, die eine Einschränkung verletzen.

Es gibt drei Arten von Einschränkungen:

- ✔ **Eine Spalteneinschränkung** legt eine Bedingung auf eine Spalte einer Tabelle.
- ✔ **Eine Tabelleneinschränkung** ist eine Einschränkung, die für eine ganze Tabelle gilt.
- ✔ **Eine Assertion** ist eine Einschränkung, die mehr als eine Tabelle betrifft.

Spalteneinschränkungen

Das folgende DDL-Beispiel zeigt eine Spalteneinschränkung:

```
CREATE TABLE Kunde (
     KundenName         CHARACTER(30)     NOT NULL,
     Anschrift1         CHARACTER(30),
     Anschrift2         CHARACTER(30),
     Ort                CHARACTER(25),
     Land               CHARACTER(2),
     Postleitzahl       CHARACTER(10),
     Telefon            CHARACTER(13),
     Fax                CHARACTER(13),
     Kontaktperson      CHARACTER(30)
     ) ;
```

Diese Anweisung definiert für die Spalte KundenName die Einschränkung NOT NULL, wodurch diese Spalte keine Nullwerte enthalten kann. UNIQUE ist eine weitere Einschränkung, die Sie auf eine Spalte anwenden können. Diese Einschränkung legt fest, dass jeder Wert in der betreffenden Spalte eindeutig sein muss. Die Einschränkung CHECK ist besonders nützlich, da sie jeden gültigen Ausdruck als Argument entgegennehmen kann. Beachten Sie das folgende Beispiel:

```
CREATE TABLE Test (
   Testname          CHARACTER(30)     NOT NULL,
   Gebühr            NUMERIC(6,2)
      CHECK (Gebühr >= 0.0
         AND Gebühr <= 200.0)
   ) ;
```

Die Standardgebühr für einen Test unserer Beispielfirma VetLab muss immer größer oder gleich null sein und darf nicht mehr als 200 Euro betragen. Die Klausel CHECK sorgt dafür, dass alle Einträge, die außerhalb des Wertebereichs `0 <= Gebühr <= 200` liegen, abgewiesen werden. Alternativ kann dieselbe Einschränkung folgendermaßen formuliert werden:

```
CHECK (Gebühr BETWEEN 0.0 AND 200.0)
```

Tabelleneinschränkungen

Die Einschränkung PRIMARY KEY legt fest, dass die Spalte, auf die die Einschränkung angewendet wird, als Primärschlüssel dienen soll. Diese Einschränkung ist somit eine Einschränkung, die für die ganze Tabelle gilt. Sie ist ein Äquivalent für die Kombination der Spalteneinschränkungen NOT NULL und UNIQUE. Die Einschränkung wird folgendermaßen in eine CREATE-Anweisung eingebunden:

```
CREATE TABLE Kunde (
     KundenName        CHARACTER(30)     PRIMARY KEY,
     Anschrift1        CHARACTER(30),
     Anschrift2        CHARACTER(30),
     Ort               CHARACTER(25),
     Land              CHARACTER(2),
     Postleitzahl      CHARACTER(10),
     Telefon           CHARACTER(13),
     Fax               CHARACTER(13),
     Kontaktperson     CHARACTER(30)
     ) ;
```

Benannte Einschränkungen wie etwa die NameFK-Einschränkung aus dem Beispiel in dem Abschnitt *Kaskadierende Löschungen – vorsichtig verwenden* weiter vorne können über eine zusätzliche Funktionalität verfügen. Angenommen, Sie wollten mehrere Tausend Interessenten in einem Batchlauf in Ihre Interessent-Tabelle einfügen. Sie verfügen über eine Datei, die hauptsächlich Interessenten aus den Vereinigten Staaten sowie verstreut einige Interessenten aus Kanada enthält. Eigentlich soll Ihre Interessent-Tabelle nur Einwohner der

USA enthalten; doch Sie wollen den Batchlauf nicht jedes Mal unterbrechen, wenn ein Datensatz einen Interessenten aus Kanada enthält. (Kanadische Postleitzahlen enthalten sowohl Buchstaben als auch Ziffern, während USA-Postleitzahlen nur Ziffern enthalten.) Dann können Sie festlegen, dass eine Einschränkung für die `Postleitzahl` solange nicht greifen soll, bis der Batchlauf abgeschlossen ist. Danach können Sie die Einschränkung wieder aktivieren.

Ursprünglich haben Sie Ihre `Interessent`-Tabelle mit dem folgenden `CREATE TABLE`-Befehl erstellt:

```
CREATE TABLE Interessent (
     KundenName        CHARACTER(30)     PRIMARY KEY,
     Anschrift1        CHARACTER(30),
     Anschrift2        CHARACTER(30),
     Ort               CHARACTER(25),
     Land              CHARACTER(2),
     Postleitzahl      CHARACTER(10),
     Telefon           CHARACTER(13),
     Fax               CHARACTER(13),
     Kontaktperson     CHARACTER(30)
     CONSTRAINT Zip CHECK (Postleitzahl BETWEEN 0 AND 99999)
     ) ;
```

Vor dem Batchlauf können Sie die Durchsetzung der `Zip`-Einschränkung deaktivieren:

```
ALTER TABLE PROSPECT
   CONSTRAINT Zip NOT ENFORCED;
```

Wenn der Batchlauf abgeschlossen ist, können Sie die Durchsetzung der `Zip`-Einschränkung wieder aktivieren:

```
ALTER TABLE PROSPECT
   CONSTRAINT Zip ENFORCED;
```

Dann können Sie alle Zeilen, die die Einschränkung nicht erfüllen, mit folgendem Befehl eliminieren:

```
DELETE FROM PROSPECT
   WHERE Postleitzahl NOT BETWEEN 0 AND 99999 ;
```

Assertionen

Eine *Assertion* (Zusicherung) legt eine Einschränkung fest, die für mehr als eine Tabelle gilt. Das folgende Beispiel verwendet eine Suchbedingung, die auf den Daten von zwei Tabellen basiert, um eine Assertion zu generieren:

```
CREATE TABLE Auftrag (
   Auftragsnummer    INTEGER           NOT NULL,
   KundenName        CHARACTER(30),
   BestellterTest    CHARACTER(30),
   Verkäufer         CHARACTER(30),
   Auftragsdatum     DATE
   ) ;

CREATE TABLE Ergebnis (
   Ergebnisnummer    INTEGER           NOT NULL,
   Auftragsnummer    INTEGER,
   Ergebnistext      CHARACTER(50),
   Ergebnisdatum     DATE,
   VorlEndg          CHARACTER(1)
   ) ;

CREATE ASSERTION
   CHECK (NOT EXISTS (SELECT * FROM Auftrag, Ergebnis
      WHERE Auftrag.Auftragsnummer = Ergebnis.Auftragsnummer
      AND Auftrag.Auftragsdatum > Ergebnis.Ergebnisdatum)) ;
```

Diese Assertion sorgt dafür, dass Sie keine Berichte über Ergebnisse erhalten, bevor Sie nicht den zugehörigen Test bestellt haben.

Die Datenbank normalisieren

Einige Methoden, Daten zu organisieren, sind besser als andere. Einige sind logischer. Einige sind einfacher. Einige sind besser dafür geeignet, Inkonsistenzen zu vermeiden, wenn Sie die Datenbank in Betrieb nehmen.

Änderungsanomalien und Normalformen

Verschiedene Probleme – sogenannte *Änderungsanomalien* – bedrohen eine Datenbank, wenn Sie sie nicht sauber strukturieren. Um diese Probleme zu vermeiden, können Sie die Datenbankstruktur *normalisieren*. Die Normalisierung besteht im Allgemeinen darin, eine Datenbanktabelle in zwei Tabellen zu zerlegen, die jeweils einfacher als das Original sind.

Änderungsanomalien tragen diesen Namen, weil sie beim Hinzufügen, Ändern oder Löschen von Daten einer Datenbanktabelle entstehen.

Um zu sehen, wie Änderungsanomalien auftreten können, betrachten Sie die Tabelle in Abbildung 5.2.

Verkauf		
Kundennummer	Produkt	Preis
1001	Waschmittel	12
1007	Zahnpasta	3
1010	Chlorbleiche	4
1024	Zahnpasta	3

Abbildung 5.2: Diese Verkaufstabelle führt zu Änderungsanomalien.

Nehmen wir an, dass Ihre Firma Haushalts- und Körperpflegeartikel vertreibt und dass Sie allen Kunden den gleichen Preis für jedes Produkt berechnen. Mit der Tabelle `Verkauf` verwalten Sie Ihren Geschäftsbetrieb. Nehmen wir weiter an, dass der Kunde 1001 aus der Stadt wegzieht und nicht länger bei Ihnen einkauft. Es ist Ihnen egal, was er in der Vergangenheit gekauft hat, weil er in Zukunft nicht mehr bei Ihnen einkaufen wird. Deshalb wollen Sie seine Zeile aus der Tabelle löschen. Wenn Sie dies tun, verlieren Sie nicht nur die Information, die Aufschluss darüber gibt, dass Kunde 1001 ein Waschmittel gekauft hat, sondern auch die Information darüber, dass dieses Waschmittel 12 Euro kostet. Diese Situation wird als *Löschanomalie* bezeichnet. Indem Sie eine Information löschen (dass der Kunde 1001 Waschmittel gekauft hat), löschen Sie unabsichtlich eine weitere Information (dass das Waschmittel 12 Euro kostet).

Wir können dieselbe Tabelle benutzen, um eine *Einfügeanomalie* zu zeigen. Angenommen, Sie möchten einen Deostick zum Preis von zwei Euro in Ihr Angebot aufnehmen. Sie können diese Daten erst dann in die Tabelle `Verkauf` aufnehmen, wenn Sie einen Kunden haben, der einen Deostick kauft.

Das Problem mit der Tabelle `Verkauf` in der Abbildung ist darauf zurückzuführen, dass diese Tabelle mit zu vielen Daten gleichzeitig arbeitet. Die Tabelle verzeichnet einerseits die Verkäufe (also welcher Kunde was kauft) und andererseits die Artikelpreise. Sie müssen die Tabelle `Verkauf` in zwei Tabellen zerlegen, die jeweils nur mit einem Thema zu tun haben (siehe Abbildung 5.3).

Abbildung 5.3 zeigt, dass die Tabelle `Verkauf` in zwei Tabellen aufgeteilt worden ist:

- ✔ Die Tabelle `Kauf`, die nur mit dem Thema der Käufe der Kunden zu tun hat
- ✔ Die Tabelle `Artikel`, die nur mit dem Thema der Artikel und ihrer Preise zu tun hat

Jetzt können Sie die Zeile des Kunden 1001 aus der Tabelle `KAUF` löschen, ohne die Information zu verlieren, dass das Waschmittel zwölf Euro kostet. Diese Information ist jetzt in der Tabelle `Artikel` gespeichert. Sie können nun auch den neuen Artikel, den Deostick, in die Tabelle `Artikel` aufnehmen, ohne dafür bereits einen Kunden zu haben. Daten über den Kauf werden ab sofort an anderer Stelle, nämlich in der Tabelle `Kauf`, gespeichert.

Kauf

Kundennummer	Produkt
1001	Waschmittel
1007	Zahnpasta
1010	Chlorbleiche
1024	Zahnpasta

Artikel

Produkt	Preis
Waschmittel	12
Zahnpasta	3
Chlorbleiche	4

Abbildung 5.3: Die Zerlegung der Tabelle `Verkauf` in zwei Tabellen

Dieser Prozess, eine Tabelle in mehrere Tabellen zu zerlegen, die jede für sich ein einziges Thema zum Inhalt hat, wird als *Normalisierung* bezeichnet. Eine Normalisierung löst einige Probleme, aber selten alle. Oft müssen Sie mehrere Normalisierungen nacheinander durchführen, um die Tabellen so zu zerlegen, dass jede nur noch ein Thema zum Inhalt hat. Jede Tabelle in einer Datenbank sollte ein – und nur ein – Hauptthema zum Inhalt haben. Manchmal ist es jedoch nicht so einfach festzustellen, dass eine Tabelle mehr als einen Gegenstand behandelt.

Man kann Tabellen danach klassifizieren, welchen Änderungsanomalien sie unterworfen sind. E.F. Codd hat im Jahre 1970 das relationale Modell erstmalig beschrieben. Dabei hat er drei Quellen für Änderungsanomalien identifiziert und die erste, die zweite und die dritte *Normalform* (1NF, 2NF, 3NF) als Abhilfen gegen diese Anomalien formuliert. In den folgenden Jahren haben Codd und andere weitere Arten von Anomalien entdeckt und zusätzliche Normalformen festgelegt, um mit diesen Anomalien umzugehen. Die *Boyce-Codd-Normalform* (BCNF), die vierte Normalform (4NF) und die fünfte Normalform (5NF) brachten jeweils einen höheren Schutz gegen Änderungsanomalien. Aber erst 1981 beschrieb Ronald Fagin in einer Arbeit die *Domain-Key-Normalform* (*DK/NF*). Mit dieser letzten Normalform können Sie *garantieren*, dass eine Tabelle frei von Änderungsanomalien ist.

Die Normalformen sind in dem Sinne *verschachtelt*, dass eine Tabelle, die in der 2NF vorliegt, automatisch auch in der 1NF vorliegt. Auf ähnliche Weise liegen Tabellen in der 3NF automatisch auch in der 2NF vor und so weiter. Für die meisten praktischen Anwendungen genügt es normalerweise, die Datenbank in die 3NF zu bringen, um einen hohen Grad von Integrität zu erreichen. Wenn Sie jedoch absolut sicher sein wollen, dass die Integrität hundertprozentig gewahrt bleibt, müssen Sie die Datenbank in die DK/NF bringen (mehr darüber später in diesem Kapitel).

Nachdem Sie eine Datenbank so weit wie möglich normalisiert haben, können Sie gezielt einzelne Komponenten wieder denormalisieren, um das Leistungsverhalten der Datenbank zu verbessern. Sie sollten sich dann aber darüber im Klaren sein, welche Anomalien dadurch ermöglicht werden.

Erste Normalform

Um in der ersten Normalform (1NF) zu sein, muss eine Tabelle folgende Eigenschaften haben:

- ✔ Die Tabelle ist zweidimensional und hat Zeilen und Spalten.
- ✔ Jede Zeile enthält Daten, die zu einer Sache oder einem Teil einer Sache gehören.
- ✔ Jede Spalte enthält Daten eines einzigen Attributs der Sache, die sie beschreibt.
- ✔ Jede Zelle (Schnittpunkt einer Zeile und einer Spalte) enthält nur einen einzigen Wert.
- ✔ Alle Einträge einer Spalte sind von derselben Art. Wenn beispielsweise in einer Zeile der Eintrag einer Spalte ein Mitarbeitername ist, müssen alle anderen Zeilen in dieser Spalte ebenfalls Mitarbeiternamen enthalten.
- ✔ Jede Spalte hat einen eindeutigen Namen.
- ✔ Keine zwei Zeilen sind identisch (das heißt, jede Zeile ist eindeutig).
- ✔ Die Reihenfolge der Spalten und die Reihenfolge der Zeilen spielt keine Rolle.

Eine Tabelle (Relation) in der ersten Normalform ist gegen einige Arten von Änderungsanomalien immun, aber nicht gegen alle. Die Tabelle `Verkauf` in Abbildung 5.2 liegt in der ersten Normalform vor und ist – wie gezeigt – anfällig für Lösch- und Einfügeanomalien. Die erste Normalform ist für viele Anwendungen nicht zuverlässig genug.

Zweite Normalform

Um die zweite Normalform verstehen zu können, müssen Sie wissen, was funktionelle Abhängigkeit bedeutet. Eine *funktionelle Abhängigkeit* ist eine Beziehung zwischen Attributen. Ein Attribut ist funktionell von einem anderen abhängig, wenn der Wert des zweiten Attributs den Wert des ersten Attributs bestimmt. Wenn Sie den Wert des zweiten Attributs kennen, können Sie den Wert des ersten Attributs bestimmen.

Angenommen, eine Tabelle habe die Attribute (Spalten) `Gebühr`, `Testanzahl` und `Gesamtgebühr`, die durch die folgende Gleichung in Beziehung stehen:

```
Gesamtgebühr = Gebühr * Testanzahl
```

Das Attribut `Gesamtgebühr` ist funktionell sowohl von dem Attribut `Gebühr` als auch von dem Attribut `Testanzahl` abhängig. Kennen Sie die Werte von `Gebühr` und `Testanzahl`, können Sie den Wert von `Gesamtgebühr` berechnen.

Jede Tabelle in der ersten Normalform muss einen eindeutigen Primärschlüssel besitzen. Dieser Schlüssel kann aus einer oder mehreren Spalten bestehen. Ein Schlüssel, der aus mehr als einer Spalte besteht, wird als *zusammengesetzter Schlüssel* bezeichnet. Um in der zweiten Normalform (2NF) zu sein, müssen alle Nicht-Schlüssel-Attribute (Spalten) vom gesamten Schlüssel abhängen. Deshalb liegt jede Relation, die in der 1NF vorliegt und einen Schlüssel besitzt, der nur aus einem einzigen Attribut besteht, automatisch auch in der zweiten Normalform vor. Wenn eine Relation einen zusammengesetzten Schlüssel verwendet, müssen alle Nicht-Schlüssel-Attribute von allen Komponenten dieses Schlüssels abhängen. Wenn eine

Tabelle Nicht-Schlüssel-Attribute enthält, die nicht von allen Komponenten des Schlüssels abhängig sind, müssen Sie die Tabelle so in zwei oder mehr Tabellen zerlegen, dass in den neuen Tabellen alle Nicht-Schlüssel-Attribute von allen Komponenten des Primärschlüssels abhängig sind.

Alle Klarheiten beseitigt? Ein Beispiel soll dies verdeutlichen. Betrachten Sie eine Tabelle wie die Tabelle `Verkauf` aus Abbildung 5.2. Statt nur einen einzigen Kauf pro Kunde zu speichern, fügen Sie jedes Mal eine Zeile in die Tabelle ein, wenn ein Kunde einen bestimmten Artikel zum ersten Mal kauft. Ein weiterer Unterschied soll darin bestehen, dass spezielle Kunden – die mit einer Kundennummer von 1001 bis 1009 – einen Sonderrabatt auf den normalen Preis erhalten. Abbildung 5.4 zeigt einige Zeilen dieser Tabelle.

Verkauf2

Kundennummer	Produkt	Preis
1001	Waschmittel	11.00
1007	Zahnpasta	2.70
1010	Chlorbleiche	4.00
1024	Zahnpasta	3.00
1010	Waschmittel	12.00
1001	Zahnpasta	2.70

Abbildung 5.4: In der Tabelle `Verkauf2` bilden die Spalten `Kundennummer` und `Produkt` einen zusammengesetzten Schlüssel.

Beachten Sie, dass in dieser Tabelle die Spalte `Kundennummer` eine Zeile nicht eindeutig identifiziert. Es gibt zwei Zeilen, in denen die Spalte `Kundennummer` den Wert 1001 hat, und zwei mit dem Wert 1010. Die Zeilen werden jedoch durch die Kombination der Spalten `Kunden nummer` und `Produkt` eindeutig identifiziert. Diese beiden Spalten bilden einen zusammengesetzten Schlüssel.

Wenn kein Kunde einen Rabatt bekäme, würde die Tabelle nicht in der zweiten Normalform vorliegen, da das Nicht-Schlüssel-Attribut `Preis` in diesem Fall nur von einem Teil des Schlüssels `Produkt` abhängig ist. Da jedoch einige Kunden einen Rabatt erhalten, ist `Preis` sowohl von der Spalte `Kundennummer` als auch von der Spalte `Produkt` abhängig, wodurch die Tabelle in der zweiten Normalform vorliegt.

Dritte Normalform

Auch Tabellen in der zweiten Normalform sind noch durch bestimmte Arten von Änderungsanomalien gefährdet. Diese Anomalien können durch transitive Abhängigkeiten verursacht werden.

Eine *transitive Abhängigkeit* tritt auf, wenn ein Attribut von einem zweiten Attribut abhängig ist, das wiederum von einem dritten Attribut abhängt. Löschungen in einer Tabelle mit einer solchen Abhängigkeit können dazu führen, dass ungewollt Daten verloren gehen. Eine Relation in der dritten Normalform ist eine Relation in der zweiten Normalform, die keine transitive Abhängigkeit hat.

Betrachten Sie noch einmal die Tabelle `Verkauf` (aus Abbildung 5.2), die sich in der ersten Normalform befindet. Solange Sie die Einträge so beschränken, dass es für jede `Kundennummer` nur eine Zeile gibt, verfügen Sie über einen Primärschlüssel, der aus einem einzigen Attribut besteht, und die Tabelle liegt in der zweiten Normalform vor. Die Tabelle ist jedoch durch Anomalien gefährdet. Was passiert beispielsweise, wenn der Kunde 1010 mit seiner Chlorbleiche unzufrieden ist, den Artikel zurückgibt und eine Erstattung seiner Zahlung verlangt? Sie wollen die dritte Zeile aus der Tabelle löschen, in der der Verkauf der Chlorbleiche an den Kunden 1010 gespeichert ist, und stehen vor einem Problem: Wenn Sie diese Zeile löschen, verlieren Sie auch die Information, dass Chlorbleiche vier Euro kostet. Dies ist ein Beispiel für eine transitive Abhängigkeit. Die Spalte `Preis` ist von der Spalte `Produkt` abhängig, die ihrerseits von der Spalte `Kundennummer`, dem Primärschlüssel, abhängig ist.

Wenn Sie die Tabelle `Verkauf` in zwei Tabellen zerlegen, wird das Problem der transitiven Abhängigkeit gelöst. Die beiden Tabellen in Abbildung 5.3, `Kauf` und `Artikel`, bilden eine Datenbank, die in der dritten Normalform vorliegt.

Domain-Key-Normalform (DK/NF)

Bei einer Datenbank in der dritten Normalform sind die meisten, aber nicht alle Möglichkeiten für Änderungsanomalien beseitigt. Die Normalformen jenseits der dritten dienen dazu, auch die restlichen Probleme zu verhindern. Die Boyce-Codd-Normalform (BCNF), die vierte Normalform (4NF) und die fünfte Normalform (5NF) sind Beispiele solcher Formen. Jede dieser Formen eliminiert eine Möglichkeit für den Eintritt von Änderungsanomalien, aber garantiert nicht, dass alle möglichen Änderungsanomalien ausgeschlossen sind. Diese Garantie wird erst durch die Domain-Key-Normalform (DK/NF) gegeben.

Eine Relation liegt in der Domain-Key-Normalform (DK/NF) vor, wenn jede Einschränkung der Relation logisch aus der Definition der Schlüssel *(Key)* und Domänen folgt. Eine *Einschränkung* ist gemäß dieser Definition eine Regel, die so genau ist, dass Sie feststellen können, ob sie wahr ist oder nicht. Ein *Schlüssel* ist ein eindeutiger Bezeichner einer Zeile in einer Tabelle. Eine *Domäne* ist die Menge der zulässigen Werte eines Attributs.

Betrachten Sie noch einmal die Datenbank aus Abbildung 5.2, die in der 1NF vorliegt, um herauszufinden, wie Sie diese Datenbank in die DK/NF bringen können:

Tabelle:	`Verkauf` (`Kundennummer`, `Produkt`, `Preis`)
Schlüssel:	`Kundennummer`
Einschränkungen:	1. `Kundennummer` bestimmt `Produkt` 2. `Produkt` bestimmt `Preis` 3. `Kundennummer` muss eine Ganzzahl > 1000 sein.

Um die dritte Einschränkung in Kraft zu setzen, nämlich dass die `Kundennummer` eine Ganzzahl größer als 1000 sein muss, können Sie einfach eine Domäne für `Kundennummer` definieren, die diese Einschränkung enthält. Dadurch wird die Einschränkung eine logische Folge der Domäne der Spalte `Kundennummer`. Weil die Spalte `Produkt` von der Spalte `Kundennummer` abhängig und die Spalte `Kundennummer` ein Schlüssel ist, gibt es kein Problem mit der ersten Einschränkung, die logisch aus der Definition des Schlüssels folgt. Dagegen stellt die zweite Einschränkung ein Problem dar. Die Spalte `Preis` ist von der Spalte `Produkt` abhängig (das heißt, sie ist eine logische Folge davon) und die Spalte `Produkt` ist kein Schlüssel. Die Lösung besteht darin, die Tabelle `Verkauf` in zwei Tabellen zu zerlegen, von denen die eine die Spalte `Kundennummer` als Schlüssel und die andere die Spalte `Produkt` als Schlüssel benutzt. Diese Lösung wird in Abbildung 5.3 gezeigt. Die Datenbank in dieser Abbildung ist also nicht nur in der 3NF, sondern auch in der DK/NF.

Entwerfen Sie Ihre Datenbanken möglichst so, dass sie in der DK/NF vorliegen. Wenn Sie dieses Ziel erreichen, sorgt die Durchsetzung der Restriktionen der Schlüssel und Domänen dafür, dass alle Einschränkungen erfüllt werden. Änderungsanomalien sind dann nicht mehr möglich. Wenn die Struktur einer Datenbank so beschaffen ist, dass Sie sie nicht in die Domain-Key-Normalform bringen können, müssen Sie die Einschränkungen in das Anwendungsprogramm einbauen, das die Datenbank benutzt. Denn die Datenbank selbst kann dann nicht garantieren, dass die Einschränkungen erfüllt werden.

Abnorme Formen

Manchmal zahlt es sich aus, abnorm zu sein. Manchmal können Sie sich in der Normalisierung verrennen und Ihre Anstrengungen zu weit treiben. Sie können eine Datenbank in so viele Tabellen zerlegen, dass sie unhandlich und ineffizient wird. Ihr Leistungsverhalten geht in den Keller. Oft finden Sie die optimale Struktur, wenn Sie die Datenbank etwas denormalisieren. Tatsächlich ist kaum eine Datenbank in der Praxis bis zur DK/NF normalisiert. Sie sollten Ihre Datenbanken jedoch so weit wie möglich normalisieren, um das Risiko zu vermindern, dass die Daten aufgrund von Änderungsanomalien beschädigt werden.

Wenn Sie die Datenbank so weit wie möglich normalisiert haben, sollten Sie probeweise einige Daten abrufen. Lässt das Leistungsverhalten zu wünschen übrig, prüfen Sie, ob nicht eine gezielte Denormalisierung des Datenbankentwurfs zu mehr Leistung führt, ohne dabei die Integrität zu opfern. Indem Sie *vorsichtig* Redundanzen an strategischen Stellen hinzufügen und damit die Datenbank denormalisieren, können Sie erreichen, dass eine Datenbank sowohl effizient als auch sicher vor Anomalien ist.

Teil III

Daten speichern und abrufen

In diesem Teil ...

- ✔ Daten verwalten
- ✔ Die Zeit verfolgen
- ✔ Werte verarbeiten
- ✔ Abfragen erstellen

Daten einer Datenbank bearbeiten

6

In diesem Kapitel

- Mit Daten arbeiten
- Daten aus einer oder mehreren Tabellen abrufen
- Ausgewählte Daten aus einer oder mehreren Tabellen anzeigen
- Daten in Tabellen und Sichten aktualisieren
- Eine neue Zeile in eine Tabelle einfügen
- Daten einer Tabellenzeile ändern
- Eine Tabellenzeile löschen

In den Kapiteln 3 und 4 haben Sie erfahren, dass der Entwurf einer soliden Datenbankstruktur für die Integrität der Daten sehr wichtig ist. Ihr eigentliches Interesse gilt jedoch nicht der Struktur der Datenbank, sondern den Daten, die in ihr gespeichert sind. Für die Arbeit mit Daten stehen Ihnen vier Funktionen zur Verfügung: Sie können Daten in Tabellen einfügen, aus Tabellen abrufen und anzeigen, in Tabellen ändern und aus Tabellen löschen.

Im Prinzip ist das Bearbeiten einer Datenbank recht einfach. Es ist nicht schwer zu verstehen, wie Daten einer Tabelle hinzugefügt werden. Man kann Daten entweder einzeln Zeile für Zeile oder mehrere Zeilen gleichzeitig einfügen. Tabellenzeilen zu ändern, zu löschen oder abzurufen, ist ebenfalls nicht schwer. Die Herausforderung besteht darin, die entsprechenden Zeilen auszuwählen. Die gesuchten Daten sind häufig in einer riesigen Menge anderer Daten verborgen. Glücklicherweise können Sie mit der SQL-Anweisung `SELECT` dem Computer sehr genau sagen, was Sie suchen, und der erledigt den Job für Sie. Also eigentlich ist das Bearbeiten einer Datenbank ein Kinderspiel. Hinzufügen, Ändern, Löschen und Abrufen – alles ist so einfach! Na, vielleicht ist das ein wenig übertrieben. Lassen Sie uns mit etwas ganz Leichtem anfangen, mit einem einfachen Datenabruf.

Daten abrufen

Die häufigste Arbeit, die Benutzer mit Daten erledigen, ist das Abrufen ausgewählter Daten. Sie können die Inhalte einer einzigen Zeile abrufen, die sich inmitten Tausender anderer Zeilen befindet. Sie können auch alle Zeilen abrufen, die eine bestimmte Bedingung oder eine Kombination von Bedingungen erfüllen. Sie können sogar alle Zeilen einer Tabelle abrufen. In SQL werden all diese Aufgaben mit einer einzigen Anweisung, `SELECT`, gelöst.

Die einfachste Form der Anweisung `SELECT` gibt die Daten aller Zeilen einer angegebenen Tabelle zurück. Verwenden Sie dazu die folgende Syntax:

```
SELECT * FROM Kunde ;
```

Das Sternchen (*) ist ein Platzhalter, der *alles* bedeutet. In diesem Beispiel ist dieses Zeichen die Kurzform einer Liste mit allen Spaltennamen der Tabelle Kunde. Das Ergebnis dieser Anweisung zeigt alle Daten in allen Zeilen und Spalten der Tabelle Kunde auf dem Bildschirm an.

Die Anweisung SELECT kann sehr viel komplizierter als in diesem Beispiel aufgebaut sein. Tatsächlich kann sie so kompliziert sein, dass sie praktisch nicht mehr zu entziffern ist. Diese Komplexität wird durch sogenannte *Klauseln* verursacht, die Sie in fast beliebiger Zahl in die Anweisung einfügen können. In Kapitel 9 beschreibe ich diese Klauseln im Detail. In diesem Kapitel hier gehe ich kurz auf die Klausel WHERE ein, die dazu dient, die Anzahl an Zeilen zu begrenzen, die eine SELECT-Anweisung zurückliefert.

Die Anweisung SELECT mit einer WHERE-Klausel hat die folgende allgemeine Form:

```
SELECT Spaltenliste FROM Tabellenname
   WHERE Bedingung ;
```

Spaltenliste gibt die Spalten der Tabelle an, die Sie anzeigen lassen wollen. Die FROM-Klausel gibt an, aus welcher Tabelle Sie sich Daten liefern lassen wollen. Die WHERE-Klausel schließt die Zeilen von der Anzeige aus, die eine bestimmte Bedingung nicht erfüllen. Diese Bedingung kann einfach (beispielsweise WHERE Bundesland = 'NRW') oder zusammengesetzt sein (beispielsweise WHERE Bundesland = 'NRW' AND Umsatz > 500000).

Das folgende Beispiel zeigt eine SELECT-Anweisung mit einer zusammengesetzten Bedingung:

```
SELECT Kundenname, KundeTelefon FROM Kunde
   WHERE Bundesland = 'NRW'
   AND Umsatz > 500000 ;
```

Diese Anweisung gibt die Namen und Telefonnummern aller Kunden zurück, die in Nordrhein-Westfalen ihren Sitz haben und einen Umsatz von über 500.000 Euro ausweisen. Das Schlüsselwort AND besagt, dass eine Zeile beide Bedingungen erfüllen muss:

Bundesland = 'NRW' und Umsatz > 500000.

Eine Sicht erstellen

Eine korrekt entworfene Datenbank, die entsprechend normalisiert ist (siehe Kapitel 5), weist eine Struktur auf, die die Integrität der Daten so weit wie möglich schützt. Diese Struktur stellt jedoch oft nicht die beste Methode dar, um die Daten zu betrachten. Mehrere Anwendungen können auf dieselben Daten zugreifen, aber verschiedene Schwerpunkte darauf legen. Eines der mächtigsten Features von SQL ist seine Fähigkeit, Daten in sogenannten *Sichten* (englisch *Views*) darzustellen, deren Struktur von den Strukturen abweichen, in denen Datenbanktabellen Daten ablegen. Die Tabellen, die Sie als Quellen für die Spalten und Zeilen in einer Sicht benutzen, nennt man *Basistabellen*. In Kapitel 3 habe ich Sichten bereits im Zusammenhang mit der Datendefinitionssprache (DDL) erwähnt. In diesem Kapitel liegt der Schwerpunkt auf Sichten im Zusammenhang mit dem Abruf und der Bearbeitung von Daten.

Die Anweisung SELECT gibt ihr Ergebnis immer in Form einer virtuellen Tabelle zurück. Eine *Sicht* ist eine besondere Art von virtueller Tabelle. Sie können eine Sicht von anderen virtuellen Tabellen unterscheiden, da die Metadaten einer Datenbank die Definition einer Sicht enthalten. Dieser Unterschied sorgt dafür, dass Sichten einen Grad an Dauerhaftigkeit besitzen, den andere virtuelle Tabellen nicht haben. Sie können Sichten wie echte Tabellen bearbeiten.

Betrachten Sie die VetLab-Datenbank aus Kapitel 5. Diese Datenbank enthält die fünf Tabellen Kunde, Test, Mitarbeiter, Auftrag und Ergebnis. Der Marketingleiter möchte wissen, aus welchem Land die Aufträge der Firma stammen. Ein Teil dieser Daten ist in der Tabelle Kunde und ein anderer Teil in der Tabelle Auftrag gespeichert. Der Chef der Qualitätskontrolle möchte das Auftragsdatum eines Tests mit dem Datum des Endergebnisses vergleichen. Dieser Vergleich benötigt Daten der Tabellen Auftrag und Ergebnis. Sie können Sichten erstellen, die diese unterschiedlichen Informationswünsche problemlos erfüllen.

FROM-Tabellen

Für den Marketingleiter definieren Sie die Sicht, die in Abbildung 6.1 gezeigt wird.

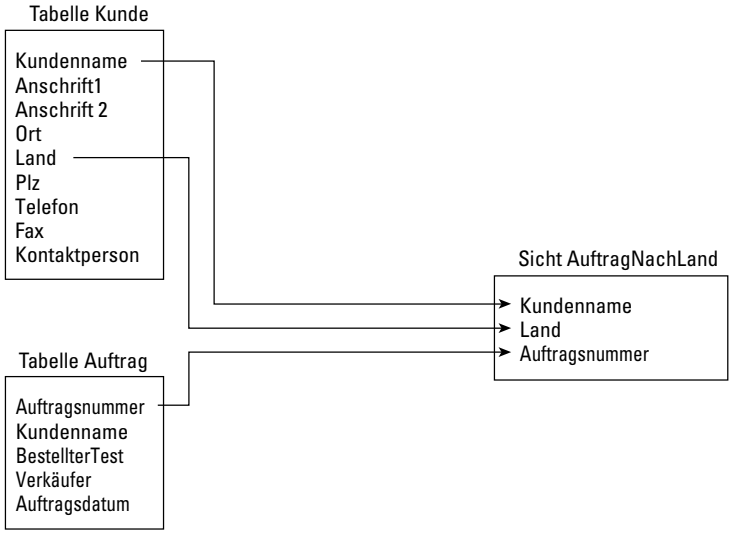


Abbildung 6.1: Die Sicht AuftragNachLand für den Marketingleiter

Die folgende Anweisung erzeugt diese Sicht:

```
CREATE VIEW AuftragNachLand
      (Kundenname, Land, Auftragsnummer)
   AS SELECT Kunde.Kundenname, Land, Auftragsnummer
   FROM Kunde, Auftrag
   WHERE Kunde.Kundenname = Auftrag.Kundenname ;
```

Die neue Sicht verfügt über die drei Spalten Kundenname, Land und Auftragsnummer. Kundenname erscheint sowohl in der Tabelle Kunde als auch in der Tabelle Auftrag und dient dazu, beide Tabellen zu verknüpfen. Die neue Sicht holt die Land-Daten aus der Tabelle Kunde und die Auftragsnummer aus der Tabelle Auftrag. In diesem Beispiel geben Sie explizit die Namen der Spalten an, die in der neuen Sicht erscheinen sollen.

Beachten Sie, dass ich vor Kundenname den Namen der Tabelle gesetzt habe, aus der die Daten stammen, nicht aber bei Land und Auftragsnummer, weil Land nur in der Tabelle Kunde und die Auftragsnummer nur in der Tabelle Auftrag vorkommt, Kundenname jedoch in beiden Tabellen.

Diese Deklaration ist nicht notwendig, wenn die Namen mit den Namen der entsprechenden Spalten in den Quelltabellen übereinstimmen. Das Beispiel im folgenden Abschnitt zeigt eine ähnliche CREATE VIEW-Anweisung, bei der die Spaltennamen der Sicht implizit statt explizit angegeben werden.

Mit einer Auswahlbedingung

Der Chef der Qualitätskontrolle benötigt eine andere Sicht als der Marketingmanager. Diese Sicht ist in Abbildung 6.2 zu sehen.

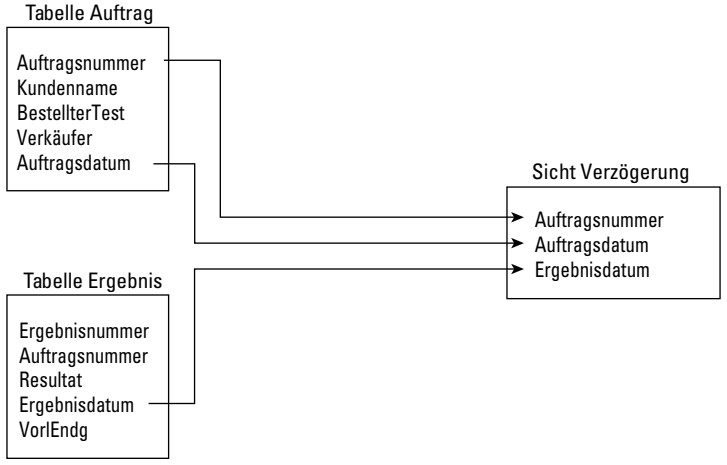


Abbildung 6.2: Die Sicht Verzögerung für den Chef der Qualitätskontrolle

Sie wird folgendermaßen definiert:

```
CREATE VIEW Verzögerung
   AS SELECT Auftrag.Auftragsnummer, Auftragsdatum, Ergebnisdatum
   FROM Auftrag, Ergebnis
   WHERE Auftrag.Auftragsnummer = Ergebnis.Auftragsnummer
   AND Ergebnis.VorlEndg = 'E' ;
```

Diese Sicht enthält das Auftragsdatum der Tabelle Auftrag und das Ergebnisdatum der Tabelle Ergebnis. Nur Zeilen, die in der Spalte VorlEndg der Tabelle Ergebnis ein E enthalten, werden in die Sicht aufgenommen.

Mit einem geänderten Attribut

Die SELECT-Klauseln der beiden vorangegangenen Beispiele enthalten nur Spaltennamen. Sie können in diesen Klauseln jedoch auch Ausdrücke verwenden. Angenommen, der Besitzer von VetLab hat Geburtstag und möchte aus diesem Grund allen Kunden zur Feier des Tages einen zehnprozentigen Rabatt gewähren. Er erstellt eine Sicht, die auf den Tabellen AUFTRAG *und* TEST basiert:

```
CREATE VIEW Geburtstag
   (Kundenname, Test, Auftragsdatum, Geburtstagsrabatt)
   AS SELECT Kundenname, BestellterTest, Auftragsdatum,
      Gebühr * .9
   FROM Auftrag, Test
   WHERE BestellterTest = Testname ;
```

Beachten Sie, dass die Spalte Test in der Sicht Geburtstag der Spalte BestellterTest in der Tabelle Auftrag entspricht, die ihrerseits der Spalte Testname in der Tabelle Test entspricht (siehe Abbildung 6.3).

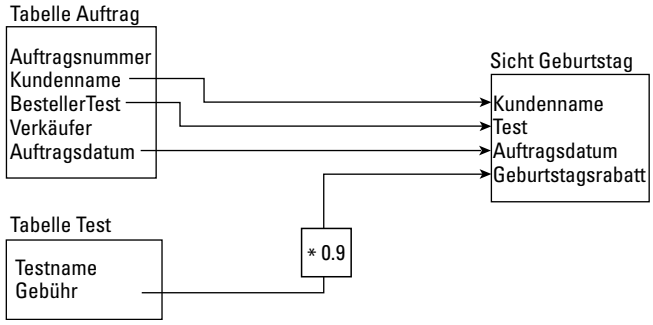


Abbildung 6.3: Die Sicht zur Ermittlung des Geburtstagsrabatts

Sie können, wie das vorherige Beispiel zeigt, eine Sicht erstellen, die auf einer oder auf mehreren Tabellen basiert. Wenn Sie nicht alle Spalten oder Zeilen einer Tabelle benötigen, definieren Sie eine Sicht, die die unerwünschten Daten filtert, und arbeiten dann mit dieser Sicht statt mit der Originaltabelle. Diese Lösung sorgt dafür, dass die Benutzer nur die Daten sehen, die sie für ihre Arbeit benötigen.

Sichten aktualisieren

Nachdem Sie eine Tabelle erstellt haben, können Sie darin problemlos Daten einfügen, ändern und löschen. Sichten bieten nicht notwendigerweise dieselben Möglichkeiten. Wenn Sie eine Sicht ändern, aktualisieren Sie tatsächlich die zugrunde liegende Tabelle. Einige der potenziellen Probleme, die während der Aktualisierung von Sichten auftreten können, sind:

- ✔ **Einige Sichten beziehen ihre Komponenten aus zwei oder mehr Tabellen.** Wenn Sie eine solche Sicht ändern, stehen Sie vor der Frage, welche der zugrunde liegenden Tabellen aktualisiert wird.
- ✔ **Die SELECT-Liste einer Sicht kann einen Ausdruck enthalten.** Weil sich Ausdrücke nicht unmittelbar in den Zeilen einer Tabelle abbilden, weiß Ihr DBMS nicht, wie ein Ausdruck aktualisiert werden kann.

Stellen Sie sich vor, dass Sie folgende Sicht definieren:

```
CREATE VIEW (Gesamtgehalt, Auszahlung)
   AS SELECT Name, Gehalt+Provision AS Auszahlung
   FROM Mitarbeiter ;
```

Sie könnten jetzt natürlich der Meinung sein, dass es möglich ist, AUSZAHLUNG mit der folgenden Anweisung zu ändern:

```
UPDATE Gesamtgehalt SET Auszahlung = Auszahlung + 100 ;
```

Unglücklicherweise macht diese Vorgehensweise keinen Sinn, weil die zugrunde liegende Tabelle keine Spalte Auszahlung besitzt. Sie können nichts aktualisieren, was nicht in der Basistabelle enthalten ist.

Denken Sie an folgende Regel, wenn Sie eine Sicht ändern wollen: Sie können eine Spalte einer Sicht nicht ändern und aktualisieren, wenn diese Spalte nicht mit einer Spalte der zugrunde liegenden Tabelle übereinstimmt.

Neue Daten hinzufügen

Anfangs ist jede Datenbanktabelle leer. Nachdem Sie eine Tabelle mit SQL oder einem RAD-Werkzeug erstellt haben, ist sie nur eine leere Struktur ohne Daten. Damit die Tabelle Nutzen bringt, müssen Sie sie mit Daten füllen. Vielleicht haben Sie Glück und Ihre Daten liegen bereits in digitaler Form vor. Auf jeden Fall existieren Daten normalerweise in einer der folgenden Formen:

- ✔ **Noch nicht digital kompiliert.** Falls Ihre Daten nicht bereits in digitaler Form vorliegen, muss jeder einzelne Datensatz manuell eingegeben werden. Sie können zwar auch optische Scanner und Spracherkennungssysteme für diesen Zweck benutzen, aber der Einsatz dieser Geräte für die Dateneingabe ist heute noch nicht weit verbreitet.

- ✔ **Irgendwie digitalisiert.** Falls Ihre Daten zwar in digitaler Form, aber nicht in dem Datenformat Ihrer Datenbanktabellen vorliegen, müssen Sie die Daten erst in das passende Format umwandeln und dann in die Datenbank einfügen.
- ✔ **Im richtigen Format digitalisiert.** Wenn Ihre Daten bereits in digitaler Form und im korrekten Format vorliegen, können sie direkt in eine neue Datenbank importiert werden.

Die folgenden Abschnitte behandeln das Hinzufügen von Daten, wobei ich alle drei Formen berücksichtige. Es hängt jetzt davon ab, in welcher Form Ihre Daten vorliegen, ob Sie sie in einem Arbeitsgang in Ihre Datenbank aufnehmen können oder ob Sie Datensatz für Datensatz manuell eingeben müssen. Jeder Datensatz, den Sie eingeben, entspricht einer einzelnen Zeile einer Datentabelle.

Daten zeilenweise einfügen

Die meisten Datenbanksysteme unterstützen eine formularbasierte Dateneingabe. Sie können ein Bildschirmformular erstellen, das für jede Spalte einer Datenbanktabelle ein Eingabefeld enthält. Zusätzliche Bezeichnungsfelder auf dem Formular teilen dem Benutzer mit, welche Daten in die einzelnen Felder eingegeben werden sollen. Der Benutzer gibt alle Daten einer einzelnen Zeile in das Formular ein. Wenn das DBMS die neue Zeile annimmt, wird das Formular geleert, und es ist bereit, eine weitere Zeile aufzunehmen. So können Sie leicht Daten zeilenweise in eine Tabelle einfügen.

Die formularbasierte Dateneingabe ist einfach und weniger anfällig für Fehler als die Verwendung *kommabegrenzter Listen* mit Spaltenwerten. Leider ist die formularbasierte Dateneingabe nicht standardisiert. Jedes DBMS bietet ein eigenes Verfahren zur Erstellung von Formularen an. (Für den Benutzer spielt diese Vielfalt keine Rolle, weil man das Aussehen von Formularen unabhängig von dem DBMS weitgehend vereinheitlichen kann. Der Anwendungsentwickler wird jedoch bei jedem Wechsel seines Entwicklungswerkzeugs mit dem Problem konfrontiert, dass er wieder am Anfang der Lernkurve steht.) Eine weitere mögliche Herausforderung bei der formularbasierten Dateneingabe kann darin bestehen, dass Ihre Implementierung nicht alle Möglichkeiten zur Gültigkeitsprüfung der Eingabedaten bietet.

Die Datenintegrität einer Datenbank können Sie am besten dadurch bewahren, dass Sie verhindern, dass falsche Daten in die Datenbank aufgenommen werden. Einige fehlerhafte Daten können Sie durch Einschränkungen abfangen, die Sie auf die Felder eines Dateneingabeformulars anwenden. Damit sorgen Sie dafür, dass die Eingabewerte vom richtigen Datentyp sind und innerhalb des definierten Wertebereichs liegen. Sie können mit Einschränkungen nicht alle möglichen Fehler abfangen, aber zumindest einige.

Wenn Ihr DBMS-Werkzeug zum Entwurf von Bildschirmformularen nicht alle Gültigkeitsprüfungen ermöglicht, die Sie zum Sichern der Datenintegrität benötigen, können Sie eigene Formulare entwickeln, die Daten in Variablen zwischenspeichern und diese dann in Ihren Anwendungsprogrammen auf Gültigkeit prüfen. Wenn alle Prüfungen korrekte Werte ergeben, können Sie die entsprechende Zeile mit der SQL-Anweisung `INSERT` in Ihre Tabelle einfügen.

Wenn Sie die Daten einer einzelnen Zeile eingeben wollen, hat INSERT die folgende Syntax:

```
INSERT INTO Tabelle_1 [(Spalte_1, Spalte_2, ..., Spalte_n)]
   VALUES (Wert_1, Wert_2, ..., Wert_n) ;
```

Die eckigen Klammern ([]) zeigen an, dass die Liste der Spaltennamen optional ist. Standardmäßig werden die Spalten in der Reihenfolge gefüllt, in der sie in der Tabelle vorliegen. Wenn Sie also VALUES in der Reihenfolge angeben, wie die Spalten der Tabellen existieren, werden die Werte richtig abgelegt – und zwar unabhängig davon, ob Sie die Spalten explizit angeben oder nicht. Wenn Sie die Werte nicht in der Reihenfolge der Spalten in der Tabelle festlegen wollen, müssen Sie die Spaltennamen in der Reihenfolge angeben, die der Reihenfolge der Werte in der Werteliste entspricht.

Verwenden Sie beispielsweise die folgende Syntax, um einen Datensatz in die Tabelle KUNDE einzufügen:

```
INSERT INTO Kunde (KundenID, Vorname, Nachname,
   Straße, Stadt, Land, Postleitzahl, Telefon)
   VALUES (:vkundid, 'Martin', 'Morlock', 'Neustr. 321',
   'Köln', 'D', '50733', '0221/9705630') ;
```

Der erste Wert in der Werteliste, :vkundid, ist eine Variable, die Sie in Ihrem Programmcode inkrementieren, nachdem Sie eine neue Zeile in die Tabelle eingefügt haben. Damit sorgen Sie dafür, dass Sie KundenID nicht doppelt erzeugen. KundenID ist der Primärschlüssel für diese Tabelle und muss deshalb eindeutig sein. Die anderen Werte sind Datenelemente und keine Variablen, die Datenelemente enthalten. Natürlich könnten Sie auch hier Variablen benutzen. Für INSERT spielt es keine Rolle, ob Sie als Argument des Schlüsselworts VALUES Variablen oder eine Kopie der Daten selbst angeben.

Daten nur in ausgewählte Spalten einfügen

Manchmal verfügen Sie beim Einfügen einer Zeile nicht über alle Spaltendaten. In einem solchen Fall können Sie eine Zeile anlegen, die nur die bekannten Spalten enthält. Wenn die Tabelle in der ersten Normalform vorliegt, müssen Sie genügend Daten einfügen, um die neue Zeile von allen anderen Zeilen zu unterscheiden. (Die erste Normalform wird in Kapitel 5 beschrieben.) Diese Bedingung wird erfüllt, wenn Sie den Primärschlüssel der neuen Zeile einfügen. Zusätzlich können Sie alle Daten angeben, die Ihnen zu diesem Datensatz vorliegen. Spalten, in die Sie keine Daten eingeben, werden mit Nullwerten gefüllt.

Das folgende Beispiel zeigt die Eingabe einer solchen nicht vollständigen Zeile:

```
INSERT INTO Kunde (KundenID, Vorname, Nachname)
   VALUES (:vkundid, 'Martin', 'Morlock') ;
```

Sie geben nur die eindeutige Kundennummer und den Namen des Kunden in die Tabelle ein. Die anderen Spalten dieser Zeile enthalten Nullwerte.

Zeilen blockweise in eine Tabelle einfügen

Eine Datenbanktabelle zeilenweise über die `INSERT`-Anweisungen mit Daten zu füllen, ist eine langweilige Angelegenheit – besonders dann, wenn Sie nichts anderes zu tun haben. Selbst wenn die Daten über ein ergonomisch sorgfältig gestaltetes Bildschirmformular eingegeben werden, ermüdet diese Aufgabe den Benutzer mit der Zeit. Falls es technisch machbar ist, die Daten automatisch in die Datenbank einzufügen, ist dies der bessere Weg.

Eine automatische Dateneingabe ist beispielsweise dann möglich, wenn die Daten bereits in elektronischer Form vorliegen, weil sie schon in der Vergangenheit manuell erfasst worden sind. In diesem Fall gibt es keinen überzeugenden Grund, das Ganze zu wiederholen. Die Übertragung von Daten einer Datei in eine andere ist eine Aufgabe, die mit minimalen menschlichen Eingriffen von einem Computer ausgeführt werden kann. Wenn Sie die Eigenschaften der Quelldaten und das Format der Zieltabelle kennen, kann ein Computer die Daten (im Prinzip) automatisch übertragen.

Daten aus einer fremden Datei kopieren

Angenommen, Sie erstellen eine Datenbank für eine neue Anwendung. Einige der benötigten Daten existieren bereits in einer Computerdatei. Bei dieser Datei kann es sich um eine flache Datei oder um eine Tabelle einer Datenbank handeln, die aus einem anderen DBMS stammt. Die Daten können im ASCII- oder EBCDIC-Code oder in einem ungebräuchlichen, anbieterspezifischen Format vorliegen. Was sollten Sie tun?

Zunächst einmal sollten Sie hoffen und beten, dass die Daten in einem weitverbreiteten Format vorliegen. Dann bestehen gute Aussichten, dass Sie ein Werkzeug zur Formatumwandlung finden, mit dem Sie die Daten so umwandeln können, dass sie sich problemlos in Ihre Datenbank importieren lassen. Ihre Entwicklungsumgebung kann vielleicht mindestens eines dieser Formate importieren. Falls Sie *wirklich* Glück haben, kann Ihre Entwicklungsumgebung das Format der Daten direkt verarbeiten. Auf PCs sind die Access-, xBASE- und MySQL-Formate weit verbreitet. Falls die vorliegenden Daten in einem dieser Formate abgespeichert sind, sollte die Umwandlung keine Schwierigkeiten bereiten. Wenn das Format der Daten weniger gebräuchlich ist, müssen Sie die Umwandlung möglicherweise in zwei Schritten durchführen.

Zeilen von einer Tabelle in eine andere übertragen

Es ist schwerer, mit fremden Daten umzugehen, als Daten zu nehmen, die bereits in einer Tabelle Ihrer Datenbank existieren, und diese Daten mit kompatiblen Daten in einer anderen Tabelle zu kombinieren. Dieser Prozess funktioniert besonders einfach, wenn beide Tabellen die gleiche Struktur aufweisen, wenn also jede Spalte der ersten Tabelle eine korrespondierende Spalte in der zweiten Tabelle hat und wenn die einander entsprechenden Spalten vom gleichen Datentyp sind. Falls dies der Fall ist, können Sie die Inhalte der beiden Tabellen mit dem relationalen Operator `UNION` kombinieren. Das Ergebnis ist eine virtuelle Tabelle mit den Daten beider Quelltabellen. Relationale Operatoren, einschließlich `UNION`, behandle ich in Kapitel 11.

Ausgewählte Spalten und Zeilen von einer Tabelle in eine andere übertragen

In der Regel sind jedoch die Strukturen der Quell- und Zieltabellen verschieden. Manchmal stimmen nur einige Spalten überein – dabei handelt es sich dann um die Spalten, in die Sie die Daten einfügen wollen. Sie können angeben, welche Spalten der Quelltabellen in die virtuelle Ergebnistabelle übertragen werden sollen, indem Sie SELECT-Anweisungen mit einer UNION-Klausel verknüpfen. Mit WHERE-Klauseln, die Sie in die SELECT-Anweisungen einfügen, können Sie außerdem Bedingungen für die Zeilen festlegen, die für die Ergebnistabelle ausgewählt werden sollen. WHERE-Klauseln werden ausführlich in Kapitel 10 behandelt.

Stellen Sie sich beispielsweise vor, dass Sie mit den beiden Tabellen Prospekt und Kunde arbeiten und alle Personen in beiden Tabellen auswählen wollen, die in Nordrhein-Westfalen leben. Sie können mit der folgenden Anweisung eine virtuelle Ergebnistabelle erstellen, die die gewünschten Daten enthält:

```
SELECT Vorname, Nachname
   FROM Prospekt
   WHERE Land = 'NRW'
UNION
SELECT Vorname, Nachname
   FROM Kunde
   WHERE Land = 'NRW' ;
```

Was macht diese Anweisung im Einzelnen?

- ✔ Die SELECT-Anweisungen geben an, dass die Ergebnistabelle die Spalten Vorname und Nachname enthalten soll.
- ✔ Die WHERE-Klauseln wählen nur die Zeilen aus, bei denen die Spalte Land den Wert 'NRW' enthält.
- ✔ Die Spalte Land erscheint nicht in der Ergebnistabelle, ist aber in den beiden Ausgangstabellen Prospekt und Kunde enthalten.
- ✔ Der Operator UNION kombiniert das Ergebnis der SELECT-Anweisung für die Tabelle Prospekt mit dem Ergebnis der SELECT-Anweisung für die Tabelle Kunde, löscht doppelte Zeilen und zeigt dann das Ergebnis an.

Eine weitere Methode, Daten von einer Tabelle einer Datenbank in eine andere zu kopieren, besteht darin, eine SELECT-Anweisung in einer INSERT-Anweisung zu verschachteln. Diese Methode (eine *Unterabfrage*; siehe Kapitel 12) erstellt keine virtuelle Tabelle, sondern dupliziert die ausgewählten Daten. Sie können beispielsweise alle Zeilen der Tabelle Kunde in die Tabelle Prospekt einfügen. Dies funktioniert natürlich nur dann, wenn die Strukturen der Tabellen Kunde und Prospekt identisch sind. Wenn Sie beispielsweise die Kunden aus Nordrhein-Westfalen in die Tabelle Prospekt einfügen wollen, benutzen Sie einfach SELECT mit einer WHERE-Klausel, wie das folgende Beispiel zeigt:

```
INSERT INTO Prospekt
   SELECT * FROM Kunde
   WHERE Land = 'NRW' ;
```

Obwohl Sie so redundante Daten erzeugen (weil Sie dieselben Personen sowohl in der Tabelle `Prospekt` als auch in der Tabelle `Kunde` speichern), ist diese Art des Arbeitens manchmal sinnvoll, weil sie das Leistungsverhalten des Systems verbessern kann. Achten Sie aber immer auf Redundanzen! Sorgen Sie dafür, dass alle Änderungen der betreffenden Daten in beiden Tabellen parallel ausgeführt werden. Potenziell problematisch ist auch, dass `INSERT` vielleicht Duplikate von Primärschlüsseln erzeugt. Hat ein Prospekt den Primärschlüssel (`Prospektnummer`), der mit einem Primärschlüssel eines Kunden (`Kundennummer`) übereinstimmt, der wiederum in die Tabelle `Prospekt` eingefügt wird, wird dieser Einfügevorgang fehlschlagen. Haben beide Tabellen Primärschlüssel, die automatisch inkrementiert werden, sollten diese nicht mit demselben Wert starten. Achten Sie darauf, dass die beiden Zahlenblöcke weit voneinander entfernt sind.

Vorhandene Daten aktualisieren

Weil sich die Welt laufend ändert, müssen sich auch die Datenbanken ändern, mit denen Sie bestimmte Aspekte der realen Welt abbilden. Kunden ziehen um. Lagerbestände ändern sich (weil, wie Sie hoffen, jemand Ihre Produkte kauft). Mit jedem Spiel ändert sich die Leistungsstatistik eines Basketballspielers. Diese Beispiele sind typisch für die Ereignisse, die eine Änderung der Daten in einer Datenbank notwendig machen.

Die SQL-Anweisung `UPDATE` dient dazu, Daten einer Tabelle zu ändern. Mit einem einzigen `UPDATE` können Sie eine, einige oder alle Zeilen einer Tabelle ändern. `UPDATE` hat folgende Syntax:

```
UPDATE Tabellenname
   SET Spalte_1 = Ausdruck_1, Spalte_2 = Ausdruck_2,
   ..., Spalte_n = Ausdruck_n
   [WHERE Bedingungen] ;
```

Die Klausel `WHERE` ist optional. Sie können damit die Zeilen auswählen, die Sie aktualisieren wollen. Wenn Sie keine `WHERE`-Klausel angeben, werden alle Zeilen der Tabelle geändert. In der Klausel `SET` geben Sie die neuen Werte der Spalten an, die Sie ändern wollen.

Betrachten Sie die Tabelle `Kunde` in Tabelle 6.1.

Name	Ort	Vorwahl	Telefon
Andreas Anders	Sonnenhausen	0714	5551111
Bernd Besser	Doldendorf	0714	5552222
Christian Weniger	Blumenberg	0714	5553333
Dieter Schneller	Blumenberg	0714	5554444
Daniele Schneller	Blumenberg	0714	5555555

Tabelle 6.1: Die Tabelle `Kunde`

Der Inhalt der Tabelle Kunde ändert sich gelegentlich: Personen ziehen um, bekommen eine andere Telefonnummer und so weiter. Angenommen, Andreas Anders ziehe von Sonnenhausen nach Neustadt um. Sie können seinen Datensatz in der Tabelle mit der folgenden UPDATE-Anweisung aktualisieren:

```
UPDATE Kunde
   SET Ort = 'Neustadt', TELEFON = '6666666'
   WHERE Name = 'Andreas Anders' ;
```

Diese Anweisung ändert die Tabelle, wie Tabelle 6.2 zeigt.

Name	Ort	Vorwahl	Telefon
Andreas Anders	Neustadt	0714	6666666
Bernd Besser	Doldendorf	0714	5552222
Christian Weniger	Blumenberg	0714	5553333
Dieter Schneller	Blumenberg	0714	5554444
Daniele Schneller	Blumenberg	0714	5555555

Tabelle 6.2: Die Tabelle Kunde *nach dem* UPDATE *einer Zeile*

Mit einer ähnlichen Anweisung können Sie mehrere Zeilen auf einmal ändern. Angenommen, die Bevölkerungszahl der Stadt Blumenberg explodiere und Blumenberg bekäme deshalb eine eigene Vorwahl. Sie können die Zeilen aller Kunden, die in Blumenberg leben, folgendermaßen mit einer einzigen UPDATE-Anweisung ändern:

```
UPDATE Kunde
   SET Vorwahl = '(0619)'
   WHERE Ort = 'Blumenberg' ;
```

Tabelle 6.3 zeigt, wie die Tabelle KUNDE nach der Änderung aussieht.

Name	Ort	Vorwahl	Telefon
Andreas Anders	Neustadt	0714	6666666
Bernd Besser	Doldendorf	0714	5552222
Christian Weniger	Blumenberg	0619	5553333
Dieter Schneller	Blumenberg	0619	5554444
Daniele Schneller	Blumenberg	0619	5555555

Tabelle 6.3: Die Tabelle Kunde *nach dem* UPDATE *mehrerer Zeilen*

Es ist noch einfacher, alle Zeilen einer Tabelle als nur einige Zeilen zu ändern. In diesem Fall brauchen Sie keine WHERE-Klausel, um die UPDATE-Anweisung einzuschränken. Angenommen, die Stadt Großstadt hat durch eine Gebietsreform die Städte Neustadt, Doldendorf und Blumenberg geschluckt. Mit einem einzigen Befehl können Sie alle Zeilen der Tabelle auf einmal ändern:

```
UPDATE Kunde
   SET Stadt = 'Großstadt' ;
```

Tabelle 6.4 stellt das Ergebnis dar.

Name	Ort	Vorwahl	Telefon
Andreas Anders	Großstadt	0714	6666666
Bernd Besser	Großstadt	0714	5552222
Christian Weniger	Großstadt	0619	5553333
Dieter Schneller	Großstadt	0619	5554444
Daniele Schneller	Großstadt	0619	5555555

Tabelle 6.4: Die Tabelle Kunde *nach dem* UPDATE *aller Zeilen*

Die WHERE-Klausel, mit der Sie die Zeilen einschränken, auf die die Anweisung UPDATE angewendet werden soll, kann eine *Unterabfrage* enthalten. Damit können Sie die Änderung der Zeilen einer Tabelle von dem Inhalt einer anderen Tabelle abhängig machen.

Für ein Beispiel einer Unterabfrage innerhalb einer UPDATE-Anweisung möchte ich einen Großhändler nehmen, der in seiner Datenbank die Tabelle Anbieter mit den Namen seiner Lieferanten verwaltet. Außerdem enthält die Datenbank die Tabelle Artikel mit den Namen und Preisen aller Artikel, die er verkauft. Die Tabelle Anbieter enthält die Spalten AnbieterID, AnbieterName, Straße, Stadt, Land und Plz. Die Tabelle Artikel hat die Spalten ArtikelID, ArtikelName, Anbieter_ID und Verkaufspreis.

Der Lieferant Cumulonimbus Corporation teilt dem Großhändler mit, dass er die Preise seiner Produkte generell um zehn Prozent erhöhe. Damit die Gewinnspanne erhalten bleibt, müssen die Verkaufspreise der Produkte von Cumulonimbus auch um zehn Prozent angehoben werden. Wir verwenden dafür folgende UPDATE-Anweisung:

```
UPDATE Artikel
   SET Verkaufspreis = (Verkaufspreis * 1.1)
   WHERE AnbieterID IN
      (SELECT AnbieterID FROM Anbieter
       WHERE Anbietername = 'Cumulonimbus Corporation') ;
```

Die Unterabfrage ermittelt die AnbieterID von Cumulonimbus. Mit dieser AnbieterID können die Zeilen der Tabelle Artikel ausgewählt werden, die geändert werden müssen. Die Preise aller Cumulonimbus-Produkte steigen um zehn Prozent, während die Preise aller anderen Produkte gleich bleiben. Unterabfragen werden ausführlich in Kapitel 12 behandelt.

Daten übertragen

Zusätzlich zu den Anweisungen INSERT und UPDATE steht Ihnen die Anweisung MERGE zur Verfügung, um einer Tabelle oder Sicht Daten hinzuzufügen. Mit dieser Anweisung können Sie Daten aus einer Quelltabelle oder Quellsicht mit den Daten einer Zieltabelle oder Zielsicht zusammenführen. Dabei fügt MERGE der Zieltabelle entweder neue Zeilen hinzu oder es werden bereits vorhandene Zeilen aktualisiert. MERGE bietet eine praktische Möglichkeit, um Daten, die irgendwo in einer Datenbank vorhanden sind, an einen anderen Ort zu kopieren.

Werfen Sie einen Blick auf die VetLab-Datenbank aus Kapitel 5. Stellen Sie sich vor, dass einige Personen in der Tabelle Mitarbeiter Verkäufer sind, die Aufträge entgegengenommen haben, während andere Personen keine kaufmännischen Angestellten oder Verkäufer sind und deshalb bisher keine Aufträge entgegennehmen konnten. Das gerade abgelaufene Geschäftsjahr war sehr erfolgreich, und Sie möchten diesen Erfolg mit Ihren Mitarbeitern teilen. Sie entschließen sich deshalb dazu, jedem, der mindestens einen Auftrag entgegengenommen hat, eine Gratifikation in Höhe von 100 Euro zukommen zu lassen. Alle anderen Mitarbeiter sollen einen Bonus in Höhe von 50 Euro erhalten. Sie legen somit zunächst eine Tabelle namens Bonus an und fügen dieser einen Datensatz für jeden Mitarbeiter hinzu, der mindestens einmal in der Tabelle Auftrag vorkommt. Jedem dieser Datensätze wird standardmäßig die Gratifikation von 100 Euro zugewiesen.

Danach benutzen Sie die Anweisung MERGE, um neue Datensätze für die Mitarbeiter einzufügen, die keine Aufträge entgegengenommen haben. Diese erhalten einen Bonus von 50 Euro. Der Code, der die Tabelle Bonus erstellt und mit Daten füllt, sieht so aus:

```
CREATE TABLE Bonus (
   Mitarbeitername CHARACTER (30)      PRIMARY KEY,
   Bonusbetrag     NUMERIC             DEFAULT 100 ) ;

INSERT INTO Bonus (Mitarbeitername)
   (SELECT Mitarbeitername FROM Mitarbeiter, Auftrag
    WHERE Mitarbeiter.Mitarbeitername = Auftrag.Verkäufer
    GROUP BY Mitarbeiter.Mitarbeitername) ;
```

Fragen Sie nun die Tabelle Bonus ab, um zu ermitteln, welche Daten darin gespeichert sind:

```
SELECT * FROM Bonus ;
Mitarbeitername   Bonusbetrag
---------------   -----------
Brynna Jones              100
Chris Bancroft            100
Greg Bosser               100
Kyle Weeks                100
```

Indem Sie nun eine MERGE-Anweisung ausführen lassen, vergeben Sie an die restlichen Mitarbeiter einen Bonus von 50 Euro:

```
MERGE INTO Bonus
   USING Mitarbeiter
   ON (Bonus.Mitarbeitername = Mitarbeiter.Mitarbeitername)
   WHEN NOT MATCHED THEN INSERT
      (Bonus.Mitarbeitername, Bonus.Bonusbetrag)
      VALUES (Mitarbeiter.Mitarbeitername, 50) ;
```

Die Datensätze der Personen in der Tabelle Mitarbeiter, die nicht mit den Datensätzen der Personen übereinstimmen, die bereits in der Tabelle Bonus abgelegt worden sind (dies sind die Personen, die die Gratifikation in Höhe von 100 Euro erhalten haben), werden nun der Tabelle Bonus hinzugefügt. Wenn Sie Bonus jetzt abfragen, sieht das Ergebnis so aus:

```
SELECT * FROM Bonus ;
Mitarbeitername   Bonusbetrag
---------------   -----------
Brynna Jones              100
Chris Bancroft            100
Greg Bosser               100
Kyle Weeks                100
Neth Doze                  50
Matt Bak                   50
Sam Saylor                 50
Nic Foster                 50
```

Die ersten vier Datensätze, die mit der INSERT-Anweisung erzeugt wurden, sind nach dem Mitarbeiternamen sortiert. Die restlichen, mit MERGE hinzugefügten Datensätze befinden sich in der Reihenfolge, in der sie auch in der Tabelle Mitarbeiter angeordnet sind.

Die MERGE-Anweisung ist eine relativ neue Erweiterung von SQL und wird möglicherweise von einigen DBMS-Produkten noch nicht unterstützt. Eine weitere Fähigkeit von MERGE ist noch neuer und wurde erst in SQL:2011 eingeführt, nämlich die (etwas paradoxe) Funktion, Datensätze mit MERGE zu löschen.

Angenommen, Sie beschlössen nach einem INSERT, dass Mitarbeiter, die wenigstens einen Auftrag entgegengenommen haben, keinen Bonus erhalten sollen; alle anderen sollen einen Bonus von 50 € erhalten. Mit der folgenden MERGE-Anweisung können Sie die Verkaufsboni entfernen und Nicht-Verkaufsboni hinzufügen:

```
MERGE INTO Bonus
   USING Mitarbeiter
   ON (Bonus.Mitarbeitername = Mitarbeiter.Mitarbeitername)
   WHEN MATCHED THEN DELETE
   WHEN NOT MATCHED THEN INSERT
      (Bonus.Mitarbeitername, Bonus.Bonusbetrag)
      VALUES (Mitarbeiter.Mitarbeitername, 50);
```

```
Das Ergebnis ist:
SELECT * FROM BONUS;

Mitarbeitername   Bonusbetrag
---------------   -----------
Neth Doze                  50
Matt Bak                   50
Sam Saylor                 50
Nic Foster                 50
```

Überholte Daten löschen

Im Laufe der Zeit veralten einige Daten und werden nutzlos. Sie sollten überholte Daten aus Tabellen löschen. Überflüssige Daten belasten das Leistungsverhalten, belegen Speicherplatz und können die Benutzer irritieren. Sie können ältere Daten in eine Archivdatei übertragen und diese dann offline aufbewahren. Falls diese Daten aus irgendeinem unwahrscheinlichen Grunde noch einmal benötigt werden, können Sie sie problemlos wiederherstellen. In der Zwischenzeit beeinträchtigen sie Ihr Tagesgeschäft nicht. Unabhängig davon, ob die überflüssigen Daten archiviert worden sind oder nicht, gelangen Sie irgendwann an den Punkt, an dem Sie entscheiden, diese Daten endgültig zu löschen. Zu diesem Zweck stellt SQL die Anweisung `DELETE` zur Verfügung.

Sie können alle Zeilen einer Tabelle mit einer einzigen `DELETE`-Anweisung ohne Parameter löschen oder Sie können den Löschvorgang mit einer `WHERE`-Klausel auf bestimmte Zeilen beschränken. Die Syntax ist ähnlich der Syntax einer `SELECT`-Anweisung, wobei Sie keine Spalten angeben müssen. Wenn Sie eine Tabellenzeile löschen, werden alle Daten in allen Spalten dieser Zeile gelöscht.

Angenommen, Ihr Kunde *David Müller* ist nach Tahiti gezogen und wird nicht mehr bei Ihnen kaufen. Sie können ihn mit folgender Anweisung aus Ihrer Tabelle `Kunde` löschen:

```
DELETE FROM Kunde
   WHERE Vorname = 'David' AND Nachname = 'Müller' ;
```

Wenn wir voraussetzen, dass Sie nur einen Kunden mit dem Namen David Müller haben, wird dieser Kunde gelöscht. Wenn Sie mehrere Kunden mit demselben Namen haben, können Sie mit der `WHERE`-Klausel weitere Bedingungen wie `Straße`, `Telefon` oder `KundeID` hinzufügen, um den richtigen David Müller zu löschen. Sonst werden alle David Müller gelöscht.

Temporale Daten verarbeiten

7

In diesem Kapitel

- Zeiten und Perioden
- Verfolgen, was zu bestimmten Zeitpunkten passierte
- Ein Protokoll der Datenänderungen erstellen
- Verarbeiten, welches Ereignis wann aufgezeichnet wurde

Vor SQL:2011 verfügte das ISO/IEC-Standard-SQL nicht über einen Mechanismus, um Daten zu verarbeiten, die zu einem bestimmten Zeitpunkt gültig und zu einem anderen ungültig waren. Jede Anwendung, die Änderungsnachweise liefern muss, braucht diese Fähigkeit. Dies bedeutet, dass die Last der Protokollierung, was zu einem bestimmten Zeitpunkt wahr war, nicht auf der Datenbank, sondern auf dem Anwendungsprogrammierer lag. Dies hört sich wie ein Rezept für komplizierte, überteuerte, verspätete und fehleranfällige Anwendungen an.

In SQL:2011 wurden neue Funktionen eingefügt, mit denen temporale Daten verarbeitet werden können, ohne die Verarbeitung nicht-temporaler Daten zu stören. Dies ist für alle, die eine vorhandene SQL-Datenbank um die Fähigkeit der Verarbeitung temporaler Daten erweitern wollen, ein großer Vorteil.

Doch was sind eigentlich *temporale Daten*? ? Im ISO/IEC-SQL:2011-Standard kommt dieser Terminus gar nicht vor, obwohl er in der Datenbank-Community verbreitet ist. In SQL:2011 sind temporale Daten alle Daten, die mit einer oder mehreren Zeitperioden verbunden sind und während dieser Perioden in einer zeitlichen Dimension als wirksam oder gültig gelten. Einfach ausgedrückt: Mit der Funktion, temporale Daten zu verwalten, können Sie bestimmen, wann ein bestimmtes Datenelement wahr ist.

In diesem Kapitel führe ich den Begriff der *Zeitperiode* ein und definiere ihn sehr spezifisch. Sie lernen verschiedene Arten der Zeit und die Auswirkungen temporaler Daten auf die Definition von Primärschlüsseln und Einschränkungen der referenziellen Integrität kennen. Schließlich beschreibe ich, wie sehr komplexe Daten in bitemporalen Tabellen gespeichert und bearbeitet werden können.

Zeiten und Perioden in SQL:2011 verstehen

Obwohl die Versionen des SQL-Standards vor SQL:2011 die Datentypen `DATE`, `TIME`, `TIMESTAMP` und `INTERVAL` zur Verfügung stellen, boten sie keine Möglichkeit, das Konzept einer *Zeitperiode* mit einer definierten Anfangszeit und einer definierten Endzeit zu verarbeiten. Eine Möglichkeit, dieses Problem zu lösen, besteht darin, einen neuen Datentyp `PERIOD` zu definieren. Doch SQL:2011 tut dies nicht. Zu einem so späten Zeitpunkt der Entwicklung von

SQL einen neuen Datentyp einzuführen, hätte das bestehende SQL-Ökosystem nachhaltig gefährdet. Um einen neuen Datentyp einzuführen, wären größere Eingriffe in praktisch alle vorhandenen Datenbankprodukte erforderlich gewesen.

Anstatt einen `PERIOD`-Datentyp einzuführen, löst SQL:2011 das Problem, indem es *Periodendefinitionen* als Metadaten zu Tabellen hinzufügt. Eine Periodendefinition ist eine benannte Tabellenkomponente, die ein Paar von Spalten spezifiziert, die den Anfangs- und den Endzeitpunkt der Periode definieren. Die Syntax der Anweisungen `CREATE TABLE` und `ALTER TABLE`, mit denen Tabellen erstellt oder modifiziert wurden, wurde erweitert, um Perioden zu erstellen oder zu zerstören, die mit dieser neuen Periodendefinition spezifiziert werden.

Eine `PERIOD` wird durch zwei Spalten definiert: eine Anfangsspalte und eine Endspalte. Es handelt sich um konventionelle Spalten, die denen der vorhandenen Datums- und Zeitdatentypen entsprechen und jeweils einen eindeutigen Namen haben. Wie bereits erwähnt, ist eine Periodendefinition eine benannte Tabellenkomponente. Sie ist im selben Namensraum wie die Spaltennamen angesiedelt; deshalb darf sie nicht denselben Namen wie eine vorhandene Spalte haben.

SQL definiert Perioden als halboffene (rechtsoffene) Intervalle: Die Anfangszeit gehört zum Intervall, die Endzeit dagegen nicht. In einer Zeile muss die Endzeit eine Periode größer als ihre Anfangszeit sein. Dies ist eine Einschränkung, die von dem DBMS durchgesetzt wird.

Beim Arbeiten mit temporalen Daten sind zwei Zeitdimensionen relevant:

- ✔ Die **gültige Zeit (valid time)** ist die Zeitperiode, in der eine Tabellenzeile die Realität widerspiegelt.
- ✔ Die **Transaktionszeit** ist die Zeitperiode, in der eine Zeile in einer Datenbank gespeichert wird.

Die gültige Zeit und die Transaktionszeit einer Tabellenzeile müssen nicht übereinstimmen. Ein Beispiel: In einer Geschäftsdatenbank, in der die gültige Vertragsdauer eines Vertrags gespeichert wird, werden die Informationen über einen Vertrag wahrscheinlich schon vor Beginn der Vertragsdauer eingefügt.

In SQL:2011 können separate Tabellen erstellt werden, um die beiden verschiedenen Arten von Zeit zu verwalten. Alternativ kann auch eine einzelne bitemporale Tabelle (siehe später in diesem Kapitel) zu diesem Zweck verwendet werden. Daten über Transaktionszeiten werden in systemversionierten Tabellen verwaltet, die die System-Zeitperiode enthalten, die mit dem Schlüsselwort `SYSTEM_TIME` bezeichnet wird. Daten über die gültige Zeit werden in Tabellen verwaltet, die eine Anwendungszeitperiode enthalten. Eine Anwendungszeitperiode kann einen beliebigen Namen haben, der noch nicht für andere Komponenten verwendet wird. Sie dürfen maximal eine System-Zeitperiode und eine Anwendungszeitperiode definieren.

Obwohl temporale Daten in SQL erst seit SQL:2011 unterstützt werden, mussten solche Daten schon lange, bevor die entsprechenden Konstrukte in SQL:2011 eingeführt wurden, in Datenbanken verwaltet werden. Zu diesem Zweck wurden üblicherweise zwei Tabellenspalten definiert, eine für den Anfangszeitpunkt und eine für den Endzeitpunkt. Weil SQL:2011 keinen neuen `PERIOD`-Datentyp definiert, sondern mit Periodendefinitionen als Metadaten arbeitet, können vorhandene Tabellen mit solchen Anfangs- und Endspalten leicht aktualisiert

werden, um diese neue Fähigkeit zu nutzen. Die Logik für das Arbeiten mit Perioden kann aus vorhandenen Anwendungsprogrammen entfernt werden. Dadurch können diese vereinfacht, beschleunigt und sicherer gemacht werden.

Mit Anwendungszeitperioden-Tabellen arbeiten

Betrachten Sie ein Beispiel, das mit Anwendungszeitperioden-Tabellen arbeitet. Angenommen, ein Unternehmen wolle nachhalten, in welchen Abteilungen seine Mitarbeiter während ihrer Beschäftigung in dem Unternehmen tätig waren. Zu diesem Zweck können wie folgt Anwendungszeitperiode-Tabellen für Mitarbeiter und Abteilungen angelegt werden:

```
CREATE TABLE Mitarb_Zeiten(
    MitarbID INTEGER,
    MitarbStart DATE,
    MitarbEnd DATE,
    MitarbAbt VARCHAR(30),
    PERIOD FOR MitarbPeriode (MitarbStart, MitarbEnd)
);
```

Der Anfangszeitpunkt (`MitarbStart` im Beispiel) gehört zu der Periode, aber der Endzeitpunkt (`MitarbEnd` im Beispiel) nicht. Dies wird als rechtsoffenes Intervall bezeichnet.

Ich habe bis jetzt noch keinen Primärschlüssel spezifiziert, weil dies bei temporalen Daten etwas komplizierter ist. Dieses Problem wird später in diesem Kapitel behandelt.

Zunächst wollen wir einige Daten in diese Tabelle einfügen:

```
INSERT INTO Mitarb_Zeiten
VALUES (12345, DATE '2011-01-01', DATE '9999-12-31', 'Verkauf');
```

Die Tabelle enthält danach eine Zeile (siehe Tabelle 7.1).

12345	2011-01-01	9999-12-31	Verkauf

Tabelle 7.1: Die Anwendungszeitperiode-Tabelle enthält eine Zeile.

Das Enddatum `9999-12-31` zeigt an, dass dieser Mitarbeiter noch bei dem Unternehmen beschäftigt ist. Der Einfachheit halber habe ich in diesem und den folgenden Beispielen Stunden, Minuten, Sekunden und Sekundenbruchteile weggelassen.

Nehmen Sie jetzt an, dass der Mitarbeiter 12345 vom 15. März 2012 bis zum 15. Juli 2012 temporär in der Entwicklungsabteilung gearbeitet und danach wieder in die Verkaufsabteilung zurückgekehrt ist. Sie können diese Daten mit der folgenden `UPDATE`-Anweisung erfassen:

```
UPDATE Mitarb_Zeiten
    FOR PORTION OF MitarbPeriode
    FROM DATE '2012-03-15'
    TO DATE '2012-07-15'
SET MitarbAbt = 'Entwicklung'
WHERE MitarbID = 12345;
```

Nach der Aktualisierung enthält die Tabelle drei Zeilen (siehe Tabelle 7.2).

MitarbID	MitarbStart	MitarbEnd	MitarbAbt
12345	2011-01-01	2012-03-15	Verkauf
12345	2012-03-15	2012-07-15	Entwicklung
12345	2012-07-15	9999-12-31	Verkauf

Tabelle 7.2: Die Anwendungszeitperiode-Tabelle nach der Aktualisierung

Angenommen, Mitarbeiter 12345 ist immer noch in der Verkaufsabteilung tätig. Dann gibt die Tabelle korrekt seine Abteilungszugehörigkeit vom 1. Januar 2011 bis heute wieder.

Wenn man neue Daten in eine Tabelle eingeben und vorhandene Daten aktualisieren kann, darf natürlich auch eine Löschfunktion nicht fehlen. Doch Daten aus einer Anwendungszeitperiode-Tabelle zu löschen, kann etwas schwieriger sein, als einfach Zeilen aus einer gewöhnlichen, nicht-temporalen Tabelle zu entfernen. Nehmen Sie etwa an, Mitarbeiter 12345 ist am 15. März 2012 nicht der Entwicklungsabteilung zugeteilt worden, sondern hat das Unternehmen an diesem Tag verlassen. Dann ist er am 15. Juli desselben Jahres wieder eingestellt worden. Ursprünglich enthält die Anwendungszeitperiode-Tabelle eine Zeile (siehe Tabelle 7.3).

MitarbID	MitarbStart	MitarbEnd	MitarbAbt
12345	2011-01-01	9999-12-31	Verkauf

Tabelle 7.3: Die Anwendungszeitperiode-Tabelle vor einer Aktualisierung oder Löschung

Eine `DELETE`-Anweisung aktualisiert die Tabelle, um die Periode zu zeigen, in der Mitarbeiter 12345 nicht angestellt war:

```
DELETE Mitarb_Zeiten
    FOR PORTION OF MitarbPeriode
    FROM DATE '2012-03-15'
    TO DATE '2012-07-15'
    WHERE MitarbID = 12345;
```

Nach diesem Befehl sieht die Tabelle wie Tabelle 7.4 aus.

MitarbID	MitarbStart	MitarbEnd	MitarbAbt
12345	2011-01-01	2012-03-15	Verkauf
12345	2012-07-15	9999-12-31	Verkauf

Tabelle 7.4: Die Anwendungszeitperiode-Tabelle nach der Löschung

Die Tabelle zeigt jetzt die Zeitperioden an, in denen Mitarbeiter 12345 in dem Unternehmen beschäftigt war, und zeigt die Lücke, in der er nicht dort beschäftigt war.

Vielleicht ist Ihnen bei den Tabellen in diesem Abschnitt etwas Fragliches aufgefallen. Bei einer gewöhnlichen, nicht-temporalen Tabelle mit den Mitarbeitern eines Unternehmens reicht die Mitarbeiter-ID-Nummer als Primärschlüssel in der Tabelle aus, weil sie jeden Mitarbeiter eindeutig identifiziert. Doch eine Anwendungszeitperiode-Tabelle von Mitarbeitern kann für einen einzelnen Mitarbeiter mehrere Zeilen enthalten. Die Mitarbeiter-ID-Nummer reicht alleine nicht mehr aus, um den Primärschlüssel der Tabelle zu definieren. Die temporalen Daten müssen in den Schlüssel aufgenommen werden.

Primärschlüssel in Anwendungszeitperiode-Tabellen definieren

In Tabelle 7.2 und Tabelle 7.4 ist klar, dass die Mitarbeiter-ID (`MitarbID`) keine Eindeutigkeit garantiert. Es gibt mehrere Zeilen mit derselben `MitarbID`. Um zu garantieren, dass keine Zeilen doppelt vorkommen, müssen der Anfangszeitpunkt (`MitarbStart`) und der Endzeitpunkt (`MitarbEnd`) in den Primärschlüssel aufgenommen werden. Es reicht jedoch nicht aus, diese beiden Felder in den Primärschlüssel aufzunehmen. Betrachten Sie etwa Tabelle 7.5: Sie zeigt den Fall, in dem Mitarbeiter 12345 nur für einige Monate in die Entwicklungsabteilung versetzt worden war und dann in seine Stammabteilung zurückkehrte.

MitarbID	MitarbStart	MitarbEnd	MitarbAbt
12345	2011-01-01	9999-12-31	Verkauf
12345	2012-03-15	2012-07-15	Entwicklung

Tabelle 7.5: Eine unerwünschte Situation

Die beiden Zeilen der Tabelle sind durch die Aufnahme der Felder `MitarbStart` und `MitarbEnd` in den Primärschlüssel garantiert eindeutig, aber die beiden Zeitperioden überlappen sich. Es sieht so aus, als wäre Mitarbeiter 12345 vom 15. März 2012 bis zum 15. Juli 2012 Mitglied sowohl der Verkaufs- als auch der Entwicklungsabteilung gewesen. In einigen Unternehmen mag dies möglich sein; aber es macht die Dinge komplizierter und könnte die Daten beschädigen. Die meisten Unternehmen würden wahrscheinlich eine Einschränkung definieren, die festlegt, dass ein Mitarbeiter immer nur zu einer einzigen Abteilung gehören darf. Mit einer `ALTER TABLE`-Anweisung können Sie eine solche Einschränkung zu einer Tabelle hinzufügen:

```
ALTER TABLE Mitarb_Zeiten
ADD PRIMARY KEY (MitarbID, MitarbPeriode WITHOUT OVERLAPS);
```

Es gibt eine bessere Methode, als zunächst eine Tabelle zu erstellen und später eine Primärschlüssel-Einschränkung zu definieren. Sie können die Primärschlüssel-Einschränkung in die ursprüngliche CREATE-Anweisung einfügen. Ein Beispiel:

```
CREATE TABLE Mitarb_Zeiten
    MitarbID INTEGER NOT NULL,
    MitarbStart DATE NOT NULL,
    MitarbEnd DATE NOT NULL,
    MitarbAbt VARCHAR(30),
    PERIOD FOR MitarbPeriode (MitarbStart, MitarbEnd)
    PRIMARY KEY (MitarbID, MitarbPeriode WITHOUT OVERLAPS)
);
```

Jetzt sind überlappende Zeilen verboten. Außerdem habe ich zugleich zu allen Elementen des Primärschlüssels NOT NULL-Einschränkungen hinzugefügt. Ein NULL-Wert in einem dieser Felder wäre eine Quelle künftiger Fehler. Normalerweise kümmert das DBMS sich um diesen Fall, doch warum sollte man dieses Risiko eingehen?

Referenzielle Einschränkungen auf Anwendungszeitperiode-Tabellen anwenden

Jede Datenbank, mit der mehr als eine einfache Liste von Objekten verwaltet werden soll, erfordert wahrscheinlich mehrere Tabellen. Wenn eine Datenbank mehrere Tabellen enthält, müssen die Beziehungen zwischen den Tabellen und die Einschränkungen der referenziellen Integrität definiert werden.

In dem Beispiel in diesem Kapitel arbeiten Sie mit einer Anwendungszeitperiode-Tabelle der Mitarbeiter und einer Anwendungszeitperiode-Tabelle der Abteilungen. Es gibt eine Eins-zu-viele-Beziehung zwischen der Abteilung-Tabelle und der Mitarbeiter-Tabelle, weil eine Abteilung mehrere Mitarbeiter haben kann, aber jeder Mitarbeiter zu genau einer Abteilung gehört. Dies bedeutet, dass Sie einen Fremdschlüssel in die Mitarbeiter-Tabelle einfügen müssen, der den Primärschlüssel der Abteilung-Tabelle referenziert. Dies sollten Sie berücksichtigen, wenn Sie die Mitarbeiter-Tabelle erneut mit einer diesmal vollständigeren CREATE-Anweisung erstellen. Die Abteilung-Tabelle wird in ähnlicher Weise erstellt:

```
CREATE TABLE Mitarb_Zeiten (
    MitarbID INTEGER NOT NULL,
    MitarbStart DATE NOT NULL,
    MitarbEnd DATE NOT NULL,
    MitarbName VARACHAR (30),
    MitarbAbt VARCHAR (30),
    PERIOD FOR MitarbPeriode (MitarbStart, MitarbEnd)

    PRIMARY KEY (MitarbID, MitarbPeriode WITHOUT OVERLAPS)
    FOREIGN KEY (MitarbAbt, PERIOD MitarbPeriode)
    REFERENCES Abt_Zeiten (AbtID, PERIOD DeptPeriod)
);
```

```
CREATE TABLE Abt_Zeiten (
    AbtID VARCHAR (30) NOT NULL,
    Manager VARCHAR (40) NOT NULL,
    AbtStart DATE NOT NULL,
    AbtEnde DATE NOT NULL,
    PERIOD FOR AbtPeriode (AbtStart, AbtEnde),
    PRIMARY KEY (AbtID, AbtPeriode WITHOUT OVERLAPS)
);
```

Anwendungszeitperiode-Tabellen abfragen

Jetzt können Sie mit SELECT-Anweisungen detaillierte Daten aus den Datenbanktabellen mit den temporalen Daten abrufen.

So können Sie etwa alle gegenwärtigen Mitarbeiter des Unternehmens anzeigen. Schon vor SQL:2011 konnten Sie diese Daten mit einer Anweisung abrufen, die etwa wie folgt aussah:

```
SELECT *
    FROM Mitarb_Zeiten
        WHERE MitarbStart <= CURRENT_DATE()
        AND MitarbEnd > CURRENT_DATE();
```

Mit der neuen PERIOD-Syntax können Sie dasselbe Resultat etwas einfacher erzielen:

```
SELECT *
    FROM Mitarb_Zeiten
        WHERE EmpPERIOD CONTAINS CURRENT_DATE();
```

Sie können auch Mitarbeiter abrufen, die während einer bestimmten Zeitperiode angestellt waren:

```
SELECT *
    FROM Mitarb_Zeiten
        WHERE MitarbPeriode OVERLAPS
        PERIOD (DATE ('2012-01-01'), DATE ('2012-09-16'));
```

Neben CONTAINS und OVERLAPS können Sie in diesem Kontext auch folgende Prädikate verwenden: EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES und IMMEDIATELY SUCCEEDS.

Diese Prädikate arbeiten wie folgt:

- ✔ EQUALS: Eine Periode ist gleich einer anderen.
- ✔ PRECEDES: Eine Periode beginnt früher als eine andere.
- ✔ SUCCEEDS: Eine beginnt später als eine andere.
- ✔ IMMEDIATELY PRECEDES: Eine Periode beginnt früher als eine andere und geht unmittelbar in diese über.
- ✔ IMMEDIATELY SUCCEEDS: Eine Periode beginnt später als eine andere und setzt sie unmittelbar fort.

Mit systemversionierten Tabellen arbeiten

Systemversionierte Tabellen erfüllen einen anderen Zweck als Anwendungszeitperiode-Tabellen und arbeiten deshalb auch anders. Mit Anwendungszeitperiode-Tabellen können Sie Zeitperioden definieren und mit den Daten arbeiten, die in diese Perioden fallen. Dagegen dienen systemversionierte Tabellen dazu, ein kontrollierbares Protokoll zu erstellen, das genau ausweist, wann welche Daten eingefügt, geändert oder gelöscht wurden. So ist es etwa für eine Bank wichtig, genau zu wissen, wann Zahlungen ein- und ausgegangen sind. Diese Daten müssen für eine gesetzlich vorgeschriebene Zeitspanne aufbewahrt werden. Ähnlich müssen Börsenmakler ihre Kaufs- und Verkaufstransaktionen genau aufzeichnen. Auch in vielen ähnlichen Fällen ist es wichtig, bis zum Bruchteil einer Sekunde genau festzuhalten, wann ein bestimmtes Ereignis eingetreten ist.

Anwendungen für Banken oder Börsenhändler müssen strenge Anforderungen erfüllen:

- ✔ Jede Aktualisierungs- oder Löschoperation muss den ursprünglichen Zustand der Zeile vor der Ausführung der Operation bewahren.
- ✔ Anfangs- und Endzeitpunkte der Perioden in den Zeilen werden vom System, nicht vom Anwender verwaltet.
- ✔ Die ursprünglichen Zeilen, die aktualisiert oder gelöscht wurden, bleiben in der Tabelle und werden ab dem Bearbeitungszeitpunkt als *historische Zeilen* bezeichnet. Die Anwender werden daran gehindert, die Inhalte von historischen Zeilen oder die Perioden zu ändern, die mit solchen Zeilen verbunden sind. Nur das System, nicht der Anwender, kann die Perioden von Zeilen in einer systemversionierten Tabelle ändern. Dies erfolgt, indem nicht auf die Periode bezogene Spalten der Tabelle geändert werden, oder als Folge einer Löschung von Zeilen.
- ✔ Diese Einschränkungen garantieren, dass der Verlauf der Datenänderungen immun gegen Manipulationen ist sowie die Anforderungen an eine ordentliche Buchführung und gesetzliche Vorschriften erfüllt.

Systemversionierte Tabellen unterscheiden sich von Anwendungszeitperiode-Tabellen durch einige Differenzen in den `CREATE`-Anweisungen, mit denen sie erstellt werden:

- ✔ In einer Anwendungszeitperiode-Tabelle kann der Anwender der Periode einen beliebigen Namen zuweisen; in einer systemversionierten Tabelle muss die Periode `SYSTEM_TIME` heißen.
- ✔ Die `CREATE`-Anweisung muss die Schlüsselwörter `WITH SYSTEM VERSIONING` enthalten. Obwohl der Datentyp für den Periodenanfang und das Periodenende laut SQL:2011 entweder der `DATE`-Typ oder einer der Zeitstempeltypen sein kann, sollten Sie fast immer einen der Zeitstempeltypen verwenden, weil diese eine viel höhere Genauigkeit als einen Tag ermöglichen. Natürlich müssen die Anfangsspalte und die Endspalte denselben Datentyp haben.

Um die Anwendung von systemversionierten Tabellen zu demonstrieren, setze ich das Mitarbeiter- und Abteilungsbeispiel fort. Mit dem folgenden Code können Sie eine systemversionierte Tabelle erstellen:

```
CREATE TABLE Mitarbeiter_sys (
    MitarbID INTEGER,
    Sys_Start TIMESTAMP(12) GENERATED ALWAYS AS ROW START,
    Sys_End TIMESTAMP(12) GENERATED ALWAYS AS ROW END,
    MitarbName VARCHAR(30),
    PERIOD FOR SYSTEM_TIME (SysStart, SysEnd)
) WITH SYSTEM VERSIONING;
```

Eine Zeile in einer systemversionierten Tabelle wird als die *aktuelle Systemzeile* betrachtet, wenn die aktuelle Zeit in der Systemzeitperiode enthalten ist. Anderfalls gilt sie als *historische Systemzeile*.

Systemversionierte Tabellen ähneln in vielen Aspekten Anwendungszeitperiode-Tabellen, aber es gibt auch Unterschiede:

- ✔ Anwender können den `Sys_Start`- und `Sys_End`-Spalten weder Werte zuweisen noch deren Werte ändern. Diese Werte werden automatisch von dem DBMS zugewiesen und geändert. Dies wird durch die Schlüsselwörter `GENERATED ALWAYS` garantiert.
- ✔ Wenn Sie mit einer `INSERT`-Operation Daten in eine systemversionierte Tabelle einfügen, wird der Wert in der `Sys_Start`-Spalte automatisch auf den Transaktionszeitstempel gesetzt, der jeder Transaktion zugewiesen wird. Der Wert, der der `Sys_End`-Spalte zugewiesen wird, ist der höchste Wert des Datentyps dieser Spalte.
- ✔ In systemversionierten Tabellen bearbeiten `UPDATE`- und `DELETE`-Operationen nur aktuelle Systemzeilen. Anwender können historische Systemzeilen nicht aktualisieren oder löschen.
- ✔ Anwender können den Anfang und das Ende der Systemzeitperiode weder bei aktuellen noch bei historischen Systemzeilen ändern.
- ✔ Wenn Sie eine aktuelle Systemzeile mit einer `UPDATE`- oder `DELETE` Operation bearbeiten, wird automatisch eine historische Systemzeile eingefügt.

  Eine `UPDATE`-Anweisung für eine systemversionierte Tabelle fügt zunächst eine Kopie der alten Zeile ein, wobei deren System-Endzeit auf den Transaktionszeitstempel gesetzt wird. Dies zeigt an, dass die Zeile zum Zeitpunkt dieses Zeitstempels aufgehört hat, aktuell zu sein. Als Nächstes führt das DBMS die Aktualisierung aus und setzt dabei die Anfangszeit der Systemperiode auf den Transaktionszeitstempel. Durch diesen Transaktionszeitstempel wird die aktualisierte Zeile jetzt als die aktuelle Systemzeile ausgewiesen. `UPDATE`-Trigger für die fraglichen Zeilen werden ausgelöst, doch `INSERT`-Trigger werden nicht ausgelöst, obwohl historische Zeilen als Teil dieser Operation eingefügt werden. (Trigger werden in Kapitel 22 beschrieben.)

Eine `DELETE`-Operation bei einer systemversionierten Tabelle löscht die spezifizierten Zeilen nicht physisch, sondern setzt den Endzeitpunkt der Systemzeitperiode der betreffenden Zeilen auf den Systemzeitstempel. Dies zeigt an, dass diese Zeilen zum Zeitpunkt des Transaktionszeitstempels aufgehört haben, aktuell zu sein. Jetzt gehören diese Zeilen nicht mehr zum aktuellen, sondern zum historischen System. Bei einer `DELETE`-Operation werden etwaige `DELETE`-Trigger für die betroffenen Zeilen ausgelöst.

Primärschlüssel für systemversionierte Tabellen definieren

Primärschlüssel für systemversionierte Tabellen zu definieren, ist viel einfacher als in Anwendungszeitperiode-Tabellen, weil es keine Probleme mit Zeitperioden gibt. In systemversionierten Tabellen können die historischen Zeilen nicht geändert werden. Als diese noch aktuelle Zeilen waren, wurde ihre Eindeutigkeit geprüft. Weil sie jetzt nicht mehr geändert werden können, muss auch ihre Eindeutigkeit nicht mehr geprüft werden.

Wenn Sie mit einer `ALTER`-Anweisung eine Primärschlüssel-Einschränkung zu einer vorhandenen systemversionierten Tabelle hinzufügen, müssen Sie keine Periodeninformation in die Anweisung einfügen, weil die Änderung nur die aktuellen Zeilen betrifft. Ein Beispiel:

```
ALTER TABLE Mitarbeiter_sys
    ADD PRIMARY KEY (MitarbID);
```

Das ist alles. Kurz und schmerzlos.

Referenzielle Einschränkungen auf systemversionierte Tabellen anwenden

Aus demselben Grund ist auch die Anwendung von referenziellen Einschränkungen auf systemversionierte Tabellen unkompliziert. Ein Beispiel:

```
ALTER TABLE Mitarbeiter_sys
    ADD FOREIGN KEY (MitarbAbt)
    REFERENCES dept_sys (AbtID);
```

Weil nur aktuelle Zeilen betroffen sind, müssen Sie den Anfangs- und Endzeitpunkt der Periodenspalten nicht angeben.

Systemversionierte Tabellen abfragen

Die meisten Abfragen von systemversionierten Tabellen sollen ermitteln, was zu einem bestimmten Zeitpunkt oder einer bestimmten Zeitperiode in der Vergangenheit wahr war. Zu diesem Zweck wurden in SQL:2011 einige neue Befehle eingefügt. Um eine Tabelle nach Daten abzufragen, die angeben, was zu einem bestimmten Zeitpunkt wahr war, werden die Schlüsselwörter `SYSTEM_TIME AS OF` verwendet. Angenommen, Sie wollten wissen, wer am 15. Juli 2013 in dem Unternehmen beschäftigt war. Dann erhalten Sie das Ergebnis mit folgender Abfrage:

```
SELECT MitarbID, MitarbName, Sys_Start, Sys_End
    FROM Mitarbeiter_sys FOR SYSTEM_TIME AS OF
        TIMESTAMP '2013-07-15 00:00:00';
```

Diese Anweisung gibt alle Zeilen zurück, deren Anfangszeit gleich oder kleiner (früher) als der Wert des Zeitstempels und deren Endzeit größer (später) als der Zeitstempelwert ist.

Um herauszufinden, was während einer Zeitperiode wahr war, können Sie mit den entsprechenden neuen Befehlen eine ähnliche Anweisung verwenden. Ein Beispiel:

```
SELECT MitarbID, MitarbName, Sys_Start, Sys_End
    FROM Mitarbeiter_sys FOR SYSTEM_TIME FROM
        TIMESTAMP '2013-07-01 00:00:00' TO
        TIMESTAMP '2013-08-01 00:00:00';
```

Diese Abfrage gibt alle Zeilen vom ersten Zeitstempel (einschließlich) bis zum zweiten Zeitstempel (ausschließlich) zurück. Alternativ könnten Sie folgenden Befehl verwenden:

```
SELECT MitarbID, MitarbName, Sys_Start, Sys_End
    FROM Mitarbeiter_sys FOR SYSTEM_TIME BETWEEN
        TIMESTAMP '2013-07-01 00:00:00' AND
        TIMESTAMP '2013-07-31 24:59:59';
```

Diese Abfrage gibt alle Zeilen vom ersten Zeitstempel (einschließlich) bis zum zweiten Zeitstempel (diesmal einschließlich!) zurück.

Wenn eine Abfrage einer systemversionierten Tabelle keinen bestimmten Zeitstempel enthält, werden standardmäßig nur die aktuellen Systemzeilen zurückgegeben. Die Abfrage sähe wie folgt aus:

```
SELECT MitarbID, MitarbName, Sys_Start, Sys_End
    FROM Mitarbeiter_sys;
```

Mit der folgenden Anweisung können Sie alle, also sowohl historische als auch aktuelle, Zeilen einer systemversionierten Tabelle abrufen:

```
SELECT MitarbID, MitarbName, Sys_Start, Sys_End
    FROM Mitarbeiter_sys FOR SYSTEM_TIME FROM
        TIMESTAMP '2013-07-01 00:00:00' TO
        TIMESTAMP '9999-12-31 24:59:59';
```

Noch mehr Daten mit bitemporalen Tabellen verwalten

Manchmal wollen Sie sowohl wissen, wann ein Ereignis in der Realität stattgefunden hat, als auch, wann es in der Datenbank aufgezeichnet wurde. In solchen Fällen können Sie Tabellen verwenden, die die Eigenschaften einer systemversionierten Tabelle und einer Anwendungszeitperiode-Tabelle kombinieren. Diese werden als *bitemporale Tabellen* bezeichnet.

Es gibt mehrere Fälle, in denen bitemporale Tabellen sinnvoll eingesetzt werden können. Angenommen, ein Mitarbeiter zöge von einem Bundesstaat (etwa Oregon) in einen anderen Bundesstaat (etwa Washington) um. Sie müssen berücksichtigen, dass sich der Steuersatz für den automatischen Abzug der Einkommensteuer am offiziellen Tag des Umzugs ändert. Doch es ist unwahrscheinlich, dass die Datenbank genau am selben Tag geändert wird. Deshalb müssen beide Zeiten festgehalten werden, wozu eine bitemporale Tabelle gut geeignet ist. Die

systemversionierte Zeitperiode hält fest, wann die Änderung in der Datenbank bekannt wurde, und die Anwendungszeitperiode hält fest, wann der Umzug gesetzlich verbindlich wurde. Hier ist ein Beispiel, wie eine solche Tabelle erstellt wird:

```
CREATE TABLE Mitarbeiter_bt (
    MitarbID INTEGER,
    MitarbStart DATE,
    MitarbEnd DATE,
    MitarbAbt Integer
    PERIOD FOR MitarbPeriode (MitarbStart, MitarbEnd),
    Sys_Start TIMESTAMP (12) GENERATED ALWAYS
        AS ROW START,
    Sys_End TIMESTAMP (12) GENERATED ALWAYS
        AS ROW END,
    MitarbName VARCHAR (30),
    MitarbStrasse VARCHAR (40),
    MitarbStadt VARCHAR (30),
    MitarbProvinz VARCHAR (2),
    MitarbPostleitzahl VARCHAR (10),
    PERIOD FOR SYSTEM_TIME (Sys_Start, Sys_End),
    PRIMARY KEY (MitarbID, EPeriode WITHOUT OVERLAPS),
    FOREIGN KEY (EAbt, PERIOD EPeriode)
    REFERENCES Abt (AbtID, PERIOD DPeriode)
) WITH SYSTEM VERSIONING;
```

Bitemporale Tabellen erfüllen die Zwecke sowohl von systemversionierten Tabellen als auch von Abfragezeit-Tabellen. Der Anwender gibt Werte für die Spalten mit dem Anfang und dem Ende der Anwendungszeitperiode an. Eine `INSERT`-Operation in einer solchen Tabelle setzt automatisch den Wert der Systemzeitperiode auf den Transaktionszeitstempel. Der Wert für das Ende der Systemzeitperiode wird automatisch auf den höchsten zulässigen Wert des Datentyps dieser Spalte gesetzt.

Die `UPDATE`- und `DELETE`-Operationen funktionieren wie für standardmäßige Anwendungszeitperiode-Tabellen. Und wie bei systemversionierten Tabellen betreffen `UPDATE`- und `DELETE`-Operationen nur aktuelle Zeilen, und mit jeder dieser Operationen wird automatisch eine historische Zeile eingefügt.

Eine Abfrage einer bitemporalen Tabelle kann eine Anwendungszeitperiode, eine systemversionierte Periode oder beides spezifizieren. Hier ist ein Beispiel für die Spezifikation beider Perioden:

```
SELECT MitarbID
    FROM Mitarbeiter_bt FOR SYSTEM TIME AS OF
        TIMESTAMP '2013-07-15 00:00:00'
        WHERE MitarbID = 314159 AND
            MitarbPeriode CONTAINS DATE '2013-06-20 00:00:00';
```

Werte festlegen

8

In diesem Kapitel

- Mit Variablen redundanten Code vermeiden
- Häufig benötigte Daten aus einem Tabellenfeld auslesen
- Einfache Werte zu komplexen Ausdrücken verknüpfen

In den vorangegangenen Kapiteln habe ich betont, wie wichtig die Datenbankstruktur für die Integrität der Datenbank ist. Die Datenbankstruktur wird leider häufig vernachlässigt. Sie dürfen aber nicht vergessen, dass die Daten selbst das Wichtigste sind. Schließlich bilden die Werte der Daten, die Sie in den Feldern Ihrer Tabellen speichern, das Rohmaterial, aus dem Sie Ihre geschäftlichen Entscheidungen ableiten.

Werte können auf verschiedene Weise abgebildet werden. Sie können sie direkt darstellen oder mit Funktionen oder Ausdrücken ableiten. Dieses Kapitel beschreibt verschiedene Arten von Werten sowie Funktionen und Ausdrücke.

Funktionen sind Programmteile, die Daten untersuchen und einen Wert berechnen, der auf diesen Daten basiert. *Ausdrücke* sind Kombinationen von Datenelementen, die von SQL ausgewertet werden, um einen einzelnen Wert zu generieren.

Werte

SQL unterscheidet mehrere Arten von Werten: Zeilenwerte, Literale, Variablen, spezielle Variablen und Spaltenreferenzens

Zeilenwerte

Die sichtbarsten Werte einer Datenbank sind die *Zeilenwerte* der Tabellen (häufig auch *Datensatz* genannt). Dabei handelt es sich um die Werte, die jede Zeile einer Datentabelle enthält. Ein Zeilenwert besteht für gewöhnlich aus mehreren Komponenten, weil jedes Feld einer Zeile einen Wert enthält. In einer Datenbanktabelle ist ein *Feld* die Schnittstelle einer einzelnen Spalte mit einer einzelnen Zeile. Ein Feld enthält einen *skalaren* oder *atomaren Wert*. Ein skalarer oder atomarer Wert besteht nur aus einer einzigen Komponente.

Literale

In SQL können Werte entweder durch *Variablen* oder durch *Konstanten* dargestellt werden. Bezeichnenderweise können Variablen ihren Wert ändern, während sich die Werte von Konstanten nie ändern. Eine wichtige Art von Konstanten sind die sogenannten *Literale*. Betrachten Sie ein *Literal* als einen *WYSIWYG*-Wert *(W*hat *Y*ou *S*ee *I*s *W*hat *Y*ou *G*et; wörtlich *Was Sie sehen, ist, was Sie bekommen)*. Literale stellen sich selbst dar.

SQL besitzt nicht nur viele verschiedene Datentypen, sondern auch viele verschiedene Typen von Literalen. Tabelle 8.1 zeigt einige Beispiele für Literale der verschiedenen Datentypen.

Datentyp	Beispiele für ein Literal
BIGINT	8589934592
INTEGER	186282
SMALLINT	186
NUMERIC	186282.42
DECIMAL	186282.42
REAL	6.02257E23
DOUBLE PRECISION	3.1415926535897E00
FLOAT	6.02257E23
CHARACTER(15)	'GREECE '
Anmerkung: Insgesamt stehen in der vorigen Zeile zwischen den Anführungsstrichen 15 Zeichen und Leerzeichen.	
VARCHAR (CHARACTER VARYING)	'lepton'
NATIONAL CHARACTER(15)	'@@eps@@lam@@ lam@@alp@@sig ' 'ΕΛΛΑ∑' Das Wort, mit dem die Griechen ihr Land in ihrer Landessprache bezeichnen (Hellas).
Anmerkung: Insgesamt stehen in der vorigen Zeile zwischen den Anführungsstrichen 15 Zeichen und Leerzeichen.	
NATIONAL CHARACTER VARYING(15)	' @@lclam@@lceps@@lcpi@@lctau@@lcomi@@lcnu' 'λεππου' (Dies ist das Wort »lepton« in Griechisch.)
CHARACTER LARGE OBJECT(512) (CLOB(512))	(Eine wirklich lange Zeichenkette)
BINARY(4)	'01001100011100001111000111001010'
VARBINARY(4) (BINARY VARYING(4))	'0100110001110000'
BINARY LARGE OBJECT(512) (BLOB(512))	(Eine wirkliche lange Kette von Nullen und Einsen)
DATE	DATE '1969-07-20'
TIME(2)	TIME '13.41.32.50'
TIMESTAMP(0)	TIMESTAMP '2013-02-25-13.03.16.000000'
TIME WITH TIMEZONE(4)	TIME '13.41.32.5000-08.00'
TIMESTAMP WITH TIMEZONE(0)	TIMESTAMP '2013-02-25-13.03.16.0000+02.00'
INTERVAL DAY	INTERVAL '7' DAY

Tabelle 8.1: Beispiele für Literale der verschiedenen Datentypen

Was passiert, wenn ein Literal aus einem String besteht, der selbst ein einfaches Anführungszeichen enthält? In diesem Fall müssen Sie zwei einfache Anführungszeichen hintereinander eingeben. Diese Konvention bedeutet, dass das Anführungszeichen Teil des Strings ist und nicht sein Ende markiert. Beispielsweise geben Sie `'Er war"s'` ein, um das Literal `Er war's` darzustellen.

Variablen

Die Fähigkeit, während der Arbeit mit einer Datenbank Literale und andere Arten von Konstanten bearbeiten zu können, ist gut und schön. In vielen Fällen ist dies jedoch mit viel Arbeit verbunden, gäbe es nicht die Variablen. Variablen sind Größen, die ihren Wert ändern können. Betrachten Sie das folgende Beispiel, um zu erkennen, warum Variablen so wertvoll sind:

Angenommen, Sie seien ein Händler, der seine Kunden in verschiedene Umsatzgruppen eingeteilt hat. Kunden mit hohem Umsatz bekommen die besten Preise, Kunden mit einem mittleren, durchschnittlichen Umsatz bekommen einen weniger guten Preis und Kunden mit geringem Umsatz zahlen am meisten. Sie wollen alle Preise mit einem Index an die Kosten der Artikel binden. Der Preis Ihres Artikels F-117A soll für Ihre besten Kunden (Klasse C) das 1,4-Fache Ihrer Kosten betragen, für die mittlere Gruppe (Klasse B) das 1,5-Fache und für die schlechteste Gruppe (Klasse A) das 1,6-Fache. Sie verwalten die Kosten und Preise in der Tabelle `Preistabelle`. Um die neue Preisstruktur anzulegen, geben Sie folgende SQL-Anweisung ein:

```
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.4
   WHERE Artikel = 'F–117A'
      AND Klasse = 'C' ;
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.5
   WHERE Artikel = 'F–117A'
      AND Klasse = 'B' ;
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.6
   WHERE Artikel = 'F–117A'
      AND Klasse = 'A' ;
```

Dieser Code erfüllt zunächst seinen Zweck. Aber was machen Sie, wenn Ihnen ein aggressiver Mitbewerber Marktanteile abjagt? Dann müssen Sie möglicherweise Ihre Gewinnspanne reduzieren, um wettbewerbsfähig zu bleiben. Dazu geben Sie beispielsweise folgende Anweisungen ein:

```
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.25
   WHERE Artikel = 'F–117A'
      AND Klasse = 'C' ;
```

```
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.35
   WHERE Artikel = 'F-117A'
      AND Klasse = 'B' ;
UPDATE Preistabelle
   SET Preis = Einkaufspreis * 1.45
   WHERE Artikel = 'F-117A'
      AND Klasse = 'A' ;
```

Falls Sie in einem sehr dynamischen Markt tätig sind, müssen Sie Ihre Preise möglicherweise sehr oft anpassen und deshalb den SQL-Code sehr häufig ändern. Diese Aufgabe kann sehr schnell zur Last werden, besonders wenn die Preise an mehreren Stellen Ihres Codes vorkommen. Sie können dieses Problem verringern, indem Sie für die Faktoren nicht Literale (zum Beispiel `1,45`), sondern Variablen (zum Beispiel `:faktorA`) benutzen. Dann können Sie die Preise folgendermaßen ändern:

```
UPDATE Preistabelle
   SET Preis = Einkaufspreis * :faktorC
   WHERE Artikel = 'F-117A'
      AND Klasse = 'C' ;
UPDATE Preistabelle
   SET Preis = Einkaufspreis * :faktorB
   WHERE Artikel = 'F-117A'
      AND Klasse = 'B' ;
UPDATE Preistabelle
   SET Preis = Einkaufspreis * :faktorA
   WHERE Artikel = 'F-117A'
      AND Klasse = 'A'
```

Wenn Sie jetzt Ihre Preise ändern müssen, brauchen Sie nur die Werte der Variablen `:faktorC`, `:faktorB` und `:faktorA` zu ändern. Diese Variablen sind Parameter, die Sie an den SQL-Code übergeben, der damit die neuen Preise berechnet.

Manchmal werden die Variablen, die Sie auf diese Weise verwenden, als *Parameter* und manchmal als *Host-Variablen* bezeichnet. Die Bezeichnung *Parameter* wird verwendet, wenn die Variablen in Anwendungen erscheinen, die in der SQL-Modulsprache geschrieben worden sind. Man nennt sie *Host-Variablen*, wenn sie in eingebettetem SQL-Code stehen.

Eingebetteter SQL-Code bedeutet, dass SQL-Befehle in den Code einer Anwendung eingebettet sind, die in einer anderen Programmiersprache, der sogenannten *Host-Sprache*, geschrieben ist. Alternativ können Sie separate SQL-Module erstellen, die nur SQL-Code enthalten, in der Datenbank gespeichert sind und von der Host-Sprache der Anwendung aus aufgerufen werden. Beide Methoden sind gleichwertig. Welche davon Sie nutzen, hängt von Ihrer SQL-Implementierung ab.

Spezielle Variablen

Wenn sich ein Benutzer auf einem Client-Rechner mit einer Datenbank eines Servers verbindet, wird diese Verbindung als *Sitzung* (englisch *session*) bezeichnet. Wenn der Benutzer sich mit mehreren Datenbanken verbindet, sagt man, dass die Sitzung mit der jüngsten Verbindung die *aktuelle Sitzung* ist; die älteren Sitzungen werden als ruhend betrachtet. SQL definiert mehrere *spezielle Variablen*, die in Mehrbenutzersystemen verwendet werden. Diese Variablen, die ich in der folgenden Liste zusammenfasse, verwalten die verschiedenen Benutzer:

- ✔ **SESSION_USER:** Diese spezielle Variable enthält einen Wert, der identisch ist mit dem Authentifizierungsschlüssel der aktuellen Sitzung des Benutzers. Wenn Sie ein Programm mit Kontrollfunktionen schreiben, können Sie die Variable `SESSION_USER` abfragen, um herauszufinden, wer SQL-Anweisungen ausführt.
- ✔ **CURRENT_USER:** Ein SQL-Modul kann über eine benutzerspezifische Autorisierungskennung verfügen. Die Variable `CURRENT_USER` speichert diesen Wert. Wenn das Modul keine solche Kennung kennt, hat die Variable `CURRENT_USER` denselben Wert wie die Variable `SESSION_USER`.
- ✔ **SYSTEM_USER:** Die Variable `SYSTEM_USER` enthält die betriebssystemspezifische Kennung eines Benutzers. Diese Kennung kann von der Benutzerkennung in einem SQL-Modul abweichen. Ein Benutzer kann sich beispielsweise im System unter dem Namen `LARRY` anmelden, sich aber in einem Modul als `Betriebsleiter` identifizieren. Der Wert in `SESSION_USER` lautet dann `Betriebsleiter`. Wenn er die Modulkennung nicht explizit angibt, enthält `CURRENT_USER` ebenfalls `BETRIEBSLEITER`. `SYSTEM_USER` enthält den Wert `LARRY`.

Die speziellen Variablen `SYSTEM_USER`, `SESSION_USER` und `CURRENT_USER` halten nach, wer das System benutzt. Sie können eine Protokolltabelle anlegen und periodisch die Werte dieser Variablen in diese Tabelle eintragen. Ein Beispielcode zur Erledigung dieser Aufgabe sieht so aus:

```
INSERT INTO Logbuch (SNAPSHOT)
   VALUES ('Benutzer ' || SYSTEM_USER ||
      ' mit ID ' || SESSION_USER ||
      ' aktiv um ' || CURRENT_TIMESTAMP) ;
```

Diese Anweisung erzeugt Protokolleinträge der folgenden Art:

```
Benutzer LARRY mit ID Betriebsleiter aktiv um 2013-04-07-14:18:00
```

Spaltenreferenzen

Jede Spalte enthält in jeder Zeile einer Tabelle einen Wert. SQL-Anweisungen verweisen oft auf diese Werte. Ein vollständiger Verweis auf eine bestimmte Spalte besteht aus dem Namen der Tabelle und einem Punkt, dem der Name der Spalte folgt (beispielsweise `Preistabelle.Artikel`). Betrachten Sie die folgende Anweisung:

```
SELECT Preistabelle.Einkaufspreis
   FROM Preistabelle
   WHERE Preistabelle.Artikel = 'F–117A' ;
```

`Preistabelle.Artikel` ist eine Spaltenreferenz. Diese Referenz enthält den Wert `F–117A`. `Preistabelle.Einkaufspreis` ist ebenfalls eine Spaltenreferenz, aber Sie kennen ihren Wert erst, wenn die `SELECT`-Anweisung ausgeführt worden ist.

Weil es eigentlich nur Sinn macht, auf Spalten in der aktuellen Tabelle zu verweisen, können Sie normalerweise darauf verzichten, vollständige Spaltenreferenzen anzugeben. Die folgende Anweisung ist mit den letzten weiter vorn stehenden vergleichbar:

```
SELECT Einkaufspreis
   FROM Preistabelle
   WHERE Artikel = 'F–117A' ;
```

Manchmal arbeiten Sie jedoch mit mehr als einer Tabelle gleichzeitig. Zwei Tabellen in einer Datenbank können gleichnamige Spalten enthalten. Falls dies der Fall ist, müssen Sie für diese Spalten die vollständigen Spaltenreferenzen angeben, um die gewünschten Spalten eindeutig zu kennzeichnen.

Angenommen, Ihre Firma habe Niederlassungen in Kingston und Jefferson und Sie führten für jede Niederlassung zwei separate Tabellen zur Verwaltung der jeweiligen Mitarbeiter. Die Tabellen heißen `MitarbKingston` und `MitarbJefferson`. Wenn Sie eine Liste aller Mitarbeiter erstellen wollen, die in beiden Niederlassungen arbeiten, müssen Sie alle Mitarbeiter ermitteln, deren Namen in beiden Tabellen vorkommen. Die folgende `SELECT`-Anweisung liefert das gewünschte Ergebnis:

```
SELECT MitarbKingston.Vorname, MitarbKingston.Nachname
   FROM MitarbKingston, MitarbJefferson
   WHERE MitarbKingston.MitarbID
      = MitarbJefferson.MitarbID ;
```

Weil die Mitarbeiter-ID jeden Mitarbeiter innerhalb des gesamten Unternehmens eindeutig identifiziert, ist sie von der Niederlassung unabhängig und kann als Bindeglied zwischen den beiden Tabellen benutzt werden. Diese Abfrage liefert nur die Namen von Mitarbeitern, die in beiden Tabellen gespeichert sind.

Wertausdrücke

Ein Ausdruck kann mehr oder weniger komplex sein und Kombinationen von Literalen, Spaltennamen, Parametern, Host-Variablen, Unterabfragen, logischen Verknüpfungen und arithmetischen Operatoren enthalten. Unabhängig davon, wie komplex der Ausdruck zusammengesetzt ist, muss er sich auf einen einzigen Wert reduzieren lassen.

Aus diesem Grund werden SQL-Ausdrücke auch als *Wertausdrücke* bezeichnet. Sie können mehrere Ausdrücke zu einem einzigen Ausdruck kombinieren, wenn sich die Teilwertausdrücke auf Werte von kompatiblen Datentypen reduzieren lassen.

SQL unterscheidet fünf Arten von Wertausdrücken: Numerische, String-, Datums- und Zeit-, Intervall- und Bedingungs-Wertausdrücke.

String-Wertausdrücke

Der einfachste *String-Wertausdruck* besteht aus der Angabe eines einzelnen Strings. Andere Formen umfassen eine Spaltenreferenz, eine Mengenfunktion, eine skalare Unterabfrage, einen `CASE`-Ausdruck, einen `CAST`-Ausdruck oder einen komplexen String-Ausdruck. (Ich gehe in Kapitel 9 auf `CASE`- und `CAST`-Wertausdrücke ein. Unterabfragen werden in Kapitel 12 behandelt.)

Für String-Wertausdrücke gibt es nur einen Operator – den *Verkettungsoperator*. Mit diesem Operator können Sie alle oben aufgelisteten Ausdrücke zu komplexeren String-Wertausdrücken zusammensetzen. Der Verkettungsoperator wird durch ein Paar senkrechter Linien (||) dargestellt. Die folgende Tabelle zeigt einige Beispiele für String-Wertausdrücke.

Ausdruck	Ergebnis
`'Erdnuss' || 'butter'`	`'Erdnussbutter'`
`'Weiße ' || ' ' || 'Bohnen'`	`'Weiße Bohnen'`
`VorName || ' ' || NachName`	`'Hans Schmitz'`
`B'1100111' || B'01010011'`	`B'110011101010011'`
`'' || 'Spargel'`	`'Spargel'`
`'Spargel' || ''`	`'Spargel'`
`'Sp' || '' || 'ar' || '' || 'gel'`	`'Spargel'`

Tabelle 8.2: String-Wertausdrücke

Wie Sie in der Tabelle sehen können, ergibt die Verkettung eines Strings mit einem leeren String den ursprünglichen String.

Numerische Wertausdrücke

Sie können in *numerischen Wertausdrücken* numerische Daten addieren, subtrahieren, multiplizieren und dividieren. Der Ausdruck muss auf einen numerischen Wert reduzierbar sein. Die Komponenten eines numerischen Wertausdrucks können zwar unterschiedliche numerische Datentypen haben, müssen aber grundsätzlich numerisch sein. Der Datentyp des Ergebnisses hängt von den Datentypen der Komponenten ab, aus denen Sie es ableiten. Der SQL-Standard schreibt nicht vor, welcher Ergebnisdatentyp von den verschiedenen möglichen Kombinationen der Wertausdruckkomponenten abgeleitet werden soll, weil es unterschiedliche Plattformen gibt. Lesen Sie in der Dokumentation Ihres Systems nach, wenn Sie numerische Datentypen mischen wollen.

Hier sind einige Beispiele numerischer Wertausdrücke:

- ✔ `-27`
- ✔ `49+83`
- ✔ `5*(12-3)`
- ✔ `Protein + Fett + Kohlenhydrate`
- ✔ `Meter/5280`
- ✔ `Einkaufspreis * :faktorA`

Datums- und Zeit-Wertausdrücke

Datums- und Zeit-Wertausdrücke arbeiten mit Datumsangaben und Zeiten. Diese Wertausdrücke können Komponenten der Datentypen `DATE`, `TIME`, `TIMESTAMP` oder `INTERVAL` enthalten. Das Ergebnis eines Datums- und Zeit-Wertausdrucks ist immer einer der Datentypen `DATE`, `TIME` oder `TIMESTAMP`. Der folgende Wertausdruck gibt beispielsweise das Datum in einer Woche von heute an gerechnet zurück:

```
CURRENT_DATE + INTERVAL '7' DAY
```

Zeiten werden in der koordinierten Weltzeit (UTC, Universal Time Coordinated; auch als GMT, Greenwich Mean Time, bekannt) verwaltet, aber Sie können einen Startwert festlegen, um die Zeit an eine bestimmte Zeitzone anzupassen. Für die lokale Zeitzone Ihres Systems können Sie folgenden einfachen Ausdruck verwenden:

```
TIME '22:55:00' AT LOCAL
```

Alternativ können Sie diesen Wert auch in langer Form ausdrücken:

```
TIME '22:55:00' AT TIME ZONE INTERVAL '-08:00' HOUR TO MINUTE
```

Dieser Ausdruck zeigt eine Zeit, die sich auf die Zeitzone für Portland im US-Bundesstaat Oregon bezieht, die acht Stunden vor der Zeit von Greenwich in England liegt.

Intervall-Wertausdrücke

Wenn Sie ein Datums- und Zeit-Datenelement von einem anderen abziehen, erhalten Sie ein *Intervall*. Es macht keinen Sinn, zwei Datums- und Zeit-Datenelemente zu addieren, deshalb gibt es in SQL diese Möglichkeit nicht. Wenn Sie zwei Intervalle addieren oder voneinander subtrahieren, ist das Ergebnis ein Intervall. Außerdem können Sie Intervalle miteinander oder mit einem numerischen Wert multiplizieren. Analoges gilt für das Dividieren.

SQL unterscheidet zwischen zwei Intervall-Arten: Jahr-Monat *(Year-Month)* und Tag-Zeit *(Day-Time)*. Um Mehrdeutigkeiten zu vermeiden, müssen Sie angeben, welche Art Sie in einem Intervall-Ausdruck verwenden wollen. Der folgende Wertausdruck gibt beispielsweise das Intervall bis zum Rentenalter in Jahren und Monaten an:

```
(GEBURTSTAG_65 - CURRENT_DATE) YEAR TO MONTH
```

Das folgende Beispiel gibt dagegen ein Intervall von 40 Tagen an:

```
INTERVAL '17' DAY + INTERVAL '23' DAY
```

Das nächste Beispiel gibt die Gesamtzahl der Schwangerschaftsmonate einer Mutter von fünf Kindern an (vorausgesetzt, es waren keine Zwillinge dabei):

```
INTERVAL '9' MONTH * 5
```

Intervalle können positiv oder negativ sein. Sie können aus einer beliebigen Kombination von Wertausdrücken bestehen, deren Gesamtwert ein Intervall ergibt.

Bedingungs-Wertausdrücke

Der Wert eines *Bedingungs-Wertausdrucks* hängt von einer Bedingung ab. Die Bedingungs-Ausdrücke `CASE`, `NULLIF` und `COALESCE` sind erheblich komplexer als andere Arten von Wertausdrücken. Tatsächlich sind sie so komplex, dass ich hier nicht genügend Platz habe, um sie zu behandeln. Bedingungsausdrücke werden deshalb in Kapitel 9 ausführlich vorgestellt.

Funktionen

Eine *Funktion* ist eine mehr oder weniger komplexe Operation, die nicht mit den üblichen SQL-Befehlen ausgeführt wird, die Sie aber in der Praxis häufig verwenden. Die Funktionen, die SQL zur Verfügung stellt, führen Aufgaben aus, die sonst der Anwendungscode in der Host-Sprache (in die Sie Ihre SQL-Befehle einbetten) erledigen müsste. SQL enthält zwei Hauptkategorien von Funktionen: *Mengenfunktionen* (auch *Aggregatfunktionen* genannt) und *Wertfunktionen*.

Mit Mengenfunktionen summieren

Mengenfunktionen arbeiten mit *Mengen* von Datensätzen in einer Tabelle statt mit einzelnen Datensätzen. Diese Funktionen fassen einige Eigenschaften einer Datenmenge in einem Wert zusammen. Die Menge kann alle Datensätze der Tabelle oder eine durch eine `WHERE`-Klausel angegebene Untermenge von Datensätzen umfassen. (`WHERE`-Klauseln werden detailliert in Kapitel 10 behandelt.) Manchmal werden die Mengenfunktionen auch als *Aggregatfunktionen* bezeichnet, weil sie mehreren Datensätzen Daten entnehmen, diese verarbeiten und das Ergebnis als einzelne Zeile zurückgeben. Dieses Ergebnis ist eine *Aggregation* (Summierung oder Zusammenfassung) der Daten aus den Datensätzen, die die Datensatzmenge bilden.

Die Mengenfunktionen sollen anhand einer Nährwerttabelle näher erläutert werden. Tabelle 8.3 enthält die Nährwerte für 100 Gramm ausgewählter Nahrungsmittel.

Nahrungsmittel	Kalorien	Eiweiß (Gramm)	Fett (Gramm)	Kohlenhydrate (Gramm)
Mandeln, geröstet	627	18,6	57,7	19,6
Spargel	20	2,2	0,2	3,6
Bananen, roh	85	1,1	0,2	22,2
Rindfleisch, magerer Hamburger	219	27,4	11,3	
Huhn, helles Fleisch	166	31,6	3,4	
Opossum, geröstet	221	30,2	10,2	
Schwein, Schinken	394	21,9	33,3	
Limabohnen	111	7,6	0,5	19,8
Cola	39			10,0
Weißbrot	269	8,7	3,2	50,4
Vollkornbrot	243	10,5	3,0	47,7
Brokkoli	26	3,1	0,3	4,5
Butter	716	0,6	81,0	0,4
Bohnen	367		0,5	93,1
Erdnusssplitter	421	5,7	10,4	81,0

Tabelle 8.3: Nährwerte von 100 Gramm ausgewählter Nahrungsmittel

Eine Datenbanktabelle mit dem Namen `Nahrungsmittel` soll die Daten der Tabelle 8.3 speichern. Leere Felder enthalten den Wert `NULL`. Die Mengenfunktionen `COUNT`, `AVG`, `MAX`, `MIN` und `SUM` können uns wichtige Fakten über die Daten in dieser Tabelle liefern.

COUNT

Die Funktion `COUNT` gibt an, wie viele Zeilen in der Tabelle enthalten sind oder wie viele Zeilen eine bestimmte Bedingung erfüllen. In der einfachsten Form lautet die Funktion:

```
SELECT COUNT (*)
   FROM Nahrungsmittel ;
```

Diese Funktion liefert das Ergebnis 15, weil sie alle Zeilen in der Tabelle `Nahrungsmittel` zählt. Die folgende Anweisung liefert dasselbe Ergebnis:

```
SELECT COUNT (Kalorien)
   FROM Nahrungsmittel ;
```

Weil die Spalte `Kalorien` in jeder Zeile der Tabelle einen Eintrag hat, ist die Zahl, den beide Anweisungen zurückliefern, gleich. Falls eine Spalte jedoch einen Nullwert enthält, zählt die Funktion die entsprechende Zeile nicht mit.

Die folgende Anweisung liefert den Wert 11, weil vier der 15 Zeilen der Tabelle in der Spalte `Kohlenhydrate` einen Nullwert enthalten.

```
SELECT COUNT (Kohlenhydrate)
   FROM Nahrungsmittel ;
```

Es gibt eine Reihe von Gründen dafür, dass ein Feld einer Datenbanktabelle einen Nullwert enthält. Oft ist der tatsächliche Wert nicht oder noch nicht bekannt. Oder er ist bekannt, wurde aber noch nicht eingegeben. Manchmal lässt der Benutzer, wenn er weiß, dass der Wert null ist (also die Zahl 0), das betreffende Eingabefeld einfach leer. Diese Vorgehensweise ist nicht zu empfehlen. Null ist ein definierter Wert, der in Berechnungen verwendet werden kann. Ein Nullwert in einer Tabelle ist dagegen *kein* bestimmter Wert, und SQL kann nicht mit Nullwerten rechnen.

Wenn Sie die Funktion `COUNT` in Verbindung mit `DISTINCT` benutzen, können Sie feststellen, *wie viele verschiedene Werte* in einer Spalte vorhanden sind. Betrachten Sie die folgende Anweisung:

```
SELECT COUNT (DISTINCT Fett)
   FROM Nahrungsmittel ;
```

Das Ergebnis lautet 12. In der Tabelle können Sie sehen, dass 100 Gramm Spargel genauso viel Fett enthalten wie 100 Gramm Bananen (0,2 Gramm) und dass 100 Gramm Limabohnen denselben Fettgehalt wie 100 Gramm Bohnen haben (0,5 Gramm). Deshalb enthält die Tabelle nur zwölf verschiedene Fettwerte.

AVG

Die Funktion `AVG` berechnet den Durchschnitt der Werte einer angegebenen Spalte. Natürlich können Sie diese Funktion nur bei numerischen Spalten benutzen, wie das folgende Beispiel zeigt:

```
SELECT AVG (Fett)
   FROM Nahrungsmittel ;
```

Das Ergebnis ist 15,37. Diese Zahl ist hauptsächlich deshalb so groß, weil Butter in der Datenbank gespeichert ist. Wenn Sie wissen wollen, wie hoch der durchschnittliche Fettgehalt ohne die Butter ist, können Sie die folgende `WHERE`-Klausel in Ihre Anweisung einfügen:

```
SELECT AVG (Fett)
   FROM Nahrungsmittel
   WHERE Nahrung <> 'Butter' ;
```

Der Durchschnittswert fällt nun auf 10,32 Gramm pro 100 Gramm Nahrung.

MAX

Die Funktion MAX gibt den größten Wert in der angegebenen Spalte zurück. Die folgende Anweisung gibt den Wert 81 (den Fettgehalt von 100 Gramm Butter zurück):

```
SELECT MAX (Fett)
   FROM Nahrungsmittel ;
```

MIN

Die Funktion MIN gibt den kleinsten Wert in der angegebenen Spalte zurück. Die folgende Anweisung liefert zum Beispiel den Wert 0,4, weil auch diese Funktion keine Nullwerte berücksichtigt:

```
SELECT MIN (Kohlenhydrate)
   FROM Nahrungsmittel ;
```

SUM

Die Funktion SUM gibt die Summe der Werte in der angegebenen Spalte zurück. Die folgende Anweisung gibt 3.924 zurück, die Summe der Kalorien aller 15 Nahrungsmittel:

```
SELECT SUM (Kalorien)
   FROM Nahrungsmittel ;
```

Wertfunktionen

Einige Operationen werden in verschiedenen Kontexten benötigt. Deshalb sind sie als Funktionen in SQL aufgenommen worden. Im Vergleich zu anderen Datenbankverwaltungssystemen wie Access, Oracle oder SQL-Server verfügt ISO/IEC-SQL über relativ wenige, dafür aber häufig benötigte Wertfunktionen. Diese Funktionen bilden folgenden Gruppen: Stringfunktionen, Numerische Funktionen, Datums- und Zeitfunktionen und Intervallwertfunktionen.

Stringfunktionen

Stringfunktionen nehmen einen String als Eingabe und geben einen anderen String zurück. SQL enthält folgende Stringfunktionen:

- ✔ SUBSTRING
- ✔ SUBSTRING SIMILAR
- ✔ SUBSTRING_REGEX
- ✔ TRANSLATE_REGEX
- ✔ OVERLAY
- ✔ UPPER
- ✔ LOWER
- ✔ TRIM

- ✔ TRANSLATE
- ✔ CONVERT

SUBSTRING

Mit der Funktion SUBSTRING können Sie einen Teil eines Strings aus einem Quellstring extrahieren. Der extrahierte Teil des Strings ist vom selben Typ wie der Quellstring. Wenn beispielsweise der Quellstring vom Typ CHARACTER VARYING ist, ist der Substring ebenfalls von diesem Typ. Die Funktion SUBSTRING hat folgende Syntax:

```
SUBSTRING (Quellstring FROM Start [FOR Länge])
```

Die Klausel in den eckigen Klammern ([]) ist optional. Der Substring, der aus dem Quellstring extrahiert wird, beginnt mit dem Zeichen an der Position, die durch Start angegeben wird, und ist so viele Zeichen lang, wie Länge angibt. Wenn die Klausel FOR fehlt, reicht der Substring bis zum Ende des Quellstrings. Ein Beispiel:

```
SUBSTRING ('Brot, Roggen und Weizen' FROM 7 FOR 6)
```

Der extrahierte Substring lautet hier Roggen. Er beginnt mit dem siebten Zeichen des Quellstrings und ist sechs Zeichen lang. Welchen Zweck erfüllt diese Funktion? Warum sollte schließlich jemand wissen wollen, wie die Zeichen sieben bis zwölf des Strings Brot, Roggen und Weizen lauten? SUBSTRING ist nützlich, weil der Stringwert kein Literal sein muss. Der Quellstring, der als Argument an die Funktion übergeben wird, kann ein Literal, eine Variable oder ein Ausdruck sein, der einen String bildet. So können Sie beispielsweise mit einer Variablen mit der Bezeichnung nahrungsmittel arbeiten, die zu verschiedenen Zeiten unterschiedliche Werte annimmt. Der folgende Ausdruck würde den gewünschten Substring unabhängig von dem String extrahieren, den die Variable im Moment enthält:

```
SUBSTRING (:nahrungsmittel FROM 8 FOR 7)
```

Alle Wertfunktionen können Literale, Variablen oder Ausdrücke, die einen Wert ergeben, als Argumente verarbeiten.

Bei der Funktion SUBSTRING müssen Sie einige Dinge beachten: Der gewünschte Substring muss innerhalb des Quellstrings liegen. Wenn Sie einen Substring abfragen, der an der Position acht beginnen soll, obwohl der Quellstring nur vier Zeichen lang ist, gibt die Funktion einen Nullwert zurück. Sie müssen deshalb die Quelldaten ungefähr kennen, bevor Sie SUBSTRING einsetzen. Außerdem dürfen Sie keinen negativen Wert für die Länge des Substrings angeben, weil das Ende eines Strings nicht vor seinem Anfang liegen kann.

Bei Spalten vom Datentyp VARCHAR wissen Sie eventuell nicht, wie lang die Daten in einer Spalte sind. Dieses fehlende Wissen macht SUBSTRING nichts aus. Geht die angegebene Länge über den Feldinhalt hinaus, gibt die Funktion den gesamten Inhalt zurück; sie meldet keinen Fehler. Ein Beispiel:

```
SELECT * FROM Nahrungsmittel
   WHERE SUBSTRING (Nahrung FROM 9 FOR 12) = 'geröstet' ;
```

Diese Anweisung gibt die Zeile mit dem gerösteten Opossum aus der Tabelle `Nahrungsmittel` zurück, obwohl der Wert in der Spalte `Nahrung` (`'Opossum, geröstet'`) kürzer als 20 Zeichen ist.

Wenn ein *Operand* (ein Wert, aus dem ein Operator einen anderen Wert ableitet) in der `SUBSTRING`-Funktion einen Nullwert hat, gibt die Funktion einen Nullwert als Ergebnis zurück.

SUBSTRING SIMILAR

Die normale Substringfunktion für reguläre Ausdrücke ist triadisch, das heißt, sie hat drei Parameter: den Quellstring, einen Pattern-String (Musterstring) sowie ein Escapezeichen. Das Ergebnis wird mit einem Musterabgleich ermittelt, der mit POSIX-basierten regulären Ausdrücken erfolgt, und zurückgegeben.

Zwei Instanzen des Escapezeichens, jeweils gefolgt von einem doppelten Anführungszeichen, unterteilen den Pattern-String in drei Teile. Ein Beispiel:

Der Quellstring sei `S`: `'Vielleicht muss er noch sieben Jahre arbeiten.'`. Der Pattern-String sei `R`: `'noch '/""sieben"/"" Jahre'`. Das Escapezeichen sei der Schrägstrich: `/`. Dann gibt

```
SUBSTRING S SIMILAR TO R ;
```

den mittleren String des Pattern-Strings als Ergebnis zurück, `'sieben'` in diesem Fall.

SUBSTRING_REGEX

`SUBSTRING_REGEX` durchsucht einen String nach einem regulären XQuery-Ausdruck und gibt ein Vorkommen des passenden Substrings zurück.

Der internationale ISO/IEC-Standard JTC 1/SC 32 schreibt folgende Syntax für einen regulären Substring-Ausdruck vor:

```
SUBSTRING_REGEX <linke Klammer>
     <XQuery–Pattern> [ FLAG <XQuery–Optionsflag> ]
     IN <Regex–Quellstring>
     [ FROM <Startposition> ]
     [ USING <Zeichenlängeneinheit> ]
     [ OCCURRENCE <Regex–Vorkommen> ]
     { GROUP <Regex–Capture–Gruppe> ] <rechte Klammer>
```

`<XQuery–Pattern>` ist ein Stringausdruck, dessen Wert ein regulärer XQuery-Ausdruck ist.

`<XQuery–Optionsflag>` ist ein optionaler String, der dem `$flags`-Argument der `[XQuery F&O]`-Funktion `fn:match` entspricht.

`<Regex–Quellstring>` ist der String, in dem Übereinstimmungen mit dem `<XQuery–Pattern>` gesucht werden sollen.

<Startposition> ist ein optionaler genauer numerischer Wert (Skalierung 0), der die Zeichenposition angibt, an der die Suche beginnen soll (Vorgabe 1).

<Zeichenlängeneinheit> ist CHARACTERS oder OCTETS und zeigt die Einheit an, in der <Startposition> gemessen wird (Vorgabe CHARACTERS).

<Regex-Vorkommen> ist ein optionaler genauer numerischer Wert (Skalierung 0), der angibt, welches Vorkommen einer Übereinstimmung zurückgegeben werden soll (Vorgabe 1).

<Regex-Capture-Gruppe> ist ein optionaler genauer numerischer Wert (Skalierung 0), der angibt, welche Capture-Gruppe einer Übereinstimmung zurückgegeben werden soll (Vorgabe 0, was den gesamten String bedeutet).

Hier sind einige Beispiele für die Anwendung von SUBSTRING_REGEX:

```
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.')='Just'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.' FROM 2)='ust'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.' OCCURRENCE 2)
= 'do'
SUBSTRING_REGEX ( '(do) (\p{L}*' IN 'Just do it.' GROUP 2) = 'it'
```

TRANSLATE_REGEX

TRANSLATE_REGEX durchsucht einen String nach einem regulären XQuery-Ausdruck und gibt einen String zurück, in dem entweder eins oder jedes Vorkommen des passenden Substrings durch einen XQuery-Ersatzstring ersetzt worden ist.

Der internationale ISO/IEC-Standard JTC 1/SC 32 schreibt folgende Syntax für eine Regex-Transliteration vor:

```
TRANSLATE_REGEX <linke Klammer>
<XQuery-Pattern> [ FLAG <XQuery-Optionsflag> ]
IN <Regex-Quellstring>
[ WITH <Regex-Ersatzstring> ]
[ FROM <Startposition> ]
[ USING <Zeichenlängeneinheit> ]
[ OCCURRENCE <Regex-Transliterationsvorkommen> ] <rechte Klammer>
<regex transliteration occurrence> ::=
<regex occurrence>
| ALL
```

Es bedeuten:

<Regex-Ersatzstring> ist ein String, dessen Wert als $replacement-Argument der [XQuery F&O]-Funktion fn:replace verwendet werden kann (Vorgabe Nullstring).

<Regex-Transliterationsvorkommen> ist entweder das Schlüsselwort ALL oder ein genauer numerischer Wert (Skalierung 0), der angibt, welches Vorkommen einer Übereinstimmung zurückgegeben werden soll (Vorgabe ALL).

Hier sind einige Beispiele ohne Ersatzstring:

```
TRANSLATE_REGEX ('i' IN 'Bill did sit.') = 'Bll dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' OCCURRENCE ALL) = 'Bll dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' FROM 5) = 'Bill dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' Occurrence 2) = 'Bill dd sit.'
```

Hier sind einige Beispiele mit Ersatzstring:

```
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a') = 'Ball dad sat.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' OCCURRENCE ALL)=
                'Ball dad sat.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' OCCURRENCE 2) =
                'Bill dad sit.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' FROM 5) = 'Bill dad sat.'
```

OVERLAY

Die Funktion `OVERLAY` ersetzt einen gegebenen Substring eines Strings (numerisch durch Startposition und Länge spezifiziert) durch einen Ersatzstring. Wenn die spezifizierte Länge des Substrings null beträgt, wird aus dem Originalstring nichts entfernt, doch der Ersatzstring wird an der angegebenen Startposition in den Originalstring eingefügt.

UPPER

Die Funktion `UPPER` wandelt alle Zeichen eines Strings in Großbuchstaben um, wie die Beispiele in Tabelle 8.4 zeigen:

Anweisung	Ergebnis
`UPPER ('e. e. cummings')`	`'E. E. CUMMINGS'`
`UPPER ('Isaac Newton, PhD')`	`'ISAAC NEWTON, PHD'`

Tabelle 8.4: Die Wirkungsweise der Funktion `UPPER`

Strings, die nur Großbuchstaben enthalten, werden von der Funktion `UPPER` nicht bearbeitet.

LOWER

Die Funktion `LOWER` wandelt alle Zeichen eines Strings in Kleinbuchstaben um, wie die Beispiele in Tabelle 8.5 zeigen:

Anweisung	Ergebnis
`LOWER ('STEUERN')`	`'steuern'`
`LOWER ('E. E. Cummings')`	`'e. e. cummings'`

Tabelle 8.5: Die Wirkungsweise der Funktion `LOWER`

Strings, die nur Kleinbuchstaben enthalten, werden von der Funktion `LOWER` nicht bearbeitet.

TRIM

Die Funktion TRIM schneidet Leerzeichen oder andere angegebene Zeichen am Anfang und am Ende eines Strings ab. Die Beispiele in Tabelle 8.6 zeigen ihre Verwendung:

Anweisung	Ergebnis
`TRIM (LEADING ' ' FROM '   test   ')`	`'test   '`
`TRIM (TRAILING ' ' FROM '   test   ')`	`'   test'`
`TRIM (BOTH ' ' FROM '   test   ')`	`'test'`
`TRIM (BOTH 't' from 'test')`	`'es'`

Tabelle 8.6: Die Wirkungsweise der Funktion TRIM

Das Leerzeichen ist das Zeichen, das standardmäßig abgeschnitten wird, wodurch auch folgende Syntax gültig ist:

```
TRIM (BOTH FROM ' test ')
```

Diese Version ergibt dasselbe Ergebnis wie das dritte Beispiel in der Tabelle.

TRANSLATE und CONVERT

Die Funktionen TRANSLATE und CONVERT wandeln einen Quellstring in einen anderen Zeichensatz um, beispielsweise von Englisch in Kanji oder von Hebräisch in Französisch. Sie sind implementierungsspezifisch. Lesen Sie die Dokumentation zu Ihrer Implementierung, um Details zu erfahren.

Es wäre großartig, wenn das Übersetzen einer Sprache in eine andere so einfach wäre wie der Aufruf der SQL-Funktion TRANSLATE. Leider ist es nicht so. Die Funktion TRANSLATE übersetzt nur die Zeichen im ersten Zeichensatz in die entsprechenden Zeichen im zweiten. Es kann zum Beispiel 'ΕΛΛΑ∑' in 'Ellas', aber nicht in 'Griechenland' übersetzen.

Numerische Funktionen

Numerische Funktionen können Daten verschiedener Datentypen als Eingabe haben, aber die Ausgabe ist immer ein numerischer Wert. SQL enthält 15 numerische Funktionen:

- ✔ Positionsausdrücke (POSITION)
- ✔ Regex-Vorkommen-Funktion (OCCURRENCES_REGEX)
- ✔ Regex-Position-Funktion (POSITION_REGEX)
- ✔ Auszugsausdrücke (EXTRACT)
- ✔ Längenausdrücke (CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGTH)
- ✔ Kardinalitätsausdrücke (CARDINALITY)

- ✔ Absolute Wertausdrücke (ABS)
- ✔ Moduloausdrücke (MOD)
- ✔ Natürlicher Logarithmus (LN)
- ✔ Exponentialfunktion (EXP)
- ✔ Potenzfunktion (POWER)
- ✔ Quadratwurzel (SQRT)
- ✔ Abrundungsfunktion (FLOOR)
- ✔ Aufrundungsfunktion (CEIL, CEILING)
- ✔ Bereichszuordnungsfunktion (WIDTH BUCKET)

POSITION

Die Funktion POSITION gibt die Position eines angegebenen Substrings innerhalb eines angegebenen Quellstrings zurück. Die Syntax lautet:

```
POSITION (Substring IN Quellstring [USING Zeichenlängeneinheit])
```

Optional können Sie eine andere Zeichenlängeneinheit als CHARACTER spezifizieren, was jedoch selten vorkommt. Wenn Sie Unicode-Zeichen verwenden, kann ein Zeichen je nach Typ 8, 16 oder 32 Bits lang sein. Sind Zeichen 16 oder 32 Bits lang, können Sie mit USING OCTETS ausdrücklich die Zeichenlänge auf 8 Bits setzen. Für Binärstrings lautet die Syntax:

```
POSITION (Substring IN Quellstring)
```

Wenn der Wert von Substring gleich dem Wert eines Teilstrings zusammenhängender Oktette identischer Länge in Quellstring ist, ist das Ergebnis um eins größer als die Anzahl der Oktette vor dem Start eines solchen Teilstrings. Die folgende Tabelle zeigt einige Beispiele.

Anweisung	Ergebnis
POSITION ('V' IN 'Vollkornbrot')	1
POSITION ('Vol' IN 'Vollkornbrot')	1
POSITION ('ko' IN 'Vollkornbrot')	5
POSITION ('brei' IN 'Vollkornbrot')	0
POSITION (" IN 'Vollkornbrot')	1
POSITION ('01001001' IN '001100010100100100100110')	2

Tabelle 8.7: Die Wirkungsweise der Funktion POSITION

Wenn die Funktion den Substring nicht findet, gibt sie sowohl bei Zeichen- als auch bei Binärstrings den Wert 0 zurück. Wenn der Substring die Länge null hat (wie im vorletzten Beispiel), gibt die Funktion den Wert 1 zurück. Wenn ein Operand in der Funktion einen Nullwert hat, ist das Ergebnis ein Nullwert.

OCCURRENCES_REGEX

OCCURRENCES_REGEX ist eine numerische Funktion, die die Anzahl der Übereinstimmungen mit einem regulären Ausdruck in einem String zurückgibt. Die Syntax lautet:

```
OCCURRENCES_REGEX <linke Klammer>
<XQuery-Pattern> [ FLAG <XQuery-Optionsflag> ]
IN <Regex-Quellstring>
[ FROM <Startposition> ]
[ USING <Zeichenlängeneinheit> ] <rechte Klammer>
```

Hier sind einige Beispiele:

```
OCCURRENCES_REGEX ( 'i' IN 'Bill did sit.' ) = 3
OCCURRENCES_REGEX ( 'i' IN 'Bill did sit.' FROM 5) = 2
OCCURRENCES_REGEX ( 'I' IN 'Bill did sit.' ) = 0
```

POSITION_REGEX

POSITION_REGEX ist eine numerische Funktion, die die Startposition einer Übereinstimmung oder eins mehr als das Ende einer Übereinstimmung mit einem regulären Ausdruck zurückgibt. Hier ist die Syntax:

```
POSITION_REGEX <linke Klammer> [ <regex position start or after> ]
<XQuery pattern> [ FLAG <XQuery option flag> ]
IN <Regex-Quellstring>
[ FROM <Startposition> ]
[ USING <Zeichenlängeneinheit> ]
[ OCCURRENCE <Regex-Vorkommen> ]
[ GROUP <Regex-Capture-Gruppe> ] <rechte Klammer>

<regex position start or after> ::= START | AFTER
```

Einige Beispiele sollten dies klarer machen:

```
POSITION_REGEX ( 'i' IN 'Bill did sit.' ) = 2
POSITION_REGEX ( START 'i' IN 'Bill did sit.' ) = 2
POSITION_REGEX ( AFTER 'i' IN 'Bill did sit.' ) = 3
POSITION_REGEX ( 'i' IN 'Bill did sit.' FROM 5) = 7
POSITION_REGEX ( 'i' IN 'Bill did sit.' OCCURRENCE 2 ) = 7
POSITION_REGEX ( 'I' IN 'Bill did sit.' ) = 0
```

EXTRACT

Die Funktion EXTRACT gibt eine einzelne Komponente eines Datums- und Zeit-Datenelements oder eines Intervall-Datenelements zurück. Die folgende Anweisung gibt beispielsweise 08 zurück:

```
EXTRACT (MONTH FROM DATE '2000-08-20')
```

CHARACTER_LENGTH

Die Funktion `CHARACTER_LENGTH` gibt die Anzahl der Zeichen in einem String zurück. Die folgende Anweisung gibt beispielsweise `17` zurück:

```
CHARACTER_LENGTH ('Opossum, geröstet')
```

Diese Funktion ist nicht besonders nützlich, wenn ihr Argument aus einem Literal besteht. Statt `CHARACTER_LENGTH ('Opossum, geröstet')` könnten Sie schneller und einfacher 17 schreiben. Diese Funktion hat einen größeren Nutzen, wenn ihr Argument eine Variable oder ein Ausdruck ist.

OCTET_LENGTH

In der Musik wird ein Ensemble mit acht Sängern als *Oktett* bezeichnet. Es besteht üblicherweise aus dem ersten und zweiten Sopran, dem ersten und zweiten Alt, dem ersten und zweiten Tenor und dem ersten und zweiten Bass. In der Computerterminologie wird ein Ensemble aus acht Datenbits als *Byte* bezeichnet. Leider bringt diese Bezeichnung die Anzahl nicht zum Ausdruck. Durch den Rückgriff auf den musikalischen Begriff *Oktett* wird diese Zusammenstellung von acht Bits plastischer beschrieben.

Praktisch alle modernen Computer verwenden acht Bits, um ein einzelnes alphanumerisches Zeichen darzustellen. Komplexere Zeichensätze (zum Beispiel Chinesisch) erfordern 16 Bits für ein Zeichen. Die Funktion `OCTET_LENGTH` gibt die Anzahl der Oktette (Bytes) in einem String zurück. Wenn es sich bei dem String um einen Bitstring handelt, gibt die Funktion die Anzahl der Oktette zurück, die zur Speicherung der Bits benötigt werden. Wenn der String einen Ausdruck der deutschen Sprache enthält (mit einem Oktett pro Zeichen), gibt die Funktion die Anzahl der Zeichen in dem String zurück. Wenn der String ein chinesischer String ist, gibt die Funktion eine Zahl zurück, die doppelt so groß wie die Anzahl der chinesischen Schriftzeichen ist. Dazu ein Beispiel:

```
OCTET_LENGTH ('Limabohnen')
```

Dieser Funktionsaufruf gibt den Wert 10 zurück, da zur Speicherung eines jeden Zeichens jeweils ein Oktett erforderlich ist.

Einige Zeichensätze benutzen eine variable Anzahl von Oktetten für verschiedene Zeichen. Zeichensätze, die eine Mischung von Kanji und lateinischen Buchstaben unterstützen, benutzen sogenannte *Escape-Zeichen*, auch *Fluchtzeichen* genannt (englisch *Escape Characters*), um zwischen den beiden Zeichensätzen zu wechseln. Beispielsweise kann ein String mit 30 Zeichen, der sowohl aus lateinischen Buchstaben als auch aus Kanji besteht, 30 Oktette benötigen, wenn alle Zeichen aus lateinischen Buchstaben bestehen. 62 Oktette sind erforderlich, wenn alle Zeichen in Kanji vorliegen (60 Oktette plus je ein Umschaltzeichen am Anfang und am Ende), und 150 Oktette, wenn die Zeichen zwischen Latein und Kanji wechseln (jedes Kanji-Zeichen benötigt zwei Oktette und jeweils ein Umschaltzeichen am Anfang und am Ende). Die Funktion `OCTET_LENGTH` gibt die Anzahl der Oktette zurück, die Sie für den gegenwärtigen Wert des Strings benötigen.

CARDINALITY

Die Kardinalität wird für Elementsammlungen benötigt, zum Beispiel für Arrays oder Multisets, bei denen der Wert eines jeden Elements von einem bestimmten Datentyp ist. Die Kardinalität einer solchen Sammlung gibt die Anzahl der Elemente zurück, die darin enthalten sind. Ein Beispiel für den Einsatz der Funktion `CARDINALITY` sieht so aus:

```
CARDINALITY (Dienstplan)
```

Dieser Funktionsaufruf gibt beispielsweise den Wert 12 zurück, wenn zwölf Mitarbeiter auf dem Dienstplan vermerkt sind. `Dienstplan`, eine Spalte in der Tabelle `Mitarbeiter`, kann entweder ein Array oder ein Multiset sein. Ein Array ist eine geordnete Auflistung von Elementen, während ein Multiset eine ungeordnete Sammlung ist. Wenn sich der Dienstplan häufig ändert, sollte das Multiset verwendet werden.

ARRAY_MAX_CARDINALITY

Die `CARDINALITY`-Funktion gibt die Anzahl der Elemente in dem spezifizierten Array oder Multiset zurück. Sie erfahren aber nicht die maximale Kardinalität, die für dieses Array deklariert wurde. Manchmal müssen Sie dies jedoch wissen.

Deshalb wurde in SQL:2011 die Funktion `ARRAY_MAX_CARDINALITY` eingeführt. Sie gibt die maximale Kardinalität des spezifizierten Arrays zurück. Für ein Multiset gibt es keine deklarierte maximale Kardinalität.

TRIM_ARRAY

Während die `TRIM`-Funktion die ersten oder letzen Zeichen eines Strings abschneidet, entfernt die `TRIM_ARRAY`-Funktion die letzten Elemente eines Arrays.

Mit dem folgenden Befehl werden die letzten drei Elemente des Arrays `Dienstplan` entfernt:

```
TRIM_ARRAY (Dienstplan, 3)
```

ABS

Die Funktion `ABS` gibt den absoluten Wert eines numerischen Wertausdrucks zurück:

```
ABS (–273)
```

Dieser Funktionsaufruf gibt 273 zurück.

MOD

Die Funktion `MOD` gibt den Modulo von zwei numerischen Wertausdrücken zurück:

```
MOD (3,2)
```

Dieser Funktionsaufruf gibt 1 zurück, den Modulo von 3 geteilt durch 2.

LN

Die Funktion `LN` gibt den natürlichen Logarithmus eines numerischen Wertausdrucks zurück:

```
LN (9)
```

Dieser Funktionsaufruf gibt so etwas wie 2,197224577 zurück. Die Anzahl der Nachkommastellen ist von Ihrer SQL-Implementierung abhängig.

EXP

Die Funktion `EXP` gibt die Basis des natürlichen Logarithmus *e* potenziert mit dem angegebenen numerischen Wertausdruck zurück:

```
EXP (2)
```

Dieser Funktionsaufruf gibt einen Wert wie 7,389056 zurück. Die Anzahl der Nachkommastellen ist von Ihrer SQL-Implementierung abhängig.

POWER

Die Funktion `POWER` potenziert den Wert des ersten numerischen Wertausdrucks mit dem zweiten numerischen Wertausdruck:

```
POWER (2,8)
```

Dieser Funktionsaufruf gibt 256 (2 hoch 8) zurück.

SQRT

Die Funktion `SQRT` gibt die Quadratwurzel des numerischen Wertausdrucks zurück.

```
SQRT (4)
```

Dieser Funktionsaufruf gibt 2 zurück, die Quadratwurzel von 4.

FLOOR

Die Funktion `FLOOR` rundet einen numerischen Wertausdruck auf die nächstniedrigere Ganzzahl ab:

```
FLOOR (3.141592)
```

Dieser Funktionsaufruf gibt 3.0 zurück.

CEIL oder CEILING

Die Funktion `CEIL` oder `CEILING` rundet einen numerischen Wertausdruck auf die nächsthöhere Ganzzahl auf:

```
CEIL (3.141592)
```

Dieser Funktionsaufruf gibt 4 zurück.

WIDTH_BUCKET

Die Funktion WIDTH_BUCKET, die in der Online-Analyse von Daten (OLAP, Online Analytical Processing) benutzt wird, hat vier Argumente und gibt eine Ganzzahl zwischen 0 und dem Wert des letzten Arguments plus 1 zurück. Sie teilt den Bereich zwischen dem zweiten und dem dritten Argument in gleich große Zahlenbereiche und weist das erste Argument diesem Bereich zu. Werte außerhalb dieses in gleich große Zahlenbereiche unterteilten Segments führen dazu, dass die Funktion entweder den Wert 0 oder den Wert des letzten Arguments plus 1 zurückgibt. Dazu ein Beispiel:

```
WIDTH_BUCKET ( PI, 0, 9, 5)
```

In diesem Beispiel ist PI ein numerischer Ausdruck mit einem Wert von 3,141592. Das Beispiel teilt den Bereich von 0 bis 9 in 5 gleich große Zahlenbereiche. Würden Sie die »Breite« der Zahlenbereiche von einer Skala ablesen, betrüge diese zwei Einheiten pro Zahlenbereich (0 bis 1 und 1 bis 2 = 2 Einheiten für den ersten Zahlenbereich; 2 bis 3 und 3 bis 4 = 2 Einheiten für den zweiten Zahlenbereich und so weiter). Der Funktionsaufruf gibt den Wert 2 zurück, da der Wert 3,141592 in den zweiten Zahlenbereich fällt, der von 2 bis 3,999999… reicht.

Datums- und Zeitfunktionen

SQL enthält drei Funktionen, die Daten über das aktuelle Datum (CURRENT_DATE), die aktuelle Zeit (CURRENT_TIME) oder über beides (CURRENT_TIMESTAMP) zurückgeben. CURRENT_DATE hat kein Argument, CURRENT_TIME und CURRENT_TIMESTAMP haben jeweils ein einzelnes Argument. Das Argument gibt die Genauigkeit des Sekundenbruchteils der Zeitkomponente an, die die Funktion zurückgibt. Die Datums- und Zeit-Datentypen und die Genauigkeit der Sekundenbruchteile sind in Kapitel 2 näher beschrieben.

Die folgende Tabelle zeigt einige Beispiele für die Datums- und Zeitfunktionen.

Anweisung	Ergebnis
CURRENT_DATE	2011-01-31
CURRENT_TIME(1)	08:36:57.3
CURRENT_TIMESTAMP(2)	2011-01-31 08:36:57.38

Tabelle 8.8: Datums- und Zeitfunktionen

Das Datum, das die Funktion CURRENT_DATE zurückgibt, ist vom Datentyp DATE, nicht vom Typ CHARACTER. Analog dazu gibt CURRENT_TIME(p) einen Wert vom Typ TIME und CURRENT_TIMESTAMP(p) einen Wert vom Typ TIMESTAMP zurück. Da SQL das Datum und die Zeit aus der Systemuhr Ihres Computers abliest, geben die Daten die Zeitzone Ihres Computers korrekt wieder.

In einigen Anwendungen werden Sie bisweilen die Daten vom Typ DATE, TIME und TIMESTAMP als Strings benötigen, um mit den Stringfunktionen arbeiten zu können. Mit dem Ausdruck CAST können Sie Datums- und Zeit-Werte in Strings umwandeln und diese dann mit Stringfunktionen bearbeiten. CAST wird in Kapitel 9 beschrieben.

Intervallwertfunktionen

In SQL:1999 wurde eine Intervallwertfunktion namens ABS eingeführt. Sie ähnelt der numerischen Funktion ABS, arbeitet aber nicht mit Zahlen, sondern mit Intervalldaten. Sie hat einen einzigen Operanden und gibt ein Intervall mit identischer Genauigkeit zurück, das garantiert keinen negativen Wert hat. Ein Beispiel:

```
ABS ( TIME '11:31:00' – TIME '12:31:00' )
```

Das Ergebnis lautet:

```
INTERVAL +'1:00:00' HOUR TO SECOND
```

SQL-Wertausdrücke – fortgeschrittener Teil

9

In diesem Kapitel

- Mit CASE-Bedingungsausdrücken arbeiten
- Datentypen umwandeln
- Mit Zeilenwertausdrücken Zeit bei der Dateneingabe sparen

In Kapitel 2 habe ich SQL als *Datenuntersprache* bezeichnet. Die einzige Funktion von SQL besteht tatsächlich nur darin, mit den Daten in einer Datenbank zu arbeiten. Deshalb fehlen SQL viele Funktionen konventioneller prozeduraler Sprachen, und Entwickler, die mit SQL arbeiten, müssen häufig zwischen SQL und ihrer Host-Sprache wechseln, um den Programmablauf zu steuern. Dieses ständige Hin-und-her-Springen verlängert die Entwicklungszeit und verschlechtert das Leistungsverhalten zur Laufzeit.

Diese auf die Einschränkungen von SQL zurückzuführenden Leistungseinbußen haben dazu geführt, SQL mit jeder neuen Version der internationalen Spezifikation um immer neue Funktionalitäten zu erweitern. Eine von ihnen, der CASE-Ausdruck, stellt unter SQL die lange vermissten Bedingungsstrukturen zur Verfügung. Eine weitere, der Ausdruck CAST, erleichtert die Umwandlung von Datentypen. Und mit einer dritten Funktionalität, dem Zeilenwertausdruck, sind Sie jetzt in der Lage, mit einer Liste von Werten zu arbeiten, wo früher nur Einzelwerte verwendet werden konnten. Wenn Ihre Werteliste beispielsweise eine Liste von Tabellenspalten enthält, können Sie jetzt mit einer einfachen Syntax alle Spalten auf einmal verarbeiten.

CASE-Bedingungsausdrücke

Jede vollständige Computersprache verfügt über eine oder mehrere Bedingungsanweisungen. Die wahrscheinlich gebräuchlichste ist die Struktur `IF..THEN..ELSE..ENDIF`. Wenn die Bedingung nach dem Schlüsselwort `IF` den Wert `TRUE` annimmt, wird der Anweisungsblock abgearbeitet, der hinter dem Schlüsselwort `THEN` steht, andernfalls werden die Befehle nach dem Schlüsselwort `ELSE` ausgeführt. Das Schlüsselwort `ENDIF` markiert das Ende der Struktur. Diese Struktur eignet sich besonders gut für Entscheidungen mit zwei Alternativen. Wenn es mehr als zwei Alternativen gibt, ist sie weniger geeignet.

Die meisten vollständigen Sprachen kennen eine CASE-Anweisung, mit der Sie mehr als zwei Bedingungen mit entsprechenden Reaktionen auf diese Bedingungen verknüpfen können.

SQL verfügt über eine CASE-*Anweisung* und einen CASE-*Ausdruck*. Ein CASE-Ausdruck ist nur Teil einer Anweisung. In SQL können Sie einen CASE-Ausdruck an fast jeder Stelle benutzen, an der ein Wert verwendet werden darf. Der Wert des Ausdrucks CASE wird zur Laufzeit ermittelt. Die Anweisung CASE von SQL ergibt keinen Wert, sondern führt einen Anweisungsblock aus.

Der CASE-Ausdruck durchsucht eine Tabelle Zeile für Zeile und nimmt den Wert eines spezifizierten Ergebnisses an, wenn ein Wert aus einer Liste von Bedingungen TRUE ist. Wenn die erste Bedingung für eine Zeile nicht erfüllt wird, wird die zweite Bedingung getestet; und wenn sie TRUE ist, wird das ihr zugeordnete Ergebnis dem Ausdruck zugewiesen. So werden alle Bedingungen verarbeitet. Wird keine Übereinstimmung gefunden, wird dem Ausdruck der Wert NULL zugewiesen. Dann wird die nächste Zeile verarbeitet.

Der Ausdruck CASE wird auf zwei verschiedene Arten benutzt:

- ✔ **Verwenden Sie den Ausdruck in Suchbedingungen.** Wenn CASE feststellt, dass eine Suchbedingung für eine Tabellenzeile erfüllt ist und den Wert TRUE zurückgibt, nimmt die Anweisung, in der der Ausdruck CASE enthalten ist, die angegebene Änderung an dieser Zeile vor. CASE sucht in Tabellen nach Zeilen, die bestimmte Bedingungen erfüllen.
- ✔ **Verwenden Sie den Ausdruck, um ein Tabellenfeld mit einem angegebenen Wert zu vergleichen.** Das Ergebnis der Anweisung, die den Ausdruck CASE enthält, ist davon abhängig, ob einer von mehreren angegebenen Werten in dem entsprechenden Tabellenfeld einer Tabellenzeile enthalten ist.

Die nächsten beiden Abschnitte *CASE mit Suchbedingungen verwenden* und *CASE mit Werten verwenden* sollen dabei helfen, ein wenig Klarheit in diese Begriffe zu bringen. Im ersten Abschnitt benutzen zwei Beispiele CASE mit Suchbedingungen. Ein Beispiel durchsucht eine Tabelle und ändert deren Werte, wenn eine bestimmte Bedingung erfüllt ist. Der zweite Abschnitt zeigt zwei Beispiele dafür, wie CASE für die Arbeit mit Werten eingesetzt werden kann.

CASE mit Suchbedingungen verwenden

Eine der mächtigsten Möglichkeiten, die der CASE-Ausdruck bietet, ist, nach Tabellenzeilen zu suchen, in denen die angegebene Suchbedingung erfüllt ist. Diese Verwendung von CASE hat folgende Syntax:

```
CASE
   WHEN Bedingung1 THEN Ergebnis1
   WHEN Bedingung2 THEN Ergebnis2
   ...
   WHEN BedingungN THEN ErgebnisN
   ELSE ErgebnisX
END
```

CASE prüft die erste *qualifizierte Zeile* (das ist die erste Zeile, die einer eventuell vorhandenen WHERE-Klausel entspricht) im Hinblick darauf, ob Bedingung1 TRUE ist. Falls dies der Fall ist, nimmt der CASE-Ausdruck den Wert Ergebnis1 an. Falls Bedingung1 nicht TRUE ist, prüft CASE die zweite Bedingung. Falls diese TRUE ist, nimmt der CASE-Ausdruck den Wert

Ergebnis2 an und so weiter. Falls keine der angegebenen Bedingungen TRUE ist, nimmt der CASE-Ausdruck den Wert ErgebnisX an. Die Klausel ELSE ist optional. Wenn der Ausdruck keine ELSE-Klausel enthält und keine der angegebenen Bedingungen TRUE ist, wird dem Ausdruck ein Nullwert zugewiesen. Nachdem die SQL-Anweisung, die den CASE-Ausdruck enthält, die erste qualifizierte Zeile verarbeitet und die entsprechende Aktion ausgeführt hat, wendet sich die Anweisung der nächsten Zeile zu und so weiter, bis die gesamte Tabelle abgearbeitet worden ist.

Werte bedingungsabhängig aktualisieren

Weil Sie einen CASE-Ausdruck fast überall in einer SQL-Anweisung unterbringen können, können Sie diese sehr flexibel gestalten. So können Sie etwa mit einem CASE in einer UPDATE-Anweisung Tabellenwerte in Abhängigkeit von einer bestimmten Bedingung ändern. Ein Beispiel:

```
UPDATE Nahrungsmittel
   SET Bewertung = CASE
                      WHEN Fett < 1
                         THEN 'sehr fettarm'
                      WHEN Fett < 5
                         THEN 'fettarm'
                      WHEN Fett < 20
                         THEN 'durchwachsen'
                      WHEN Fett < 50
                         THEN 'fettreich'
                      ELSE 'herzgefährdend'
                   END
```

Diese Anweisung wertet die WHEN-Bedingungen nacheinander aus, bis sie auf die erste wahre Bedingung stößt. Danach werden die restlichen Bedingungen von der Anweisung ignoriert.

Tabelle 8.3 in Kapitel 8 listet unter anderem den Fettgehalt von 100 Gramm ausgewählter Nahrungsmittel auf. Eine Tabelle mit diesen Informationen könnte die Spalte Bewertung enthalten, um den Fettgehalt kurz und knapp zu bewerten. Wenn Sie die letzte UPDATE-Anweisung für die Tabelle Nahrungsmittel in Kapitel 8 ausführen, wird Spargel als *sehr fettarm*, Hühnchen als *fettarm* und geröstete Mandeln als *herzgefährdend* bewertet.

Bedingungen vermeiden, die Fehler verursachen

Eine weitere nützliche Funktion von CASE besteht darin, *Fehler abzufangen*, indem es Bedingungen überprüft, die Fehler hervorrufen könnten.

Betrachten wir ein Entlohnungsmodell für Außendienstmitarbeiter. Manche Unternehmen, die ihre Außendienstler provisionsabhängig entlohnen, zahlen neuen Mitarbeitern einen festen Gehaltsanteil oder Sockelbetrag, der mit der Höhe der Provisionen allmählich abnimmt:

```
UPDATE Vertrieb
   SET Zahlung = Provision + CASE
      WHEN Provision <> 0
         THEN Sockelbetrag/Provision
      WHEN Provision = 0
         THEN Sockelbetrag
      END
```

Wenn die Provision eines Außendienstlers null ist, wird durch diese Anweisung verhindert, dass es zu einer ungültigen Division durch null kommt, die einen Fehler verursachen würde. Wenn die Provision ungleich null ist, erhält der Vertriebsmitarbeiter einen provisionsabhängigen Anteil des Sockelbetrags.

Alle `THEN`-Ausdrücke in einem `CASE`-Ausdruck müssen vom selben Typ sein – alle numerisch, alle Zeichenfolgen oder alle Datums- und Zeitwerte. Das Ergebnis des `CASE`-Ausdrucks ist dann ebenfalls von diesem Typ.

CASE mit Werten verwenden

Sie können eine kompaktere Form des `CASE`-Ausdrucks benutzen, wenn Sie einen Testwert mit einer Reihe anderer Werte vergleichen wollen. Diese Form ist innerhalb einer Anweisung vom Typ `SELECT` oder `UPDATE` nützlich, wenn eine Tabellenspalte eine begrenzte Anzahl von Werten enthält und Sie jeden Spaltenwert mit einem bestimmten Ergebnis verbinden wollen. Wenn Sie `CASE` auf diese Weise verwenden, hat der Ausdruck die folgende Syntax:

```
CASE WertT
   WHEN Wert1 THEN Ergebnis1
   WHEN Wert2 THEN Ergebnis2
   ...
   WHEN WertN THEN ErgebnisN
   ELSE ErgebnisX
END
```

Wenn der Testwert `WertT` gleich `Wert1` ist, vergleicht der Ausdruck alle Werte nacheinander, bis er einen übereinstimmenden Wert findet, und nimmt den Wert `Ergebnis1` an. Wenn `WertT` gleich `Wert2` ist, nimmt der Ausdruck den Wert `Ergebnis2` an und so weiter. Wenn keiner der Werte mit dem Testwert übereinstimmt, nimmt der Ausdruck den Wert `ErgebnisX` der optionalen Klausel `ELSE` an. Wenn diese fehlt, nimmt der Ausdruck einen Nullwert an.

Um die Arbeitsweise des Ausdrucks besser zu verstehen, nehmen Sie an, Sie verfügten über eine Tabelle, die für Anschreiben Namen und Anrede mit Rang verschiedener Offiziere enthält. Sie wollen für einen Brief die korrekte Anrede eines jeden Dienstgrads ermitteln. Dies ist mit der folgenden Anweisung möglich:

```
SELECT CASE Dienstgrad
      WHEN 'General'         THEN 'Herrn General'
      WHEN 'Oberst'          THEN 'Herrn Oberst'
      WHEN 'Oberstleutnant' THEN 'Herrn Oberstleutnant'
      WHEN 'Major'           THEN 'Herrn Major'
      WHEN 'Hauptmann'       THEN 'Herrn Hauptmann'
      WHEN 'Oberleutnant'    THEN 'Herrn Oberleutnant'
      WHEN 'Leutnant'        THEN 'Herrn Leutnant'
   ELSE NULL
   END,
      Nachname
   FROM OFFIZIER ;
```

Diese Anweisung erzeugt eine Liste wie diese:

```
Herrn Hauptmann Beierlein
Herrn Oberst Schmitz
Herrn General Schwarzkopf
Herrn Major Mittendorf
Herrn Nimitz
```

Chester Nimitz war im Zweiten Weltkrieg Admiral der US-Marine. Weil sein Dienstgrad nicht im `CASE`-Ausdruck auftaucht, legt die Klausel `ELSE` seine Anrede ohne Rang fest.

Stellen Sie sich für ein weiteres Beispiel vor, dass Hauptmann Beierlein zum Major befördert worden ist. In diesem Fall muss die Tabelle `Offizier` geändert werden. Angenommen, die Variable `offizierNachname` enthält den Wert `'Beierlein'` und die Variable `neuerRang` die Ganzzahl `4`, die dem neuen Dienstgrad von Beierlein entspricht, wie die folgende Tabelle zeigt:

neuerRang	Dienstgrad
1	`General`
2	`Oberst`
3	`Oberstleutnant`
4	`Major`
5	`Hauptmann`
6	`Oberleutnant`
7	`Leutnant`
8	`Herrn`

Tabelle 9.1: Dienstgrade

Sie können jetzt die Beförderung mit folgendem SQL-Code registrieren:

```
UPDATE Offizier
   SET Dienstgrad = CASE :neuerRang
                WHEN 1 THEN 'General'
                WHEN 2 THEN 'Oberst'
                WHEN 3 THEN 'Oberstleutnant'
                WHEN 4 THEN 'Major'
                WHEN 5 THEN 'Hauptmann'
                WHEN 6 THEN 'Oberleutnant'
                WHEN 7 THEN 'Leutnant'
                WHEN 8 THEN NULL
            END ;
   WHERE Nachname = :offizierNachname
```

Alternativ könnten Sie den CASE-Ausdruck auch folgendermaßen formulieren:

```
CASE
   WHEN WertT = Wert1 THEN Ergebnis1
   WHEN WertT = Wert2 THEN Ergebnis2
   ...
   WHEN WertT = Wertn THEN Ergebnisn
   ELSE resultX
END
```

Ein Sonderfall: CASE – NULLIF

Dass sich alles ändert, ist die einzige Konstante in dieser Welt. Manchmal vollziehen sich Veränderungen bewusst, und manchmal glauben Sie, etwas zu kennen, nur um später herauszufinden, dass Ihnen dieses Etwas eigentlich völlig unbekannt ist. Sowohl die klassische Thermodynamik als auch die moderne Chaostheorie erzählen uns, dass Systeme aus einem bekannten, geordneten Zustand in einen ungeordneten Zustand migrieren können, den niemand vorhersagen kann. Jeder, der schon einmal eine Woche lang ein anfänglich aufgeräumtes Kinderzimmer beobachtet hat, weiß, was ich meine.

Datenbanktabellen enthalten definierte Werte in Feldern, die bekannte Inhalte enthalten. Wenn der Wert eines Feldes einmal nicht bekannt ist, enthält das Feld einen Nullwert. In SQL können Sie mit einem CASE-Ausdruck den Inhalt eines Tabellenfeldes von einem definitiven Wert in einen Nullwert ändern. Der Nullwert zeigt an, dass Sie den Wert des Feldes nicht mehr kennen. Betrachten Sie das folgende Beispiel.

Angenommen, Ihnen gehöre eine kleine Fluglinie, die Flüge zwischen Südkalifornien und dem Bundesstaat Washington anbietet. Bis vor Kurzem landeten einige Ihrer Flugzeuge auf dem Flughafen von San Jose zwischen, um aufzutanken. Leider haben Sie das Recht verloren, San Jose anzufliegen. Ab jetzt müssen Sie entweder auf dem Flughafen von San Francisco oder in Oakland zwischenlanden. Im Moment wissen Sie noch nicht, welche Flüge auf welchem Flughafen landen, aber Sie wissen mit Sicherheit, dass die Flüge nicht mehr San Jose

anfliegen. Sie haben eine Tabelle `Flug`, die wichtige Informationen über Ihre Routen enthält, und Sie wollen diese Tabelle jetzt aktualisieren, um alle Verweise auf San Jose zu entfernen. Das folgende Beispiel zeigt einen der möglichen Wege auf, die zum gewünschten Ziel führen:

```
UPDATE Flug
   SET ZwischenStop = CASE
                        WHEN ZwischenStop = 'San Jose'
                           THEN NULL
                        ELSE ZwischenStop
                      END ;
```

Weil Situationen wie diese, in der Sie einen bekannten Wert durch einen Nullwert ersetzen, recht häufig vorkommen, gibt es in SQL eine Kurzform dieser Anweisung. Das letzte Beispiel sieht in dieser Kurzform folgendermaßen aus:

```
UPDATE Flug
   SET ZwischenStop = NULLIF(ZwischenStop, 'San Jo-
se') ;
```

Sie können diesen Ausdruck folgendermaßen lesen: »Setze die Spalte `Zwischen Stop` der Tabelle `Flug` auf einen Nullwert, wenn ihr Wert `'San Jose'` lautet. Andernfalls ändere nichts.«

`NULLIF` ist besonders nützlich, wenn Sie Daten, die ursprünglich mit einem Programm einer Standardprogrammiersprache wie COBOL oder FORTRAN erfasst worden sind, in Ihre Datenbank importieren wollen. Die meisten Standardprogrammiersprachen kennen Nullwerte nicht und benutzen deshalb besondere Werte, um die Zustände *unbekannt* oder *nicht anwendbar* darzustellen. Beispielsweise könnte in einer Spalte `Gehalt` ein numerisches `–1` für den Wert *unbekannt* und in einer Spalte `Positionscode` die Zeichenfolge `"***"` für den Wert *unbekannt* oder *nicht anwendbar* stehen. Wenn Sie die Werte *unbekannt* und *nicht anwendbar* in einer SQL-kompatiblen Datenbank durch Nullwerte darstellen wollen, müssen Sie diese Sonderwerte in Nullwerte umwandeln. Das folgende Beispiel zeigt eine solche Umwandlung für eine Mitarbeitertabelle, in der einige Gehaltswerte unbekannt sind:

```
UPDATE Mitarbeiter
   SET Gehalt = CASE Gehalt
                   WHEN –1 THEN NULL
                   ELSE Gehalt
                END ;
```

Mit `NULLIF` geht es bequemer:

```
UPDATE Mitarbeiter
   SET Gehalt = NULLIF(Gehalt, –1) ;
```

Ein weiterer Sonderfall: CASE – COALESCE

COALESCE ist wie NULLIF eine Kurzform eines speziellen CASE-Ausdrucks. COALESCE hat mit einer Liste von Werten zu tun, die einen normalen oder einen Nullwert enthalten können. Der Ausdruck arbeitet wie folgt:

- ✔ **Wenn ein Wert in der Liste kein Nullwert ist:** Der COALESCE-Ausdruck nimmt diesen Wert an.
- ✔ **Wenn mehr als ein Wert in der Liste kein Nullwert ist:** Der Ausdruck nimmt den ersten Wert der Liste an, der kein Nullwert ist.
- ✔ **Wenn alle Werte in der Liste Nullwerte sind:** Der Ausdruck nimmt einen Nullwert an.

Ein CASE-Ausdruck mit dieser Funktion hat folgende Form:

```
CASE
   WHEN Wert1 IS NOT NULL
      THEN Wert1
   WHEN Wert2 IS NOT NULL
      THEN Wert2
   ...
   WHEN WertN IS NOT NULL
      THEN WertN
   ELSE NULL
END
```

Die COALESCE-Kurzform lautet folgendermaßen:

```
COALESCE(Wert1, Wert2, ..., WertN)
```

Ein COALESCE-Ausdruck kann nach einer Operation vom Typ OUTER JOIN (die ich in Kapitel 11 behandle) viel Tipparbeit sparen.

Umwandlungen von Datentypen mit CAST

In Kapitel 2 beschreibe ich die Datentypen, die SQL erkennt und unterstützt. Idealerweise passen die Datentypen der Tabellenspalten zu den Anwendungen, die auf diese Spalten zugreifen. In unserer nicht gerade perfekten Welt ist es nicht immer klar, welcher Datentyp am besten geeignet ist. Stellen Sie sich beispielsweise vor, dass Sie einer Spalte einen Datentyp zuweisen, der perfekt mit Ihrer aktuellen Anwendung zusammenarbeitet. Nach einiger Zeit möchten Sie jedoch die Anwendung erweitern oder eine völlig neue Anwendung schreiben, die dieselben Daten auf eine andere Weise verwendet. Diese neue Art der Verwendung könnte dazu führen, dass Sie einen anderen Datentyp als den ursprünglichen benötigen.

Stellen Sie sich vor, dass Sie zwei Spalten mit verschiedenen Datentypen in zwei verschiedenen Tabellen vergleichen wollen. Sie haben beispielsweise Datumswerte gespeichert, wobei diese Werte in der einen Tabelle als Zeichenkette vorkommen und in der anderen Tabelle als echte Datumsangaben abgelegt worden sind. Selbst wenn beide Spalten die gleichen Informa-

tionen (Datumsangaben) enthalten, hindern die unterschiedlichen Datentypen Sie daran, einen Vergleich vorzunehmen. In SQL-86 und SQL-89 war die Typeninkompatibilität ein großes Problem. SQL-92 hingegen führte in Form des `CAST`-Ausdrucks eine einfach zu benutzende Lösung dieses Problems ein.

Mit dem Ausdruck `CAST` können Sie die Datentypen von Tabellendaten oder Host-Variablen umwandeln. Nach der Umwandlung können Sie dann, wie geplant, mit der ursprünglichen Arbeit oder Datenanalyse fortfahren.

Die Umwandlung von Datentypen ist natürlich gewissen Einschränkungen unterworfen. Sie können Datentypen nicht beliebig ineinander umwandeln, sondern die Daten müssen zum neuen Datentyp kompatibel sein. Sie können beispielsweise mit `CAST` eine Zeichenfolge vom Typ `CHAR(10)` und dem Inhalt `2013-01-31` in den Datentyp `DATE` umwandeln. Mit der Zeichenfolge `'Rhinozeros'` vom selben Typ ist dies jedoch nicht möglich. Auch können Sie eine Zahl vom Datentyp `INTEGER` nicht in eine Zahl vom Datentyp `SMALLINT` umwandeln, wenn sie größer als der größtmögliche Wert des Datentyps `SMALLINT` ist.

Sie können die Elemente, die als beliebige Zeichenfolgendaten vorliegen, in andere Datentypen (zum Beispiel numerische Datentypen oder Datums- und Zeit-Datentypen) umwandeln, wenn der Ausgangswert die Form eines Literals des neuen Typs hat. Umgekehrt können Sie beliebige Literale dieser Datentypen in Strings umwandeln.

Die folgende Liste beschreibt einige weitere Umwandlungsmöglichkeiten:

- ✔ Sie können jeden numerischen Typ in jeden anderen numerischen Typ umwandeln. Das System rundet oder beschneidet das Ergebnis, wenn der Zieldatentyp weniger Nachkommastellen oder eine geringere Genauigkeit hat.
- ✔ Sie können jeden genauen numerischen Typ in einen Datentyp umwandeln, der als Intervall mit einer einzelnen Komponente, zum Beispiel `INTERVAL DAY` oder `INTERVAL SECOND` vorliegt.
- ✔ Sie können jeden `DATE`-Typ in einen `TIMESTAMP`-Typ umwandeln. Die Zeitkomponente von `TIMESTAMP` wird mit Nullen gefüllt.
- ✔ Sie können jeden `TIME`-Typ in einen `TIME`-Typ mit einem anderen Sekundenbruchteil oder in einen `TIMESTAMP`-Typ umwandeln. Die Datumskomponente von `TIMESTAMP` wird mit dem aktuellen Datum gefüllt.
- ✔ Sie können jeden `TIMESTAMP`-Typ in einen `DATE`-Typ, einen `TIME`-Typ oder einen `TIMESTAMP` mit einem anderen Sekundenbruchteil umwandeln.
- ✔ Sie können jeden `INTERVAL`-Typ aus dem Bereich Jahr-Monat in einen genauen numerischen Typ oder einen anderen `INTERVAL`-Typ aus dem Bereich Jahr-Monat mit anderer Genauigkeit des führenden Feldes umwandeln.
- ✔ Sie können jeden `INTERVAL`-Typ aus dem Bereich Tag-Zeit in einen genauen numerischen Typ oder einen anderen `INTERVAL`-Typ aus dem Bereich Tag-Zeit mit anderer Genauigkeit des führenden Feldes umwandeln.

CAST in SQL verwenden

Angenommen, Sie arbeiteten für eine Vertriebsfirma, die Interessenten und Kunden verwaltet. Die Interessenten werden in der Tabelle `Interessent` gespeichert und in der Spalte `InteressentID` vom Datentyp `CHAR(9)` eindeutig gekennzeichnet. Die Kunden werden in der Tabelle `Kunde` gespeichert und in der Spalte `KundenID` vom Typ `INTEGER` eindeutig gekennzeichnet. Sie haben nun vor, sich alle Personen anzeigen zu lassen, die in beiden Tabellen vorkommen. Diese Aufgabe können Sie erledigen, indem Sie `CAST` verwenden:

```
SELECT * FROM Kunde
   WHERE Kunde.KundenID =
      CAST(Interessent.InteressentID AS INTEGER) ;
```

CAST als Mittler zwischen SQL und Host-Sprachen

Die Hauptfunktion von `CAST` besteht darin, mit Datentypen umzugehen, die zwar in SQL, aber nicht in der von Ihnen verwendeten Host-Sprache vorhanden sind. Die folgende Liste zeigt einige Beispiele für diese Datentypen:

- ✔ SQL hat `DECIMAL` und `NUMERIC`, aber FORTRAN und Pascal kennen diese Datentypen nicht.
- ✔ SQL hat `FLOAT` und `REAL`, nicht aber Standard-COBOL.
- ✔ SQL verfügt über den Typ `DATETIME`, den keine andere Sprache kennt.

Angenommen, Sie wollten etwa mit FORTRAN oder Pascal auf Tabellenspalten vom Datentyp `DECIMAL(5,3)` zugreifen, wollten aber die Ungenauigkeiten nicht in Kauf nehmen, die mit einer Umwandlung dieser Werte in den `REAL`-Datentyp von FORTRAN und Pascal verbunden sind. Dann können Sie die Werte mit `CAST` in Host-Zeichenfolgenvariablen umwandeln. Sie könnten beispielsweise ein numerisches Gehalt von `198,37` als `'0000198,37'` in einer Zeichenfolge vom Typ `CHAR(10)` speichern. Wenn Sie dann das Gehalt in `203,74` ändern möchten, speichern Sie diesen Wert einfach erneut als `'0000203,74'` in einer Zeichenfolge vom Typ `CHAR(10)`. Zunächst wandeln Sie mit `CAST` den SQL-Datentyp `DECIMAL(5,3)` in den Datentyp `CHAR(10)` um. Mit der folgenden Anweisung können Sie diese Operation für einen Mitarbeiter ausführen, dessen Kennung Sie in der Host-Variablen `:varMitarbeiterID` ablegen:

```
SELECT CAST(Gehalt AS CHAR(10)) INTO :varGehalt
   FROM Mitarbeiter
      WHERE MitarbeiterID = :varMitarbeiterID;
```

Die FORTRAN- oder Pascal-Anwendung überprüft das Ergebnis, das als Zeichenfolge in der Variablen `:varGehalt` vorliegt, setzt gegebenenfalls die Zeichenketten auf den neuen Wert `'000203,74'` und aktualisiert die Datenbank, indem folgende Anweisung verwendet wird:

```
UPDATE Mitarbeiter
   SET Gehalt = CAST(:varGehalt AS DECIMAL(5,3))
      WHERE MitarbeiterID = :varMitarbeiterID ;
```

Werte, wie beispielsweise '000198,37' in Form von Zeichenfolgen zu bearbeiten, ist in FORTRAN oder Pascal umständlich, aber es ist mit Unterroutinen machbar und ermöglicht dann den Zugriff auf und die exakte Manipulation von SQL-Daten.

Der Hauptzweck von CAST besteht darin, Datentypen zwischen Host-Sprachen und der Datenbank statt innerhalb der Datenbank selbst umzuwandeln.

Datensatzwertausdrücke

In den ursprünglichen SQL-Standards wie SQL-86 und SQL-89 hatten die meisten Operationen nur mit einem einzelnen Wert oder einer einzelnen Spalte einer Tabellenzeile zu tun. Um nun mit mehreren Werten arbeiten zu können, müssen Sie komplexe Ausdrücke bilden, indem Sie mit *logischen Verknüpfungen* arbeiten (die ich in Kapitel 10 beschreibe).

SQL-92 führt *Datensatzwertausdrücke* ein, die eine Liste von Werten oder Spalten statt eines einzelnen Werts oder einer einzelnen Spalte ansprechen. Ein Datensatzwertausdruck besteht aus einer Liste von Ausdrücken, die von Klammern eingeschlossen und durch Kommata voneinander getrennt sind. Sie können entweder mit einem kompletten Datensatz oder nur mit einer ausgewählten Untermenge des Datensatzes arbeiten.

In Kapitel 6 beschreibe ich, wie Sie mit INSERT eine neue Zeile in eine Tabelle einfügen können. Diese Anweisung benutzt einen Datensatzwertausdruck. Betrachten Sie das folgende Beispiel:

```
INSERT INTO Nahrungsmittel
   (Bezeichnung, Kalorien, Eiweiß, Fett, Kohlenhydrate)
   VALUES
   ('Käse, Edamer', 398, 25, 32.2, 2.1);
```

In diesem Beispiel ist ('Käse, Edamer', 398, 25, 32.2, 2.1) ein Datensatzwertausdruck. Bei einer Anweisung vom Typ INSERT kann dieser Datensatzwertausdruck Nullwerte und Standardwerte enthalten. (Ein *Standardwert* ist ein Wert, den eine Tabellenspalte annimmt, wenn Sie keinen anderen Wert angeben.) Bei der folgenden Zeile handelt es sich um einen gültigen Datensatzwertausdruck:

```
('Käse, Edamer', 398, NULL, 32.2, DEFAULT)
```

Sie können mehrere Datensätze auf einmal in eine Tabelle einfügen, indem Sie in einer VALUES-Klausel mehrere Datensatzwertausdrücke angeben:

```
INSERT INTO Nahrungsmittel
   (Bezeichnung, Kalorien, Eiweiß, Fett, Kohlenhydrate)
   VALUES
   ('Salat', 14, 1.2, 0.2, 2.5),
   ('Margarine', 720, 0.6, 81.0, 0.4),
   ('Senf', 75, 4.7, 4.4, 6.4),
   ('Spaghetti', 148, 5.0, 0.5, 30.1);
```

Mit Datensatzwertausdrücken können Sie bei Vergleichen Tipparbeit sparen. Wenn Sie etwa zwei Nährwerttabellen, eine in Deutsch und die andere in Spanisch, vergleichen und die Zeilen ermitteln wollen, die einander genau entsprechen, müssen Sie ohne den Einsatz von Datensatzwertausdrücken folgende Anweisung formulieren:

```
SELECT * FROM Nahrungsmittel
   WHERE Nahrungsmittel.Kalorien = Comida.Caloria
      AND Nahrungsmittel.Eiweiß = Comida.Proteina
      AND Nahrungsmittel.Fett = Comida.Gordo
      AND Nahrungsmittel.Kohlenhydrate
         = Comida.Carbohidrato;
```

Mit Datensatzwertausdrücken können Sie dieselbe Logik folgendermaßen ausdrücken:

```
SELECT * FROM Nahrungsmittel
   WHERE (Nahrungsmittel.Kalorien, Nahrungsmittel.Eiweiß,
      Nahrungsmittel.Fett, Nahrungsmittel.Kohlenhydrate)
    =
      (Comida.Caloria, Comida.Proteina, Comida.Gordo,
       Comida.Carbohidrato);
```

In diesem Beispiel sparen Sie nicht viel Schreibarbeit. Der Nutzen wäre etwas größer, wenn Sie eine größere Zahl von Spalten vergleichen. In den Fällen, in denen die alternative Schreibweise nur so wenig Nutzen bringt wie in diesem Beispiel, sollten Sie bei der älteren Syntax bleiben, weil deren Bedeutung klarer ist.

Datensatzwertausdrücke haben gegenüber einzeln codierten Ausdrücken den Vorteil, dass sie viel schneller sind. Im Prinzip könnte eine sehr clevere Implementierung auch die einzeln codierte Version analysieren und wie eine Zeilenwertversion implementieren, aber praktisch ist diese Optimierung so schwierig, dass (meines Wissens nach) kein DBMS auf dem Markt sie durchführt.

Daten zielsicher finden

10

In diesem Kapitel

- Die Tabellen festlegen, mit denen Sie arbeiten wollen
- Die gewünschten Zeilen herausfiltern
- Effiziente WHERE-Klauseln schreiben
- Mit Nullwerten umgehen
- Zusammengesetzte Ausdrücke über logische Verknüpfungen erzeugen
- Abfrageergebnisse nach Spalten gruppieren
- Abfrageergebnisse sortieren

Ein Datenbankverwaltungssystem hat zwei Hauptfunktionen: Daten zu speichern und einen leichten Zugriff auf diese Daten zu ermöglichen. Daten zu speichern ist nichts Besonderes. Das leistet auch ein Ablageschrank. Der schwierige Teil besteht darin, einen leichten Zugriff auf die Daten zu ermöglichen. Dabei geht es darum, die wenigen benötigten Daten von der (normalerweise) großen verbleibenden Datenmenge zu trennen, die in diesem Augenblick von Ihnen nicht gebraucht wird.

Mit SQL können Sie anhand der Merkmale von Daten entscheiden, ob eine bestimmte Zeile für Sie von Interesse ist oder nicht. Die Anweisungen SELECT, DELETE und UPDATE zeigen der Datenbankengine (der Teil des Datenbankverwaltungssystems, der direkt mit den Daten interagiert), welche Zeilen ausgewählt, gelöscht oder aktualisiert werden sollen. Um diese Datensatzauswahl Ihren Anforderungen entsprechend zu verfeinern, können Sie den genannten Anweisungen *modifizierende Klauseln* hinzufügen.

Modifizierende Klauseln

Die modifizierenden Klauseln in SQL lauten: FROM, WHERE, HAVING, GROUP BY und ORDER BY. Mit der Klausel FROM legen Sie die Tabelle oder die Tabellen fest, mit denen Sie arbeiten möchten. Mit den Klauseln WHERE und HAVING legen Sie die Auswahlkriterien fest, die von den Daten eines Datensatzes erfüllt werden müssen, damit die gewünschte Operation für diesen Datensatz ausgeführt wird. Mit den Klauseln GROUP BY und ORDER BY geben Sie an, wie die abgerufenen Datensätze angezeigt werden sollen. Tabelle 10.1 fasst diese Aussagen noch einmal übersichtlich zusammen.

Modifizierende Klausel	Funktion
FROM	Gibt die Tabellen an, deren Daten verwendet werden sollen
WHERE	Filtert Datensätze heraus, die die Suchbedingungen erfüllen
GROUP BY	Gruppiert Datensätze anhand der Werte, die sich in den angegebenen Spalten befinden
HAVING	Filtert Gruppen heraus, die bestimmte Suchkriterien erfüllen
ORDER BY	Sortiert das Ergebnis der vorherigen Klauseln, um die endgültige Ausgabe zu generieren

Tabelle 10.1: Modifizierende Klauseln und ihre Funktionen

Werden mehrere Klauseln verwendet, müssen sie in der folgenden Reihenfolge angegeben werden:

```
SELECT Spaltenliste
   FROM Tabellenliste
   [WHERE Suchkriterium]
   [GROUP BY Gruppierungsspalten]
   [HAVING Suchkriterium]
   [ORDER BY Sortierkriterien] ;
```

Was machen diese Klauseln?

- ✔ Die Klausel WHERE filtert die Datensätze heraus, die ein bestimmtes Suchkriterium erfüllen, und weist die anderen Datensätze zurück.
- ✔ Die Klausel GROUP BY ordnet Datensätze, die von der WHERE-Klausel zurückgegeben werden, in Abhängigkeit von dem Wert neu an, der in der angegebenen Gruppierungsspalte steht.
- ✔ Die Klausel HAVING ist ein zusätzlicher Filter, der jede von der GROUP BY-Klausel gebildete Gruppe herausfiltert, die die Suchbedingungen erfüllt. Alle anderen Gruppen werden verworfen.
- ✔ Die Klausel ORDER BY sortiert die übrig gebliebenen Daten anhand bestimmter Sortierkriterien.

Die eckigen Klammern [] zeigen an, dass die Klauseln WHERE, GROUP BY, HAVING und ORDER BY optional sind.

SQL verarbeitet diese Klauseln in der Reihenfolge FROM, WHERE, GROUP BY, HAVING und schließlich SELECT. Dabei funktionieren die Klauseln wie eine Pipeline – jede Klausel erhält das Ergebnis der vorherigen Klausel und erzeugt die Verarbeitungsgrundlage für die nächste Klausel. Die Auswertungsreihenfolge wird funktional folgendermaßen dargestellt:

```
SELECT(HAVING(GROUP BY(WHERE(FROM...))))
```

ORDER BY wird nach SELECT ausgeführt, deshalb kann ORDER BY nur auf Spalten der SELECT-Liste verweisen. ORDER BY kann keine anderen Spalten der Tabellen in der FROM-Liste verarbeiten.

Die Klausel FROM

Die Klausel FROM ist leicht zu verstehen, wenn Sie, wie im folgenden Beispiel, nur eine Tabelle angeben:

```
SELECT * FROM Verkauf ;
```

Diese Anweisung gibt alle Daten aller Spalten und Zeilen der Tabelle Verkauf zurück. Sie können natürlich auch mehr als eine Tabelle in einer FROM-Klausel angeben, wie das folgende Beispiel zeigt:

```
SELECT *
   FROM Kunde, Verkauf ;
```

Diese Anweisung bildet eine virtuelle Tabelle (siehe Kapitel 6), die die Daten der Tabelle Kunde mit den Daten der Tabelle Verkauf kombiniert. Jede Zeile in der Tabelle Kunde wird mit jeder Zeile der Tabelle Verkauf verbunden, um die neue virtuelle Tabelle zu bilden. Diese Tabelle enthält so viele Zeilen, wie die Zeilenzahl der Tabelle Kunde multipliziert mit der Zeilenzahl der Tabelle Verkauf ergibt. Wenn die Tabelle Kunde zehn und die Tabelle Verkauf hundert Zeilen enthält, verfügt die neue virtuelle Tabelle über tausend Zeilen.

Diese Operation wird als *kartesisches Produkt* der beiden Tabellen bezeichnet. Das kartesische Produkt ist eine Form einer JOIN-Operation, die ich in Kapitel 11 detailliert beschreibe.

Bei den meisten Anwendungen enthält der überwiegende Teil der Zeilen, die durch das kartesische Produkt entstehen, nur unwichtige Informationen. Bei der virtuellen Tabelle, die aus den Tabellen Kunde und Verkauf gebildet wird, sind nur die Zeilen von Interesse, bei denen die Spalte KundenID der Tabelle Kunde mit der Spalte KundenID der Tabelle Verkauf übereinstimmt. Die anderen Zeilen können mit einer WHERE-Klausel herausgefiltert werden.

Die Klausel WHERE

Ich habe in diesem Buch die Klausel WHERE schon wiederholt verwendet, ohne sie wirklich zu erklären, weil Bedeutung und Verwendung der Klausel eigentlich offensichtlich sind: Eine Anweisung führt eine Operation (zum Beispiel SELECT, DELETE oder UPDATE) nur für die Zeilen aus, bei denen die Bedingung erfüllt ist, die in der Klausel WHERE angegeben wird. Die Syntax der WHERE-Klausel lautet:

```
SELECT Spaltenliste
   FROM Tabellenname
   WHERE Bedingung ;

DELETE FROM Tabellenname
   WHERE Bedingung ;

UPDATE Tabellenname
   SET Spalte1=Wert1, Spalte2=Wert2, ..., SpalteN=WertN
   WHERE Bedingung ;
```

Die Bedingung in der WHERE-Klausel kann einfach oder beliebig komplex sein. Sie können mehrere Bedingungen durch die logischen Operatoren AND, OR und NOT (die ich weiter hinten in diesem Kapitel beschreibe) zu einer einzigen Bedingung verknüpfen.

Die folgenden Anweisungen zeigen einige typische WHERE-Klauseln:

```
WHERE Kunde.KundenID = Verkauf.KundenID
WHERE Nahrungsmittel.Kalorien = Comida.Caloria
WHERE Nahrungsmittel.Kalorien < 219
WHERE Nahrungsmittel.Kalorien > 3 * :basisWert
WHERE Nahrungsmittel.Kalorien < 219 AND
      Nahrungsmittel.Protein > 27.4
```

Die Bedingungen dieser WHERE-Klauseln werden auch als Prädikate bezeichnet. Ein *Prädikat* ist ein Ausdruck, der eine Behauptung über einen Wert aufstellt.

So ist beispielsweise das Prädikat `Nahrungsmittel.Kalorien < 219` wahr, wenn der Wert der Spalte `Nahrungsmittel.Kalorien` des aktuellen Datensatzes kleiner als 219 ist. Wenn die Behauptung *wahr* ist, erfüllt sie die Bedingung. Eine Behauptung kann wahr (TRUE), falsch (FALSE) oder unbekannt (UNKNOWN) sein. Sie ist unbekannt, wenn ein oder mehrere Elemente in der Behauptung Nullwerte sind. Am häufigsten kommen die *Vergleichsprädikate* =, <, >, <>, <= und >= zum Einsatz, aber SQL enthält eine Reihe weiterer Prädikate, mit denen die Möglichkeiten, gewünschte Datenelemente herauszufiltern, sehr erweitert werden. Die folgende Liste zeigt die Prädikate, die Sie zum Filtern benutzen können:

- ✔ Vergleichsprädikate
- ✔ BETWEEN
- ✔ IN [NOT IN]
- ✔ LIKE [NOT LIKE]
- ✔ NULL
- ✔ ALL, SOME, ANY
- ✔ EXISTS
- ✔ UNIQUE

- ✔ OVERLAPS
- ✔ MATCH
- ✔ SIMILAR
- ✔ DISTINCT

Vergleichsprädikate

Die Beispiele in dem letzten Abschnitt zeigen die typischen Verwendungsmöglichkeiten für Prädikate, mit denen Sie zwei Werte miteinander vergleichen. Für jede Zeile, für die der Vergleich den Wert TRUE ergibt, ist die WHERE-Klausel erfüllt, wodurch die Operation (SELECT, UPDATE, DELETE und so weiter) auf diesen Datensatz ausgeführt wird. Datensätze, bei denen der Vergleich den Wert FALSE ergibt, werden übersprungen. Betrachten Sie die folgende SQL-Anweisung:

```
SELECT * FROM Nahrungsmittel
   WHERE Kalorien < 219 ;
```

Diese Anweisung zeigt alle Datensätze der Tabelle Nahrungsmittel an, die in der Spalte Kalorien einen Wert enthalten, der kleiner als 219 ist.

Tabelle 10.2 stellt die sechs Vergleichsprädikate in einer Übersicht dar.

Vergleich	Symbol
Gleich	=
Ungleich	<>
Kleiner als	<
Kleiner als oder gleich	<=
Größer als	>
Größer als oder gleich	>=

Tabelle 10.2: Vergleichsprädikate in SQL

BETWEEN

Manchmal möchten Sie eine Zeile auswählen, deren Wert in einem bestimmten Wertebereich liegt. Eine Möglichkeit, um diese Auswahl zu tätigen, ist der Einsatz eines Vergleichsprädikats. Sie können beispielsweise eine WHERE-Klausel formulieren, um alle Datensätze der Tabelle Nahrungsmittel auszuwählen, die in der Spalte Kalorien einen Wert enthalten, der größer als 100 und kleiner als 300 ist:

```
WHERE Nahrungsmittel.Kalorien > 100
   AND Nahrungsmittel.Kalorien < 300
```

Dieser Vergleich schließt die Nahrungsmittel aus, deren Werte genau 100 oder 300 Kalorien betragen. Wenn Sie diese Grenzwerte ebenfalls einschließen wollen, können Sie folgende Anweisung verwenden:

```
WHERE Nahrungsmittel.Kalorien >= 100
   AND Nahrungsmittel.Kalorien <= 300
```

Ein weiterer Weg, einen Bereich vorzugeben und seine Grenzwerte in die Suche einzuschließen, ist die Verwendung des Prädikats BETWEEN:

```
WHERE Nahrungsmittel.Kalorien BETWEEN 100 AND 300
```

Diese Klausel erfüllt dieselbe Funktion wie das Beispiel davor mit den Vergleichsprädikaten. Diese Formulierung spart ein wenig Schreibarbeit und ist wahrscheinlich etwas intuitiver zu verstehen als zwei Vergleichsprädikate, die mit einem logischen AND verknüpft sind.

Das Schlüsselwort BETWEEN kann verwirrend sein, weil es nicht explizit ausdrückt, ob die Grenzwerte dazugehören oder nicht. Tatsächlich gehören sie immer zu dem angegebenen Wertebereich. BETWEEN drückt ebenfalls nicht aus, dass der erste Teil des Vergleichs kleiner oder so groß wie der zweite sein muss. Falls beispielsweise Nahrungsmittel.Kalorien den Wert 200 enthält, gibt die folgende Klausel den Wert *wahr* zurück:

```
WHERE NAHRUNGSMITTEL.KALORIEN BETWEEN 100 AND 300
```

Dagegen liefert die folgende Klausel, von der Sie möglicherweise meinen könnten, dass sie zur voranstehenden Klausel gleichwertig ist, genau das Gegenteil zurück, nämlich den Wert FALSE:

```
WHERE NAHRUNGSMITTEL.KALORIEN BETWEEN 300 AND 100
```

Wenn Sie mit BETWEEN arbeiten, müssen Sie garantieren können, dass der erste Ausdruck nicht größer als der zweite ist.

Sie können BETWEEN für Zeichenfolgen, Bitketten und Datums- und Zeitwerte sowie für die numerischen Typen verwenden. Betrachten Sie das folgende Beispiel:

```
SELECT Vorname, Nachname
   FROM Kunde
   WHERE Kunde.Nachname BETWEEN 'A' AND 'Mzzz' ;
```

Dieses Beispiel gibt alle Kunden zurück, deren Nachname in der ersten Hälfte des Alphabets liegt.

IN und NOT IN

Mit den Prädikaten `IN` und `NOT IN` können Sie prüfen, ob eine Menge (wie die Staaten Europas) bestimmte Elemente (wie DK, SE, NO und FI) enthält oder nicht. Angenommen, Ihre Firma hat in einer Tabelle mit der Bezeichnung `Lieferant` die Lieferanten eines Artikels abgelegt, den Sie regelmäßig beziehen. Sie wollen die Telefonnummern der Lieferanten wissen, die im nördlichen Teil Europas beheimatet sind. Sie können die Telefonnummern folgendermaßen durch Vergleichsprädikate ermitteln:

```
SELECT Firma, Telefon
   FROM Lieferant
   WHERE Staat = 'DK' OR
         Staat = 'SE' OR
         Staat = 'NO' OR
         Staat = 'FI';
```

Sie können aber auch das Prädikat `IN` benutzen, um die gleiche Aufgabe zu lösen:

```
SELECT Firma, Telefon
   FROM Lieferant
   WHERE Staat IN ('DK', 'SE', 'NO', 'FI') ;
```

Diese Formulierung ist etwas kompakter als diejenige, die die Vergleichsprädikate und das logische `OR` verwendet.

Die `NOT IN`-Version dieses Prädikats funktioniert auf die gleiche Weise. Angenommen, Sie suchen alle Lieferanten, die nicht in bestimmten Staaten beheimatet sind. Dann können Sie die folgende Konstruktion einsetzen:

```
SELECT Firma, Telefon
   FROM Lieferant
   WHERE Staat NOT IN ('DK', 'SE', 'NO', 'FI') ;
```

Mit dem Schlüsselwort `IN` können Sie Schreibarbeit sparen, und zwar umso mehr, je mehr Elemente in der Menge enthalten sind. Sie können dieselbe Arbeit aber auch mit den Vergleichsprädikaten des ersten Beispiels erledigen.

Selbst wenn Sie mit `IN` nicht sehr viel an Schreibarbeit sparen können, gibt es einen guten Grund, dieses Prädikat statt der Vergleichsprädikate zu nutzen. Höchstwahrscheinlich implementiert Ihr DBMS die beiden Methoden unterschiedlich, und es kann gut sein, dass eine Methode wesentlich schneller ist als die andere. Sie sollten deshalb ausprobieren, welches Verfahren auf Ihrem System schneller läuft, und dann dieses Verfahren verwenden. Ein DBMS mit einem guten Optimierer verwendet wahrscheinlich das effizienteste Verfahren, und zwar unabhängig davon, welches Prädikat Sie angeben.

Das Schlüsselwort `IN` ist auch in einem anderen Bereich nützlich. Wenn `IN` Teil einer Unterabfrage ist, erhalten Sie damit die Möglichkeit, Informationen aus zwei Tabellen zu verwerten und dadurch Ergebnisse zu erhalten, die eine einzelne Tabelle nicht liefern könnte. Ich gehe zwar erst in Kapitel 12 detailliert auf Unterabfragen ein, zeige aber im folgenden Beispiel, wie eine Unterabfrage das Schlüsselwort `IN` benutzt.

Angenommen, Sie möchten die Namen aller Kunden anzeigen, die in den letzten 30 Tagen den Artikel mit der Artikelnummer F–117A gekauft haben. Die Kundennamen sind in der Tabelle Kunde und die Verkaufstransaktionen in der Tabelle Transaktion gespeichert. Sie können folgende Abfrage formulieren:

```
SELECT Vorname, Nachname
   FROM Kunde
   WHERE KundenID IN
      (SELECT KundenID
         FROM Transaktion
         WHERE ArtikelID = 'F–117A'
         AND TransDate >= (CurrentDate – 30)) ;
```

Die innere SELECT-Anweisung der Tabelle Transaktion ist in die äußere SELECT-Anweisung der Tabelle Kunde eingebettet. Das innere SELECT ermittelt die Kennungen aller Kunden, die den Artikel F–117A in den letzten 30 Tagen gekauft haben. Das äußere SELECT gibt die Vornamen und Nachnamen dieser Kunden an.

LIKE und NOT LIKE

Mit dem Prädikat LIKE können Sie Zeichenfolgen daraufhin vergleichen, ob sie teilweise übereinstimmen. Dies ist besonders dann nützlich, wenn Sie die genaue Form eines Suchbegriffs nicht kennen. Sie können teilweise Übereinstimmungen auch dazu benutzen, mehrere Zeilen abzurufen, die in einer Tabellenspalte ähnliche Zeichenfolgen enthalten.

Teilweise Übereinstimmungen werden in SQL durch zwei Platzhalterzeichen definiert. Das Prozentzeichen (%) steht für eine beliebige Zeichenfolge mit null oder mehr Zeichen. Der Unterstrich (_) steht für ein einzelnes Zeichen. Tabelle 10.3 zeigt einige Beispiele für die Verwendung von LIKE.

Befehl	Rückgabewert
WHERE Wort LIKE 'intern%'	Intern
	Internal
	International
	Internet
	Interna
WHERE Wort LIKE '%Frieden%'	Gerechtigkeit des Friedens
	Friedenskrieger
WHERE Wort LIKE 't_p_'	Tape
	Taps
	Tipi
	Tops
	Type

Tabelle 10.3: LIKE *in SQL*

Mit dem Prädikat NOT LIKE können Sie alle Zeilen abrufen, bei denen eine teilweise Übereinstimmung, einschließlich eines oder mehrerer Platzhalterzeichen, nicht gegeben ist:

```
WHERE Telefon NOT LIKE '503%'
```

Dieses Beispiel gibt alle Tabellenzeilen zurück, in denen die Telefonnummer nicht mit 503 beginnt.

Wenn Sie nach einer Zeichenfolge suchen, die ein Prozentzeichen oder einen Unterstrich enthält, darf SQL das Prozentzeichen nicht als Platzhalter interpretieren, sondern muss es als Prozentzeichen werten. Sie können dies erreichen, indem Sie unmittelbar vor das Platzhalterzeichen ein Escape-Zeichen setzen (dieser Vorgang wird auch *Maskieren* genannt). Sie können jedes Zeichen als Escape-Zeichen verwenden, solange es selbst nicht Bestandteil des Suchstrings ist. Escape-Zeichen werden mit dem Schlüsselwort ESCAPE definiert:

```
SELECT Zitat
   FROM Bartletts
   WHERE Info LIKE '20#%'
      ESCAPE '#' ;
```

Das Prozentzeichen (%) wird durch das Escape-Zeichen (#) maskiert, damit SQL dieses Prozentzeichen buchstäblich und nicht als Platzhalter interpretiert. Sie können auf die gleiche Weise einen Unterstrich oder das Escape-Zeichen selbst maskieren. Die voranstehende Abfrage findet folgende Information:

```
20% der Verkäufer erzielen 80% des Ergebnisses
```

Die Abfrage findet außerdem alles, in dem die Zeichenfolge 20% vorkommt.

SIMILAR

In SQL:1999 wurde das Prädikat SIMILAR neu eingeführt, das eine leistungsstärkere Methode als LIKE zur Verfügung stellt, um teilweise Übereinstimmungen zu finden. Sie sind mit dem Prädikat SIMILAR in der Lage, eine Zeichenfolge mit einem regulären Ausdruck zu vergleichen. Stellen Sie sich beispielsweise vor, dass Sie die Spalte Betriebssystem einer Tabelle durchsuchen wollen, um Programme zu finden, die mit Microsoft Windows kompatibel sind. Dann könnten Sie die folgende WHERE-Klausel konstruieren:

```
WHERE Betriebssystem SIMILAR TO
'('Windows '(3.1 | 95 | 98 | ME | CE | NT | 2000 | XP | Vista | 7 | 8))'
```

Dieses Prädikat wählt alle Zeilen aus, die eines der angegebenen Betriebssysteme von Microsoft enthalten.

NULL

Mit dem Prädikat NULL können Sie alle Zeilen ermitteln, die in einer angegebenen Spalte einen Nullwert enthalten. Die Tabelle Nahrungsmittel aus Kapitel 8 enthält mehrere Zeilen, die in der Spalte Kohlenhydrate Nullwerte enthalten. Sie können diese Zeilen folgendermaßen abfragen:

```
SELECT (Nahrung)
   FROM Nahrungsmittel
   WHERE Kohlenhydrate IS NULL ;
```

Diese Abfrage gibt folgende Werte zurück:

```
Rindfleisch, magerer Hamburger
Huhn, helles Fleisch
Opossum, geröstetes
Schwein, Schinken
```

Wenn Sie zusätzlich das Schlüsselwort NOT benutzen, wird das Ergebnis umgekehrt:

```
SELECT (Nahrung)
   FROM Nahrungsmittel
   WHERE Kohlenhydrate IS NOT NULL ;
```

Diese Abfrage gibt alle Tabellenzeilen außer den eben genannten zurück.

Kohlenhydrate IS NULL ist nicht dasselbe wie Kohlenhydrate = NULL. Damit Sie sich den Unterschied besser verdeutlichen können, gehen wir davon aus, dass die beiden Spalten Kohlenhydrate und Eiweiß in der aktuellen Zeile der Tabelle Nahrungsmittel einen Nullwert enthalten. Damit gelten die folgenden Aussagen:

- ✔ Kohlenhydrate IS NULL ist wahr.
- ✔ Eiweiß IS NULL ist wahr.
- ✔ Kohlenhydrate IS NULL AND Eiweiß IS NULL ist wahr.
- ✔ Kohlenhydrate = Eiweiß ist unbekannt.
- ✔ Kohlenhydrate = NULL ist ein ungültiger Ausdruck.

Es macht keinen Sinn, das Schlüsselwort NULL in einem Vergleich zu verwenden, weil das Ergebnis immer *unbekannt* ist.

Warum wird Kohlenhydrate = Eiweiß als *unbekannt* bezeichnet, obwohl sowohl Kohlenhydrate als auch Eiweiß einen Nullwert enthalten? Weil NULL einfach nur bedeutet: »Ich weiß es nicht.« Sie wissen nicht, welchen Wert Kohlenhydrate hat, und Sie wissen nicht, welchen Wert Eiweiß hat; deshalb können Sie auch nicht wissen, ob diese beiden (unbekannten) Werte gleich sind. Vielleicht hat die Spalte Kohlenhydrate den Wert 37 und die Spalte Eiweiß den Wert 14 oder vielleicht enthält die Spalte Kohlenhydrate den Wert 93 und die Spalte Eiweiß den Wert 93. Wenn Sie beide Werte nicht kennen, können Sie auch nicht sagen, ob sie gleich sind.

ALL, SOME, ANY

Vor etwa 2400 Jahren hat der griechische Philosoph Aristoteles die Gesetze der Logik formuliert, die zur Grundlage eines großen Teils des westlichen Denkens wurden. Das Wesen dieser Logik besteht darin, von bestimmten Prämissen auszugehen, deren Wahrheitswert man kennt, und daraus Schlussfolgerungen abzuleiten und zu neuen Wahrheiten zu gelangen. Ein klassisches Beispiel für diese Prozedur lautet:

Prämisse 1: Alle (ALL) Griechen sind Menschen.

Prämisse 2: Alle Menschen sind sterblich.

Schluss: Alle Griechen sind sterblich.

Ein weiteres Beispiel:

Prämisse 1: Einige (SOME) Griechen sind Frauen.

Prämisse 2: Alle Frauen sind Menschen.

Schluss: Einige Griechen sind Menschen.

Als drittes Beispiel möchte ich das zweite Beispiel etwas anders formulieren:

Wenn einige (ANY) Griechen Frauen sind und alle Frauen Menschen sind, dann sind einige Griechen Menschen.

Das erste Beispiel benutzt den Allquantor ALL (alle) in beiden Prämissen, aus denen eine schlüssige Aussage über alle Griechen abgeleitet werden kann. Das zweite Beispiel benutzt den Existenzquantor SOME (einige) in einer Prämisse, die einen Schluss über einige Griechen ermöglicht. Das dritte Beispiel verwendet den Existenzquantor ANY, der ein Synonym für SOME ist, um zu demselben Schluss zu kommen wie das zweite Beispiel.

Wie werden SOME, ANY und ALL in SQL verwendet?

Angenommen, Sie sind ein Spielervermittler für die Erste und die Zweite Fußballbundesliga. Sie haben sich auf die Entdeckung von Talenten in der Zweiten Bundesliga spezialisiert. Sie führen zwei separate Tabellen mit allen Stürmern der Ersten und Zweiten Bundesliga mit den Namen und den Toren aller Spieler.

Mit der folgenden Anweisung können Sie ermitteln, welche Spieler in der Zweiten Bundesliga mehr Tore geschossen haben als die besten Stürmer der Ersten Bundesliga:

```
SELECT Vorname, Nachname
   FROM ZweiteLiga
   WHERE Tore > ALL
      (SELECT Tore FROM ErsteLiga) ;
```

Die Unterabfrage (die innere SELECT-Anweisung) gibt eine Liste mit allen Toren der Spieler der Ersten Bundesliga zurück. Die äußere Abfrage gibt die Vornamen und Nachnamen der Spieler in der Zweiten Bundesliga zurück, die mehr Tore geschossen haben als alle Stürmer der Ersten Bundesliga.

Wenn Sie dagegen wissen wollen, welche Spieler der Zweiten Bundesliga mehr Tore geschossen haben als die schlechtesten Stürmer der Ersten Bundesliga, können Sie folgende Anweisung verwenden:

```
SELECT Vorname, Nachname
   FROM ZweiteLiga
   WHERE Tore > ANY
      (SELECT Tore FROM ErsteLiga) ;
```

In diesem Fall benutzen Sie den Existenzquantor `ANY` statt des Allquantors `ALL`. Die Unterabfrage (der innere `SELECT`-Befehl) ist identisch mit der Unterabfrage in dem vorangegangenen Beispiel. Diese Unterabfrage gibt eine Liste mit allen Toren der Spieler der Ersten Bundesliga zurück. Die äußere Abfrage gibt die Vornamen und Nachnamen der Spieler der Zweiten Bundesliga zurück, die mehr Tore geschossen haben als der schlechteste Stürmer der Ersten Bundesliga. Weil dieser wahrscheinlich kein Tor geschossen hat, gibt diese Anweisung alle Torschützen der Zweiten Liga zurück.

Sie können das Schlüsselwort `ANY` durch das synonyme Schlüsselwort `SOME` ersetzen, das Ergebnis bleibt gleich. Wenn die Aussage, dass wenigstens ein Spieler während eines Spiels kein Tor geschossen hat, wahr ist, kann man sagen, dass `SOME` (irgendein) Spieler während eines Spiels kein Tor geschossen hat.

ANY kann mehrdeutig sein

Ursprünglich benutzte SQL das Wort `ANY` als Existenzquantor. Diese Verwendung erwies sich als verwirrend und fehleranfällig, weil das Wort *any* im Englischen manchmal die Konnotation »allumfassend« und manchmal »existierend, überhaupt ein« hat.

Deshalb wurde im SQL-92-Standard zwar das Wort `ANY` aus Kompatibilitätsgründen beibehalten, aber zusätzlich als zweiter Existenzquantor das Wort `SOME` eingeführt, was als Synonym weniger verwirrend ist. SQL unterstützt weiterhin beide Existenzquantoren.

EXISTS

Mit dem Prädikat `EXISTS` können Sie feststellen, ob eine Unterabfrage überhaupt eine Zeile zurückgibt. Wenn die Unterabfrage wenigstens eine Zeile zurückgibt, ist die `EXISTS`-Bedingung erfüllt, und die äußere Abfrage wird ausgeführt. Betrachten Sie das folgende Beispiel:

```
SELECT Vorname, Nachname
   FROM Kunde
   WHERE EXISTS
      (SELECT DISTINCT KundenID
      FROM Verkauf
      WHERE Verkauf.KundenID = Kunde.KundenID);
```

Sie finden in der Tabelle Verkauf alle Verkäufe Ihrer Firma. Sie enthält die Kennung des jeweiligen Käufers sowie andere verkaufsspezifische Daten. Die Tabelle Kunde enthält ebenfalls die Kennung des Kunden sowie seinen Vornamen und Nachnamen, aber keine Informationen über die einzelnen Verkaufsvorgänge.

Die Unterabfrage dieses Beispiels gibt für jeden Kunden eine Zeile zurück, der wenigstens einmal etwas gekauft hat. Die äußere Abfrage gibt den Vornamen und den Nachnamen der Kunden zurück.

Der Einsatz von EXISTS entspricht einem Vergleich von COUNT mit null, wie die folgende Abfrage zeigt:

```
SELECT Vorname, Nachname
  FROM Kunde
  WHERE 0 <>
    (SELECT COUNT(*)
      FROM Verkauf
      WHERE Verkauf.KundenID = Kunde.KundenID);
```

Diese Anweisung zeigt für jede Zeile in der Tabelle Verkauf, die eine Kundennummer enthält, die mit der entsprechenden Kundennummer der Tabelle Kunde übereinstimmt, die Spalten Vorname und Nachname der Tabelle Kunde an. Somit wird für jeden Verkauf, der in der Tabelle Verkauf vermerkt ist, der Name des Kunden angezeigt, der den Kauf getätigt hat.

UNIQUE

Das Prädikat UNIQUE wird wie das Prädikat EXISTS bei Unterabfragen verwendet. Während EXISTS den Wert *wahr* zurückgibt, wenn die Unterabfrage wenigstens eine Zeile enthält, wird das Prädikat UNIQUE nur dann auf *wahr* gesetzt, wenn die Zeilen, die von der Unterabfrage geliefert werden, eindeutig sind. Das Prädikat hat nur dann den Wert *wahr*, wenn die Unterabfrage keine doppelten Zeilen enthält. Betrachten Sie das folgende Beispiel:

```
SELECT Vorname, Nachname
   FROM Kunde
   WHERE UNIQUE
     (SELECT KundenID FROM Verkauf
         WHERE Verkauf.KundenID = Kunde.KundenID);
```

Diese Anweisung gibt die Namen aller neuen Kunden zurück, für die es in der Tabelle Verkauf nur einen Datensatz gibt. Da Nullwerte unbekannte Werte sind, werden sie nicht als gleich eingestuft und sind deshalb eindeutig. Wenn UNIQUE auf eine Ergebnistabelle angewendet wird, die nur zwei Zeilen mit Nullwerten enthält, erhält das Prädikat UNIQUE den Wert *wahr*.

DISTINCT

Das Prädikat DISTINCT ähnelt dem Prädikat UNIQUE, unterscheidet sich aber von diesem in der Art und Weise, wie es Nullwerte behandelt. Wenn alle Werte in einer Ergebnistabelle

UNIQUE (eindeutig) sind, sind sie auch DISTINCT (verschieden) voneinander. Doch im Gegensatz zu UNIQUE erhält DISTINCT den Wert FALSE, wenn die Ergebnistabelle nur Zeilen mit Nullwerten enthält. Zwei Nullwerte werden *nicht* als verschieden eingestuft, während sie zugleich als eindeutig betrachtet werden.

Diese seltsame Situation scheint ein Widerspruch in sich zu sein, aber es gibt einen Grund dafür. Es kann zu Situationen kommen, in denen Sie zwei Nullwerte als gleich, und andere, in denen Sie diese Werte als verschieden behandeln wollen. Im ersten Fall verwenden Sie das Prädikat UNIQUE, im zweiten DISTINCT.

OVERLAPS

Mit dem Prädikat OVERLAPS können Sie feststellen, ob sich zwei Zeitintervalle überlappen. Damit sind Sie beispielsweise in der Lage, Planungskonflikte zu vermeiden. Wenn sich die beiden Intervalle überlappen, gibt das Prädikat den Wert *wahr*, andernfalls den Wert *falsch* zurück.

Sie können Intervalle auf zwei Arten festlegen: entweder durch einen Startzeitpunkt und einen Endzeitpunkt oder durch einen Startzeitpunkt und eine Dauer. Schauen Sie sich die folgenden Beispiele an:

```
(TIME '2:55:00', INTERVAL '1' HOUR)
OVERLAPS
(TIME '3:30:00', INTERVAL '2' HOUR)
```

Dieses Beispiel gibt den Wert *wahr* zurück, weil 3:30 Uhr weniger als eine Stunde später als 2:55 Uhr ist.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:29:00', TIME '9:31:00')
```

Dieses Beispiel gibt den Wert *wahr* zurück, weil sich beide Intervalle um eine Minute überlappen.

```
(TIME '9:00:00', TIME '10:00:00')
OVERLAPS
(TIME '10:15:00', INTERVAL '3' HOUR)
```

Dieses Beispiel gibt den Wert *falsch* zurück, weil sich die beiden Intervalle nicht überlappen.

```
(TIME '9:00:00', TIME '9:30:00)
OVERLAPS
(TIME '9:30:00', TIME '9:35:00)
```

Dieses Beispiel gibt den Wert *falsch* zurück, weil die beiden Intervalle zwar zusammenhängen, sich aber nicht überlappen.

MATCH

In Kapitel 5 beschreibe ich die referenzielle Integrität, die dazu dient, die Konsistenz in einer Datenbank mit mehreren Tabellen aufrechtzuerhalten. Sie können die Integrität zerstören, indem Sie eine Zeile in eine Kind-Tabelle einfügen, zu der es keine entsprechende Zeile in der Eltern-Tabelle gibt. Ähnliche Probleme entstehen, wenn Sie eine Zeile der Eltern-Tabelle löschen, wenn es zu dieser Zeile korrespondierende Zeilen in einer Kind-Tabelle gibt.

Angenommen, Ihr Unternehmen verwaltet in einer Tabelle `Kunde` Ihre Kunden und in einer Tabelle `Verkauf` alle Verkäufe. Sie wollen verhindern, dass Zeilen in die Tabelle `Verkauf` eingefügt werden können, bevor ein Kunde in der Tabelle `Kunde` angelegt worden ist. Weiterhin dürfen Kunden nur dann in der Tabelle `Kunde` gelöscht werden, wenn es für sie in der Tabelle `Verkauf` keine Einträge gibt.

Vor dem Einfügen oder Löschen müssen Sie also die betreffende Zeile prüfen, ob Sie durch die geplante Operation nicht die Integrität der Datenbank zerstören. Mit dem Prädikat `MATCH` können Sie eine solche Prüfung durchführen.

Betrachten wir das Prädikat `MATCH` im Zusammenhang mit unserem Beispiel. Die Spalte `KundenID` soll den Primärschlüssel der Tabelle `Kunde` bilden und in der Tabelle `Verkauf` als Fremdschlüssel dienen. Jede Zeile der Tabelle `Kunde` muss eine eindeutige Kundennummer (`KundenID`) enthalten, die kein Nullwert sein darf. In der Tabelle `Verkauf` braucht `KundenID` nicht eindeutig zu sein, weil ein Kunde ruhig mehrfach Käufe tätigen sollte. Dies gefährdet die Integrität nicht, weil `KundenID` in dieser Tabelle eben ein Fremdschlüssel und kein Primärschlüssel ist.

Eigentlich könnte die Spalte `KundenID` in der Tabelle `Verkauf` sogar Nullwerte enthalten, weil jemand einfach in Ihren Laden kommen, etwas kaufen und wieder gehen kann, ohne seinen Namen und seine Adresse zu hinterlassen. Dadurch könnten Zeilen in der Kind-Tabelle (`Verkauf`) entstehen, zu denen es keine entsprechenden Zeilen in der Eltern-Tabelle (`Kunde`) gibt. Um dieses Problem zu lösen, wird gerne ein generischer »Dummy«-Kunde in der Tabelle `Kunde` angelegt, dem alle anonymen Transaktionen zugewiesen werden.

Angenommen, eine Kundin käme zur Kasse und behauptete, am 1. April 2010 den Tarnkappenjäger F-117A gekauft zu haben. Obwohl sie ihre Quittung verloren hat, will die Kundin das Flugzeug zurückgeben, weil es auf gegnerischen Radarschirmen als Flugzeug erkannt würde. Sie können jetzt prüfen, ob die Kundin die F-117A wirklich gekauft hat, indem Sie die Datenbank mit einem `MATCH` durchsuchen. Zunächst müssen Sie die Kundennummer `KundenID` in der Variablen `varKundenID` ablegen, dann benutzen Sie in Ihrer Abfrage folgende Syntax:

```
... WHERE (:varKundenID, 'F-117A', '2010-04-01')
          MATCH
          (SELECT KundenID, ArtikelID, Verkaufsdatum
              FROM Verkauf)
```

Wenn das Prädikat `MATCH` *wahr* zurückliefert, enthält die Datenbank für diese Kundennummer am 1. April 2010 den Verkauf einer F-117A. Nehmen Sie das defekte Produkt zurück und zahlen Sie Ihre Kundin aus. (***Anmerkung:*** Falls im ersten Argument des Prädikats `MATCH` irgendein Wert ein Nullwert ist, wird immer der Wert *wahr* zurückgegeben.)

Die Prädikate MATCH und UNIQUE existieren beide aus demselben Grund: Sie stellen ein Verfahren zur Verfügung, mit dem Sie explizit die referenzielle Integrität (RI) und UNIQUE-Einschränkungen testen können.

Das Prädikat MATCH hat folgende allgemeine Form:

```
Zeilenwert MATCH  [UNIQUE] [SIMPLE | PARTIAL | FULL ] Unterabfrage
```

Die Optionen UNIQUE, PARTIAL und FULL beziehen sich auf Fälle, bei denen der Zeilenwert *Z* eine oder mehrere Spalten mit Nullwerten enthält (siehe auch Kapitel 8). Die Regeln für das Prädikat MATCH sind eine Kopie der entsprechenden Regeln für die referenzielle Integrität.

Regeln der referenziellen Integrität und das Prädikat MATCH

Die *Regeln der referenziellen Integrität* verlangen, dass Spaltenwerte einer Tabelle mit Spaltenwerten in einer anderen Tabelle übereinstimmen. Die Spalten in der ersten Tabelle werden als *Fremdschlüssel* und die Spalten in der zweiten Tabelle als *Primärschlüssel* oder *eindeutige Schlüssel* bezeichnet. Sie können beispielsweise die Spalte MitarbAbtNr in einer Tabelle Mitarbeiter als Fremdschlüssel deklarieren, der auf die Spalte AbtNr einer Tabelle Abteilung verweist. Diese Verbindung sorgt dafür, dass in der Tabelle Abteilung eine Zeile mit der Abteilungsnummer 123 eingetragen wird, wenn Sie in der Tabelle Mitarbeiter einen Mitarbeiter einfügen, der in Abteilung 123 arbeitet.

Wenn Primär- und Fremdschlüssel jeweils aus einer einzigen Spalte bestehen, müssen Sie sich um die Situation keine Gedanken machen. Es besteht aber auch die Möglichkeit, dass die beiden Schlüssel aus mehreren Spalten zusammengesetzt werden. So könnte der Wert der Spalte AbtNr nur innerhalb von einem Standort eindeutig sein. Damit müssen Sie sowohl Standort als auch AbtNr festlegen, um eine Zeile Abteilung eindeutig identifizieren zu können. Wenn sowohl die Niederlassung in Boston als auch die in Tampa über eine Abteilung 123 verfügen, bleibt Ihnen nichts anderes übrig, als die Abteilungen durch ('Boston', '123') und ('Tampa', '123') zu identifizieren. In diesem Fall benötigt auch die Tabelle Mitarbeiter zwei Spalten, um eine Abteilung eindeutig zu erkennen. Diese Spalten können zum Beispiel MitarbStandort und MitarbAbtNr heißen. Wenn ein Mitarbeiter in Abteilung 123 in Boston arbeitet, enthalten die Spalten MitarbStandort und MitarbAbtNr die Werte 'Boston' und '123' und die Fremdschlüsseldeklaration in Mitarbeiter lautet wie folgt:

```
FOREIGN KEY (MitarbStandort, MitarbAbtNr)
   REFERENCES Abteilung (Standort, AbtNr)
```

Wenn die Daten Nullwerte enthalten, wird es sehr viel schwieriger, daraus gültige Schlüsse zu ziehen. Manchmal wollen Sie die Daten mit Nullwerten auf die eine Weise, manchmal auf die andere Weise behandeln. Die Schlüsselwörter UNIQUE, SIMPLE, PARTIAL und FULL bieten verschiedene Methoden an, um die Daten zu verarbeiten, die Nullwerte enthalten. Wenn Ihre Daten keine Nullwerte enthalten, können Sie sich viel Kopfzerbrechen ersparen, indem Sie einfach zum nächsten Abschnitt *Logische Verknüpfungen* weiterblättern. Falls Ihre Daten aber Nullwerte enthalten, sollten Sie die folgenden Absätze sorgfältig durchlesen. Jeder von ihnen beschreibt eine andere Situation, in der Nullwerte vorkommen, und teilt Ihnen mit, wie das Prädikat MATCH damit umgeht.

Die folgenden Szenarien erläutern die Regeln, die gelten, wenn Sie es mit Nullwerten und MATCH zu tun haben:

- ✔ **Beide Werte sind identisch:** Wenn beide Werte (von MitarbStandort und MitarbAbtNr) einen Nullwert oder keinen Nullwert enthalten, gelten dieselben Regeln für die referenzielle Integrität wie für Schlüssel, die aus einer einzigen Spalte bestehen und Nullwerte enthalten.
- ✔ **Ein Wert ist ein Nullwert, der andere nicht:** Wenn beispielsweise MitarbStandort ein Nullwert und MitarbAbtNr kein Nullwert ist, benötigen Sie neue Regeln. Wenn Sie Regeln einrichten, die für das Einfügen und Aktualisieren von Daten in der Tabelle Mitarbeiter mit den Werten (NULL, '123') oder ('Boston', NULL) für die Spalten MitarbStandort und MitarbAbtNr gelten sollen, stehen Ihnen sechs generelle Alternativen zur Verfügung: SIMPLE, FULL und PARTIAL, jeweils mit oder ohne das Schlüsselwort UNIQUE.
- ✔ **Das Schlüsselwort UNIQUE ist vorhanden:** In der Unterabfrage muss eine passende Zeile eindeutig sein, damit dem Prädikat der Wert *wahr* zugewiesen wird.
- ✔ **Beide Komponenten des Zeilenwertausdrucks *Z* enthalten einen Nullwert:** Das Prädikat MATCH gibt den Wert *wahr* zurück, und zwar unabhängig vom inhaltlichen Ergebnis der Unterabfrage.
- ✔ **Keine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, SIMPLE wurde angegeben, UNIQUE wurde nicht angegeben, und wenigstens eine Zeile in der Tabelle der Unterabfrage stimmt mit *Z* überein:** Das Prädikat MATCH gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.
- ✔ **Keine der Komponenten des Zeilenwertausdrucks *Z* enthält einen Nullwert, SIMPLE wurde angegeben, UNIQUE wurde angegeben, und wenigstens eine Zeile in der Tabelle der Unterabfrage ist eindeutig und stimmt auch mit *Z* überein:** Das Prädikat MATCH gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.
- ✔ **Eine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, und SIMPLE wurde angegeben:** Das Prädikat MATCH gibt den Wert *wahr* zurück.
- ✔ **Eine Komponente des Zeilenwertausdrucks *Z* enthält keinen Nullwert, PARTIAL wurde angegeben, UNIQUE wurde nicht angegeben, und die Nicht-Null-Teile von wenigstens einer Zeile der Ergebnistabelle der Unterabfrage stimmen mit *Z* überein:** MATCH gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.
- ✔ **Eine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, PARTIAL wurde angegeben, UNIQUE wurde angegeben, und die Nicht-Null-Teile von *Z* stimmen mit den Nicht-Null-Teilen von wenigstens einer eindeutigen Zeile in der Ergebnistabelle der Unterabfrage überein:** MATCH gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.
- ✔ **Keine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, FULL wurde angegeben, UNIQUE wurde nicht angegeben, und wenigstens eine Zeile in der Ergebnistabelle der Unterabfrage stimmt mit *Z* überein:** Das Prädikat MATCH gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.

- ✔ **Keine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, `FULL` wurde angegeben, `UNIQUE` wurde angegeben, und wenigstens eine Zeile in der Ergebnistabelle der Unterabfrage ist eindeutig und stimmt mit *Z* überein:** Das `MATCH`-Prädikat gibt den Wert *wahr* zurück, andernfalls gibt es den Wert *falsch* zurück.
- ✔ **Irgendeine Komponente des Zeilenwertausdrucks *Z* enthält einen Nullwert, und es wurde `FULL` angegeben:** Das `MATCH`-Prädikat gibt den Wert *falsch* zurück.

Komitee-Regeln

Der SQL-89-Standard schrieb die `UNIQUE`-Regel als Standard fest, ehe Alternativen vorgeschlagen und diskutiert wurden. Während der Entwicklung des SQL-92-Standards kamen die Alternativen auf den Tisch. Einige Teilnehmer bevorzugten die `PARTIAL`-Regeln und sprachen sich nachdrücklich dafür aus, dass diese die einzigen Regeln sein sollten. Diese Personen hielten die SQL-89-(`UNIQUE`-)Regeln für so negativ, dass sie sie als Fehler einstuften, der durch die `PARTIAL`-Regeln behoben werden sollte. Andere Teilnehmer zogen die `UNIQUE`-Regeln vor und hielten die `PARTIAL`-Regeln für unklar, fehleranfällig und ineffizient. Wieder andere forderten zusätzlich die `FULL`-Regeln. Das Problem wurde schließlich dadurch gelöst, dass alle drei Varianten in den Standard aufgenommen worden sind, wodurch jeder Benutzer seinen Ansatz wählen kann. SQL:1999 fügte die `SIMPLE`-Regeln hinzu. Dieses Ausufern von Regeln macht die Arbeit mit Nullwerten alles andere als einfach. Wenn weder `SIMPLE`, `PARTIAL` noch `FULL` angegeben werden, greifen die `SIMPLE`-Regeln.

Logische Verknüpfungen

Wie einige der vorangegangenen Beispiele zeigen, reicht es häufig nicht aus, in einer Abfrage nur eine einzige Bedingung einzusetzen, um die Zeilen einer Tabelle zu erhalten, die Sie benötigen. In einigen Fällen müssen die Zeilen zwei oder mehr Bedingungen gleichzeitig erfüllen. In anderen Fällen müssen die Datensätze eine beliebige von zwei oder mehr Bedingungen erfüllen, damit sie einer qualifizierten Antwort entsprechen. Es kann aber auch vorkommen, dass Sie Zeilen erhalten wollen, die eine bestimmte Bedingung nicht erfüllen. Um diesen Anforderungen gerecht zu werden, gibt es in SQL die logischen Verknüpfungen `AND`, `OR` und `NOT`.

AND

Wenn mehrere Bedingungen gleichzeitig erfüllt sein müssen, um einen Datensatz abzurufen, sollten Sie die logische Verknüpfung `AND` benutzen. Betrachten Sie das folgende Beispiel:

```
SELECT RechnungNr, Verkaufsdatum, Verkäufer, Umsatz
   FROM Verkauf
   WHERE Verkaufsdatum >= '2010-04-28'
      AND Verkaufsdatum <= '2010-05-03' ;
```

Die Klausel `WHERE` muss die folgenden beiden Bedingungen erfüllen:

- ✔ `Verkaufsdatum` muss größer oder gleich 28. April 2010 sein.
- ✔ `Verkaufsdatum` muss kleiner oder gleich 3. Mai 2010 sein.

Nur die Zeilen mit Verkäufen in diesem Zeitraum erfüllen beide Bedingungen. Die Abfrage gibt also nur diese Zeilen zurück.

Beachten Sie, dass die `AND`-Verknüpfung streng logisch wirkt. Diese Einschränkung kann manchmal recht verwirrend sein, weil *und* im normalen Sprachgebrauch nicht so eng gesehen wird. Angenommen, Ihr Chef sagt zu Ihnen: »Ich möchte die Verkäufe von Meier und Schmitz sehen.« Er sagte: »Meier **und** Schmitz«, weshalb Sie folgende SQL-Abfrage schreiben könnten:

```
SELECT *
   FROM Verkauf
   WHERE Verkäufer = 'Meier'
      AND Verkäufer = 'Schmitz';
```

Sie legen das Ergebnis dieser Abfrage Ihrem Chef besser nicht vor. Was ihm vorschwebte, wird durch folgende Abfrage besser ausgedrückt:

```
SELECT *
   FROM Verkauf
   WHERE Verkäufer IN ('Meier', 'Schmitz') ;
```

Die erste Abfrage würde nichts zurückgeben, weil die Tabelle `Verkauf` keine Verkäufe enthält, die gleichzeitig von Meier *und* Schmitz abgewickelt worden sind. Die zweite Abfrage gibt die Informationen über die Verkäufe zurück, die von Meier *oder* von Schmitz abgewickelt wurden – also das Ergebnis, das Ihr Chef wahrscheinlich haben möchte.

OR

Wenn es ausreicht, dass eine von zwei oder mehr Bedingungen erfüllt ist, um eine Zeile abzurufen, benutzen Sie die logische Verknüpfung `OR`:

```
SELECT RechnungNr, Verkaufsdatum, Verkäufer, Umsatz
   FROM Verkauf
      WHERE Verkäufer = 'Schmitz'
         OR Umsatz > 200 ;
```

Diese Abfrage gibt alle Verkäufe des Verkäufers Schmitz zurück, unabhängig davon, wie groß diese waren, sowie alle Verkäufe über 200 Euro, unabhängig davon, wer sie abgewickelt hat.

NOT

Die Verknüpfung `NOT` kehrt den Wert einer Bedingung um. Wenn eine Bedingung den Wert *wahr* hat, gibt sie bei einem vorangestellten `NOT` den Wert *falsch* zurück und umgekehrt. Betrachten Sie das folgende Beispiel:

```
SELECT RechnungNr, Verkaufsdatum, Verkäufer, Umsatz
   FROM Verkauf
      WHERE NOT (Verkäufer = 'Schmitz') ;
```

Diese Abfrage gibt alle Datensätze für alle Verkäufe zurück, die nicht vom Verkäufer Schmitz abgewickelt worden sind.

Wenn es unklar ist, welche Parameter jeweils zu den logischen Verknüpfungen `AND`, `OR` oder `NOT` gehören, sollten Sie sie in Klammern fassen, damit SQL die jeweilige Verknüpfung auf das gewünschte Prädikat anwendet. In dem vorhergehenden Beispiel bezieht sich die `NOT`-Verknüpfung auf das gesamte Prädikat (`VERKÄUFER = 'Schmitz'`).

Die Klausel GROUP BY

Manchmal möchten Sie nicht einzelne Datensätze abrufen, sondern etwas über eine ganze Gruppe von Datensätzen in Erfahrung bringen. In diesem Fall benötigen Sie die Klausel `GROUP BY`.

Angenommen, Sie möchten sich als Verkaufsleiter die Umsätze Ihrer Verkäufer anzeigen lassen. Sie könnten dazu folgende einfache `SELECT`-Anweisung verwenden:

```
SELECT RechnungNr, Verkaufsdatum, Verkäufer, Umsatz
   FROM Rechnungen
```

Sie würden ein ähnliches Ergebnis wie in Abbildung 10.1 erzielen.

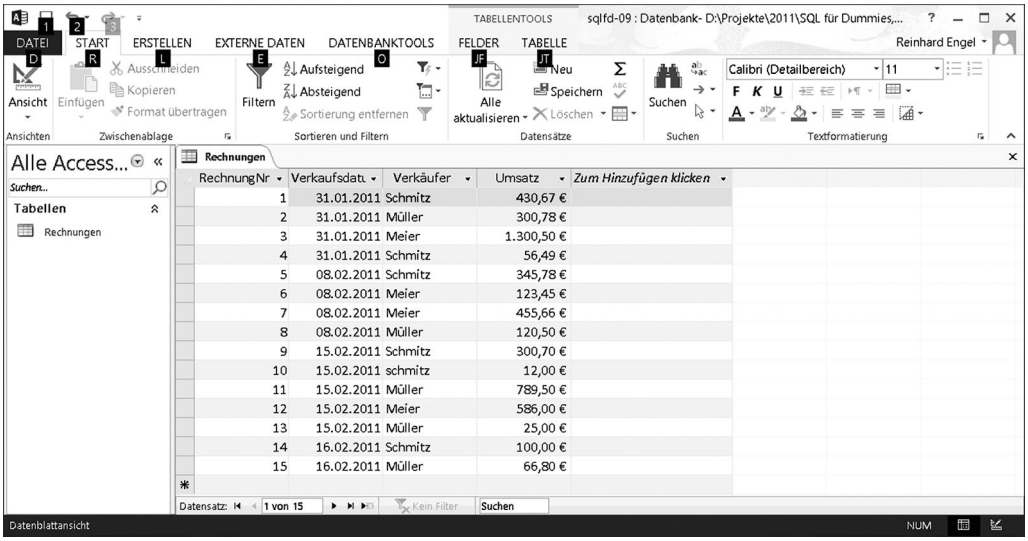


Abbildung 10.1: Das Ergebnis des Abrufs der Verkäufe vom 31.01.2011 bis zum 16.02.2011

Dieses Ergebnis verschafft Ihnen einen Überblick darüber, wie Ihre Verkäufer arbeiten, da die Tabelle nur wenige Einträge aufführt. Normalerweise finden aber in einem Unternehmen weitaus mehr Verkäufe statt, wodurch es schon etwas schwieriger wird, Aussagen darüber zu treffen, ob die Verkaufsziele erreicht worden sind oder nicht. Um das gesuchte Ergebnis zu erhalten, können Sie die Klausel `GROUP BY` mit einer der Aggregatfunktionen (die auch als Mengenfunktionen bezeichnet werden) kombinieren. Sie erhalten dann einen quantitativen Überblick über die Verkaufsleistung der Mitarbeiter. Sie könnten beispielsweise ermitteln, welcher Verkäufer durchschnittlich am meisten verkauft hat, indem Sie die Funktion `AVG` wie folgt einsetzen:

```
SELECT Verkäufer, AVG(Umsatz)
    FROM Rechnungen
    GROUP BY Verkäufer;
```

Das Ergebnis könnte aussehen wie das aus Abbildung 10.2.

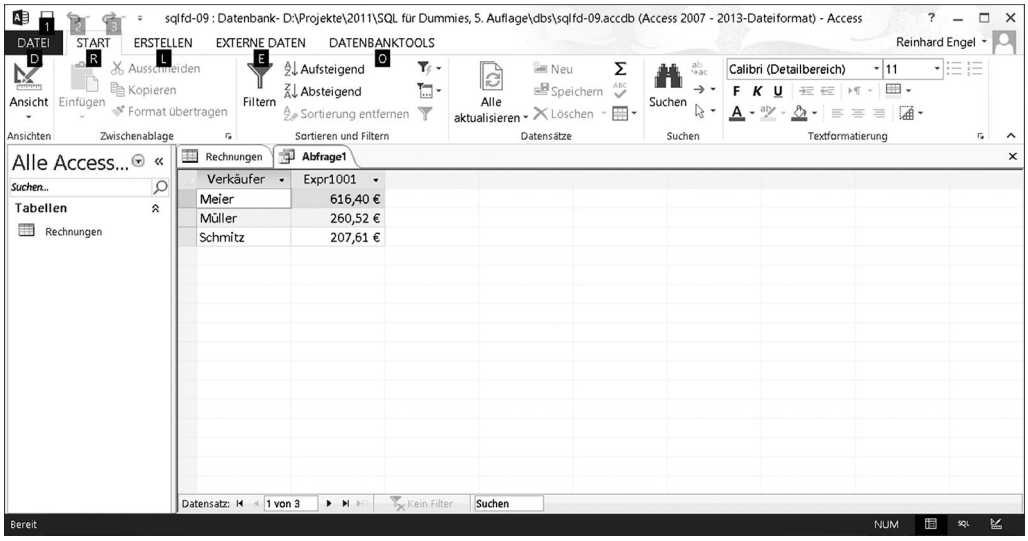


Abbildung 10.2: Die durchschnittliche Verkaufsleistung der Verkäufer

Abbildung 10.2 zeigt, dass Meiers durchschnittliche Verkaufsleistung höher ist als die der anderen beiden Verkäufer. Sie erhalten den Gesamtumsatz mit einer ähnlichen Abfrage:

```
SELECT Verkäufer, SUM(Umsatz)
    FROM Rechnungen
    GROUP BY Verkäufer;
```

Das Ergebnis ist in Abbildung 10.3 zu sehen.

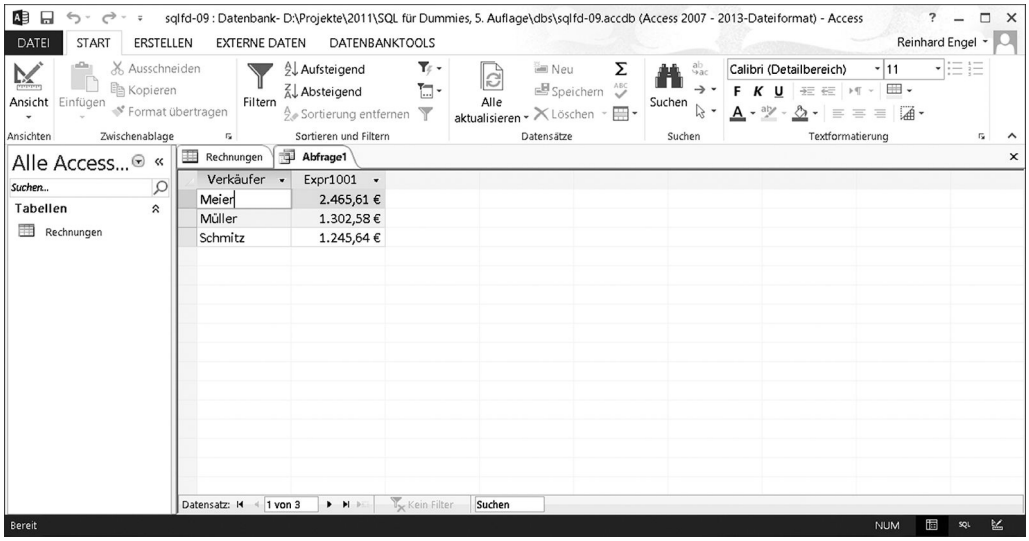


Abbildung 10.3: Der Gesamtumsatz aller Verkäufer

Meier hat auch den höchsten Gesamtumsatz erzielt, was zum höchsten durchschnittlichen Umsatz passt.

HAVING

Sie können die gruppierten Daten mit der Klausel `HAVING` weiter analysieren. Die Klausel `HAVING` ist ein Filter, der ähnlich wie die Klausel `WHERE` arbeitet, aber auf Datensatzgruppen und nicht auf einzelne Datensätze angewendet wird. Um die Funktion der Klausel `HAVING` zu illustrieren, gehen wir davon aus, dass der Verkaufsleiter den Verkäufer Meier als Klasse für sich betrachtet. Meiers Leistung verzerrt die Daten der anderen Verkäufer. Mit der folgenden Anweisung können Sie die Verkäufe von Meier mit einer `HAVING`-Klausel aus den gruppierten Daten ausschließen:

```
SELECT Verkäufer, SUM(Umsatz)
    FROM Rechnungen
    GROUP BY Verkäufer
    HAVING Verkäufer <> 'Meier';
```

Dies führt zu dem in Abbildung 10.4 dargestellten Ergebnis. Dort werden nur die Datensätze angezeigt, in denen jeweils der Verkäufer nicht Meier ist.

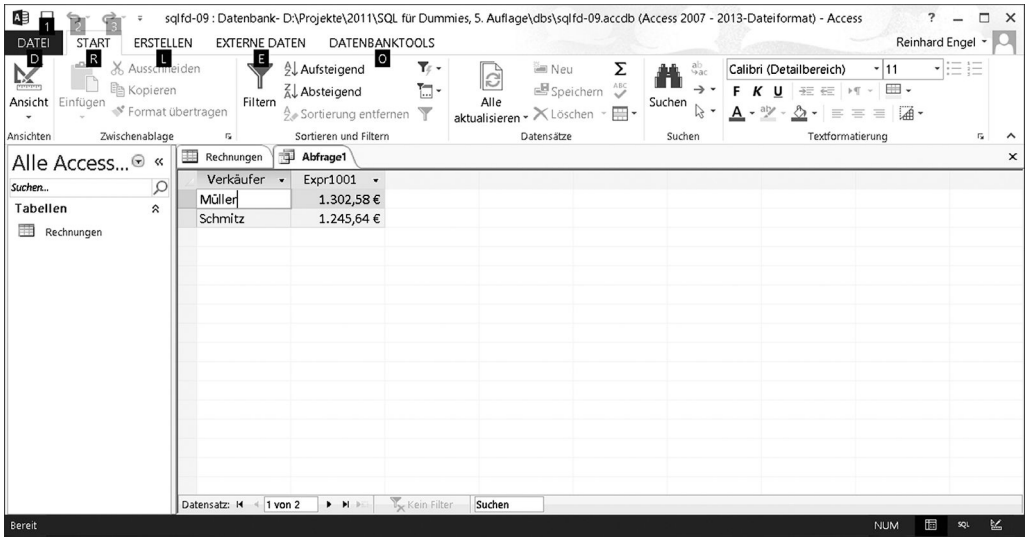


Abbildung 10.4: Der Gesamtumsatz aller Verkäufer außer Meier

ORDER BY

Mit der Klausel `ORDER BY` können Sie die Ausgabetabelle einer Abfrage absteigend oder aufsteigend sortieren. Während die Klausel `GROUP BY` Datensätze zu Gruppen zusammenfasst und diese alphabetisch sortiert, sortiert `ORDER BY` einzelne Datensätze. Die Klausel `ORDER BY` muss in einer Abfrage an letzter Stelle stehen. Wenn die Abfrage auch eine `GROUP BY`-Klausel enthält, werden erst die Gruppen gebildet. Die Klausel `ORDER BY` sortiert dann die Datensätze innerhalb jeder Gruppe. Wenn es keine `GROUP BY`-Klausel gibt, betrachtet die Anweisung die gesamte Tabelle als Gruppe und sortiert alle Zeilen, so wie es in der Klausel `ORDER BY` vorgegeben ist.

Lassen Sie uns die Daten in der Tabelle `Verkauf` ansehen, um diesen Punkt etwas zu erhellen. Diese Tabelle enthält die Spalten `RechnungNr`, `Verkaufsdatum`, `Verkäufer` und `Umsatz`. Mit dem folgenden Beispiel werden alle Verkäufe in einer beliebigen Reihenfolge angezeigt:

```
SELECT * FROM Verkauf ;
```

In der einen Implementierung entspricht diese Reihenfolge der Reihenfolge, in der Sie die Daten eingegeben haben, während die Reihenfolge in einer anderen Implementierung die der letzten Änderung der Daten ist. Die Reihenfolge kann sich unerwartet ändern, wenn die Datenbank physisch reorganisiert wird. Deshalb sollten Sie normalerweise die Reihenfolge angeben, in der die Datensätze zurückgegeben werden sollen. Wenn Sie die Datensätze nach `Verkaufsdatum` sortiert sehen wollen, geben Sie Folgendes ein:

```
SELECT * FROM Verkauf ORDER BY Verkaufsdatum ;
```

Bei Datensätzen mit demselben Verkaufsdatum hängt die Standardsortierreihenfolge von der einzelnen Implementierung ab. Sie können jedoch festlegen, wie die Zeilen sortiert werden sollen, die ein gemeinsames Verkaufsdatum haben. Sie können beispielsweise für diese Fälle als zusätzliches Sortierkriterium die Rechnungsnummer angeben:

```
SELECT * FROM Verkauf ORDER BY Verkaufsdatum, RechnungNr ;
```

Dieses Beispiel sortiert die Verkäufe erst nach `Verkaufsdatum` und innerhalb eines gleichen Verkaufsdatums nach `RechnungNr`. Verwechseln Sie das Beispiel jedoch nicht mit der folgenden Abfrage:

```
SELECT * FROM Verkauf ORDER BY RechnungNr, Verkaufsdatum ;
```

Diese Abfrage sortiert die Verkäufe erst nach `RechnungNr` und innerhalb einer Rechnungsnummer nach `Verkaufsdatum`. Dies ist wahrscheinlich nicht das Ergebnis, das Sie erzielen wollen, weil es unwahrscheinlich ist, dass es für dieselbe Rechnungsnummer verschiedene Verkaufsdaten gibt.

Die folgende Abfrage ist ein weiteres Beispiel dafür, wie SQL Daten zurückgibt:

```
SELECT * FROM Verkauf ORDER BY Verkäufer, Verkaufsdatum ;
```

Sie können die Datensätze auch erst nach `Verkaufsdatum` und dann nach `Verkäufer` sortieren lassen:

```
SELECT * FROM Verkauf ORDER BY Verkaufsdatum, Verkäufer ;
```

Alle diese Beispiele sortieren die Daten aufsteigend (`ASC`), was die Standardsortierfolge ist. Das letzte Beispiel zeigt ältere Verkäufe zuerst und ordnet die Daten innerhalb eines Datums aufsteigend nach Verkäufernamen, also *Becker* vor *Schiffer*. Wenn Sie eine absteigende (`DESC`) Sortierreihenfolge bevorzugen, gehen Sie so vor:

```
SELECT * FROM Verkauf
ORDER BY Verkaufsdatum DESC, Verkäufer ASC ;
```

Dieses Beispiel sortiert die Verkäufe absteigend nach Verkaufsdatum und innerhalb eines Verkaufsdatums aufsteigend in der normalen alphabetischen Reihenfolge nach den Namen der Verkäufer(innen).

Begrenzende FETCH-Funktion

Bei einer Änderung des ISO/IEC-SQL-Standards werden normalerweise die Funktionen der Sprache erweitert. Das ist gut so. Doch manchmal kann man nicht alle möglichen Konsequenzen einer Änderung vorhersehen. Dies passierte, als in SQL:2008 die begrenzende `FETCH`-Funktion hinzugefügt wurde.

Während eine SELECT-Anweisung eine unbestimmte Anzahl von Zeilen zurückgibt, sind Sie vielleicht nur an den ersten drei oder zehn Ergebnissen interessiert. Und genau diese Zeilen soll die begrenzende FETCH-Funktion liefern, die in SQL:2008 eingeführt wurde. Ein Beispiel:

```
SELECT Verkäufer, AVG(GesamtUmsatz)
   FROM Verkauf
   GROUP BY Verkäufer
   ORDER BY AVG(GesamtUmsatz) DESC
   FETCH FIRST 3 ROWS ONLY;
```

Dies sieht gut aus. Sie wollen nur die besten drei Verkäufer mit den besten Gesamtumsätzen ermitteln. Doch dabei gibt es ein kleines Problem. Was passiert, wenn nach den beiden besten Verkäufern drei Verkäufer denselben Gesamtumsatz erzielt haben? Nur einer dieser drei wird zurückgegeben, welcher, ist unbestimmt.

Unbestimmtheit ist für Datenbankverwalter mit Selbstachtung nicht akzeptabel. Deshalb wurde die Funktion in SQL:2011 korrigiert. Eine neue Syntax soll mögliche Gleichstände berücksichtigen:

```
SELECT Verkäufer, AVG(GesamtUmsatz)
   FROM Verkauf
   GROUP BY Verkäufer
   ORDER BY AVG(GesamtUmsatz) DESC
   FETCH FIRST 3 ROWS WITH TIES;
```

Jetzt ist das Ergebnis vollkommen determiniert: Gibt es einen Gleichstand, werden alle entsprechenden Zeilen zurückgegeben. Wenn Sie den Modifizierer WITH TIES (»mit Unentschieden«) weglassen, ist das Ergebnis wie zuvor unbestimmt.

Außerdem wurde die begrenzende FETCH-Funktion in SQL:2011 um einige Fähigkeiten erweitert.

Erstens: Sie kann jetzt nicht nur eine bestimmte Anzahl von Zeilen, sondern auch Prozentsätze handhaben. Ein Beispiel:

```
SELECT Verkäufer, AVG(GesamtUmsatz)
   FROM Verkauf
   GROUP BY Verkäufer
   ORDER BY AVG(GesamtUmsatz) DESC
   FETCH FIRST 10 PERCENT ROWS ONLY;
```

Da Gleichstände auch bei Prozentsätzen ein Problem sein könnten, können Sie auch hier den Modifizierer WITH TIES verwenden, um je nach Bedarf Zeilen mit den gleichen Werten einzuschließen oder nicht.

Zweitens: Angenommen, Sie wollten nicht die besten drei oder zehn Prozent, sondern die zweitbesten drei oder zehn Prozent auswählen. Oder Sie wollten direkt zu einem bestimmten Punkt springen, der tief in der Ergebnismenge verborgen ist. SQL:2011 bietet auch dafür eine Lösung. Der Code würde etwa wie folgt aussehen:

```
SELECT Verkäufer, AVG(GesamtUmsatz)
   FROM Verkauf
   GROUP BY Verkäufer
   ORDER BY AVG(GesamtUmsatz) DESC
        OFFSET 3 ROWS
   FETCH NEXT 3 ROWS ONLY;
```

Das Schlüsselwort `OFFSET` gibt an, wie viele Zeilen übersprungen werden sollen, bevor das Ergebnis zurückgegeben wird. Das Schlüsselwort `NEXT` gibt an, dass die Zeilen zurückgegeben werden sollen, die unmittelbar auf die Offset-Zeilen folgen. Jetzt werden die Verkäufer mit dem viert-, fünft- und sechstbesten Gesamtumsatz zurückgegeben. Doch ohne den Modifizierer `WITH TIES` gibt es auch hier das Ergebnis unbestimmt. Wenn der dritte, der vierte und der fünfte Verkäufer denselben Gesamtumsatz erzielt haben, ist unbestimmt, welche zwei in diese zweite (`NEXT`-)Gruppe und welcher in die erste (`OFFSET`-)Gruppe eingeschlossen wird.

Vielleicht ist es besser, die begrenzende `FETCH`-Funktion nicht zu verwenden, weil die Gefahr zu groß ist, irreführende Ergebnisse zu erhalten.

Ergebnismengen mit Fensterfunktionen erstellen

Fenster und Fensterfunktionen wurden in SQL:1999 eingeführt. Mit einem Fenster kann ein Anwender eine Datenmenge bei Bedarf partitionieren, die Zeilen in jeder Partition sortieren und eine Sammlung von Zeilen (einen *Fensterrahmen*) spezifizieren, der einer bestimmten Zeile zugeordnet ist.

Der Fensterrahmen einer Zeile Z ist eine Teilmenge der Partition, die Z enthält. So kann etwa der Fensterrahmen aus allen Zeilen vom Anfang der Partition bis zu Z einschließlich bestehen, je nachdem, wie die Zeilen in der Partition sortiert sind.

Eine Fensterfunktion berechnet, basierend auf den Zeilen in dem Fensterrahmen von Z, einen Wert für eine Zeile Z.

Angenommen, Sie hätten eine `Verkauf`-Tabelle mit den Spalten `KundenID`, `RechnungNr` und `GesamtUmsatz`. Ihr Verkaufsleiter möchte den Gesamtumsatz pro Kunde über einen bestimmten Bereich von Rechnungsnummern haben. Folgender SQL-Code liefert diese Werte:

```
SELECT KundenID, RechnungNr,
   SUM (GesamtUmsatz) OVER
( PARTITION BY KundenID
     ORDER BY RechnungNr
     ROWS BETWEEN
     UNBOUNDED PRECEDING
     AND CURRENT ROW )
   FROM Verkauf;
```

Die OVER-Klausel bestimmt, wie die Zeilen der Abfrage vor der Verarbeitung (hier mit SUM-Funktion) partitioniert werden sollen. Jedem Kunden wird eine Partition zugewiesen. Jede Partition enthält eine Liste mit Rechnungsnummern. Der Gesamtumsatz pro Kunde wird aus den GesamtUmsatz-Werten der einzelnen Zeilen in dem spezifizierten Bereich berechnet.

In SQL:2011 wurden mithilfe neuer Schlüsselwörter mehrere größere Erweiterungen der ursprünglichen Fensterfunktionalität eingeführt.

Ein Fenster mit NTILE in Buckets partionieren

Die NTILE-Fensterfunktion unterteilt eine geordnete Fensterpartition in eine positive Anzahl *n* von Buckets (Gruppen, Kammern), die von *1* bis *n* nummeriert sind. Wenn die Anzahl der Zeilen in einer Partition *m* nicht ohne Rest durch *n* teilbar ist, füllt die NTILE-Funktion die Buckets zunächst gleichmäßig; dann wird der Rest von *m/n*, genannt *r*, auf die ersten *r* Buckets verteilt, sodass diese größer als die anderen Buckets werden.

Angenommen, Sie wollten Ihre Mitarbeiter in fünf Gehaltsgruppen, von der höchsten bis zur niedrigsten, unterteilen. Der Code lautet:

```
SELECT Vorname, Nachname, NTILE (5)
   OVER (ORDER BY Gehalt DESC)
   AS BUCKET
   FROM Mitarbeiter;
```

Wenn Sie beispielsweise 11 Mitarbeiter haben, wird jeder Bucket bis auf den ersten mit zwei Mitarbeitern gefüllt. Der erste Bucket wird mit den drei höchstbezahlten Mitarbeitern gefüllt. Der fünfte Bucket enthält die zwei geringstbezahlten Mitarbeiter.

In einem Fenster navigieren

In SQL:2011 wurden fünf Fensterfunktionen eingeführt, die einen Ausdruck in einer Zeile Z2 auswerten, die sich irgendwo in dem Fensterrahmen der aktuellen Zeile Z1 befindet: LAG, LEAD, NTH_VALUE, FIRST_VALUE und LAST_VALUE.

Mit diesen Funktionen können Sie Daten aus bestimmten Zeilen extrahieren, die sich im Fensterrahmen der aktuellen Zeile befinden.

Mit der LAG-Funktion zurückblicken

Mit der LAG-Funktion können Sie in einem Fenster Daten aus der aktuellen Zeile sowie Daten aus einer anderen Zeile abrufen, die der aktuellen Zeile vorausgeht.

Angenommen, Sie hätten eine Tabelle, in der der Gesamtumsatz pro Tag des laufenden Jahres gespeichert wird, und wollten wissen, wie hoch der heutige Umsatz im Vergleich zum gestrigen ist. Mit der LAG-Funktion können Sie die Tage wie folgt vergleichen:

```
SELECT GesamtUmsatz AS TagesUmsatz,
   LAG (GesamtUmsatz) OVER
   (ORDER BY VerkaufsDatum) AS VortagsUmsatz
   FROM TagesSummen;
```

Diese Abfrage gibt für jede Zeile in TagesSummen eine Zeile zurück, die den Gesamtumsatz der Zeile sowie den Gesamtumsatz des Vortags enthält. Der Standardoffset beträgt 1, weswegen der Wert des Vortags (der vorhergehenden Zeile) zurückgegeben wird.

Mit der folgenden Abfrage können Sie den aktuellen Tagesumsatz mit dem Tagesumsatz vor einer Woche vergleichen:

```
SELECT GesamtUmsatz AS TagesUmsatz,
   LAG (GesamtUmsatz, 7) OVER
   (ORDER BY VerkaufsDatum) AS VortagsUmsatz
   FROM TagesSummen;
```

Die ersten sieben Zeilen in einem Fensterrahmen haben keinen Vorgänger, der eine Woche älter ist. Standardmäßig wird in diesem Fall ein NULL-Wert als VortagsUmsatz zurückgegeben. Wenn Sie in diesem Fall nicht NULL, sondern einen anderen Wert, etwa 0, zurückgeben wollen, können Sie diesen Wert wie folgt spezifizieren:

```
SELECT GesamtUmsatz AS TagesUmsatz,
   LAG (GesamtUmsatz, 7, 0) OVER
   (ORDER BY VerkaufsDatum) AS VortagsUmsatz
   FROM TagesSummen;
```

Standardmäßig werden alle Zeilen gezählt, die das in LAG spezifizierte Feld (hier Gesamt Umsatz) enthalten, auch wenn dieses einen NULL-Wert enthält. Wenn Sie solche Zeilen überspringen und nur Zeilen zählen wollen, die in diesem Feld einen echten Wert enthalten, müssen Sie die Schlüsselwörter IGNORE NULLS hinzufügen:

```
SELECT GesamtUmsatz AS TagesUmsatz,
   LAG (GesamtUmsatz, 7, 0) IGNORE NULLS
   OVER (ORDER BY VerkaufsDatum) AS VortagsUmsatz
   FROM TagesSummen;
```

Mit der LEAD-Funktion zurückblicken

Die LEAD-Funktion arbeitet genau wie die LAG-Funktion, außer dass sie nicht auf eine vorangehende, sondern auf eine nachfolgende Zeile zugreift. Sie schaut also in dem Fensterrahmen der aktuellen Zeile voraus. Ein Beispiel:

```
SELECT GesamtUmsatz AS TagesUmsatz,
   LEAD (GesamtUmsatz, 7, 0) IGNORE NULLS
   OVER (ORDER BY VerkaufsDatum) AS FolgetagsUmsatz
   FROM TagesSummen;
```

Mit der NTH_VALUE-Funktion auf eine bestimmte Zeile zugreifen

Die NTH_VALUE-Funktion arbeitet ähnlich wie die LAG- und LEAD-Funktion, außer dass sie nicht auf eine vorangehende oder nachfolgende Zeile der aktuellen Zeile, sondern auf die Zeile zugreift, die sich an einer spezifizierten Stelle (einem Offset) relativ zur ersten oder letzten Zeile des Fensterrahmens befindet.

Ein Beispiel:

```
SELECT GesamtUmsatz AS AuswahlUmsatz,
   NTH_VALUE (GesamtUmsatz, 2)
   FROM FIRST
   IGNORE NULLS
   OVER (ORDER BY VerkaufsDatum)
ROWS BETWEEN 10 PRECEDING AND 10 FOLLOWING )
        AS AlterUmsatz
FROM TagesSummen;
```

In diesem Beispiel wird `AlterUmsatz` wie folgt ausgewertet:

- ✔ Der Fensterrahmen, der mit der aktuellen Zeile verbunden ist, wird erstellt. Er enthält die ersten zehn vorhergehenden und die zehn nachfolgenden Zeilen.
- ✔ Der `GesamtUmsatz` von jeder Zeile des Fensterrahmens wird ausgewertet.
- ✔ `IGNORE NULLS` wird spezifiziert, damit Zeilen, die für `GesamtUmsatz` einen `NULL`-Wert enthalten, übersprungen werden.
- ✔ Ausgehend vom ersten Wert, der nach dem Ausschluss der Zeilen verbleibt, die für `GesamtUmsatz` einen `NULL`-Wert enthalten, wird zwei Zeilen vorwärts gegangen (vorwärts, weil `FROM FIRST` spezifiziert worden ist).

Der Wert von `AlterUmsatz` ist der Wert von `GesamtUmsatz` aus der spezifizierten Zeile.

Wenn die Zeilen, die für `GesamtUmsatz` einen `Null`-Wert enthalten, mit ausgewertet werden sollen, spezifizieren Sie nicht `IGNORE NULLS`, sondern `RESPECT NULLS`. Die `NTH_VALUE`-Funktion arbeitet ähnlich, wenn Sie nicht `FROM FIRST`, sondern `FROM LAST` spezifizieren; dann werden die Zeilen nicht vorwärts von der ersten Zeile des Fensterrahmens, sondern rückwärts von seiner letzten Zeile gezählt. Die Zahl, mit der die Anzahl der Zeilen spezifiziert wird, ist immer noch positiv, auch wenn Sie nicht vorwärts, sondern rückwärts zählen.

Mit FIRST_VALUE und LAST_VALUE auf bestimmte Werte zugreifen

Die Funktionen `FIRST_VALUE` und `LAST_VALUE` sind Sonderfälle der `NTH_VALUE`-Funktion. Die `FIRST_VALUE`-Funktion entspricht einer `NTH_VALUE`-Funktion, bei der `FROM FIRST` spezifiziert und der Offset auf `0` gesetzt ist. Die `LAST_VALUE`-Funktion entspricht einer `NTH_VALUE`-Funktion, bei der `FROM LAST` spezifiziert und der Offset auf `0` gesetzt ist. Bei beiden Funktionen können Sie entweder `IGNORE NULLS` oder `RESPECT NULLS` spezifizieren.

Fensterfunktionen verschachteln

Manche Ergebnisse lassen sich am einfachsten ermitteln, indem Funktionen verschachtelt werden. In SQL:2011 wurde die Möglichkeit eingeführt, Fensterfunktionen zu verschachteln.

Ein Beispiel: Ein Investor versucht festzustellen, ob der Zeitpunkt günstig ist, eine bestimmte Aktie zu kaufen. Deshalb will er den aktuellen Kurs mit den Kursen der unmittelbar vorhergehenden 100 Trades vergleichen und herausfinden, wie oft die Aktie dabei unter dem aktuellen Preis verkauft worden ist. Folgende Abfrage gibt die Antwort:

```
SELECT VerkaufsZeit,
   SUM ( CASE WHEN VerkaufsPreis <
   VALUE OF (VerkaufsPreis AT CURRENT ROW)
   THEN 1 ELSE 0 )
   OVER (ORDER BY VerkaufsZeit
   ROWS BETWEEN 100 PRECEDING AND CURRENT ROW )
  FROM AktienVerkaeufe;
```

Das Fenster umfasst die 100 Zeilen vor der aktuellen Zeile. Sie enthalten die 100 Trades unmittelbar vor dem aktuellen Zeitpunkt. Jedes Mal, wenn der Wert von `VerkaufsPreis` in einer Zeile kleiner als der aktuelle Kurs ist, wird die Summe um `1` erhöht. Das Ergebnis gibt die Anzahl der Trades aus den vorhergehenden hundert Trades an, bei denen der Kurs unter dem aktuellen Kurs lag.

Gruppen von Zeilen auswerten

Manchmal enthält der Sortierschlüssel, mit der Sie eine Partition sortieren, Duplikate. Vielleicht wollen Sie alle Zeilen mit demselben Sortierschlüssel als Gruppe auswerten. In solchen Fällen können Sie die `GROUPS`-Option verwenden. Damit können Sie Gruppen von Zeilen zählen, die denselben Sortierschlüssel haben.

Ein Beispiel:

```
SELECT KundenID, VerkaufsDatum,
   SUM (RechnungsSumme) OVER
( PARTITION BY KundenID
     ORDER BY VerkaufsDatum
   GROUPS BETWEEN 2 PRECEDING AND 2 FOLLOWING )
  FROM Kunde;
```

Der Fensterrahmen in diesem Beispiel besteht aus fünf Gruppen von Zeilen: zwei Gruppen vor der Gruppe mit der aktuellen Zeile, die Gruppe mit der aktuellen Zeile und die beiden Gruppen nach der aktuellen Zeile. Die Zeilen in jeder Gruppe enthalten dasselbe `VerkaufsDatum`, und das `VerkaufsDatum` jeder Gruppe unterscheidet sich von den `VerkaufsDatum`-Werten der anderen Gruppen.

Relationale Operatoren

11

In diesem Kapitel

- Tabellen mit ähnlichen Strukturen kombinieren
- Tabellen mit unterschiedlichen Strukturen kombinieren
- Daten aus mehreren Tabellen abfragen

Mittlerweile wissen Sie, dass SQL eine Abfragesprache für relationale Datenbanken ist. In den vorangegangenen Kapiteln habe ich einfache Datenbanken vorgestellt, und die meisten Beispiele haben nur mit einer Tabelle zu tun. Jetzt wird es Zeit, sich mit dem *Relationalen* der relationalen Datenbanken zu beschäftigen. Schließlich werden relationale Datenbanken so bezeichnet, weil sie aus mehreren Tabellen bestehen, die miteinander in Beziehung stehen.

Weil die Daten in einer relationalen Datenbank über mehrere Tabellen verteilt sind, müssen Abfragen normalerweise auf mehr als eine Tabelle zugreifen. SQL verfügt über Operatoren, die Daten aus mehreren Quellen in einer einzigen Ergebnistabelle zusammenführen können. Dazu zählen die Operatoren `UNION`, `INTERSECTION` und `EXCEPT` sowie die Familie der `JOIN`-Operatoren. Jeder Operator kombiniert Daten aus mehreren Tabellen auf seine Art.

UNION

Der Operator `UNION` ist die SQL-Umsetzung des Vereinigungsoperators der relationalen Algebra. Er dient dazu, Daten aus zwei oder mehr Tabellen mit derselben Struktur zu vereinigen. *Dieselbe Struktur* meint:

- ✔ Die Tabellen müssen dieselbe Anzahl von Spalten haben.
- ✔ Die entsprechenden Spalten müssen von demselben Datentyp sein und die gleiche Spaltenbreite haben.

Wenn dies der Fall ist, sind die Tabellen *UNION-kompatibel*. Die Vereinigung zweier Tabellen gibt alle Zeilen zurück, die in einer der beiden Tabellen vorkommen. Dabei werden Dubletten entfernt.

Die Tabellen `ErsteLiga` und `ZweiteLiga` der Fußballspielerdatenbank in Kapitel 10 sind beispielsweise `UNION`-kompatibel. Beide Tabellen verfügen über drei Spalten desselben Typs. Tatsächlich haben die entsprechenden Spalten sogar identische Spaltennamen (was für die `UNION`-Kompatibilität nicht erforderlich ist).

Die beiden Tabellen enthalten die Vornamen, Nachnamen und geschossenen Tore von Stürmern der Ersten und Zweiten Bundesliga. Die Vereinigung (`UNION`) der beiden Tabellen erzeugt eine virtuelle Ergebnistabelle, in der alle Spieler beider Tabellen zusammengefasst sind.

Um dieses Beispiel zu verdeutlichen, habe ich in jede Tabelle einige (fiktive) Stürmer eingefügt.

```
SELECT * FROM ErsteLiga ;

Vorname    Nachname   Tore
---------  ---------  ----
Franz      Becker       11
Helmut     Langer        9
Toni       Doppler      13
Helmut     Berthold     12

SELECT * FROM ZweiteLiga ;

VORNAME    Nachname   Tore
---------  ---------  ----
Paul       Peters       12
Helmut     Schmitz      10
Gerd       Müller        8
Arnold     Schwarzer    14

SELECT * FROM ErsteLiga
UNION
SELECT * FROM ZweiteLiga ;
Vorname    Nachname   Tore
---------  ---------  ----
Arnold     Schwarzer    14
Gerd       Müller        8
Helmut     Berthold     12
Helmut     Schmitz      10
Helmut     Langer        9
Franz      Becker       11
Toni       Doppler      13
Paul       Peters       12
```

Der Operator `UNION DISTINCT` arbeitet genauso wie der Operator `UNION` ohne das Schlüsselwort `DISTINCT`. In beiden Fällen werden doppelt vorhandene Zeilen aus der Ergebnistabelle gelöscht.

Ich habe das Sternchen (*) als Kurzform für die Auswahl aller Spalten einer Tabelle benutzt. Meistens ist die Verwendung dieser Kurzform auch in Ordnung. Wenn Sie aber relationale Operatoren in eingebettetem SQL oder in SQL-Modulen verwenden, kann dies zu Problemen führen. Wenn Sie eine oder mehrere neue Spalten in die eine Tabelle und nicht auch in die andere einfügen oder wenn Sie in beide Tabellen unterschiedliche Spalten einfügen, sind die beiden Tabellen nicht mehr `UNION`-kompatibel. Das hat zur Folge, dass Ihr Programm nicht mehr

funktioniert. Selbst wenn Sie dieselben neuen Spalten in beide Tabellen einfügen, damit diese UNION-kompatibel bleiben, kann es passieren, dass Ihr Programm mit den zusätzlichen Daten nicht umgehen kann. Deshalb sollten Sie die Spalten immer ausdrücklich angeben und sich nicht auf die Kurzform mit dem Sternchen verlassen. Bei Abfragen, die Sie ad hoc an der Konsole eingeben, können Sie das Sternchen ruhig benutzen, um schnell die Tabellenstrukturen anzeigen zu lassen und auf UNION-Kompatibilität hin zu überprüfen, wenn Ihre Abfrage fehlschlagen sollte.

UNION ALL

Wie ich bereits weiter vorn erwähne, entfernt der Operator UNION normalerweise alle doppelten Zeilen aus der Ergebnistabelle – was in den meisten Fällen auch gewünscht wird. Es sind aber Situationen denkbar, in denen Sie diese Duplikate unbedingt haben wollen. Hier verwenden Sie UNION ALL.

Wenn in unserem Bundesligabeispiel ein Spieler in der laufenden Saison von einem Verein der Zweiten Liga zu einem Verein der Ersten Liga wechselt, finden Sie ihn naturgemäß in beiden Tabellen. Wenn dieser Spieler nun in der Ersten Liga zufällig genauso viele Tore schießt wie vorher in der Zweiten Liga, entfernt der normale Operator UNION eine der beiden doppelten Zeilen und zeigt nur halb so viele Tore an, wie der Spieler tatsächlich geschossen hat. Die folgende Abfrage liefert den wahren Sachverhalt:

```
SELECT * FROM ErsteLiga
UNION ALL
SELECT * FROM ZweiteLiga
```

Manchmal können Sie zwei Tabellen mit dem Operator UNION verknüpfen, selbst wenn sie nicht UNION-kompatibel sind. Wenn die Spalten in Ihrer Ergebnistabelle in beiden Ausgangstabellen vorhanden und miteinander kompatibel sind, können Sie den Operator UNION CORRESPONDING verwenden. Der Operator wählt nur die angegebenen Spalten aus.

UNION CORRESPONDING

Fußballstatistiker führen über Torhüter und Feldspieler unterschiedliche Statistiken. Torhüter kassieren Tore, Feldspieler schießen Tore. Auf jeden Fall werden aber Vornamen, Nachnamen und Anzahl der Spiele gespeichert. Diese Informationen, die in den Tabellen Tormann und Feldspieler gesammelt werden, können mit dem Operator UNION in einer Tabelle zusammengefasst werden:

```
SELECT *
   FROM Tormann
UNION CORRESPONDING
   (Vorname, Nachname, Spiele)
SELECT *
   FROM Feldspieler ;
```

Die Ergebnistabelle enthält die Vornamen, Nachnamen und Spiele aller Spieler. Wie beim einfachen `UNION`-Operator werden Dubletten entfernt. Wenn ein Spieler beide Funktionen ausgeübt hat, können einige Informationen verloren gehen. Dies geschieht auch bei der `CORRESPONDING`-Operation, wenn ein Spieler sowohl als Feldspieler als auch als Torwart eingesetzt wird. Um dieses Problem zu vermeiden, sollten Sie `UNION ALL CORRESPONDING` verwenden.

Jede Spalte, deren Name in der Liste hinter dem Schlüsselwort `CORRESPONDING` steht, muss in beiden Tabellen vorkommen. Wenn Sie diese Namensliste nicht angeben, wird automatisch eine Liste aller Namen verwendet, die in beiden Tabellen übereinstimmen. Diese »interne« Liste von Namen kann sich jedoch ändern, wenn neue Spalten in eine oder beide Tabellen eingefügt werden. Deshalb ist es besser, die Spaltennamen explizit anzugeben.

INTERSECT

Der Operator `UNION` liefert eine Ergebnistabelle mit allen Zeilen aller Ausgangstabellen. Wenn Sie nur die Zeilen suchen, die in allen Ausgangstabellen gemeinsam vorkommen, können Sie den Operator `INTERSECT` benutzen, der der Schnittmengenoperator der relationalen Algebra in SQL ist. Um ein Beispiel für den Operator `INTERSECT` zu geben, wollen wir Gerd Müller in der laufenden Saison von der Zweiten in die Erste Liga transferieren.

```
SELECT * FROM ErsteLiga;

Vorname    Nachname   Tore
---------  ---------  ----
Franz      Becker       11
Helmut     Langer        9
Toni       Doppler      13
Helmut     Berthold     12
Gerd       Müller        8

SELECT * FROM ZweiteLiga;

Vorname    Nachname   Tore
---------  ---------  ----
Paul       Peters       12
Helmut     Schmitz      10
Gerd       Müller        8
Arnold     Schwarzer    14
```

Wenn Sie mit INTERSECT die Schnittmenge beider Tabellen bilden, erhalten Sie nur eine einzige Zeile als Ergebnis:

```
SELECT *
   FROM ErsteLiga
INTERSECT
SELECT *
   FROM ZweiteLiga;

Vorname    Nachname   Tore
---------  ---------  ----
Gerd       Müller        8
```

Die Ergebnistabelle sagt uns, dass Gerd Müller der einzige Spieler ist, der in beiden Ligen gespielt und dabei dieselbe Anzahl von Toren geschossen hat. Beachten Sie, dass, wie bei UNION, der Operator INTERSECT DISTINCT zu demselben Ergebnis führt, wie INTERSECT ohne das Schlüsselwort DISTINCT. In beiden Fällen wird nur eine identische Zeile (Gerd Müller) zurückgegeben.

Die Schlüsselwörter ALL und CORRESPONDING funktionieren mit INTERSECT genauso wie mit UNION. Bei ALL werden keine Dubletten aus der Ergebnistabelle entfernt. Bei CORRESPONDING brauchen die Ausgangstabellen nicht UNION-kompatibel zu sein, obwohl die entsprechenden Spalten gleiche Datentypen und Breiten haben müssen.

INTERSECT ALL liefert folgendes Ergebnis:

```
SELECT *
   FROM ErsteLiga
INTERSECT ALL
SELECT *
   FROM ZweiteLiga;

Vorname    Nachname   Tore
---------  ---------  ----
Gerd       Müller        8
Gerd       Müller        8
```

Betrachten wir ein weiteres Beispiel. Eine Stadtverwaltung speichert die Pager, die von Polizeibeamten, Feuerwehrleuten, Straßenreinigern und anderen Mitarbeitern der Verwaltung benutzt werden. Eine Datenbanktabelle mit dem Namen Pager enthält die Daten aller aktiven Pager. Eine weitere, identisch strukturierte Tabelle mit dem Namen NichtInBenutzung speichert die Daten aller Pager, die nicht benutzt werden. Es darf keinen Pager geben, der in beiden Tabellen existiert. Mit einer INTERSECT-Operation können Sie feststellen, ob eine solche unerwünschte Dopplung vorhanden ist:

```
SELECT *
   FROM Pager
INTERSECT CORRESPONDING (PagerID)
SELECT *
   FROM NichtInBenutzung;
```

Wenn die Ergebnistabelle jetzt eine Zeile enthält, haben Sie ein Problem. Sie müssen dann feststellen, welchen Status der Pager mit der `PagerID` in der Ergebnistabelle hat. Er kann nur aktiv oder inaktiv, aber nicht beides gleichzeitig sein. Wenn Sie die Ursache entdeckt haben, löschen Sie den Pager aus der entsprechenden Tabelle, um die Integrität der Datenbank wiederherzustellen.

EXCEPT

Eine Operation mit `UNION` arbeitet mit zwei Ausgangstabellen und gibt alle Zeilen zurück, die in beiden Tabellen enthalten sind. Eine `INTERSECT`-Operation gibt alle Zeilen zurück, die in den Ausgangstabellen übereinstimmen. Im Gegensatz dazu liefert eine Operation mit `EXCEPT` (oder `EXCEPT DISTINCT`) alle Zeilen, die zwar in der ersten, aber *nicht* in der zweiten Tabelle existieren.

Lassen Sie uns zu dem Beispiel mit der Pager-Datenbank (aus dem Abschnitt *INTERSECT* weiter vorne in diesem Kapitel) zurückkehren und annehmen, dass einige Pager, die beim Hersteller zur Reparatur waren, wieder in Gebrauch genommen werden. Die Tabelle `Pager` wurde aktualisiert und enthält jetzt die reparierten, wieder in Gebrauch genommenen Pager. Die reparierten Pager wurden jedoch noch nicht – wie es hätte sein sollen – aus der Tabelle `NichtInBenutzung` gelöscht. Sie können die Kennungen der Pager, die in der Tabelle `NichtInBenutzung` stehen, anzeigen lassen, wobei die reaktivierten Pager ausgeblendet werden, indem Sie `EXCEPT` verwenden:

```
SELECT *
   FROM NichtInBenutzung
EXCEPT CORRESPONDING (PagerID)
SELECT *
   FROM Pager;
```

Diese Abfrage gibt alle Zeilen der Tabelle `NichtInBenutzung` zurück, deren `PagerID` nicht auch in der Tabelle `Pager` gespeichert ist.

Verknüpfungsoperatoren

Mit den Operatoren `UNION`, `INTERSECT` und `EXCEPT` können Sie mehrere `UNION`-kompatible Tabellen einer Datenbank miteinander verknüpfen. In vielen Fällen wollen Sie jedoch Daten aus Tabellen extrahieren, die sehr wenig gemeinsam haben. Verknüpfungen (englisch *Joins*, daher auch das Schlüsselwort `JOIN`, mit dem die Verknüpfungsoperationen im Regelfall durchgeführt werden) sind ein sehr mächtiges Werkzeug, das Daten aus mehreren Tabellen in

einer einzigen Ergebnistabelle zusammenführt. Die Ausgangstabellen brauchen nur wenig (oder sogar nichts) gemeinsam zu haben.

SQL unterstützt eine Reihe unterschiedlicher Verknüpfungen. Welche davon Sie verwenden, ist von den Ergebnissen abhängig, die Sie erzielen möchten.

Die einfache Verknüpfung

Jede Abfrage, die mehrere Tabellen betrifft, ist eine Art von Verknüpfung. Die Daten der Ausgangstabellen werden in dem Sinne miteinander verknüpft, dass die Informationen, die Sie aus ihnen herausziehen, in einer einzigen Ergebnistabelle zusammengefasst werden. Die einfachste Verknüpfung ist ein `SELECT` über zwei Tabellen ohne `WHERE`-Klausel. Jede Zeile der ersten Tabelle wird mit jeder Zeile der zweiten Tabelle verknüpft. Die Ergebnistabelle ist das kartesische Produkt der beiden Ausgangstabellen. Die Anzahl der Zeilen in der Ergebnistabelle ist die Anzahl der Zeilen der ersten Ausgangstabelle, multipliziert mit der Anzahl der Zeilen der zweiten Ausgangstabelle.

Angenommen, Sie seien Personalchef einer Firma und verwalteten die Mitarbeiter in einer Datenbank. Die meisten Daten der Mitarbeiter, zum Beispiel Adressen oder Telefonnummern, sind nicht besonders schutzbedürftig. Einige Daten jedoch, zum Beispiel das aktuelle Gehalt, sollten nur autorisierten Personen zugänglich sein. Um die Sicherheit dieser vertraulichen Informationen zu gewährleisten, speichern Sie diese Daten in einer separaten Tabelle, die durch ein Kennwort geschützt ist. Betrachten Sie die beiden folgenden Tabellen:

```
Mitarbeiter         Bezahlung
-----------         ------------
MitID               Mitarb
Vorname             Gehalt
Nachname            Bonus
Stadt
Telefon
```

Wir wollen die Tabellen mit einigen Beispieldaten füllen:

```
MitID Vorname Nachname Stadt       Telefon
----- ------- -------- ----------- ----------
    1 Walter  Meier    Altdorf     5551001
    2 Dieter  Schulz   Neustadt    5553221
    3 Sarah   Müller   Kleinhausen 5556905
    4 Anna    Schmidt  Oberdorf    5558908

Mitarb   Gehalt   Bonus
------   ------   -----
     1    33000   10000
     2    18000    2000
     3    24000    5000
     4    22000    7000
```

Mit der folgenden Abfrage wird eine virtuelle Tabelle erstellt:

```
SELECT *
   FROM Mitarbeiter, Bezahlung ;
```

Die Ergebnistabelle sieht folgendermaßen aus:

```
MitID  Vorname Nachname Stadt       Telefon  Mitarb  Gehalt Bonus
-----  ------- -------- ----------- -------  ------  ------ -----
    1  Walter  Meier    Altdorf     5551001      1   33000  10000
    1  Walter  Meier    Altdorf     5551001      2   18000   2000
    1  Walter  Meier    Altdorf     5551001      3   24000   5000
    1  Walter  Meier    Altdorf     5551001      4   22000   7000
    2  Dieter  Schulz   Neustadt    5553221      1   33000  10000
    2  Dieter  Schulz   Neustadt    5553221      2   18000   2000
    2  Dieter  Schulz   Neustadt    5553221      3   24000   5000
    2  Dieter  Schulz   Neustadt    5553221      4   22000   7000
    3  Sarah   Müller   Kleinhausen 5556905      1   33000  10000
    3  Sarah   Müller   Kleinhausen 5556905      2   18000   2000
    3  Sarah   Müller   Kleinhausen 5556905      3   24000   5000
    3  Sarah   Müller   Kleinhausen 5556905      4   22000   7000
    4  Anna    Schmidt  Oberdorf    5558908      1   33000  10000
    4  Anna    Schmidt  Oberdorf    5558908      2   18000   2000
    4  Anna    Schmidt  Oberdorf    5558908      3   24000   5000
    4  Anna    Schmidt  Oberdorf    5558908      4   22000   7000
```

Die Ergebnistabelle ist das kartesische Produkt der Tabellen `Mitarbeiter` und `Bezahlung`. Sie enthält viele redundante Daten. Ihr Inhalt ist ziemlich nutzlos, weil sie alle Zeilen beider Tabellen miteinander kombiniert. Sinnvoll sind nur die Zeilen, in denen die Kennung `MitID` der Tabelle `Mitarbeiter` mit der Kennung `Mitarb` der Tabelle `Bezahlung` übereinstimmt. In diesen Zeilen wird die Adresse eines Mitarbeiters mit dem entsprechenden Einkommen kombiniert.

Wenn Sie versuchen, von einer Datenbank mit mehreren Tabellen nützliche Informationen zu erhalten, ist das kartesische Produkt, das eine einfache Verknüpfung erzeugt, zwar so gut wie nie das, was Sie gebrauchen können, aber immer der erste Schritt auf dem Weg zu Ihrem Ziel. Wenn Sie einen `JOIN` mit einer `WHERE`-Klausel um Einschränkungen erweitern, können Sie unerwünschte Zeilen herausfiltern. Die nächsten Abschnitte erklären Ihnen, wie Sie die Informationen filtern, die Sie nicht sehen wollen.

Gleichheitsverknüpfung – Equi-Join

Die am häufigsten verwendete Verknüpfungsart ist eine, die auf Gleichheit setzt. Ein sogenannter *Equi-Join* ist eine einfache Verknüpfung mit einer `WHERE`-Klausel, die als Bedingung angibt, dass der Wert in einer Spalte der ersten Tabelle gleich dem Wert der entsprechenden Spalte in der zweiten Tabelle sein muss. Wenn Sie die Tabellen des letzten Beispiels mit einem Equi-Join verknüpfen, erhalten Sie ein sinnvolleres Ergebnis:

```
SELECT *
   FROM Mitarbeiter, Bezahlung
   WHERE Mitarbeiter.MitID = Bezahlung.Mitarb ;
```

Die Abfrage liefert dieses Ergebnis:

```
MitID  Vorname Nachname Stadt       Telefon  Mitarb  Gehalt  Bonus
------ ------- -------- ----------- ------  -------  ------  -----
    1  Walter  Meier    Altdorf     5551001      1   33000  10000
    2  Dieter  Schulz   Neustadt    5553221      2   18000   2000
    3  Sarah   Müller   Kleinhausen 5556905      3   24000   5000
    4  Anna    Schmidt  Oberdorf    5558908      4   22000   7000
```

In dieser Ergebnistabelle gehören die Adressdaten auf der linken Seite und die Gehaltsdaten auf der rechten Seite zu demselben Mitarbeiter. Die Tabelle enthält immer noch redundante Daten, weil die Kennung des Mitarbeiters (`MIT_ID` und `MITARB`) dupliziert wird. Sie können dieses Problem dadurch beheben, dass Sie die Abfrage ein wenig umformulieren:

```
SELECT Mitarbeiter.*,Bezahlung.Gehalt,Bezahlung.Bonus
   FROM Mitarbeiter, Bezahlung
   WHERE Mitarbeiter.MitID = Bezahlung.Mitarb ;
```

Das Ergebnis lautet jetzt:

```
MitID  Vorname Nachname Stadt       Telefon  Gehalt  Bonus
-----  ------- -------- ----------- -------  ------  -----
    1  Walter  Meier    Altdorf     5551001   33000  10000
    2  Dieter  Schulz   Neustadt    5553221   18000   2000
    3  Sarah   Müller   Kleinhausen 5556905   24000   5000
    4  Anna    Schmidt  Oberdorf    5558908   22000   7000
```

Diese Tabelle zeigt Ihnen genau das, was Sie wissen wollen, und nicht mehr. Es ist nur etwas langweilig, eine solche Abfrage zu schreiben. Um Mehrdeutigkeiten zu vermeiden, sollten Sie die Spaltennamen durch eine Erweiterung um den Namen der entsprechenden Tabellen eindeutig benennen – was allerdings einigen Schreibaufwand darstellt.

Sie können allerdings die Tipparbeit verringern, indem Sie Aliasse benutzen. Ein *Alias* ist ein Kurzname, der zum Beispiel anstelle eines Tabellennamens verwendet werden kann. Wenn Sie in der vorhergehenden Abfrage die Tabellennamen jeweils durch ein Alias ersetzen, sieht diese folgendermaßen aus:

```
SELECT M.*, B.Gehalt, B.Bonus
   FROM Mitarbeiter M, Bezahlung B
   WHERE M.MitID = B.Mitarb ;
```

In diesem Beispiel ist `M` das Alias für `Mitarbeiter` und `B` das Alias für `Bezahlung`. Ein Alias gilt nur für die Anweisung, in der es definiert ist. Wenn Sie ein Alias in der Klausel `FROM` deklarieren, müssen Sie es in der gesamten Anweisung benutzen. Das Alias und die Langform des Tabellennamens dürfen in der gleichen Anweisung nicht gleichzeitig verwendet werden.

Das letzte Beispiel mit den Aliassen entspricht dem folgenden SELECT ohne Aliasse:

```
SELECT T2.C, T1.C
   FROM T1 , T2
   WHERE T2.C > T1.C ;
```

In SQL können Sie mehr als zwei Tabellen verknüpfen. Die maximale Anzahl variiert von Implementierung zu Implementierung. Die Syntax ist vergleichbar mit der Verknüpfung von zwei Tabellen:

```
SELECT M.*, B.Gehalt, M.Bonus, U.Umsatz
   FROM Mitarbeiter M, Bezahlung B, Umsatz U
   WHERE M.MitID = B.Mitarb
      AND B.Mitarb = U.MitarbNR ;
```

Diese Anweisung verknüpft drei Tabellen mit einem Equi-Join. Er ruft die Namen, die Bezahlung und den Umsatz der Verkäufer aus den entsprechenden Zeilen der drei Tabellen ab und setzt daraus die Ergebnistabelle zusammen. Der Verkaufsleiter kann schnell erkennen, ob die Bezahlung den Umsätzen entspricht.

Kreuzverknüpfungen – Cross-Join

CROSS JOIN (englisch für Kreuzverknüpfung) ist das Schlüsselwort für die einfache Verknüpfung ohne WHERE-Klausel. Deshalb kann die Anweisung

```
SELECT *
   FROM Mitarbeiter, Bezahlung ;
```

auch folgendermaßen geschrieben werden:

```
SELECT *
   FROM Mitarbeiter CROSS JOIN Bezahlung ;
```

Das Ergebnis ist das kartesische Produkt (auch *Kreuzprodukt* genannt) der beiden Ausgangstabellen. Mit Kreuzverknüpfungen erhalten Sie selten das Ergebnis, das Sie benötigen, es kann aber der erste Schritt in einer Kette von Operationen zur Datenbearbeitung sein, die letztendlich das gewünschte Resultat erzielt.

Natürliche Verknüpfungen – Natural-Join

Eine natürliche Verknüpfung (englisch *Natural-Join*) ist ein Sonderfall von Equi-Join. In der WHERE-Klausel eines Equi-Join wird eine Spalte einer Tabelle mit einer Spalte einer anderen Tabelle verglichen. Die beiden Spalten müssen vom selben Typ und gleichnamig sein und die gleiche Größe besitzen. Tatsächlich werden bei einer natürlichen Verknüpfung alle Spalten in einer Tabelle, die dieselben Namen, Typen und Größen wie die entsprechenden Spalten in der zweiten Tabelle haben, auf Gleichheit geprüft.

Stellen Sie sich vor, dass die Tabelle Bezahlung des vorangegangenen Beispiels die Spalten MitID, Gehalt und Bonus statt Mitarb, Gehalt und Bonus hat. In diesem Fall können Sie die beiden Tabellen Bezahlung und Mitarbeiter natürlich verknüpfen. In der traditionellen JOIN-Syntax sieht dies folgendermaßen aus:

```
SELECT M.*, B.Gehalt, B.Bonus
   FROM Mitarbeiter M, Bezahlung B
   WHERE M.MitID = B.MitID ;
```

Diese Abfrage ist ein Sonderfall einer natürlichen Verknüpfung. Die SELECT-Anweisung gibt verknüpfte Zeilen zurück, für die M.MitID = B.MitID gilt. Betrachten Sie die folgende Anweisung:

```
SELECT M.*, B.Gehalt, B.Bonus
   FROM Mitarbeiter M Natural JOIN Bezahlung B ;
```

Diese Abfrage verknüpft Zeilen, für die M.MitID = B.MitID gilt, wobei M.Gehalt = B.Gehalt und M.Bonus = B.Bonus ist. Die Ergebnistabelle enthält nur Zeilen, in denen *alle* entsprechenden Spalten übereinstimmen. In diesem Beispiel sind die Ergebnisse beider Abfragen gleich, weil die Tabelle Mitarbeiter keine Gehalt- und keine Bonus-Spalte enthält.

Bedingte Verknüpfungen

Eine *bedingte Verknüpfung* funktioniert wie ein Equi-Join, außer dass die getestete Bedingung nicht eine Gleichheitsbedingung zu sein braucht (obwohl sie es sein kann). Es kann sich um ein beliebiges wohlgeformtes Prädikat handeln. Wenn die Bedingung erfüllt ist, wird die entsprechende Zeile in die Ergebnistabelle aufgenommen. Die Syntax unterscheidet sich etwas von der bisher betrachteten, weil die Bedingung in einer ON-Klausel statt in einer WHERE-Klausel enthalten ist.

Angenommen, Sie möchten die Stürmer der Ersten Bundesliga finden, die genauso viele Tore wie die Spieler in der Zweiten Bundesliga geschossen haben. Diese Abfrage kann durch einen Equi-Join beantwortet werden. Wenn Sie dafür eine bedingte Verknüpfung einsetzen, lautet die Bedingung folgendermaßen:

```
SELECT *
   FROM ErsteLiga JOIN ZweiteLiga
   ON ErsteLiga.Tore = ZweiteLiga.Tore ;
```

Spaltennamenverknüpfungen

Eine *Spaltennamenverknüpfung* funktioniert wie eine natürliche Verknüpfung, ist aber flexibler. Bei einer natürlichen Verknüpfung haben alle Spalten in den Ausgangstabellen dieselben Namen und werden auf Gleichheit getestet. Bei einer Spaltennamenverknüpfung geben Sie an, welche gleichnamigen Spalten verglichen werden sollen und welche nicht. Sie können natürlich alle gleichnamigen Spalten wählen und damit faktisch eine natürliche Verknüpfung ausführen. Oder Sie wählen nicht alle gleichnamigen Spalten aus. Auf diese Weise haben Sie eine größere Kontrolle über die Auswahl der Datensätze.

Angenommen, Sie stellen Schachfiguren her und verwalten Ihre Lagerbestände an weißen und schwarzen Figuren in zwei separaten Tabellen, `Weiß` und `Schwarz`. Die Tabellen enthalten folgende Daten:

```
Weiß                              Schwarz
-----                             --------
Figur    Menge  Holz              Figur    Menge     Holz
-------  -----  --------          -------  -----     --------
König      502  Eiche             König      502     Ebenholz
Dame       398  Eiche             Dame       397     Ebenholz
Turm      1020  Eiche             Turm      1020     Ebenholz
Läufer     985  Eiche             Läufer     985     Ebenholz
Springer   950  Eiche             Springer   950     Ebenholz
Bauer      431  Eiche             Bauer      453     Ebenholz
```

Die Anzahl der weißen und schwarzen Figuren sollte in beiden Tabellen übereinstimmen. Falls dies nicht der Fall ist, sind Figuren verloren gegangen oder gestohlen worden, und Sie müssen Ihre Sicherheitsmaßnahmen verstärken.

Eine natürliche Verknüpfung testet alle gleichnamigen Spalten auf Gleichheit. In diesem Fall enthält die Ergebnistabelle keine einzige Zeile, weil sich alle Tabellenzeilen in den beiden Tabellen durch den Wert in der Spalte `Holz` unterscheiden. Mit dieser Ergebnistabelle können Sie also nicht feststellen, ob Figuren fehlen. Wenn Sie dagegen eine Spaltennamenverknüpfung verwenden, die die Spalte `Holz` ignoriert, finden Sie die Antwort:

```
SELECT *
   FROM Weiß JOIN Schwarz
   USING (Figur, Menge) ;
```

Die Ergebnistabelle zeigt nur die Zeilen, bei denen die Anzahl der weißen Figuren mit der Anzahl der schwarzen Figuren übereinstimmt.

```
Figur    Menge     Holz      Figur    Menge     Holz
-------  -----     --------  -------  -----     --------
König      502     Eiche     König      502     Ebenholz
Turm      1020     Eiche     Turm      1020     Ebenholz
Läufer     985     Eiche     Läufer     985     Ebenholz
Springer   950     Eiche     Springer   950     Ebenholz
```

Ein guter Beobachter erkennt sofort, dass Dame und Bauer in der Liste fehlen, was auf eine Fehlmenge an irgendeiner Stelle hinweist.

Innere Verknüpfungen – INNER JOIN

Bis hierher haben Sie wahrscheinlich den Eindruck gewonnen, dass Verknüpfungen ein ziemlich esoterisches Thema sind, das man nur mit einem höheren Grad spiritueller Erleuchtung meistern kann. Vielleicht haben Sie sogar schon von den mysteriösen *inneren Verknüpfungen* (englisch *Inner Join*) gehört und dabei gedacht, dass diese wahrscheinlich den Kern oder das

Wesen von relationalen Operationen ausmachen. Hah! Reingefallen! Innere Verknüpfungen sind nichts Geheimnisvolles. Tatsächlich gehören alle Verknüpfungen, die Sie bis jetzt in diesem Kapitel kennengelernt haben, zu den inneren Verknüpfungen. Ich hätte die Spaltennamenverknüpfung im letzten Beispiel auch als innere Verknüpfungen ausdrücken können:

```
SELECT *
   FROM Weiß INNER JOIN Schwarz
   USING (Figur, Menge) ;
```

Das Ergebnis ist dasselbe.

Die innere Verknüpfung trägt diesen Namen, um sie von der sogenannten äußeren Verknüpfung (englisch *Outer Join*) zu unterscheiden. Eine innere Verknüpfung schließt alle Zeilen aus der Ergebnistabelle aus, zu denen es keinen entsprechenden Datensatz in beiden Ausgangstabellen gibt. Eine äußere Verknüpfung kümmert sich um alle nicht passenden Zeilen. Das ist der Unterschied. Dahinter verbirgt sich also kein metaphysisches Geheimnis.

Äußere Verknüpfungen – OUTER JOIN

Wenn Sie zwei Tabellen verknüpfen, kann es vorkommen, dass die erste (die *linke*) Tabelle Datensätze enthält, zu denen es keine Entsprechungen in der zweiten (der *rechten*) Tabelle gibt. Umgekehrt kann die rechte Tabelle Datensätze enthalten, zu denen es keine Entsprechungen in der Tabelle auf der linken Seite gibt. Wenn Sie eine innere Verknüpfung dieser Tabellen ausführen, werden alle Zeilen ohne Entsprechungen vom Ergebnis ausgeschlossen. Äußere Verknüpfungen hingegen liefern eben genau diese nicht passenden Zeilen. Es gibt drei Arten von äußeren Verknüpfungen: die linke (Left Outer Join), die rechte (Right Outer Join) und die vollständige (Full Outer Join).

Die linke äußere Verknüpfung – LEFT OUTER JOIN

Wenn eine Abfrage eine Verknüpfung enthält, wird die Tabelle vor dem Schlüsselwort JOIN als *linke Tabelle* bezeichnet. Analog dazu wird die Tabelle hinter dem Schlüsselwort *rechte Tabelle* genannt. Die linke äußere Verknüpfung *(Left Outer Join)* übernimmt die Zeilen der linken Tabelle, die keine Entsprechung in der rechten Tabelle haben, in das Ergebnis, während die Zeilen der rechten Tabelle, zu denen es keine Übereinstimmung in der linken Tabelle gibt, ausgeschlossen bleiben.

Um äußere Verknüpfungen ein bisschen besser zu verstehen, stellen Sie sich eine Firmendatenbank vor, in der es Tabellen für Mitarbeiter, Abteilungen und Standorte gibt. Tabelle 11.1, Tabelle 11.2 und Tabelle 11.3 enthalten die für dieses Beispiel benötigten Daten:

StandortID	Stadt
1	Berlin
3	Trier
5	Chemnitz

Tabelle 11.1: Tabelle `Standort`

AbteilungID	StandortID	Name
21	1	Verkauf
24	1	Verwaltung
27	5	Wartung
29	5	Lager

Tabelle 11.2: Tabelle `Abteilung`

MitarbID	AbteilungID	Name
61	24	Meier
63	27	Schulze

Tabelle 11.3: Tabelle `Mitarbeiter`

Stellen Sie sich jetzt vor, dass Sie alle Daten aller Mitarbeiter einschließlich ihrer Abteilungen und Standorte sehen wollen. Sie können dies mit einem Equi-Join erreichen:

```
SELECT *
   FROM Standort S, Abteilung A, Mitarbeiter M
   WHERE S.StandortID = A.StandortID
      AND A.AbteilungID = M.AbteilungID ;
```

Diese Anweisung liefert folgendes Ergebnis:

```
1    Berlin     24    1     Verwaltung    61    24    Meier
5    Chemnitz   27    5     Wartung       63    27    Schulze
```

Diese Ergebnistabelle enthält alle Daten aller Mitarbeiter einschließlich ihrer Standorte und Abteilungen. Dieser Equi-Join funktioniert, weil jeder Mitarbeiter einen Standort und eine Abteilung hat.

Angenommen, Sie möchten die Daten der Standorte mit den dort vorhandenen Abteilungen und Mitarbeitern sehen. Dies ist ein anderes Problem, weil es Standorte ohne Abteilungen geben kann. Um das gewünschte Ergebnis zu erzielen, müssen Sie einen Outer-Join wie diesen verwenden:

```
SELECT *
   FROM Standort S LEFT OUTER JOIN Abteilung A
      ON (S.StandortID = A.StandortID)
   LEFT OUTER JOIN Mitarbeiter M
      ON (A.AbteilungID = M.AbteilungID);
```

Diese Verknüpfung ruft Daten aus drei Tabellen ab. Zunächst wird die Tabelle `Standort` mit der Tabelle `Abteilung` verknüpft. Die daraus resultierende Tabelle wird dann mit der Tabelle `Mitarbeiter` verknüpft. Die Zeilen der Tabelle auf der linken Seite des Operators `LEFT OUTER JOIN`, zu denen es keine entsprechenden Zeilen in der Tabelle auf der rechten Seite des

Operators gibt, werden in das Ergebnis eingeschlossen. Deshalb werden bei der ersten Verknüpfung alle Standorte eingeschlossen, selbst wenn es dort keine Abteilungen gibt. Bei der zweiten Verknüpfung werden alle Abteilungen eingeschlossen, selbst wenn sie keine Mitarbeiter haben. Das Ergebnis lautet:

```
1    Berlin     24    1    Verwaltung  61    24    Meier
5    Chemnitz   27    5    Wartung     63    27    Schulze
3    Trier      NULL  NULL NULL        NULL  NULL  NULL
5    Chemnitz   29    5    Lager       NULL  NULL  NULL
1    Berlin     21    1    Verkauf     NULL  NULL  NULL
```

Die ersten beiden Zeilen entsprechen den beiden Ergebnisdatensätzen des vorangegangenen Beispiels. Der dritte Datensatz (`3 Trier`) enthält in den Spalten für die Abteilung und Mitarbeiter Nullwerte, weil es in Trier keine Abteilungen und Mitarbeiter gibt. Der vierte und fünfte Datensatz (`5 Chemnitz` und `1 Berlin`) enthalten Daten über die Abteilungen Lager und Verkauf, aber die Spalten für die Mitarbeiter enthalten Nullwerte, weil diese beiden Abteilungen keine Mitarbeiter haben. Diese äußere Verknüpfung liefert Ihnen alle Informationen, die auch ein Equi-Join zur Verfügung gestellt hätte, und zusätzlich folgende Daten:

- ✔ Alle Standorte der Firma, unabhängig davon, ob sie Abteilungen haben oder nicht
- ✔ Alle Abteilungen der Firma, unabhängig davon, ob sie Mitarbeiter haben oder nicht

Die Datensätze, die in dem obigen Beispiel zurückgegeben werden, haben keine bestimmte Reihenfolge. Die Reihenfolge ist implementierungsabhängig. Sie können die Daten mit einer zusätzlichen `ORDER BY`-Klausel in der `SELECT`-Anweisung sortieren:

```
SELECT *
   FROM Standort S LEFT OUTER JOIN Abteilung A
      ON (S.StandortID = A.StandortID)
   LEFT OUTER JOIN Mitarbeiter M
      ON (A.AbteilungID = M.AbteilungID)
   ORDER BY S.StandortID, A.AbteilungID, M.MitarbID;
```

Der Ausdruck `LEFT OUTER JOIN` kann als `LEFT JOIN` abgekürzt werden, weil es keine linke *innere* Verknüpfung gibt.

Rechte äußere Verknüpfung – RIGHT OUTER JOIN

Die rechte äußere Verknüpfung *(Right Outer Join)* funktioniert analog. Sie übernimmt Zeilen ohne Entsprechung aus der rechten Tabelle in das Ergebnis, verwirft aber die Zeilen der linken Tabelle, zu denen es keine Entsprechung gibt. Sie können sie auf dieselben Tabellen anwenden und dasselbe Ergebnis erzielen, wenn Sie die Reihenfolge umkehren, in der Sie die Tabellen im Verhältnis zum `JOIN`-Operator angeben:

```
SELECT *
   FROM Mitarbeiter M RIGHT OUTER JOIN Abteilung A
      ON (A.AbteilungID = M.AbteilungID)
   RIGHT OUTER JOIN Standort S
      ON (S.StandortID = A.StandortID) ;
```

Bei dieser Formulierung erzeugt der erste `JOIN` eine Tabelle, die alle Abteilungen enthält, und zwar unabhängig davon, ob die jeweilige Abteilung Mitarbeiter hat oder nicht. Der zweite `JOIN` erzeugt eine Tabelle, die alle Standorte enthält, und zwar unabhängig davon, ob der jeweilige Standort Abteilungen hat oder nicht.

Der Ausdruck `RIGHT OUTER JOIN` kann als `RIGHT JOIN` abgekürzt werden, weil es keine rechte innere Verknüpfung gibt.

Vollständige äußere Verknüpfung – FULL OUTER JOIN

Die vollständige äußere Verknüpfung *(Full Outer Join)* kombiniert die Funktionen der linken und rechten äußeren Verknüpfung. Sie übernimmt aus der linken und der rechten Tabelle alle Datensätze ohne Entsprechung in das Ergebnis. Betrachten Sie den allgemeinsten Fall der Firmendatenbank, die im letzten Beispiel verwendet wird. Folgende Konstellationen sind denkbar: Standorte ohne Abteilungen, Abteilungen ohne Standorte, Abteilungen ohne Mitarbeiter und Mitarbeiter ohne Abteilungen.

Wenn Sie unabhängig davon, ob es entsprechende Datensätze in den anderen Tabellen gibt, alle Standorte, Abteilungen und Mitarbeiter anzeigen wollen, benutzen Sie eine vollständige äußere Verknüpfung:

```
SELECT *
   FROM Standort S FULL JOIN Abteilung A
      ON (S.StandortID = A.StandortID)
   FULL JOIN Mitarbeiter M
      ON (A.AbteilungID = M.AbteilungID) ;
```

Der Ausdruck `FULL OUTER JOIN` kann als `FULL JOIN` abgekürzt werden, weil es keine vollständige äußere Verknüpfung gibt.

Vereinigungsverknüpfungen – Union Join

Im Gegensatz zu den anderen Verknüpfungsarten versucht die Vereinigungsverknüpfung *(Union Join)* nicht, Entsprechungen zwischen den Datensätzen der linken Ausgangstabelle und den Datensätzen der rechten Ausgangstabelle zu finden. Sie erstellt eine neue virtuelle Tabelle, die die Vereinigung aller Spalten in beiden Ausgangstabellen enthält. In der virtuellen Ergebnistabelle enthalten die Spalten, die aus der linken Ausgangstabelle stammen, alle Zeilen der linken Ausgangstabelle. In diesen Zeilen enthalten die Spalten, die aus der rechten Ausgangstabelle stammen, einen Nullwert. Ähnliches gilt für die Spalten, die aus der rechten

Ausgangstabelle stammen. Sie enthalten alle Datensätze der rechten Ausgangstabelle. In diesen Zeilen enthalten die Spalten, die aus der linken Ausgangstabelle stammen, einen Nullwert. Deshalb enthält die Ergebnistabelle einer Vereinigungsverknüpfung alle Spalten beider Ausgangstabellen, und die Anzahl der Zeilen in der Ergebnistabelle ist die Summe der Anzahl der Zeilen in den beiden Ausgangstabellen.

Das Ergebnis einer Vereinigungsverknüpfung ist in den meisten Fällen nicht direkt brauchbar. Diese Verknüpfung erzeugt Ergebnistabellen mit vielen Nullwerten. Sie können jedoch nützliche Informationen aus einer Vereinigungsverknüpfung gewinnen, wenn Sie sie in Verbindung mit einem `COALESCE`-Ausdruck (siehe Kapitel 9) einsetzen. Betrachten wir ein Beispiel.

Stellen Sie sich vor, dass Sie für eine Firma arbeiten, die Raketen entwickelt und testet. Sie betreuen mehrere Projekte. Außerdem haben Sie mehrere Entwurfsingenieure, die über Fähigkeiten auf mehreren Gebieten verfügen. Als Manager wollen Sie wissen, welcher Mitarbeiter welche Fähigkeiten hat und an welchen Projekten er mitgearbeitet hat. Im Moment sind diese Daten verstreut in den Tabellen `Mitarbeiter`, `Projekt` und `Fähigkeit` gespeichert.

Die Tabelle `Mitarbeiter` enthält die Daten der Mitarbeiter. Sie hat den Primärschlüssel `Mitarbeiter.MitarbID`. Die Tabelle `Projekte` enthält eine Zeile für jedes Projekt, an dem ein Mitarbeiter gearbeitet hat. Die Spalte `Projekte.MitarbID` ist ein Fremdschlüssel, der auf die Tabelle `Mitarbeiter` verweist. In der Tabelle `Fähigkeit` sind die Fähigkeiten eines jeden Mitarbeiters gespeichert. Die Spalte `Fähigkeit.MitarbID` ist ein Fremdschlüssel, der ebenfalls auf die Tabelle `Mitarbeiter` verweist.

Die Tabelle `Mitarbeiter` enthält für jeden Mitarbeiter eine Zeile, die Tabellen `Projekte` und `Fähigkeit` können pro Mitarbeiter null oder mehr Zeilen enthalten.

Tabelle 11.4, Tabelle 11.5 und Tabelle 11.6 zeigen die Beispieldaten, die in den drei Tabellen vorliegen.

MitarbID	Name
1	Maier
2	Meier
3	Schulze

Tabelle 11.4: Tabelle `Mitarbeiter`

ProjektName	MitarbID
X-63 Struktur	1
X-64 Struktur	1
X-63 Steuerung	2
X-64 Steuerung	2
X-63 Telemetrie	3
X-64 Telemetrie	3

Tabelle 11.5: Tabelle `Projekte`

Fähigkeit	MitarbID
Mechanikdesign	1
Aerodynamik	1
Analogdesign	2
Gyroskopbau	2
Digitaldesign	3
R/F-Design	3

Tabelle 11.6: Tabelle `Fähigkeit`

Aus den Tabellen können Sie ablesen, dass *Maier* am Strukturdesign von X-63 und X-64 mitgearbeitet hat und über Fähigkeiten auf den Gebieten *Mechanikdesign* und *Aerodynamik* verfügt.

Angenommen, Sie möchten als Manager alle Informationen über alle Mitarbeiter sehen: Sie entscheiden sich deshalb für einen Equi-Join auf die Tabellen `Mitarbeiter`, `Projekte` und `Fähigkeit`:

```
SELECT *
   FROM Mitarbeiter M, Projekte P, Fähigkeit F
   WHERE M.MitarbID = P.MitarbID
      AND M.MitarbID = F.MitarbID ;
```

Sie können dieselbe Operation auch mit einer inneren Verknüpfung formulieren:

```
SELECT *
   FROM Mitarbeiter M INNER JOIN Projekte P
      ON (M.MitarbID = P.MitarbID)
   INNER JOIN Fähigkeit F
      ON (M.MitarbID = F.MitarbID ;
```

Beide Varianten liefern dasselbe Ergebnis (siehe Tabelle 11.7).

M.MitarbID	M.Name	P.MitarbID	ProjektName	F.MitarbID	F.Fähigkeit
1	Maier	1	X-63 Struktur	1	Mechanikdesign
1	Maier	1	X-63 Struktur	1	Aerodynamik
1	Maier	1	X-64 Struktur	1	Mechanikdesign
1	Maier	1	X-64 Struktur	1	Aerodynamik
2	Meier	2	X-63 Steuerung	2	Analogdesign
2	Meier	2	X-63 Steuerung	2	Gyroskopbau
2	Meier	2	X-64 Steuerung	2	Analogdesign
2	Meier	2	X-64 Steuerung	2	Gyroskopbau
3	Schulze	3	X-63 Telemetrie	3	Digitaldesign
3	Schulze	3	X-63 Telemetrie	3	R/F-Design
3	Schulze	3	X-64 Telemetrie	3	Digitaldesign
3	Schulze	3	X-64 Telemetrie	3	R/F-Design

Tabelle 11.7: Ergebnis der inneren Verknüpfung

Diese Anordnung der Daten ist nicht besonders aussagekräftig. Die Kennungen der Mitarbeiter kommen dreimal und die Projekte und Fähigkeiten für jeden Mitarbeiter zweimal vor. Eine innere Verknüpfung ist also für diese Art von Abfrage unbrauchbar. Sie können hier stattdessen mit einer Vereinigungsverknüpfung (Union Join) arbeiten, die Sie mit einigen strategisch gewählten `SELECT`-Anweisungen kombinieren, um ein ansprechenderes Ergebnis zu erzielen. Sie beginnen mit einer einfachen Vereinigungsverknüpfung:

```
SELECT *
   FROM MITARBEITER M UNION JOIN PROJEKTE P
      UNION JOIN FÄHIGKEIT F ;
```

Beachten Sie, dass diese Vereinigungsverknüpfung keine `ON`-Klausel enthält. Sie filtert keine Daten und braucht deshalb keine `ON`-Klausel. Diese Anweisung erzeugt das Ergebnis aus Tabelle 11.8.

M.MitarbID	M.Name	P.MitarbID	P.ProjektName	F.MitarbID	F.Fähigkeit
1	Maier	NULL	NULL	NULL	NULL
NULL	NULL	1	X-63 Struktur	NULL	NULL
NULL	NULL	1	X-64 Struktur	NULL	NULL
NULL	NULL	NULL	NULL	1	Mechanikdesign
NULL	NULL	NULL	NULL	1	Aerodynamik
2	Meier	NULL	NULL	NULL	NULL
NULL	NULL	2	X-63 Steuerung	NULL	NULL
NULL	NULL	2	X-64 Steuerung	NULL	NULL
NULL	NULL	NULL	NULL	2	Analogdesign
NULL	NULL	NULL	NULL	2	Gyroskopbau
3	Schulze	NULL	NULL	NULL	NULL
NULL	NULL	3	X-63 Telemetrie	NULL	NULL
NULL	NULL	3	X-64 Telemetrie	NULL	NULL
NULL	NULL	NULL	NULL	3	Digitaldesign
NULL	NULL	NULL	NULL	3	R/F-Design

Tabelle 11.8: Ergebnis des Union-Joins

Jede Tabelle wurde links oder rechts um Spalten mit Nullwerten erweitert und dann wurden die erweiterten Tabellen vereinigt. Die Reihenfolge der Zeilen ist nicht definiert und hängt von Ihrer Implementierung ab. Jetzt können Sie die Daten »kneten«, um sie in eine brauchbarere Form zu bringen.

Beachten Sie zunächst, dass die Tabelle drei ID-Spalten enthält, von denen in jeder Zeile zwei `NULL` sind. Sie können die Anzeige verbessern, indem Sie `COALESCE` auf die `ID`-Spalten anwenden. Wie ich in Kapitel 9 beschreibe, nimmt sich der Ausdruck `COALESCE` den ersten von

NULL verschiedenen Wert einer Werteliste. Bei diesem Beispiel nimmt er den einzigen von NULL verschiedenen Wert einer Liste von Spalten:

```
SELECT COALESCE (M.MitarbID,P.MitarbID,F.MitarbID) AS ID,
      M.Name, P.ProjektName, F.Fähigkeit
   FROM Mitarbeiter M UNION JOIN Projekte P
      UNION JOIN Fähigkeit F
   ORDER BY ID ;
```

Die Klausel FROM hat sich gegenüber dem letzten Beispiel nicht geändert, aber jetzt werden die drei MitarbID-Spalten mit COALESCE zu einer einzigen Spalte mit dem Namen ID verdichtet. Außerdem sortieren Sie das Ergebnis nach der ID. Tabelle 11.9 zeigt das Ergebnis:

ID	M.Name	P.ProjektName	F.Fähigkeit
1	Maier	X-63 Struktur	NULL
1	Maier	X-64 Struktur	NULL
1	Maier	NULL	Mechanikdesign
1	Maier	NULL	Aerodynamik
2	Meier	X-63 Steuerung	NULL
2	Meier	X-64 Steuerung	NULL
2	Meier	NULL	Analogdesign
2	Meier	NULL	Gyroskopbau
3	Schulze	X-63 Telemetrie	NULL
3	Schulze	X-64 Telemetrie	NULL
3	Schulze	NULL	Digitaldesign
3	Schulze	NULL	R/F-Design

Tabelle 11.9: Ergebnis der Vereinigungsverknüpfung mit dem Ausdruck COALESCE

Jede Zeile in diesem Ergebnis enthält entweder Daten über ein Projekt oder über eine Fähigkeit, aber nicht über beides. Wenn Sie das Ergebnis betrachten, müssen Sie erst feststellen, welche Art von Informationen – Projekt oder Fähigkeit – eine bestimmte Zeile anzeigt. Wenn die Spalte ProjektName nicht NULL ist, enthält die Zeile ein Projekt, an dem der Mitarbeiter beteiligt war. Wenn die Spalte Fähigkeit nicht NULL ist, enthält die Zeile die Fähigkeit des Mitarbeiters.

Sie können das Ergebnis noch etwas übersichtlicher gestalten, wenn Sie einen weiteren COALESCE-Ausdruck in die SELECT-Anweisung einfügen:

```
SELECT COALESCE (M.MitarbID,P.MitarbID,F.MitarbID) AS ID,
      M.Name, COALESCE (P.Typ, F.Typ) AS Typ,
      ProjektName, F.Fähigkeit
```

```
FROM Mitarbeiter M
   UNION JOIN (Select 'Projekt' AS Typ, P.*
                  FROM Projekte) P
   UNION JOIN (SELECT 'Fähigkeit' AS Typ, F.*
                  FROM Fähigkeit) F
ORDER BY ID, Typ ;
```

Die Tabelle `Projekte` des vorherigen Beispiels wurde in dieser Vereinigungsverknüpfung durch eine eingebettete `SELECT`-Anweisung ersetzt, die den Spalten aus der Tabelle `Projekte` eine Spalte mit dem Namen `P.Typ` und dem konstanten Wert '`Projekt`' hinzufügt. Ähnliches gilt für die Tabelle `Fähigkeit`, die durch eine eingebettete `SELECT`-Anweisung ersetzt worden ist, die zu den Spalten aus der Tabelle `Fähigkeit` eine Spalte mit dem Namen `F.Typ` und dem konstanten Wert '`Fähigkeit`' hinzufügt. In jeder Zeile hat `P.Typ` entweder den Wert `NULL` oder '`Projekt`', und `F.Typ` hat entweder den Wert `NULL` oder '`Fähigkeit`'.

Die äußere `Select`-Liste enthält einen `COALESCE`-Ausdruck, der diese beiden `Typ`-Spalten zu einer einzigen Spalte mit dem Namen `Typ` zusammenfasst. Sie können `Typ` dann in der Klausel `ORDER BY` als Sortierkriterium angeben, damit alle Zeilen mit derselben ID so sortiert werden, dass erst die Fähigkeiten und dann die Projekte aufgeführt werden. Das Ergebnis wird in Tabelle 11.10 gezeigt.

ID	M.Name	Typ	ProjektName	F.Fähigkeit
1	Maier	Fähigkeit	NULL	Mechanikdesign
1	Maier	Fähigkeit	NULL	Aerodynamik
1	Maier	Projekt	X-63 Struktur	NULL
1	Maier	Projekt	X-64 Struktur	NULL
2	Meier	Fähigkeit	NULL	Analogdesign
2	Meier	Fähigkeit	NULL	Gyroskopbau
2	Meier	Projekt	X-63 Steuerung	NULL
2	Meier	Projekt	X-64 Steuerung	NULL
3	Schulze	Fähigkeit	NULL	Digitaldesign
3	Schulze	Fähigkeit	NULL	R/F-Design
3	Schulze	Projekt	X-63 Telemetrie	NULL
3	Schulze	Projekt	X-64 Telemetrie	NULL

Tabelle 11.10: Verfeinertes Ergebnis der Vereinigungsverknüpfung mit `COALESCE`-Ausdrücken

Die Ergebnistabelle enthält jetzt eine sehr gute Übersicht über die Fähigkeiten und Projekterfahrungen aller Mitarbeiter, die es in der Tabelle `Mitarbeiter` gibt.

In Anbetracht der vielen verschiedenen verfügbaren `JOIN`-Operationen sollte es kein Problem sein, Daten ungeachtet der Tabellenstrukturen aus verschiedenen Tabellen zu verknüpfen. Vertrauen Sie darauf, dass Sie mit SQL die Daten in eine lesbare, sinnvolle Form bringen können, wenn die Rohdaten in Ihrer Datenbank erst einmal gespeichert sind.

ON im Vergleich zu WHERE

Es kann schon ein wenig verwirrend sein, sich mit der Funktion der Klauseln `ON` und `WHERE` in den verschiedenen Verknüpfungsarten auseinanderzusetzen. Die folgende Liste soll Ihnen helfen, den Überblick zu bewahren:

- ✔ Die Klausel `ON` gehört zu einer inneren, linken, rechten oder vollständigen Verknüpfung *(Inner-, Left-, Right- oder Full-Join)*. Die Kreuzverknüpfung *(Cross-Join)* und die Vereinigungsverknüpfung *(Union-Join)* kennen keine `ON`-Klausel, weil diese Verknüpfungsarten keine Daten filtern.
- ✔ Die Klausel `ON` entspricht bei einer inneren Verknüpfung logisch einer `WHERE`-Klausel. Ein und dieselbe Bedingung kann sowohl mit einer `ON`- als auch mit einer `WHERE`-Klausel angegeben werden.
- ✔ Die `ON`-Klauseln unterscheiden sich bei äußeren Verknüpfungen (links, rechts und vollständig) von `WHERE`-Klauseln. Die `WHERE`-Klausel filtert einfach die Zeilen, die von der `FROM`-Klausel zurückgegeben werden. Zeilen, die die Filterbedingung nicht erfüllen, gehören nicht zum Ergebnis. Die `ON`-Klausel einer äußeren Verknüpfung filtert erst die Zeilen eines Kreuzprodukts und schließt dann noch die zurückgewiesenen Zeilen ein, die um Nullwerte ergänzt werden.

Mit verschachtelten Abfragen tief graben

12

In diesem Kapitel

- Daten mit einer einzigen SQL-Anweisung aus mehreren Tabellen abrufen
- Einen Wert einer Tabelle mit einer Menge von Werten aus einer anderen Tabelle vergleichen
- Die Anweisung SELECT verwenden, um einen Wert einer Tabelle mit einem einzelnen Wert einer anderen Tabelle zu vergleichen
- Einen Wert einer Tabelle mit allen dazu korrespondierenden Werten einer anderen Tabelle vergleichen
- Abfragen erstellen, die die Zeilen einer Tabelle zu entsprechenden Zeilen einer anderen Tabelle in Beziehung setzen
- Mit einer Unterabfrage festlegen, welche Zeilen geändert, gelöscht oder eingefügt werden sollen

Eine der besten Methoden, um die Integrität der Daten zu schützen und vor Änderungsanomalien zu bewahren (siehe Kapitel 5), ist das Normalisieren einer Datenbank. *Normalisierung* bedeutet, eine einzelne Tabelle in mehrere, thematisch einheitliche Tabellen zu zerlegen. Daten von Artikeln werden beispielsweise von Kundendaten getrennt, selbst wenn die Kunden Artikel gekauft haben.

Wenn Sie eine Datenbank sauber normalisieren, werden die Daten über mehrere Tabellen verteilt. Die meisten Ihrer Abfragen müssen somit Daten aus zwei oder mehr Tabellen abrufen. Eine Methode, solche Abfragen zu formulieren, besteht darin, mit dem Operator JOIN oder einem der anderen relationalen Operatoren (UNION, INTERSECT oder EXCEPT) zu arbeiten. Diese relationalen Operatoren fassen Informationen aus mehreren Tabellen in einer einzigen Tabelle zusammen, wobei sich die Vorgehensweisen der Operatoren unterscheiden.

Eine andere Methode, Daten aus zwei oder mehr Tabellen abzurufen, arbeitet mit verschachtelten Abfragen. Unter SQL handelt es sich bei einer *verschachtelten Abfrage* um eine Abfrage, die eine Unterabfrage enthält. Unterabfragen können selbst weitere Unterabfragen enthalten. Theoretisch ist die Schachtelungstiefe nicht begrenzt, praktisch ist sie abhängig von Ihrer Implementierung.

Unterabfragen bestehen ausnahmslos aus SELECT-Anweisungen, während die äußere, die Unterabfrage einschließende Anweisung auch ein INSERT, UPDATE oder DELETE sein kann.

Weil Unterabfragen auf andere Tabellen zugreifen können als die Anweisung, die sie einschließt, bilden sie eine weitere Methode, um Informationen aus mehreren Tabellen zu extrahieren.

Stellen Sie sich beispielsweise vor, dass Sie Ihre Firmendatenbank abfragen wollen, um alle Abteilungen zu ermitteln, deren Manager älter als 50 Jahre sind. Mit den Verknüpfungen, die ich in Kapitel 11 erkläre, sind Sie in der Lage, eine Abfrage wie diese zu erstellen:

```
SELECT A.AbtNr, A.Name, M.Name, M.LebAlter
   FROM Abteilung A, Mitarbeiter M
   WHERE A.ManagerID = M.ID AND M.LebAlter > 50 ;
```

A ist das Alias für die Tabelle Abteilung und M ist das Alias für die Tabelle Mitarbeiter. Die Tabelle Mitarbeiter hat eine ID-Spalte, die zugleich der Primärschlüssel ist, und die Tabelle Abteilung hat die Spalte ManagerID, die die Kennung des Mitarbeiters enthält, der die Abteilung leitet. Die Tabellen werden in der FROM-Klausel einfach verknüpft. Die Bedingungen in der WHERE-Klausel filtern nur die Zeilen heraus, die das Kriterium erfüllen. Beachten Sie, dass die Parameterliste der SELECT-Anweisung die Spalten AbtNr und Name der Tabelle Abteilung und die Spalten Name und LebAlter (Lebensalter – »ALTER« ist ein reserviertes Wort) der Tabelle Mitarbeiter enthält.

Angenommen, Sie möchten zwar im Prinzip dasselbe Ergebnis haben, dabei aber nur die Spalten der Tabelle Abteilung sehen. Sie sind also nur an den Abteilungen interessiert, deren Manager älter als 50 Jahre sind, aber es ist Ihnen eigentlich gleichgültig, wer diese Manager sind und welches Alter sie haben. An dieser Stelle bietet es sich an, lieber eine Unterabfrage zu schreiben, als eine Verknüpfung zu verwenden:

```
SELECT A.AbtNr, A.Name
   FROM Abteilung A
   WHERE EXISTS (SELECT * FROM Mitarbeiter M
      WHERE M.ID = A.ManagerID AND M.LebAlter > 50) ;
```

Diese Abfrage enthält zwei neue Elemente: das Schlüsselwort EXISTS und den Ausdruck SELECT * in der WHERE-Klausel der ersten SELECT-Anweisung. Bei der zweiten SELECT-Anweisung handelt es sich um eine Unterabfrage. Das Schlüsselwort EXISTS gehört zu den Werkzeugen, die bei Unterabfragen benutzt werden und die ich in diesem Kapitel beschreibe.

Was Unterabfragen erledigen

Unterabfragen stehen in der WHERE-Klausel der Anweisung, in der sie eingeschlossen sind. Ihre Aufgabe ist, die Suchbedingungen der WHERE-Klausel festzulegen. Jede Art von Unterabfrage erzeugt ihr eigenes Ergebnis. Einige Unterabfragen erzeugen eine Werteliste, die von der einschließenden Anweisung als Eingabe verwendet wird. Andere Unterabfragen produzieren einen einzelnen Wert, der dann von der einschließenden Anweisung mit einem Vergleichsoperator ausgewertet wird. Eine dritte Art von Unterabfragen gibt den Wert TRUE (wahr) oder FALSE (falsch) zurück.

Verschachtelte Abfragen, die eine Zeilenmenge zurückgeben

Um zu zeigen, wie eine verschachtelte Abfrage eine Zeilenmenge zurückgibt, gehen wir davon aus, dass Sie für ein Computer-Systemhaus arbeiten. Ihre Firma, die Zetec GmbH, baut Systeme aus gekauften Einzelkomponenten zusammen und verkauft sie dann an Unternehmen und Behörden. Sie verwalten das Geschäft mit einer relationalen Datenbank. Die Datenbank enthält viele Tabellen, von denen im Moment nur die Tabellen `Artikel`, `Verwendung` und `Komponente` interessant sind. Die Tabelle `Artikel` (siehe Tabelle 12.1) enthält eine Liste Ihrer Standardprodukte. Die Tabelle `Komponente` (siehe Tabelle 12.2) enthält die Komponenten, aus denen Sie Ihre Produkte zusammenbauen, und die Tabelle `Verwendung` (siehe Tabelle 12.3) zeigt, welche Komponenten bei welchem Produkt verwendet werden.

Spalte	Typ	Einschränkung
`Modell`	`Char(6)`	`PRIMARY KEY`
`Artname`	`Char(35)`	
`Artbeschr`	`Char(31)`	
`Listenpreis`	`Numeric (9,2)`	

Tabelle 12.1: Tabelle `Artikel`

Spalte	Typ	Einschränkung
`KompID`	`CHAR(6)`	`NOT NULL, PRIMARY KEY`
`KompTyp`	`CHAR(10)`	
`KompBeschr`	`CHAR(31)`	

Tabelle 12.2: Tabelle `Komponente`

Spalte	Typ	Einschränkung
`Modell`	`CHAR(6)`	`FOREIGN KEY (für Artikel)`
`KompID`	`CHAR(6)`	`FOREIGN KEY (für Komponente)`

Tabelle 12.3: Tabelle `Verwendung`

Eine Komponente kann in mehreren Produkten verwendet werden, und ein Produkt kann mehrere Komponenten enthalten (eine Viele-zu-viele-Beziehung). Diese Situation kann Probleme mit der Integrität hervorrufen. Dieses Problem können Sie dadurch umgehen, dass Sie die Verknüpfungstabelle `Verwendung` anlegen, die die Tabellen `Komponente` und `Artikel` verbindet. Eine Komponente kann in vielen Zeilen der Tabelle `Verwendung` vorkommen, aber jede dieser Zeilen referenziert nur eine Komponente (eine Eins-zu-viele-Beziehung). Analog dazu kann ein Produkt in vielen Zeilen der Tabelle `Verwendung` vorkommen, aber jede dieser Zeilen verweist nur auf ein Produkt (eine weitere Eins-zu-viele-Beziehung). Durch die Zwischentabelle `Verwendung` wird eine Viele-zu-viele-Beziehung, die Schwierigkeiten machen kann, in zwei Eins-zu-viele-Beziehungen aufgelöst. Dies ist ein Beispiel dafür, wie Komplexität durch Normalisierung reduziert werden kann.

Unterabfragen und das Schlüsselwort IN

Eine Form der verschachtelten Abfrage vergleicht einen einzelnen Wert mit einer Wertemenge, die von SELECT zurückgegeben wird. Dazu wird das Prädikat IN mit der folgenden Syntax benutzt:

```
SELECT Spaltenliste
   FROM Tabelle
   WHERE Ausdruck IN (Unterabfrage) ;
```

Der Ausdruck in der WHERE-Klausel ergibt den Wert TRUE, wenn *Ausdruck* in der Wertemenge enthalten ist, die von der Unterabfrage zurückgegeben wird. Die Spalten der aktuell verarbeiteten Tabellenzeile, die in der Abfrage festgelegt worden sind, werden dann der Ergebnistabelle hinzugefügt. Die Unterabfrage kann auf dieselbe Tabelle verweisen wie die äußere Abfrage oder auf eine andere.

Ich verwende im folgenden Beispiel die Datenbank von Zetec, um diese Art von Abfrage zu veranschaulichen. Stellen Sie sich vor, dass ein Mangel an Computerbildschirmen herrscht. Wenn Sie keine Monitore haben, können Sie die Produkte nicht mehr ausliefern, zu denen ein Monitor gehört. Wenn Sie wissen wollen, welche Produkte betroffen sind, geben Sie folgende Abfrage ein:

```
SELECT Modell
   FROM Verwendung
   WHERE KompID IN
      (SELECT KompID
          FROM Komponente
          WHERE KompTyp = 'Monitor') ;
```

Da SQL zuerst die innere Abfrage verarbeitet, ermittelt es zunächst in der Tabelle Komponente alle Zeilen mit einem Monitor als Komponententyp und liefert KompID für alle Zeilen zurück, bei denen KompTyp der Monitor ist. Das Ergebnis ist eine Liste mit den IDs aller Monitore. Die äußere Abfrage prüft dann den Wert von KompID in jeder Zeile der Tabelle Verwendung gegen diese Liste. Wenn der Vergleich erfolgreich ist, wird der Wert der Spalte Modell in die Ergebnistabelle der äußeren SELECT-Anweisung eingetragen. Das folgende Beispiel zeigt, was passiert, wenn Sie diese Abfrage laufen lassen:

```
MODELL
------
CX3000
CX3010
CX3020
MB3030
MX3020
MX3030
```

Jetzt wissen Sie, welche Produkte bald nicht mehr auf Lager sein werden, und können Ihren Kollegen im Verkauf auffordern, diese Produkte im Augenblick nicht mehr anzupreisen.

Bei dieser Art einer verschachtelten Abfrage darf in der Unterabfrage nur eine einzelne Spalte angegeben werden, deren Datentyp mit dem Datentyp des Arguments vor dem Schlüsselwort `IN` übereinstimmt.

Ich bin sicher, Sie erinnern sich an das KISS-Prinzip. »KISS« steht im Englischen für »Keep It Simple Stupid«, deutsch etwa »Mach es einfach, Dummerchen«. Aufgaben möglichst einfach zu lösen, ist beim Arbeiten mit jeder Art von Software wichtig. Aber die Forderung gilt besonders für Datenbank-Software. Anweisungen, die verschachtelte `SELECT`-Anweisungen enthalten, sind nicht leicht zu formulieren. Eine Möglichkeit, die richtige Form zu finden, besteht darin, die innere `SELECT`-Anweisung separat auszuführen und zu prüfen, ob sie das erwartete Ergebnis liefert. Funktioniert sie korrekt, können Sie sie in den äußeren Teil der Anweisung einfügen und haben so eine bessere Chance, dass die Anweisung das Gewünschte leistet.

Unterabfragen und das Schlüsselwort NOT IN

Eine Unterabfrage kann nicht nur mit dem Schlüsselwort `IN`, sondern auch mit `NOT IN` beginnen. Und tatsächlich ist es jetzt genau der richtige Zeitpunkt für die Geschäftsführung von Zetec, eine solche Abfrage ablaufen zu lassen. Mit der Abfrage im letzten Abschnitt hat das Management von Zetec erfahren, welche Produkte im Augenblick nicht verkauft werden sollten, aber davon kann man die Miete nicht bezahlen. Was das Management wirklich wissen muss, ist, welche Produkte verkauft werden können. Das Management möchte den Verkauf der Produkte forcieren, die keine Monitore enthalten. Eine verschachtelte Abfrage mit einer Unterabfrage, vor der das Schlüsselwort `NOT IN` steht, liefert die gewünschten Informationen:

```
SELECT Modell
   FROM Verwendung
   WHERE Modell NOT IN
      (SELECT KompID
          FROM Komponente
          WHERE KompTyp = 'Monitor') ;
```

Diese Abfrage liefert das folgende Ergebnis:

```
MODELL
------
PX3040
PB3050
PX3040
PB3050
```

In diesem Beispiel ist erwähnenswert, dass die Ergebnistabelle Duplikate enthält. Diese Verdopplung kommt dadurch zustande, dass es zu einem Produkt in der Tabelle `Verwendung` mehrere Komponenten geben kann, die nicht Monitore sind. Die Abfrage erzeugt für jede dieser Zeilen einen Eintrag in der Ergebnistabelle.

In unserem Beispiel stellt die Anzahl der Zeilen kein Problem dar, weil die Ergebnistabelle kurz ist. In der Praxis kann eine solche Ergebnistabelle jedoch Hunderte oder Tausende von Zeilen umfassen. Deshalb sollten Sie die Duplikate entfernen, um Verwirrung zu vermeiden. Mit dem Schlüsselwort `DISTINCT` können Sie diese Aufgabe einfach lösen. Damit werden nur solche Zeilen, die sich von allen vorher abgerufenen Zeilen unterscheiden, in die Ergebnistabelle eingefügt:

```
SELECT DISTINCT Modell
   FROM Verwendung
   WHERE Modell NOT IN
      (SELECT KompID
         FROM Komponente
         WHERE KompTyp = 'Monitor') ;
```

Das Ergebnis lautet jetzt:

```
MODELL
-------
PX3040
PB3050
```

Verschachtelte Abfragen, die einen einzelnen Wert zurückgeben

Es ist oft nützlich, eine Unterabfrage mit einem der sechs Vergleichsoperatoren (=, <>, <, <=, >, >=) einzuleiten. In diesem Fall ergibt der Ausdruck vor dem Operator einen einzelnen Wert, und die Unterabfrage hinter dem Operator muss naturgemäß ebenfalls einen einzelnen Wert zurückgeben. Eine Ausnahme bilden nur die *quantifizierenden Vergleichsoperatoren*, die aus einem Vergleichsoperator bestehen, dem ein Quantor (`ANY`, `SOME` oder `ALL`) folgt.

Als Beispiel einer Unterabfrage, die einen einzelnen Wert zurückgibt, wollen wir einen anderen Ausschnitt der Firmendatenbank von Zetec betrachten. Er enthält die Tabelle `Kunde` mit Informationen über die Firmen, die Zetec-Produkte kaufen, und die Tabelle `Kontakt` mit Informationen über die einzelnen Kontaktpersonen in diesen Kundenfirmen. Der Aufbau dieser Tabellen wird in Tabelle 12.4 und Tabelle 12.5 dargestellt.

Spalte	Typ	Einschränkung
KundID	INTEGER	PRIMARY KEY
Firma	CHAR(40)	
KundAdresse	CHAR(30)	
KundOrt	CHAR(20)	
KundStaat	CHAR(2)	
KundPlz	CHAR(10)	
KundTel	CHAR(12)	
ModEbene	INTEGER	

Tabelle 12.4: Tabelle `Kunde`

Spalte	Typ	Einschränkung
KundID	INTEGER	FOREIGN KEY
KontVorname	CHAR(10)	
KontNachname	CHAR(16)	
KontTel	CHAR(12)	
KontInfo	CHAR(50)	

Tabelle 12.5: Tabelle `Kontakt`

Wenn Sie die Kontaktdaten der Firma Verkaufsolymp sehen möchten, sich aber nicht an deren `KundID` erinnern, können Sie die Informationen mit einer verschachtelten Abfrage abrufen:

```
SELECT *
   FROM Kontakt
      WHERE KundID =
         (SELECT KundID
             FROM Kunde
                WHERE Firma = 'Verkaufsolymp') ;
```

Das Ergebnis könnte folgendermaßen aussehen:

```
KundID KontVorname KontNachname KontTel      KontInfo
------ ----------- ------------ ------------ -----------------
   118 Joseph      Alt          505-876-3456 Wird Hauptrolle
                                             beim Ausbau des
                                             funkbasierten
                                             Webs spielen.
```

Jetzt können Sie Joseph Alt von Verkaufsolymp anrufen und ihm das Monatssonderangebot von webfähigen Handys unterbreiten.

Wenn Sie eine Unterabfrage in einer Gleichheitsbedingung (=) verwenden, darf die `SELECT`-Liste der Unterabfrage nur eine einzelne Spalte (`KundID` in unserem Beispiel) benennen. Wenn die Unterabfrage ausgeführt wird, darf sie nur eine einzige Zeile zurückgeben, um für den Vergleich nur einen einzigen Wert zu haben.

In diesem Beispiel gehe ich davon aus, dass die Tabelle `Kunde` nur eine Zeile mit dem Wert '`Verkaufsolymp`' in der Spalte `Firma` enthält. Wenn Sie bei der Definition der Tabelle `Firma` der Spalte `Kunde` eine Einschränkung vom Typ `UNIQUE` zuweisen, ist dafür gesorgt, dass die Unterabfrage in dem vorangegangenen Beispiel nur einen einzigen (oder keinen) Wert zurückgibt. Häufig werden Unterabfragen wie in unserem Beispiel jedoch für Spalten benutzt, für die keine `UNIQUE`-Einschränkung angegeben wurde. In solchen Fällen müssen Sie sich auf altes Wissen über den Inhalt der Datenbank verlassen und darauf vertrauen, dass der Spalteneintrag nicht doppelt vorkommt.

Falls die Tabelle Kunde mehr als einen Kunden mit dem Namen 'Verkaufsolymp' (möglicherweise in einem anderen Land) enthält, meldet die Unterabfrage einen Fehler.

Wenn die Tabelle Kunde dagegen keinen Kunden mit diesem Namen enthält, gibt die Unterabfrage einen Nullwert zurück und der Vergleichswert wird UNKNOWN (unbekannt). In diesem Fall gibt die WHERE-Klausel keine Zeile zurück. (WHERE gibt nur Zeilen zurück, für die die Bedingung wahr ist, alle anderen Zeilen werden gefiltert.) Wahrscheinlich würde dies passieren, wenn Sie den Firmennamen aus Versehen falsch schreiben, zum Beispiel 'Verkaufsolump'.

Obwohl der Gleichheitsoperator (=) derjenige ist, der wohl am häufigsten eingesetzt wird, können Sie die anderen fünf Vergleichsoperatoren auf ähnliche Weise verwenden. Der Einzelwert, den die Unterabfrage zurückgibt, wird in die Bedingung der WHERE-Klausel eingesetzt und gemäß dem angegebenen Operator getestet. Wenn die Bedingung den Wert TRUE ergibt, wird die betreffende Zeile in die Ergebnistabelle aufgenommen.

Sie können garantieren, dass die Unterabfrage einen einzelnen Wert zurückgibt, wenn Sie eine Aggregatfunktion in der Unterabfrage verwenden. *Aggregatfunktionen* geben immer einen einzelnen Wert zurück. (Aggregatfunktionen werden in Kapitel 3 beschrieben.) Natürlich ist diese Art des Arbeitens nur sinnvoll, wenn Sie das Ergebnis einer Aggregatfunktion benötigen.

Angenommen, Sie arbeiten als Verkäufer bei Zetec. Weil privat einige Rechnungen fällig werden, müssen Sie in dieser Woche eine besonders hohe Provision verdienen. Sie glauben, dass Sie dies am ehesten erreichen, wenn Sie sich auf den Verkauf des teuersten Produkts von Zetec konzentrieren. Um herauszufinden, welches Produkt den höchsten Preis hat, benutzen Sie die folgende verschachtelte Abfrage:

```
SELECT Modell, ArtName, ListenPreis
   FROM Artikel
      WHERE ListenPreis =
         (SELECT MAX(ListenPreis)
             FROM Artikel) ;
```

Dieses Beispiel zeigt eine verschachtelte Abfrage, in der sowohl die Unterabfrage als auch die einschließende Anweisung auf dieselbe Tabelle zugreifen. Die Unterabfrage gibt einen einzelnen Wert zurück: den höchsten Listenpreis in der Tabelle Artikel. Die äußere Abfrage ermittelt alle Zeilen der Tabelle Artikel, die diesen Listenpreis haben.

Das nächste Beispiel zeigt eine vergleichende Unterabfrage, die einen anderen Vergleichsoperator als »gleich« (=) benutzt:

```
SELECT Modell, ArtName, ListenPreis
   FROM Artikel
      WHERE ListenPreis <
         (SELECT AVG(ListenPreis)
             FROM Artikel) ;
```

Die Unterabfrage gibt einen einzelnen Wert zurück: den durchschnittlichen Listenpreis in der Tabelle Artikel. Die äußere Abfrage ermittelt alle Zeilen der Tabelle Artikel mit einem Listenpreis, der kleiner als der durchschnittliche Listenpreis ist.

Im Original-SQL-Standard durfte ein Vergleich nur eine Unterabfrage enthalten, die auf der rechten Seite des Vergleichs stehen musste. In SQL:1999 konnten beide Operanden eines Vergleichs aus Unterabfragen bestehen. Spätere Versionen von SQL haben diese Möglichkeit beibehalten.

Die quantifizierenden Vergleichsoperatoren ALL, SOME und ANY

Eine weitere Methode, um dafür zu sorgen, dass eine Unterabfrage einen einzelnen Wert zurückgibt, besteht darin, mit einem quantifizierenden Vergleichsoperator (einem sogenannten *Quantor*) zu arbeiten. Der Allquantor `ALL` und die Existenzquantoren `SOME` und `ANY` reduzieren in Verbindung mit einem Vergleichsoperator die Liste, die von einer Unterabfrage zurückgegeben wird, auf einen einzigen Wert.

Die Wirkung dieser Quantoren lässt sich am besten anhand eines Beispiels zeigen. Ich möchte noch einmal auf das Bundesligabeispiel von Kapitel 11 zurückkommen.

Der Inhalt der beiden Tabellen wird durch die folgenden beiden Abfragen erzeugt:

```
SELECT * FROM ErsteLiga

Vorname     Nachname   Tore
---------   --------   ----
Franz       Becker       11
Helmut      Langer        9
Toni        Doppler      13
Helmut      Berthold     12
Gerd        Müller        8

SELECT * FROM ZweiteLiga

Vorname     Nachname   Tore
---------   --------   ----
Paul        Peters       12
Helmut      Schmitz      10
Gerd        Müller        8
Arnold      Schwarzer    14
```

Wenn Sie die Stürmer der Zweiten Bundesliga herausfinden wollen, die mehr Tore geschossen haben als die besten Stürmer der Ersten Liga, können Sie folgende Abfrage eingeben:

```
SELECT *
   FROM ZweiteLiga
   WHERE Tore > ALL
      (SELECT Tore FROM ErsteLiga) ;
```

Dies ist das Ergebnis:

```
Vorname     Nachname   Tore
--------    ---------  ----
Arnold      Schwarzer    14
```

Die Unterabfrage (`SELECT Tore FROM ErsteLiga`) gibt die Werte aller Spieler der Ersten Liga zurück, die in der Spalte `Tore` stehen. Der quantifizierende Vergleichsoperator `> ALL` gibt nur die Werte der Spalte `Tore` in der Tabelle `ZweiteLiga` zurück, die größer sind als die Werte, die die Unterabfrage liefert. Umgangssprachlich lautet die Bedingung: »Größer als der größte Wert der Unterabfrage.« In diesem Fall ist der größte Wert der Unterabfrage 13 (Toni Doppler). Die einzige Zeile in der Zweiten Liga mit einem größeren Wert ist die von Arnold Schwarzer mit 14 Toren.

Was passiert, wenn es in der Zweiten Liga keinen Spieler gibt, der mehr Tore als irgendein Spieler der Ersten Liga geschossen hat? In diesem Fall gibt die Abfrage

```
SELECT *
   FROM ZweiteLiga
   WHERE Tore > ALL
      (SELECT Tore FROM ErsteLiga) ;
```

eine Warnmeldung zurück, die besagt, dass keine Zeile die Bedingungen der Abfrage erfüllt. Das heißt, dass es in der Zweiten Liga keinen Spieler gibt, der mehr Tore als ein Spieler der Ersten Liga geschossen hat.

Verschachtelte Abfragen als Existenztest

Eine Abfrage gibt die Daten aller Zeilen einer Tabelle zurück, die die Bedingungen der Abfrage erfüllen. Manchmal werden viele Zeilen zurückgegeben, manchmal nur eine. Ab und an kommt es vor, dass keine Zeile die Bedingungen der Abfrage erfüllt und somit auch nichts zurückgegeben wird. Sie können die Prädikate `EXISTS` und `NOT EXISTS` verwenden, um eine Unterabfrage zu beginnen. Diese Struktur sagt Ihnen, ob es in der Tabelle, die vom `FROM` der Unterabfrage angesprochen wird, überhaupt Zeilen gibt, die den Bedingungen der `WHERE`-Klausel entsprechen.

Unterabfragen mit `EXISTS` und `NOT EXISTS` unterscheiden sich grundsätzlich von den übrigen Unterabfragen, die bis jetzt in diesem Kapitel behandelt worden sind. In allen vorangegangenen Fällen hat SQL erst die Unterabfrage ausgeführt und dann das Ergebnis dieser Operation an den einschließenden Befehl übergeben. Unterabfragen mit den Prädikaten `EXISTS` und `NOT EXISTS` gehören dagegen zu den sogenannten korrelierten Unterabfragen.

Eine *korrelierte Unterabfrage* ermittelt zunächst Tabelle und Zeile, die durch die einschließende Anweisung angegeben werden, und führt dann die Unterabfrage für die Zeile in der Tabelle der Unterabfrage durch, die mit der aktuellen Zeile der Tabelle der einschließenden Anweisung in Beziehung steht (korreliert).

Die Unterabfrage gibt entweder keine, eine oder mehrere Zeilen zurück. Wenn sie wenigstens eine Zeile zurückgibt, hat das Prädikat `EXISTS` den Wert `TRUE`, und die einschließende Anweisung führt ihre Aktion aus. Wenn eine Zeile der Tabelle der einschließenden Anweisung bearbeitet worden ist, wird der Vorgang auf der nächsten Tabellenzeile ausgeführt. Diese Schritte werden so lange wiederholt, bis alle Datensätze in der Tabelle der einschließenden Anweisung abgearbeitet sind.

EXISTS

Angenommen, Sie sind Verkäufer bei der Firma Zetec und möchten die Kontaktpersonen in den schwedischen Kundenfirmen anrufen. Sie versuchen es mit folgender Abfrage:

```
SELECT *
   FROM Kontakt
   WHERE EXISTS
      (SELECT *
          FROM Kunde
          WHERE KundStaat = 'SE'
             AND Kontakt.KundID = Kunde.KundID) ;
```

Achten Sie auf die Verwendung von `Kontakt.KundID`. Dieser Parameter verweist auf eine Spalte der äußeren Abfrage und vergleicht sie mit einer Spalte der inneren Abfrage, `Kunde.KundID`. Für jede geprüfte Zeile der äußeren Abfrage führen Sie die innere Abfrage aus, wobei die dabei zu prüfenden Zeilen der inneren Abfrage durch den Wert von `KundID` des aktuellen `Kontakt`-Datensatzes der äußeren Abfrage vorgegeben werden.

Folgendes passiert:

1. Die Spalte `KundID` verknüpft die Tabelle `Kontakt` mit der Tabelle `Kunde`.
2. SQL untersucht den ersten Datensatz in der Tabelle `Kontakt`, findet die Zeile in der Tabelle `Kunde`, die dieselbe `KundID` hat, und prüft das Feld `KundStaat` dieser Zeile.
3. Wenn `Kunde.KundStaat = 'SE'`, wird die aktuelle `Kontakt`-Zeile zur Ergebnistabelle hinzugefügt.
4. Die nächste `Kontakt`-Zeile wird auf die gleiche Weise verarbeitet und so weiter, bis die gesamte Tabelle `Kontakt` abgearbeitet worden ist.
5. Da die Abfrage mit `SELECT * FROM Kontakt` arbeitet, werden alle Felder der Tabelle `Kontakt` (einschließlich des Namens und der Telefonnummer des Kontakts) zurückgegeben.

NOT EXISTS

Im letzten Beispiel wollte der Zetec-Verkäufer die Kontaktpersonen der Firmenkunden in Schweden wissen. Angenommen, ein zweiter Verkäufer ist für alle Kunden in Europa außer denen in Schweden zuständig. Diese Kunden können mit einer Abfrage herausgefunden werden, die der vorherigen Abfrage ziemlich gleicht, nur dass sie das Prädikat `NOT EXISTS` verwendet:

```
SELECT *
   FROM Kontakt
   WHERE NOT EXISTS
      (SELECT *
          FROM Kunde
          WHERE KundStaat = 'SE'
             AND Kontakt.KundID = Kunde.KundID) ;
```

Jeder Datensatz in `Kontakt`, für den die Unterabfrage keine entsprechende Zeile liefert, wird in die Ergebnistabelle aufgenommen.

Weitere korrelierte Unterabfragen

Wie bereits weiter vorn in diesem Kapitel erwähnt wurde, müssen Unterabfragen, die durch `IN` oder einen Vergleichsoperator eingeleitet werden, zwar keine korrelierten Abfragen sein, sie können es aber.

Korrelierte Unterabfragen, die durch IN eingeleitet werden

Der Abschnitt *Unterabfragen und das Schlüsselwort IN* weiter vorn in diesem Kapitel beschreibt, wie eine nicht korrelierte Unterabfrage mit dem Prädikat `IN` verwendet werden kann. Um zu zeigen, wie eine korrelierte Unterabfrage das Prädikat `IN` benutzt, wollen wir dieselbe Frage stellen, die wir bereits mit dem `EXISTS`-Prädikat beantwortet haben: »Wie lauten die Namen und Telefonnummern der Kontaktpersonen aller Zetec-Kunden in Schweden?« Sie können diese Frage mit einer korrelierten `IN`-Unterabfrage beantworten:

```
SELECT *
   FROM Kontakt
   WHERE 'SE' IN
      (SELECT KundStaat
          FROM Kunde
          WHERE Kontakt.KundID = Kunde.KundID) ;
```

Die Anweisung wird für jede Zeile der Tabelle `Kontakt` ausgewertet. Wenn die Zeile `Kontakt.KundID` mit `Kunde.KundID` übereinstimmt, wird der Wert von `Kunde.KundStaat` mit `'SE'` verglichen. Das Ergebnis der Unterabfrage ist eine Liste, die höchstens ein Element enthält. Falls dieses Element `'SE'` lautet, ist die `WHERE`-Klausel der einschließenden Anweisung erfüllt, und der Datensatz wird zur Ergebnistabelle der Abfrage hinzugefügt.

Unterabfragen, die durch Vergleichsoperatoren eingeleitet werden

Eine korrelierte Unterabfrage kann auch durch einen der sechs Vergleichsoperatoren eingeleitet werden, wie das nächste Beispiel zeigt.

Die Firma Zetec zahlt ihren Verkäufern monatlich einen Bonus, der von ihrem Gesamtumsatz in dem entsprechenden Monat abhängt. Je höher der Umsatz ist, desto größer ist der Bonus. Die Bonusprozentsätze sind in der Tabelle `BonusRate` abgelegt:

```
MinBetrag        MaxBetrag      BonusPzt
-----------      ----------     ---------
      0.00         24999.99        0.
  25000.00         49999.99        0.001
  50000.00         99999.99        0.002
 100000.00        249999.99        0.003
 250000.00        499999.99        0.004
 500000.00        749999.99        0.005
 750000.00        999999.99        0.006
```

Wenn ein Verkäufer einen Monatsumsatz zwischen 100.000,00 Euro und 249.999,99 Euro tätigt, beträgt sein Bonus 0,3 Prozent des Umsatzes.

Die Verkäufe werden in einer Tabelle mit dem Namen `Transmaster` gespeichert, die als eine Art Haupttabelle für Transaktionen dient:

```
Transmaster
-----------
Spalte         Type             Constraints
---------      -------          -----------
TransID        INTEGER          PRIMARY KEY
KundID         INTEGER          FOREIGN KEY
MitID          INTEGER          FOREIGN KEY
TransDatum     DATE
NetBetrag      NUMERIC
Fracht         NUMERIC
Steuer         NUMERIC
Brutto         NUMERIC
```

Die Boni werden auf Basis des Feldes `NetBetrag` aller Transaktionen eines Verkäufers im jeweiligen Monat berechnet. Sie können die Bonusrate einer bestimmten Person mit einer korrelierten Unterabfrage ermitteln, die mit Vergleichsoperatoren arbeitet:

```
SELECT BonusPzt
   FROM BonusRate
      WHERE MinBetrag <=
         (SELECT SUM (NetBetrag)
             FROM Transmaster
                WHERE MitID = 133)
         AND MaxBetrag >=
            (SELECT SUM (NetBetrag)
                FROM Transmaster
                   WHERE MitID = 133) ;
```

Interessant sind an dieser Abfrage die beiden Unterabfragen, die mit einem logischen `AND` verknüpft sind. Die Unterabfragen benutzen die Aggregatfunktion `SUM`, die einen einzelnen Wert zurückgibt, nämlich den Gesamtumsatz des Mitarbeiters mit der Mitarbeiternummer 133.

Dieser Wert wird dann mit den Spalten MinBetrag und MaxBetrag der Tabelle BonusRate verglichen, um den Bonus dieses Mitarbeiters zu ermitteln.

Wenn Sie nicht die MitID, sondern nur den Namen des Verkäufers kennen, wird die Abfrage etwas komplexer:

```
SELECT BonusPzt
   FROM BonusRate
      WHERE MinBetrag <=
         (SELECT SUM (NetBetrag)
             FROM Transmaster
                WHERE MitID =
                   (SELECT MitID
                       FROM Mitarbeiter
                          WHERE MitNachname = 'Coffin'))
         AND Max_Betrag >=
            (SELECT SUM (NetBetrag)
                FROM Transmaster
                   WHERE MitID =
                      (SELECT MitID
                          FROM Mitarbeiter
                             WHERE MitNachname = 'Coffin'));
```

Dieses Beispiel enthält Unterabfragen, die in die Unterabfragen eingebettet sind, die wiederum in eine einschließende Abfrage eingebettet sind, um den Bonus des Verkäufers mit dem Namen Coffin zu ermitteln. Diese Struktur funktioniert nur, wenn Sie mit Sicherheit wissen, dass die Firma einen einzigen Mitarbeiter mit dem Nachnamen Coffin beschäftigt. Wenn Sie wissen, dass mehr als ein Mitarbeiter Coffin heißt, müssen Sie die einzelnen Personen durch eine WHERE-Klausel in der innersten Unterabfrage so unterscheiden, dass nur eine Zeile der Tabelle Mitarbeiter ausgewählt wird.

Unterabfragen in einer HAVING-Klausel

Sie können eine korrelierte Unterabfrage in einer HAVING-Klausel genau wie in einer WHERE-Klausel verwenden. Wie ich in Kapitel 10 ausführe, steht vor einer HAVING-Klausel normalerweise eine GROUP BY-Klausel. Die HAVING-Klausel dient als Filter für die Gruppen, die durch die GROUP BY-Klausel gebildet werden. Gruppen, die die Bedingung der HAVING-Klausel nicht erfüllen, werden nicht in das Ergebnis aufgenommen. Auf diese Art und Weise wird die HAVING-Klausel für jede Gruppe ausgewertet, die von der GROUP BY-Klausel erzeugt wird.

Wenn die Anweisung keine GROUP BY-Klausel enthält, wird die HAVING-Klausel für die Menge der Zeilen ausgewertet, die von der WHERE-Klausel übergeben wird. Diese Menge wird wie eine einzelne Gruppe behandelt. Wenn es in der Anweisung weder eine WHERE-Klausel noch eine GROUP BY-Klausel gibt, wird die HAVING-Klausel für die gesamte Tabelle ausgewertet:

```
SELECT TM1.MitID
   FROM Transmaster TM1
      GROUP BY TM1.MitID
      HAVING MAX (TM1.NetBetrag) >= ALL
         (SELECT 2 * AVG (TM2.NetBetrag)
             FROM Transmaster TM2
             WHERE TM1.MitID <> TM2.MitID) ;
```

Diese Abfrage benutzt zwei Aliasse für dieselbe Tabelle. Sie ermittelt die Mitarbeiterkennungen (`MitID`) aller Verkäufer, deren Umsatz wenigstens doppelt so groß ist wie der Durchschnittsumsatz aller anderen Verkäufer. Die Abfrage funktioniert folgendermaßen:

1. Die äußere Abfrage gruppiert die Zeilen von `Transmaster` mit den Klauseln `SELECT`, `FROM` und `GROUP BY` nach der `MitID`.
2. Die `HAVING`-Klausel filtert diese Gruppen. Für jede Gruppe berechnet sie das Maximum (`MAX`) der Spalte `NetBetrag` aller Datensätze dieser Gruppe.
3. Die innere Abfrage ermittelt den doppelten Durchschnittsbetrag der Spalte `NetBetrag` aller Zeilen von `Transmaster`, deren `MitID` sich von der `MitID` der aktuellen Gruppe der äußeren Abfrage unterscheidet.

Achten Sie darauf, dass Sie in der letzten Zeile auf zwei verschiedene `MitID`-Werte verweisen müssen, weshalb die `FROM`-Klauseln der äußeren und inneren Abfragen mit verschiedenen Aliassen für `Transmaster` arbeiten.

4. Sie benutzen dann diese beiden Aliasse in dem Vergleich in der letzten Zeile der Abfrage, um anzuzeigen, dass Sie sowohl auf `MitID` der aktuellen Zeile der inneren Unterabfrage (`TM2.MitID`) als auch auf `MitID` der aktuellen Gruppe der äußeren Unterabfrage (`TM1.MitID`) verweisen.

Die Anweisungen UPDATE, DELETE und INSERT

Zusätzlich zu den Anweisungen vom Typ `SELECT` können auch die Anweisungen `UPDATE`, `DELETE` und `INSERT` Klauseln vom Typ `WHERE` enthalten. Diese `WHERE`-Klauseln können auf dieselbe Art und Weise Unterabfragen enthalten, wie das bei `WHERE`-Klauseln in `SELECT`-Befehlen der Fall ist.

Zetec hat beispielsweise gerade einen größeren Abnahmevertrag mit der Firma Verkaufsolymp unterzeichnet und möchte dem Unternehmen rückwirkend einen zehnprozentigen Rabatt auf alle Käufe des letzten Monats gewähren. Sie können die Gutschrift mit einer `UPDATE`-Anweisung in die Tabelle eintragen:

```
UPDATE Transmaster
   SET NetBetrag = NetBetrag * 0.9
   WHERE KundID =
      (SELECT KundID
          FROM Kunde
          WHERE Firma = 'Verkaufsolymp') ;
```

Sie können in einer UPDATE-Anweisung auch eine korrelierte Unterabfrage benutzen. Angenommen, die Tabelle Kunde enthält eine Spalte mit dem Namen LetztMonatMax und Zetec möchte rückwirkend auf alle Käufe, die das Maximum des letzten Monats eines Kunden überschreiten, denselben Rabatt gewähren:

```
UPDATE Transmaster TM
   SET NetBetrag = NetBetrag * 0.9
   WHERE NetBetrag >
      (SELECT LetztMonatMax
          FROM Kunde K
          WHERE K.KundID = TM.KundID) ;
```

Beachten Sie, dass diese Unterabfrage korreliert ist: Die WHERE-Klausel in der letzten Zeile verweist sowohl auf KundID der Zeile Kunde der Unterabfrage als auch auf KundID der aktuellen Transmaster-Zeile, die Kandidat für UPDATE ist.

Eine Unterabfrage in einer UPDATE-Anweisung kann auch auf die Tabelle verweisen, die aktualisiert werden soll. Angenommen, Zetec möchte den Kunden, deren Umsätze 10.000 Euro überschreiten, einen zehnprozentigen Rabatt gewähren:

```
UPDATE Transmaster TM1
   SET NetBetrag = NetBetrag * 0.9
   WHERE 10000 < (SELECT SUM(NetBetrag)
      FROM Transmaster TM2
      WHERE TM1.KundID = TM2.KundID);
```

Die innere Unterabfrage berechnet die Summe der Spalte NetBetrag aller Transmaster-Zeilen eines Kunden. Aber was bedeutet das? Stellen Sie sich vor, dass es für den Kunden mit KundID = 37 in der Tabelle Transmaster vier Zeilen gibt, die folgende Werte für Net Betrag haben: 3000, 5000, 2000 und 1000. Die Summe von NetBetrag dieser KundID beträgt 11.000.

Die Reihenfolge, in der die UPDATE-Anweisung die Zeilen verarbeitet, ist abhängig von Ihrer Implementierung und im Allgemeinen nicht vorhersagbar. Die Reihenfolge kann von Tag zu Tag gleich sein oder sich mit der physischen Speicherung auf der Festplatte ändern. Angenommen, Ihre Implementierung verarbeitet die Zeilen für diese KundID in der folgenden Reihenfolge: erst die Transmaster-Zeile mit NetBetrag von 3000; dann die mit NetBetrag von 5000 und so weiter. Nachdem die ersten drei Zeilen für KundID 37 geändert worden sind, betragen die Werte von NetBetrag noch 2700 (90 Prozent von 3000), 4500 (90 Prozent von 5000) und 1800 (90 Prozent von 2000). Es muss jetzt noch die vierte Zeile dieses Kunden mit einem alten NetBetrag von 1000 an den neuen Wert angepasst werden. Das Problem besteht nun darin, dass die Anweisung die Aufgabe hat, Werte zu aktualisieren, wenn die Gesamtsumme aller Spalten NetBetrag eines bestimmten Kunden größer als 10000 ist. Die Summe der bereits aktualisierten Spalten plus der noch nicht aktualisierten Spalte ergibt nun aber genau den Betrag von 10000, was dazu führt, dass die ursprüngliche Bedingung der SELECT-Anweisung nicht mehr erfüllt ist und die letzte Spalte nicht mehr neu berechnet wird. Glücklicherweise ist es jetzt aber so, dass man die letzten Aussagen im Konjunktiv hätte

schreiben müssen, weil eine UPDATE-Anweisung sich so nicht definiert, wenn eine Unterabfrage auf eine Tabelle verweist, die aktualisiert werden soll.

Alle Versionen von Unterabfragen, die Sie in einer UPDATE-Anweisung einsetzen, beziehen sich immer auf die ursprünglichen Werte der Tabelle, die aktualisiert werden soll. Im letzten Beispiel gibt die Unterabfrage immer `11000` für `KundID 37` zurück – die ursprüngliche Summe.

Die Unterabfrage in einer WHERE-Klausel funktioniert genauso wie die in einer SELECT- oder UPDATE-Anweisung. Dasselbe gilt für DELETE und INSERT. Um alle Transaktionen von Verkaufsolymp zu löschen, können Sie diese Anweisung eingeben:

```
DELETE Transmaster
   WHERE KundID =
      (SELECT KundID
         FROM Kunde
         WHERE Firma = 'Verkaufsolymp') ;
```

Wie bei UPDATE können auch DELETE-Unterabfragen korreliert sein und ebenfalls auf die Tabellen verweisen, deren Datensätze gelöscht werden sollen. Die Regeln ähneln den Regeln für UPDATE-Unterabfragen. Angenommen, Sie möchten alle `Transmaster`-Zeilen für Kunden löschen, deren gesamter `NetBetrag` größer als 10.000 Euro ist:

```
DELETE Transmaster TM1
   WHERE 10000 < (SELECT SUM(NetBetrag)
      FROM Transmaster TM2
          WHERE TM1.KundID = TM2.KundID) ;
```

Diese Abfrage löscht sowohl alle Zeilen aus `Transmaster`, die `KundID 37` haben, als auch alle anderen Kunden, deren Gesamtumsatz größer als 10.000 Euro ist. Alle Verweise auf `Transmaster` in der Unterabfrage beziehen sich auf den Inhalt von `Transmaster`, bevor die aktuelle Anweisung irgendeine Zeile gelöscht hat. Deshalb arbeitet, selbst wenn Sie die letzte Zeile von `Transmaster` für `KundID 37` löschen, die Unterabfrage immer mit den ursprünglichen Daten von `Transmaster` und gibt `11000` zurück.

Immer wenn Sie in einer Datenbank Datensätze einfügen, ändern oder löschen, besteht die Gefahr, dass Sie die Daten einer Tabelle so ändern, dass sie mit anderen Tabellen in der Datenbank nicht mehr konsistent sind. Eine solche Inkonsistenz wird als *Änderungsanomalie* bezeichnet, ein Problem, das ich in Kapitel 5 beschreibe. Wenn Sie Datensätze in `Transmaster` löschen und über eine weitere Tabelle `Transdetail` verfügen, die von `Transmaster` abhängt, müssen Sie auch die entsprechenden Datensätze in `Transdetail` löschen. Diese Operation wird als *kaskadierendes Löschen* bezeichnet, weil das Löschen eines Eltern-Datensatzes wie in einer Kaskade auf alle abhängigen untergeordneten Datensätze (Kind-Datensätze) ausgeweitet werden muss. Andernfalls werden die nicht gelöschten Kind-Datensätze zu Waisen. In diesem Fall würde es sich bei ihnen um detaillierte Rechnungszeilen handeln, die in der Luft hängen, weil es keinen passenden Rechnungsdatensatz mehr gibt.

Wenn Ihre SQL-Implementierung kaskadierendes Löschen nicht unterstützt, müssen Sie die Daten selbst löschen. In diesem Fall sollten Sie zunächst die betreffenden Datensätze in den Kind-Tabellen und dann erst den zugehörigen Datensatz in der übergeordneten Tabelle löschen. So existieren zu keinem Zeitpunkt verwaiste Datensätze in der Kind-Tabelle.

Änderungen per pipelined DML abrufen

Im vorhergehenden Abschnitt zeige ich, wie eine `UPDATE`-, `DELETE`- oder `INSERT`-Anweisung in einer `WHERE`-Klausel eine verschachtelte `SELECT`-Anweisung enthalten kann. In SQL:2011 wurde eine verwandte Fähigkeit eingeführt, mit der eine Datenmaipulationsanweisung (wie `UPDATE`, `INSERT`, `DELETE` oder `MERGE`) in eine `SELECT`-Anweisung eingebettet werden kann. Diese Fähigkeit wird als *pipelined DML* bezeichnet.

Eine Möglichkeit, Datenänderungen mit einer `DELETE`-, `INSERT`- oder `UPDATE`-Operation zu betrachten, besteht darin, sich zwei Tabellen vorzustellen: die alte Tabelle vor der Änderung und die neue Tabelle nach der Änderung. Während der Änderungsoperation werden Hilfstabellen, sogenannte *Deltatabellen*, erstellt. Eine `DELETE`-Operation erstellt eine alte Deltatabelle mit den zu löschenden Zeilen. Eine `INSERT`-Operation erstellt eine neue Deltatabelle mit den einzufügenden Zeilen. Eine `UPDATE`-Operation erstellt sowohl eine alte als auch eine neue Deltatabelle, die alte mit den zu ersetzenden Zeilen und die neue mit den ersetzenden Zeilen.

Mit pipelined DML können Sie die Daten aus den Deltatabellen abrufen. Angenommen, Sie wollten aus einer `Produkt`-Tabelle alle Produkte mit einer `ProduktID` zwischen 1000 und 1399 löschen und festhalten, welche Produkte in diesem Bereich genau gelöscht worden sind. Sie können diese Aufgabe mit folgendem Code lösen:

```
SELECT AlteTabelle.ProduktID
   FROM OLD TABLE (DELETE FROM Produkt
                   WHERE ProduktID BETWEEN 1000 AND 1399)
      AS AlteTabelle ;
```

In diesem Beispiel spezifizieren die Schlüsselwörter `OLD TABLE`, dass das Ergebnis der `SELECT`-Anweisung aus der alten Deltatabelle stammt. Das Ergebnis ist die Liste der `ProduktID`-Nummern der Produkte, die gelöscht werden.

Ähnlich könnten Sie mit den Schlüsselwörtern `NEW TABLE` eine Liste aus der neuen Deltatabelle abrufen, die die `Produkt ID`-Nummern der Zeilen enthält, die mit einer `INSERT`-Operation eingefügt oder einer `UPDATE`-Operation aktualisiert werden. Weil eine `UPDATE`-Operation sowohl eine alte als auch eine neue Deltatabelle erzeugt, können Sie den Inhalt jeder der beiden Tabellen per pipelined DML abrufen.

Rekursive Abfragen

13

In diesem Kapitel

- Die rekursive Verarbeitung verstehen
- Rekursive Abfragen definieren
- Mit rekursiven Abfragen arbeiten

Einer der Hauptkritikpunkte an SQL bestand bis zur Version SQL-92 darin, dass SQL keine *rekursive Verarbeitung* ermöglichte. Viele wichtige Probleme, die mit anderen Methoden schwer zu lösen sind, lassen sich rekursiv leicht bewältigen. Erweiterungen, die in SQL:1999 enthalten sind, ermöglichen rekursive Abfragen und machen die Sprache damit erheblich leistungsfähiger. Falls Ihre Implementierung von SQL über die Rekursionserweiterungen verfügt, können Sie eine große Kategorie von Problemen effizient lösen. Da die Rekursion jedoch nicht zum Kern von SQL gehört, ist diese Funktion in vielen aktuellen Implementierungen nicht enthalten.

Was ist Rekursion?

Die Rekursion ist eine Funktionalität, die in vielen Programmiersprachen, wie beispielsweise Logo, LISP oder C++, schon seit Jahren vorhanden ist. In diesen Sprachen können Sie eine *Funktion* (einen Block aus einem oder mehreren zusammengehörigen Befehlen) definieren, die eine spezielle Operation ausführt. Das Hauptprogramm aktiviert die Funktion, indem es einen sogenannten *Funktionsaufruf* ausführt. Wenn sich die Funktion selbst als Teil ihrer Operation aufruft, liegt die einfachste Form einer Rekursion vor.

Ein einfaches Programm, das in einer seiner Funktionen die Rekursion verwendet, soll die Vorteile und Fallen von Rekursionen aufzeigen. Das folgende Programm, das in C++ geschrieben ist, zeichnet auf dem Computerbildschirm eine Spirale. Es geht davon aus, dass die Zeichnung anfangs auf den oberen Rand des Bildschirms weist. Das Programm enthält drei Funktionen:

- ✔ Die Funktion `line(`*n*`)` zeichnet eine Linie, die *n* Einheiten lang ist.
- ✔ Die Funktion `left_turn(`*d*`)` dreht die Zeichnung um *d* Grad gegen den Uhrzeigersinn.
- ✔ Die Funktion `spiral(segment)` kann wie folgt definiert werden:

```
void spiral(int segment)
  {
   line(segment)
left_turn(90)
     spiral(segment + 1)
} ;
```

Wenn Sie `spiral(1)` aus dem Hauptprogramm heraus aufrufen, werden die folgenden Aktionen ausgeführt:

`spiral(1)` zeichnet, von der Mitte ausgehend eine vertikale Linie in Richtung des oberen Bildschirmrandes. Die Linie ist eine Einheit lang.

`spiral(1)` dreht um 90 Grad nach links.

`spiral(1)` ruft `spiral(2)` auf.

`spiral(2)` zeichnet eine Linie auf den linken Bildschirmrand zu, die zwei Einheiten lang ist.

`spiral(2)` dreht um 90 Grad nach links.

`spiral(2)` ruft `spiral(3)` auf.

Und so weiter …

Auf diese Weise erzeugt das Programm eine Spiralkurve (siehe Abbildung 13.1).

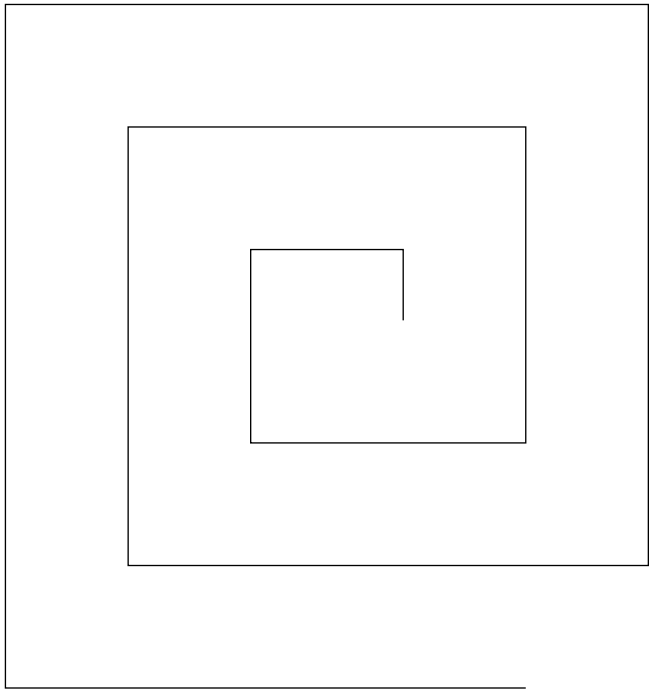


Abbildung 13.1: Das Ergebnis des Aufrufs von `spiral(1)`

Houston, wir haben ein Problem

Na ja, die Situation ist nicht ganz so ernst, wie sie für Apollo 13 war, als im Weltraum auf dem Weg zum Mond der Haupttank für den Sauerstoff explodierte. Ihr Problem besteht darin, dass das Programm nicht aufhört, sich selbst aufzurufen, und immer längere Linien zeichnet. Es fährt damit so lange fort, bis die Ressourcen des Computers erschöpft sind und er – wenn Sie Glück haben – eine dieser unsäglichen Fehlermeldungen bringt. Wenn Sie Pech haben, stürzt Ihr Computer einfach ab.

Scheitern ist keine Option

Dieses Szenario zeigt eine der Gefahren von Rekursionen. Ein Programm, das so geschrieben ist, dass es sich selbst aufruft, ruft eine neue Instanz von sich selbst auf – die ihrerseits eine weitere Instanz aufruft, *ad infinitum*. Das dürfte kaum das sein, was Sie sich wünschen.

Um dieses Problem zu lösen, enthalten rekursive Funktionen eine *Abbruchbedingung*, die begrenzt, wie tief die Rekursion gehen kann, damit das Programm die gewünschte Aktion ausführt und sich dann selbst vernünftig beendet. Sie können eine solche Abbruchbedingung in das Programm zum Zeichnen einer Spirale einfügen, um Computerressourcen zu sparen und Benutzer vor dem Wahnsinn zu bewahren:

```
void spiral2(int segment)
{
   if (segment <= 10)
   {
      line(segment)
      left_turn(90)
      spiral2(segment + 1)
   }
} ;
```

Wenn Sie `spiral2(1)` aufrufen, wird die Funktion ausgeführt und ruft sich dann (rekursiv) selbst auf, bis der Wert von `segment` größer als zehn ist. An dem Punkt, an dem `segment = 11` ist, gibt die Bedingung `if (segment <=10)` den Wert *falsch* zurück, und der Code zwischen den inneren Klammern wird übersprungen. Die Kontrolle kehrt zum vorherigen Aufruf von `spiral2` zurück. Von dort geht sie dann die ganze Aufrufkette aufwärts bis zum ersten Aufruf zurück. Danach wird das Programm beendet. Abbildung 13.2 zeigt die Folge der Aufrufe und die Rückkehr zum Aufrufer.

Immer dann, wenn sich eine Funktion selbst aufruft, führt sie dies eine Stufe weiter von dem Hauptprogramm weg, das die Aktion angestoßen hat. Damit das Hauptprogramm fortgesetzt werden kann, muss die tiefste Iteration die Kontrolle wieder an die Iteration zurückgeben, die die tiefere Iteration aufgerufen hat. Die höhere Iteration muss auf die gleiche Weise die Kontrolle weiter nach oben abgeben, bis das Hauptprogramm wieder erreicht ist, das den ersten Aufruf der rekursiven Funktion ausgelöst hat.

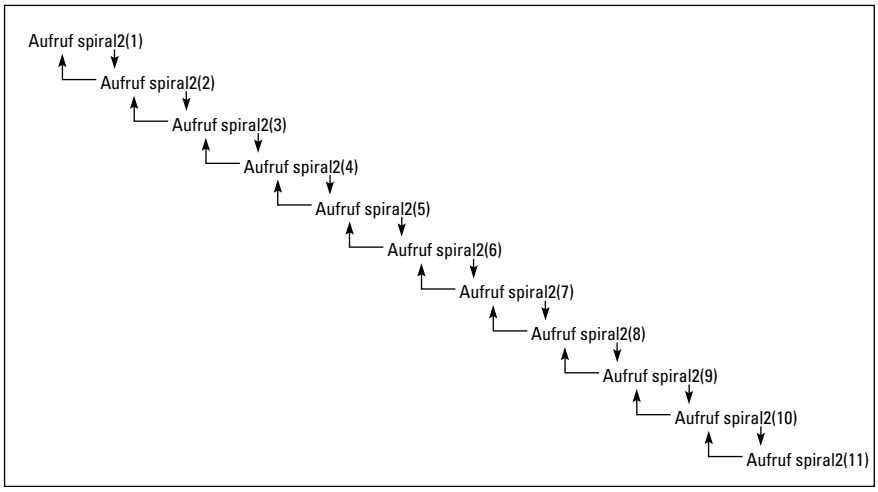


Abbildung 13.2: Die Rekursionskette hinab- und wieder hinaufsteigen

Die Rekursion ist ein leistungsstarkes Werkzeug, um Code wiederholt auszuführen, wenn man am Anfang nicht weiß, wie oft der Code wiederholt werden soll. Sie ist das ideale Werkzeug, um Baumstrukturen zu durchsuchen, beispielsweise Stammbäume von Familien, komplexe elektronische Schaltkreise oder mehrschichtige Verteilernetzwerke.

Was ist eine rekursive Abfrage?

Eine *rekursive Abfrage* ist eine Abfrage, die funktionell von sich selbst abhängig ist. Die einfachste Form einer solchen funktionellen Abhängigkeit liegt vor, wenn eine Abfrage A1 sich innerhalb ihrer eigenen Struktur selbst aufruft. Ein komplexerer Fall liegt vor, wenn eine Abfrage A1 von einer Abfrage A2 abhängig ist, die ihrerseits eine Abhängigkeit von Abfrage A1 aufweist. Eine funktionelle Abhängigkeit und eine Rekursion liegen unabhängig davon vor, wie viele Abfragen zwischen dem ersten und dem zweiten Aufruf derselben Abfrage liegen.

Wo kann ich eine rekursive Abfrage anwenden?

Mit rekursiven Abfragen können Sie bei der Lösung verschiedener Probleme Zeit und Mühe sparen. Angenommen, Sie verfügen über ein Flugticket, mit dem Sie kostenlos jeden Flug der (fiktiven) Vannevar Airlines benutzen dürfen. Da stellt sich natürlich sofort die Frage: »Wohin kann ich kostenlos fliegen?« Die Tabelle `Flug` enthält alle Flüge von Vannevar. Tabelle 13.1 zeigt Flugnummer sowie Start und Ziel eines jeden Flugs.

FlugNr	Start	Ziel
3141	Portland	Orange County
2173	Portland	Charlotte
623	Portland	Daytona Beach
5440	Orange County	Montgomery
221	Charlotte	Memphis
32	Memphis	Champaign
981	Montgomery	Memphis

Tabelle 13.1: Flüge, die von Vannevar Airlines angeboten werden

Abbildung 13.3 zeigt die Routen auf der Karte der Vereinigten Staaten.

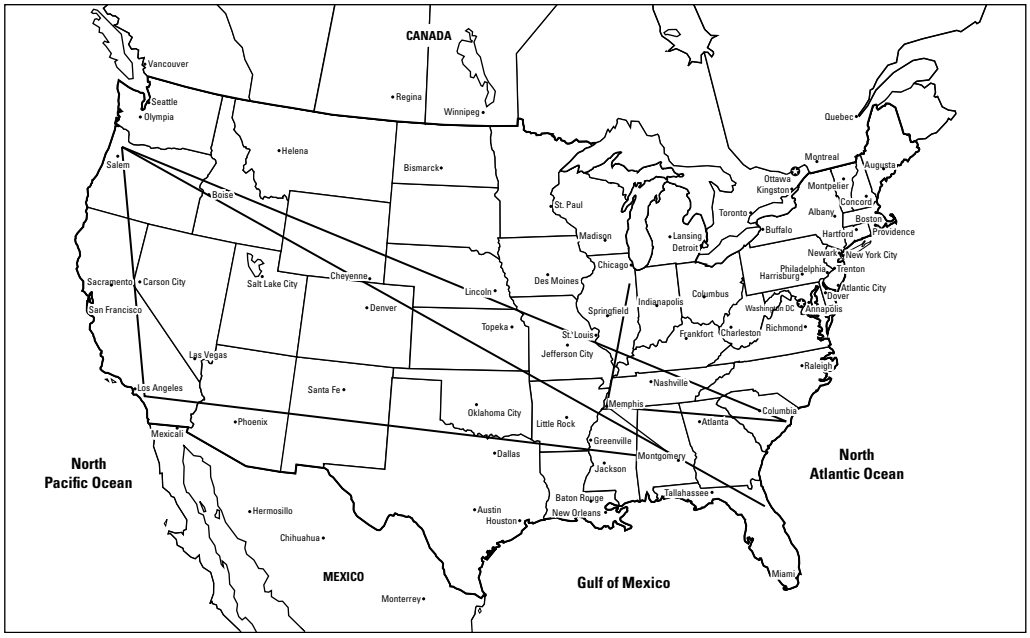


Abbildung 13.3: Routenkarte von Vannevar Airlines

Für Ihren Urlaubsplan legen Sie die Datenbanktabelle `Flug` an, indem Sie SQL wie folgt verwenden:

```
CREATE TABELLE Flug (
   Flugnr     INTEGER          NOT NULL,
   Start      CHARACTER(30),
   Ziel       CHARACTER(30)
);
```

Nachdem Sie die Tabelle erstellt haben, können Sie die Daten aus Tabelle 13.1 übernehmen.

Angenommen, Ihre Reise beginnt in Portland und Sie möchten zunächst einen Freund in Montgomery besuchen. Natürlich fragen Sie sich: »Welche Städte kann ich mit Vannevar erreichen, wenn ich in Portland starte?« und: »Welche Städte kann ich mit derselben Fluggesellschaft erreichen, wenn ich in Montgomery starte?« Einige Städte sind mit einem Flug erreichbar, andere nicht. Bei einigen müssen Sie zweimal oder häufiger umsteigen. Anhand der Routenkarte könnten Sie alle Städte ermitteln, zu denen Sie mit Vannevar von einer bestimmten Stadt aus fliegen können – schließlich beherrschen Sie doch SQL.

Abfragen auf die harte Tour erstellen

Um herauszufinden, was Sie wissen wollen, können Sie eine Reihe von Abfragen ausführen, die `Portland` als Startpunkt haben – vorausgesetzt, dass Sie über genügend Zeit und Geduld verfügen:

```
SELECT Ziel FROM Flug WHERE Start = "Portland";
```

Die erste Abfrage gibt `Orange County`, `Charlotte` und `Daytona Beach` zurück. Ihre zweite Abfrage könnte das erste dieser Ergebnisse als Startpunkt verwenden:

```
SELECT Ziel FROM Flug WHERE Start = "Orange County";
```

Die zweite Abfrage gibt `Montgomery` zurück. Ihre dritte Abfrage könnte wieder auf die erste Abfrage zurückgreifen und deren zweites Ergebnis als Startpunkt nehmen:

```
SELECT Ziel FROM Flug WHERE Start = "Charlotte";
```

Die dritte Abfrage gibt `Memphis` zurück. Ihre vierte Abfrage könnte auf die erste Abfrage zurückgreifen und deren letztes Ergebnis als Startpunkt wählen:

```
SELECT Ziel FROM Flug WHERE Start = "Daytona Beach";
```

Leider gibt die vierte Abfrage einen Nullwert zurück, weil `Vannevar` keine Flüge anbietet, die in Daytona Beach starten. Aber die zweite Abfrage hat eine weitere Stadt (`Montgomery`) als möglichen Startpunkt zurückgegeben, weshalb Ihre fünfte Abfrage dieses Ergebnis verwenden kann:

```
SELECT Ziel FROM Flug WHERE Start = "Montgomery";
```

Diese Abfrage gibt `Memphis` zurück, aber Sie wissen bereits, dass Sie diese Stadt erreichen können (in diesem Fall über Charlotte). Sie machen aber weiter und setzen dieses letzte Ergebnis als Startpunkt für eine weitere Abfrage ein:

```
SELECT Ziel FROM Flug WHERE Start = "Memphis";
```

Diese Abfrage gibt `Champaign` zurück – das Sie zu der Liste der erreichbaren Städte hinzufügen können (selbst wenn Sie dafür zwei Teilstrecken benötigen). Solange Sie bereit sind, auch mehrere Teilstrecken zu fliegen, können Sie auch `Champaign` als Startpunkt wählen:

```
SELECT Ziel FROM Flug WHERE Start = "Champaign';
```

Hoppla. Diese Abfrage gibt einen Nullwert zurück; Vannevar bietet keine Abflüge von Champaign an. (Bis jetzt haben wir sieben Abfragen ausgeführt. Wird schon jemand unruhig?)

Vannevar bietet auch keinen Flug von Daytona Beach aus an, weshalb Sie dort stecken bleiben, falls Sie dieses Ziel ansteuern – was möglicherweise gar nicht so schlimm ist, falls es gerade Frühlingsanfang ist und dort die entsprechenden Partys stattfinden. (Wenn Sie natürlich eine ganze Woche damit verbringen, einzelne Abfragen auszuführen, um Ihr nächstes Ziel zu ermitteln, könnten Sie davon größere Kopfschmerzen bekommen als von einer durchzechten Nacht.) Möglicherweise bleiben Sie auch in Champaign stecken – in diesem Fall könnten Sie sich bei der Universität von Illinois einschreiben und einige Datenbankkurse belegen.

Zugegeben – mit dieser Methode können Sie (eventuell) die Frage beantworten: »Welche Städte sind von Portland aus erreichbar?« Aber eine Abfrage nach der anderen auszuführen, wobei jede neue Abfrage von den Ergebnissen einer vorangegangenen Abfrage abhängt, ist kompliziert, zeitaufwendig und nervenaufreibend.

Zeit mit einer rekursiven Abfrage sparen

Eine einfachere Methode, um die benötigten Informationen zu ermitteln, besteht darin, eine einzige rekursive Abfrage zu entwickeln, die die gesamte Aufgabe auf einmal löst. In unserem Beispiel hat die Abfrage die folgende Syntax:

```
WITH REKURSIVE
   ErreichbarVon (Start, Ziel)
      AS (SELECT Start, Ziel
             FROM Flug
          UNION
          SELECT in.Start, out.Ziel
             FROM ErreichbarVon in, Flug out
             WHERE in.Ziel = out.Start
         )
   SELECT * FROM ErreichbarVon
   WHERE Start = "Portland";
```

Beim ersten Durchlauf der Rekursion hat `Flug` sieben Zeilen und `ErreichbarVon` keine. `UNION` nimmt die sieben Zeilen von `Flug` und kopiert sie in `ErreichbarVon`. An diesem Punkt enthält `ErreichbarVon` die Daten, die in Tabelle 13.2 gezeigt werden.

Wie bereits erwähnt, gehört Rekursion nicht zum Kern von SQL. Deshalb ist sie in einigen Implementierungen möglicherweise nicht enthalten.

Start	Ziel
Portland	Orange County
Portland	Charlotte
Portland	Daytona Beach
Orange County	Montgomery
Charlotte	Memphis
Memphis	Champaign
Montgomery	Memphis

Tabelle 13.2: `ErreichbarVon` nach einem Durchlauf durch die Rekursion

Beim zweiten Durchlauf durch die Rekursion wird es langsam interessant. Die `WHERE`-Klausel (`WHERE in.Ziel = out.Start`) bedeutet, dass Sie nur die Zeilen auswerten, bei denen das Feld `Ziel` der Tabelle `ErreichbarVon` gleich dem Feld `Start` der Tabelle `Flug` ist. Bei diesen Zeilen nehmen Sie das Feld `Start` von `ErreichbarVon` und das Feld `Ziel` von `Flug` und fügen diese beiden Felder in `ErreichbarVon` als neue Zeile ein. Tabelle 13.3 zeigt das Ergebnis dieser Iteration der Rekursion.

Start	Ziel
Portland	Orange County
Portland	Charlotte
Portland	Daytona Beach
Orange County	Montgomery
Charlotte	Memphis
Memphis	Champaign
Montgomery	Memphis
Portland	Montgomery
Portland	Memphis
Orange County	Memphis
Charlotte	Champaign

Tabelle 13.3: `ErreichbarVon` nach zwei Durchläufen durch die Rekursion

Die Ergebnisse sehen jetzt sehr viel brauchbarer aus. `ErreichbarVon` enthält jetzt alle `Ziel`-Städte, die von jeder `Start`-Stadt aus in zwei Teilstrecken oder weniger erreichbar sind. Die Rekursion verarbeitet als Nächstes die Reisen mit drei Teilstrecken und so weiter, bis alle möglichen `Ziel`-Städte erreicht worden sind.

Nachdem die Rekursion abgeschlossen ist, zieht die dritte und letzte `SELECT`-Anweisung (die sich außerhalb der Rekursion befindet) aus `ErreichbarVon` nur die Städte heraus, die Sie von Portland aus durch einen Flug mit Vannevar erreichen können. In diesem Beispiel sind

alle sechs anderen Städte von Portland aus erreichbar – wobei es so wenige Zwischenstopps sind, dass Sie nicht glauben müssen, mit einem Springstock unterwegs zu sein.

Wenn Sie den Code der rekursiven Abfrage genau betrachten, *sieht er nicht einfacher aus* als die sieben Einzelabfragen, an deren Stelle er tritt. Die rekursive Abfrage hat jedoch zwei Vorteile:

- ✔ Wenn Sie die Abfrage starten, führt sie die gesamte Operation ohne einen weiteren Eingriff aus.
- ✔ Sie kann diese Aufgabe wirklich schnell ausführen.

Stellen Sie sich jetzt eine tatsächlich existierende Fluggesellschaft vor, die eine viel größere Anzahl von Städten anfliegt. Je mehr mögliche Ziele es gibt, desto größer ist der Vorteil, den die rekursive Methode bietet.

Wodurch wird diese Abfrage rekursiv? Dies wird durch die Tatsache bewirkt, dass wir `ErreichbarVon` durch sich selbst definieren. Der rekursive Teil der Definition ist die zweite `SELECT`-Anweisung, die direkt auf `UNION` folgt. `ErreichbarVon` ist eine temporäre Tabelle, die mit dem Fortschreiten der Rekursion immer mehr mit Daten gefüllt wird. Die Verarbeitung wird so lange durchgeführt, bis alle möglichen Ziele in `ErreichbarVon` eingetragen sind. Duplikate werden ausgeschlossen, weil der `UNION`-Operator keine Duplikate in die Ergebnistabelle einfügt. Nach dem Abschluss der Rekursion enthält `ErreichbarVon` alle Städte, die von einer beliebigen `Start`-Stadt aus erreichbar sind. Die dritte und letzte `SELECT`-Anweisung gibt nur die Städte zurück, die Sie von Portland aus erreichen können. Gute Reise!

Wo könnte ich eine rekursive Abfrage sonst noch einsetzen?

Jedes Problem, das in Form einer Baumstruktur darstellbar ist, kann potenziell mit einer rekursiven Abfrage gelöst werden. Die klassische industrielle Anwendung sind Stücklisten für Produktionsprozesse, bei denen Rohmaterialien über Zwischenprodukte in Fertigprodukte umgewandelt werden. Stellen Sie sich vor, Ihr Unternehmen baut ein neues Hybridauto, das mit Benzin und Strom betrieben wird. Eine solche Maschine wird aus Baugruppen zusammengestellt – Motor, Batterien und so weiter –, die ihrerseits aus kleineren Baugruppen (Zylinder, Elektroden und so weiter) bestehen, die wiederum aus kleineren Teilen bestehen.

Die Verwaltung dieser unterschiedlichen Teile in einer relationalen Datenbank, die keine Rekursion verwendet, kann sehr schwierig sein. Wenn Sie die Rekursion einsetzen, können Sie mit der kompletten Maschine beginnen und sich bis zu den Einzelheiten hinunter durcharbeiten. Sie wollen die Einzelheiten über die Befestigungsschraube wissen, die die Klemme an der negativen Elektrode der Hilfsbatterie hält? Kein Problem. Die `WITH RECURSIVE`-Struktur stattet SQL mit der Fähigkeit aus, solche Fragen zu lösen.

Die Rekursion eignet sich auch zur Beantwortung von Was-wäre-wenn-Fragen. Was passiert in dem Beispiel der Vannevar Airlines, wenn die Geschäftsleitung beschließen sollte, die Flüge von Portland nach Charlotte einzustellen? Welchen Einfluss hätte dies auf die Städte, die von Portland aus erreichbar sind? Eine rekursive Abfrage liefert schnell eine Antwort.

Teil IV

Kontrollmechanismen

In diesem Teil ...

- ✔ Den Zugang kontrollieren
- ✔ Daten vor Beschädigung schützen
- ✔ Prozedurale Sprachen anwenden

Datenbanken schützen

14

In diesem Kapitel

- Den Zugriff auf Datenbanktabellen kontrollieren
- Entscheiden, wer worauf zugreifen darf
- Zugriffsrechte vergeben
- Zugriffsrechte entziehen
- Nicht autorisierte Zugriffsversuche verhindern
- Das Recht zur Vergabe von Rechten delegieren

Ein Systemadministrator muss genau wissen, wie eine Datenbank arbeitet. Deshalb beschreibe ich in den vorangegangenen Kapiteln die Komponenten von SQL, mit denen Sie eine Datenbank erstellen und Daten in Datenbanken bearbeiten können. In Kapitel 3 gebe ich Ihnen eine Einführung in die Möglichkeiten von SQL, Datenbanken vor Schäden und Missbrauch zu bewahren. In diesem Kapitel möchte ich detailliert auf das Thema des Missbrauchs eingehen.

Die Person, die für eine Datenbank verantwortlich ist, kann darüber bestimmen, welche Benutzer darauf zugreifen und was diese dabei tun dürfen. Diese Person kann selektiv die Zugriffsrechte der Benutzer für bestimmte Teile des Systems vergeben oder entziehen. Sie kann sogar dieses Recht zur Vergabe und zum Entzug von Rechten auf andere Personen übertragen. Richtig eingesetzt sind die Sicherheitswerkzeuge von SQL mächtige Schutzfaktoren wichtiger Daten. Falsch eingesetzt können dieselben Werkzeuge zu frustrierenden Störfaktoren werden, die legitime Benutzer bei ihrer Arbeit behindern.

Da Datenbanken oft vertrauliche Informationen enthalten, die nicht für jeden verfügbar sein sollen, bietet SQL verschiedene Zugriffsebenen – von keinem bis zum totalen Zugriff mit mehreren Zwischenstufen. Indem der Datenbankadministrator kontrolliert, welche Operationen die berechtigten Personen durchführen können, ist er in der Lage, den einzelnen Benutzern nur die Daten zur Verfügung zu stellen, auf die sie zugreifen müssen – die anderen Teile der Datenbank, die sie nichts angehen, sind für sie gesperrt.

Die Datenkontrollsprache von SQL

Die SQL-Anweisungen, mit denen Sie Datenbanken erstellen, werden als *Datendefinitionssprache (DDL, Data Definition Language)* bezeichnet. Nachdem Sie eine Datenbank erstellt haben, können Sie eine andere Gruppe von SQL-Anweisungen verwenden, mit denen Sie in einer Datenbank Daten hinzufügen, ändern und löschen. Diese Anweisungen werden zusammengefasst als *Datenbearbeitungssprache (DML, Data Manipulation Language)* bezeichnet. Darüber hinaus enthält SQL Anweisungen, die nicht zu diesen beiden Kategorien gehören.

Programmierer verwenden für diese Anweisungen ab und an den kollektiven Begriff *Datenkontrollsprache (DCL, Data Control Language)*. Die DCL-Anweisungen dienen hauptsächlich dazu, die Datenbank vor unautorisierten Zugriffen, schädlichen Interaktionen zwischen mehreren Datenbankbenutzern sowie Strom- und Geräteausfällen zu schützen. In diesem Kapitel behandle ich den Schutz vor unautorisierten Zugriffen.

Zugriffsebenen für Benutzer

SQL ermöglicht einen kontrollierten Zugriff auf die folgenden neun Datenbankverwaltungsfunktionen:

- ✔ **Erstellen, sehen, modifizieren und löschen:** Diese Funktionen entsprechen den Operationen `INSERT`, `SELECT`, `UPDATE` und `DELETE` (siehe Kapitel 6).
- ✔ **Verweisen (referenzieren):** Das Schlüsselwort `REFERENCES` (siehe Kapitel 3 und 5) hat mit der Anwendung von Einschränkungen zu tun, die die referenzielle Integrität einer Tabelle aufrechterhalten sollen, die von anderen Tabellen der Datenbank abhängt.
- ✔ **Benutzen:** Das Schlüsselwort `USAGE` bezieht sich auf Domänen, Zeichensätze, Sortierreihenfolgen und Übersetzungstabellen (siehe Kapitel 5).
- ✔ **Definieren neuer Datentypen:** Um mit benutzerdefinierten Typennamen zu arbeiten, verwenden Sie das Schlüsselwort `UNDER`.
- ✔ **Reagieren auf Ereignisse:** Der Einsatz des Schlüsselworts `TRIGGER` führt dazu, dass eine SQL-Anweisung oder ein Satz von SQL-Anweisungen immer dann ausgeführt wird, wenn ein vorgegebenes Ereignis eintritt.
- ✔ **Ausführen:** Mit dem Schlüsselwort `EXECUTE` wird eine Routine ausgeführt.

Der Datenbankadministrator

Die höchste Autorität in einer Datenbank, die von mehreren Benutzern verwendet wird, ist der Datenbankadministrator (*DBA*). Der DBA hat in Bezug auf die Datenbank alle Rechte. Die Position eines DBA ist mit Macht, aber auch mit Verantwortung verbunden. Mit dieser Macht können Sie Ihre Datenbank leicht beschädigen und Tausende von Arbeitsstunden zunichtemachen. Ein DBA sollte sich immer über die Konsequenzen seiner Aktionen im Klaren sein.

Ein DBA verfügt nicht nur über alle Rechte an der Datenbank, sondern kontrolliert auch die Rechte der anderen Benutzer. Er kann beispielsweise besonders zuverlässigen Mitarbeitern größere Zugriffsrechte auf mehr Tabellen einräumen als der Mehrzahl der anderen Benutzer.

Der einfachste Weg, DBA zu werden, besteht darin, das Datenbankverwaltungssystem selbst zu installieren. Dabei nennt Ihnen das Installationshandbuch ein *Konto*, auch *Login* oder *Benutzername* genannt, und ein Kennwort. Dieser Benutzername identifiziert Sie als speziellen, privilegierten Benutzer. Bei manchen Systemen wird dieser privilegierte Benutzer als *DBA*, bei anderen als *Systemadministrator* oder auch als *Superuser* (leider ohne Umhang und Stiefel) bezeichnet. Auf jeden Fall sollte Ihre erste Amtshandlung nach Ihrer ersten offiziellen Anmeldung darin bestehen, das Kennwort zu ändern.

Wenn Sie das Kennwort nicht ändern, kann sich jeder, der das Handbuch kennt, als DBA mit allen Rechten anmelden. Nachdem Sie das Kennwort geändert haben, können sich nur die Benutzer als DBA anmelden, die das neue Kennwort kennen. Ich empfehle Ihnen, das neue DBA-Kennwort nur einigen wenigen, sehr zuverlässigen Personen mitzuteilen. Schließlich könnten Sie morgen von einem Meteoriten getroffen werden, im Lotto gewinnen oder auf eine andere Weise für Ihre Firma unerreichbar werden. Ihre Kollegen müssen auch ohne Sie weiterarbeiten können. Jeder, der sich mit dem Benutzernamen und dem Kennwort des DBA anmeldet, gilt für das System als DBA.

Selbst wenn Sie über DBA-Rechte verfügen, sollten Sie sich nur dann als DBA anmelden, wenn Sie spezielle Aufgaben ausführen wollen, für die DBA-Rechte erforderlich sind. Wenn Sie damit fertig sind, melden Sie sich wieder ab und melden sich dann mit Ihrem normalen Benutzernamen und Kennwort wieder an, um normal weiterzuarbeiten. Damit können Sie verhindern, dass Sie Fehler machen, die für Ihre Tabellen und die der anderen Benutzer fatale Folgen haben könnten.

Besitzer von Datenbankobjekten

Neben dem DBA gibt es eine weitere Klasse privilegierter Benutzer, die sogenannten *Besitzer von Datenbankobjekten*. Beispielsweise zählen Tabellen und Sichten zu den Datenbankobjekten. Jeder Benutzer, der ein solches Objekt erstellt, kann dessen Besitzer festlegen. Der Besitzer einer Tabelle verfügt über jedes mögliche Recht, das mit dieser Tabelle verbunden ist, einschließlich des Rechts, anderen Benutzern das Zugriffsrecht auf seine Tabelle einzuräumen. Weil Sichten auf Tabellen beruhen, können Sie eine Sicht erstellen, die auf die Tabelle eines fremden Besitzers zugreift. Der Besitzer der Sicht verfügt aber nur über die Rechte, die er auch für die darunter liegende Tabelle besitzt. Sie können sich also nicht dadurch den Zugriff auf Tabellen anderer Benutzer erschleichen, dass Sie einfach eine neue Sicht erstellen, die auf diese Tabellen zugreift.

Die Öffentlichkeit

Die Öffentlichkeit (in vielen Datenbanksystemen auch als `PUBLIC` bezeichnet) besteht aus allen Benutzern, die keine speziellen Rechte (entweder als DBA oder Objektbesitzer) haben und denen kein privilegierter Benutzer spezielle Rechte eingeräumt hat. Wenn ein privilegierter Benutzer dem allgemeinen Benutzer `PUBLIC` bestimmte Rechte einräumt, verfügt jeder, der sich beim System anmeldet, über diese Rechte.

Bei den meisten Installationen existiert eine Hierarchie von Benutzerrechten, in der der DBA auf der obersten Ebene und die Öffentlichkeit auf der untersten Ebene stehen. Abbildung 14.1 stellt eine solche privilegierte Hierarchie dar.

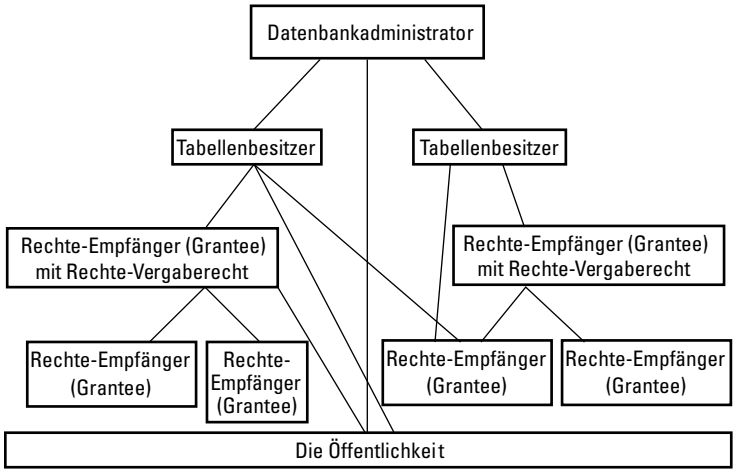


Abbildung 14.1: Beispiel für die Rechtehierarchie

Rechte an Benutzer vergeben

Der DBA verfügt kraft seiner Position über alle Rechte an allen Objekten in der Datenbank. Der Besitzer eines Objekts verfügt über alle Rechte an diesem Objekt – und auch die Datenbank selbst ist ein Objekt. Sonst verfügt niemand über irgendwelche Rechte an irgendwelchen Objekten, es sei denn, ihm werden diese Rechte von jemandem eingeräumt, der bereits über diese Rechte verfügte und der autorisiert ist, diese Rechte an andere Personen zu übertragen. Rechte werden mit der Anweisung `GRANT` eingeräumt, die folgende Syntax hat:

```
GRANT Rechteliste
   ON Objekt
   TO Benutzerliste
   [WITH GRANT OPTION]
```

In dieser Anweisung wird *`Rechteliste`* folgendermaßen definiert:

```
Recht [, Recht] ....
```

oder

```
ALL PRIVILEGES
```

`Recht` ist hier folgendermaßen definiert:

```
SELECT
| DELETE
| INSERT [(Spaltenname [, Spaltenname]...)]
| UPDATE [(Spaltenname [, Spaltenname]...)]
| REFERENCES [(Spaltenname [, Spaltenname]...)]
```

```
| USAGE
| UNDER
| TRIGGER
| EXECUTE
```

Objekt ist folgendermaßen definiert:

```
[TABLE] <Tabellenname>
| DOMAIN <Domänenname>
| COLLATION <Sortierfolgename>
| CHARACTER SET <Zeichensatzname>
| TRANSLATION <Übersetzungsname>
| TYPE <Benutzerdefinierter Typname>
| SEQUENCE <Sequenzgeneratorname>
| <spezieller Routinenbezeichner>
```

Benutzerliste ist folgendermaßen definiert:

```
  Login-ID [, Login-ID]...
| PUBLIC
```

Der *grantor* ist entweder der *CURRENT_USER* oder die *CURRENT_ROLE*.

Die vorhergehende Syntax behandelt eine Sicht als Tabelle. Die Rechte `SELECT`, `DELETE`, `INSERT`, `UPDATE`, `TRIGGER` und `REFERENCES` gelten nur für Tabellen und Sichten. Das Recht `USAGE` bezieht sich auf Domänen, Zeichensätze, Sortierfolgen und Übersetzungstabellen. Das Recht `UNDER` betrifft Typen, und das Recht `EXECUTE` gilt nur für Routinen. Die folgenden Abschnitte zeigen verschiedene Beispiele für die Anweisung `GRANT` und ihre Ergebnisse.

Rollen

Ein *Benutzername* ist ein Mittel, um eine Authentifizierung zu kennzeichnen, aber es ist nicht das einzige. Der Benutzername identifiziert eine Person (oder ein Programm), die oder das dazu berechtigt ist, eine oder mehrere Funktionen einer Datenbank auszuführen. In einem großen Unternehmen mit vielen Benutzern ist es mühsam und zeitaufwendig, jedem einzelnen Mitarbeiter Rechte zuzuweisen. SQL löst dieses Problem mit sogenannten *Rollen*.

Eine *Rolle* wird durch einen *Rollennamen* gekennzeichnet und besteht aus einem Satz von einem oder mehreren Rechten, die mehreren Benutzern eingeräumt werden können, die alle dieselben Zugriffsrechte für eine Datenbank benötigen. Beispielsweise erhält jeder, der die Rolle eines Sicherheitsbeauftragten ausübt, über die Rolle `Sicherheitsbeauftragter` dieselben Rechte. Diese Rechte unterscheiden sich von den Rechten der Personen, die die Funktion `Vertriebsmitarbeiter` haben.

Sie wissen bereits, dass nicht jedes Feature der SQL-Spezifikation in allen Implementierungen verfügbar ist. Lesen Sie in Ihrer DBMS-Dokumentation nach, bevor Sie versuchen, mit Rollen zu arbeiten.

Die Syntax zum Erstellen von Rollen entspricht in etwa dem folgenden Beispiel:

```
CREATE ROLE Vertrieb ;
```

Nachdem Sie eine Rolle erstellt haben, können Sie ihr mit der Anweisung GRANT Benutzer zuweisen, wie das folgende Beispiel zeigt:

```
GRANT Vertrieb TO Jutta ;
```

Sie können einer Rolle auf genau dieselbe Weise Rechte zuweisen wie einem Benutzer. Die einzige Ausnahme besteht darin, dass die Rolle mit Ihnen nicht diskutiert oder sich bei Ihnen beschwert.

Daten einfügen

Das Recht, Daten in eine Tabelle einzufügen, wird einer Rolle folgendermaßen eingeräumt:

```
GRANT INSERT
   ON Kunde
   TO Verkauf ;
```

Dieses Recht räumt jedem Mitarbeiter der Verkaufsabteilung das Recht ein, neue Kundendatensätze in die Tabelle Kunde einzufügen.

Daten lesen

Das Recht, die Daten einer Tabelle zu sehen, wird folgendermaßen eingeräumt:

```
GRANT SELECT
   ON Artikel
   TO PUBLIC ;
```

Dieses Recht räumt jeder Person, die Zugang zu dem System hat (PUBLIC), das Recht ein, die Daten der Tabelle Artikel zu betrachten.

Diese Anweisung kann gefährlich sein. Manche Spalten der Tabelle Artikel können Informationen enthalten, die nicht für jeden bestimmt sind, zum Beispiel Einkaufspreis. Um einen selektiven Zugriff auf bestimmte Spalten zu ermöglichen und andere Spalten zu unterdrücken, können Sie eine Sicht auf die Tabelle definieren, die keine Spalten mit vertraulichen Inhalten enthält. Gewähren Sie dann das SELECT-Recht für diese Sicht statt für die darunter liegende Tabelle. Das folgende Beispiel zeigt die Syntax für diese Prozedur:

```
CREATE VIEW ArtikelSicht AS
   SELECT Modell, ArtName, ArtBeschr, Listenpreis
      FROM Artikel ;
GRANT SELECT
   ON Artikelsicht
   TO PUBLIC ;
```

Die Sicht `ArtikelSicht` gewährt `PUBLIC` nur den Blick auf die in der zweiten Zeile aufgeführten vier Spalten, während `Einkaufspreis` und die übrigen Spalten verborgen bleiben.

Tabellendaten ändern

In jedem Unternehmen ändern sich Tabellendaten im Laufe der Zeit. Deshalb müssen Sie einigen Personen das Recht einräumen, die Daten zu ändern, während andere dies nicht dürfen. Änderungsrechte werden folgendermaßen vergeben:

```
GRANT UPDATE (BonusProzent)
   ON BonusRate
   TO VerkaufLtr ;
```

Der Verkaufsleiter kann als Reaktion auf die Marktbedingungen die Quote für den Bonus ändern, den Verkäufer für ihre Umsätze bekommen (die Spalte `BonusProzent`). Er kann jedoch nicht die Werte in den Spalten `MinBetrag` und `MaxBetrag` ändern, mit denen die Wertebereiche für die einzelnen Stufen des Bonusplans definiert werden. Wenn Sie das Änderungsrecht für alle Spalten einräumen wollen, müssen Sie entweder alle oder überhaupt keine Spaltennamen festlegen:

```
GRANT UPDATE
   ON BonusRate
   TO VPVerkauf ;
```

Tabellenzeilen löschen

Kunden ziehen sich aus dem Geschäftsleben zurück oder kaufen aus anderen Gründen nichts mehr von Ihrem Unternehmen. Mitarbeiter scheiden aus, gehen in Rente, werden entlassen oder sterben. Produkte werden ausgemustert. Das Leben geht weiter und Informationen, die Sie in der Vergangenheit gespeichert haben, sind heute überflüssig. Jemand muss veraltete Zeilen in Ihren Tabellen löschen. Mit dieser Aufgabe sollten jedoch nur zuverlässige Mitarbeiter betraut werden. Mit der Anweisung `GRANT` können Sie Löschrechte folgendermaßen vergeben:

```
GRANT DELETE
   ON Mitarbeiter
   TO PersonalLtr ;
```

Der Personalleiter kann jetzt Datensätze aus der Tabelle `Mitarbeiter` löschen. Über dieses Recht verfügen außerdem noch der DBA und die Person, die die Tabelle `Mitarbeiter` angelegt hat (wahrscheinlich ebenfalls der DBA). Niemand sonst ist in der Lage, Personaldatensätze zu löschen (es sei denn, eine weitere `GRANT`-Anweisung räumt dieser Person das Recht dazu ein).

Verknüpfte Tabellen referenzieren

Wenn eine Tabelle den Primärschlüssel einer zweiten Tabelle als Fremdschlüssel enthält, sind die Informationen in der zweiten Tabelle für die Benutzer der ersten Tabelle zugänglich. Diese Situation schafft eine potenziell gefährliche »Hintertür«, durch die nicht autorisierte Benutzer auf vertrauliche Informationen zugreifen können. In einem solchen Fall braucht ein Benutzer keine besonderen Zugriffsrechte, um etwas über den Inhalt einer Tabelle zu erfahren. Oft reicht es aus, dass der Benutzer Zugriffsrechte für eine Tabelle besitzt, die die Zieltabelle referenziert, um auch auf die Zieltabelle zugreifen zu können.

Angenommen, die Tabelle `Entlassung` enthält die Namen der Mitarbeiter, die im nächsten Monat entlassen werden sollen. Nur autorisierte Abteilungsleiter verfügen über ein `SELECT`-Recht für diese Tabelle. Ein nicht autorisierter Mitarbeiter vermutet nun, dass der Primärschlüssel dieser Tabelle die Mitarbeiterkennung `MitID` ist. Er erstellt eine neue Tabelle mit dem Namen `Spion` mit `MitID` als Fremdschlüssel, wodurch er auf die Tabelle `Entlassung` verweisen kann. (Wie Fremdschlüssel mit der Klausel `REFERENCES` erstellt werden, wird in Kapitel 5 beschrieben. Jeder Systemadministrator sollte diese Technik kennen.)

```
CREATE TABLE Spion
   (MitID INTEGER REFERENCES Entlassung) ;
```

Jetzt muss der Mitarbeiter nur noch versuchen, bestimmte Mitarbeiterkennungen in seine neue Tabelle `Spion` einzufügen. Da die Tabelle nur solche `MitID`-Werte annimmt, die in der Tabelle `Entlassungen` enthalten sind, kann der Mitarbeiter ermitteln, wer entlassen wird und wer nicht. Alle abgelehnten Einfügungen betreffen Mitarbeiter, die nicht in der Liste stehen.

Neuere Versionen von SQL verhindern diese Art von Sicherheitsverletzung, indem sie verlangen, dass ein berechtigter Benutzer den anderen Benutzern explizit Referenzrechte einräumen muss. Das folgende Beispiel zeigt, wie dies geschieht:

```
GRANT REFERENCES (MitID)
   ON Entlassung
   TO PersonalMit ;
```

Sie sollten prüfen, ob Ihr DBMS über diese aktualisierte Funktion verfügt.

Domänen, Zeichensätze, Sortierreihenfolgen und Übersetzungstabellen

Domänen, Zeichensätze, Sortierreihenfolgen und Übersetzungstabellen haben ebenfalls Einfluss auf die Sicherheit. Gerade Domänen müssen sorgfältig definiert werden, um zu verhindern, dass Ihre Sicherheitsmaßnahmen unterlaufen werden können.

Sie können eine Domäne definieren, die mehrere Spalten umfasst. Damit legen Sie fest, dass alle diese Spalten denselben Datentyp haben und denselben Einschränkungen unterliegen. Die Spalten, die Sie mit Ihrer `CREATE DOMAIN`-Anweisung erstellen, erben den Typ und die Einschränkungen der Domäne. Sie können bei Bedarf diese Eigenschaften für einzelne Spalten umdefinieren, aber Domänen bieten eine einfache Methode, um mit einer einzigen Deklaration zahlreiche Eigenschaften für mehrere Spalten gleichzeitig festzulegen.

Domänen sind nützlich, wenn Sie mit mehreren Tabellen arbeiten, die Spalten mit ähnlichen Eigenschaften enthalten. Beispielsweise kann Ihre Geschäftsdatenbank aus mehreren Tabellen bestehen, die alle eine `Preis`-Spalte vom Datentyp `DECIMAL(10,2)` enthalten und deren Werte nicht negativ und nicht größer als 10.000 Euro werden dürfen. Ehe Sie die Tabellen mit diesen Spalten erstellen, können Sie eine Domäne mit den gewünschten Spalteneigenschaften definieren:

```
CREATE DOMAIN PreisDomain DECIMAL (10,2)
   CHECK (VALUE >= 0 AND VALUE <= 10000) ;
```

Vielleicht kennzeichnen Sie Ihre Artikel in mehreren Tabellen durch die Spalte `Artikel Code`, die immer vom Datentyp `CHAR(5)` ist und die immer mit einem `X`, `C` oder `H` beginnen und mit `0` oder `9` enden. Auch für diese Spalten können Sie eine Domäne definieren:

```
CREATE DOMAIN ArtikelCodeDomain CHAR (5)
   CHECK (SUBSTR (VALUE, 1,1) IN ("X", "C", "H")
      AND SUBSTR (VALUE, 5, 1) IN ("9", "0") ) ;
```

Haben Sie diese Domänen eingerichtet, können Sie Tabellen wie folgt erstellen:

```
CREATE TABLE Artikel
   (ArtikelCode   ArtikelCodeDomain,
   ArtikelName    CHAR (30),
   Preis          PreisDomain) ;
```

Bei anderen ISO/IEC-Standardfunktionen von SQL habe ich bereits erwähnt, dass kein DBMS alle Funktionen unterstützt. Das gilt auch für `CREATE DOMAIN`. Die Datenbank iAnywhere von Sybase und PostgreSQL unterstützen diese Funktion, Oracle 11g und SQL Server 2012 tun dies nicht.

In dieser Tabellendefinition geben Sie nicht den Datentyp für `ArtikelCode` und `Preis` an, sondern die entsprechende Domäne. Damit werden den Spalten die Datentypen und Einschränkungen zugewiesen, die Sie bei der Definition der Domäne angegeben haben.

Das Arbeiten mit Domänen hat Einfluss auf die Sicherheit der Daten. Kann es Probleme geben, wenn jemand die Domänen benutzen möchte, die Sie definiert haben? Ja. Was ist, wenn jemand eine Tabelle mit einer Spalte vom Typ `PreisDomain` erstellt? Diese Person kann dann dieser Spalte immer größere Werte zuweisen, bis das System den Wert zurückweist. Damit hat der Benutzer herausgefunden, welche Obergrenze Sie in der Klausel `CHECK` Ihrer Anweisung `CREATE DOMAIN` für `Preis` festgelegt haben. Wenn diese Obergrenze geheim bleiben soll, darf niemand `PreisDomain` benutzen. Um die Sicherheit in solchen Situationen zu gewährleisten, können Sie in SQL ein Benutzungsrecht (`USAGE`) für Domänen erteilen. Nur der Benutzer, der eine Domäne definiert hat (und natürlich der DBA), kann ein solches Recht vergeben. Es wird folgendermaßen erteilt:

```
GRANT USAGE ON PreisDomain TO VerkaufLtr ;
```

Wenn Sie Domänen per `DROP` löschen, können verschiedene Sicherheitsprobleme auftreten. Tabellen mit Spalten, die durch eine Domäne definiert wurden, können Probleme verursachen, wenn Sie versuchen, die Domäne zu löschen. Möglicherweise müssen Sie vorher alle betroffenen Tabellen löschen. Oder das System weigert sich, die Domäne zu löschen. Wie Domänen gelöscht werden, ist von Ihrer Implementierung abhängig. SQL Server macht es auf die eine und Oracle auf die andere Art. Auf jeden Fall sollten Sie das Recht, Domänen zu löschen, auf zuverlässige Mitarbeiter beschränken. Dasselbe gilt für Zeichensätze, Sortierfolgen und Übersetzungstabellen.

Das Ausführen von SQL-Anweisungen bewirken

Manchmal löst die Ausführung einer SQL-Anweisung die Ausführung einer anderen SQL-Anweisung oder eines ganzen Anweisungsblocks aus. SQL unterstützt sogenannte Trigger (deutsch *Auslöser*). Ein *Trigger* legt ein Trigger-Ereignis, eine Trigger-Aktionszeit und eine oder mehrere Trigger-Aktionen fest.

- ✔ Das *Trigger-Ereignis* löst die Trigger-Aktion aus.
- ✔ Die *Trigger-Aktionszeit* legt fest, wann die ausgelöste Aktion stattfindet: vor oder nach dem Trigger-Ereignis.
- ✔ Die *Trigger-Aktion* ist die Ausführung einer oder mehrerer SQL-Anweisungen.

  Wenn mehr als eine SQL-Anweisung ausgelöst wird, müssen die Anweisungen innerhalb einer `BEGIN ATOMIC...END`-Struktur angeordnet sein. Das Trigger-Ereignis kann einer der Anweisungen vom Typ `INSERT`, `UPDATE` oder `DELETE` sein.

Sie können beispielsweise einen Trigger benutzen, um eine Anweisung ausführen zu lassen, die die Gültigkeit eines neuen Wertes überprüft, bevor ein `UPDATE` durchgeführt wird. Wenn der neue Wert ungültig ist, kann die Aktualisierung abgebrochen werden.

Ein Benutzer oder eine Rolle muss über ein `TRIGGER`-Recht verfügen, um einen Trigger erstellen zu können. Dazu ein Beispiel:

```
CREATE TRIGGER KundeLoeschen BEFORE DELETE
   ON Kunde FOR EACH ROW
   WHEN Staat = 'DE'
   INSERT INTO Kundenprotokoll VALUES ('DE–Kunde gelöscht') :
```

Immer dann, wenn in der Tabelle `Kunde` ein Kunde aus Deutschland gelöscht worden ist, wird in der Protokolltabelle `Kundenprotokoll` das Löschen mit einem neuen Eintrag vermerkt.

Rechte über Ebenen hinweg einräumen

In Kapitel 2 beschreibe ich strukturierte Typen als eine Art von User-definierten Typen (UDT). Ein großer Teil der Architektur strukturierter Typen ist von den Ideen der objektorientierten Programmierung abgeleitet. Eine dieser Ideen ist das Konzept der Hierarchie. Danach kann ein Typ Untertypen haben, die einige ihrer Attribute von ihrem übergeordneten Typ (ihrem Supertyp) erben. Zusätzlich zu diesen geerbten Attributen können sie auch Attribute haben, die exklusiv ihnen selbst gehören. Eine solche Hierarchie kann mehrere Ebenen enthalten. Der Typ am unteren Ende der Hierarchie wird als *Leaf-Typ* (*Blatt-Typ*) bezeichnet.

Eine typisierte Tabelle ist eine Tabelle, in der jede Zeile, die in der Tabelle gespeichert ist, eine Instanz des zugehörigen strukturierten Typs ist. Eine typisierte Tabelle hat eine Spalte für jedes Attribut des zugeordneten strukturierten Typs. Name und Datentyp der Spalte sind dieselben wie der Name und Datentyp des Attributs.

Nehmen Sie beispielsweise an, Sie seien ein Maler, der seine Gemälde über Galerien verkauft. Zusätzlich zu den Originalen verkaufen Sie auch signierte, nummerierte begrenzte Ausgaben, unsignierte, nicht nummerierte offene Ausgaben sowie Poster. Wie folgt können Sie einen strukturierten Typ für ein Kunstwerk erstellen:

```
CREATE TYPE Kunstwerk (
     Künstler        CHARACTER VARYING (30),
     Titel           CHARACTER VARYING (50),
     Beschreibung    CHARACTER VARYING (256),
     Medium          CHARACTER VARYING (20),
     ErstellDatum    DATE )
     NOT FINAL
```

Dies ist ein weiteres Beispiel für eine Funktion, die nicht in allen DBMS-Produkten implementiert ist. PostgreSQL verfügt über die `CREATE TYPE`-Anweisung, Oracle 11g und SQL Server 2012 dagegen nicht.

Wenn Sie als Künstler Ihren Bestand überwachen wollen, sollten Sie zwischen Originalen und Reproduktionen und bei diesen vielleicht noch die verschiedenen Arten unterscheiden. Abbildung 14.2 zeigt eine mögliche Anwendung einer Hierarchie, um die erforderlichen Unterscheidungen zu erleichtern. Der Kunstwerk-Typ kann Untertypen haben, die selbst wieder eigene Untertypen haben können.

Es gibt eine Eins-zu-eins-Entsprechung zwischen den Typen in der Typenhierarchie und den Tabellen in der typisierten Tabellenhierarchie. Standardtabellen, wie sie in den Kapiteln 4 und 5 behandelt werden, dürfen nicht in eine Hierarchie eingefügt werden, die der hier für typisierte Tabellen beschriebenen ähnelt.

Anstelle eines Primärschlüssels verfügt eine typisierte Tabelle über eine selbst referenzierende Spalte, die die Eindeutigkeit nicht nur für die übergreifende Supertabelle einer Hierarchie, sondern auch für alle ihre Untertabellen garantiert. Die selbst referenzierende Spalte wird durch eine `REF IS`-Klausel in der `CREATE`-Anweisung der übergreifenden Supertabelle spezifiziert. Wenn die Referenz von dem System generiert wird, ist ihre Eindeutigkeit im System garantiert.

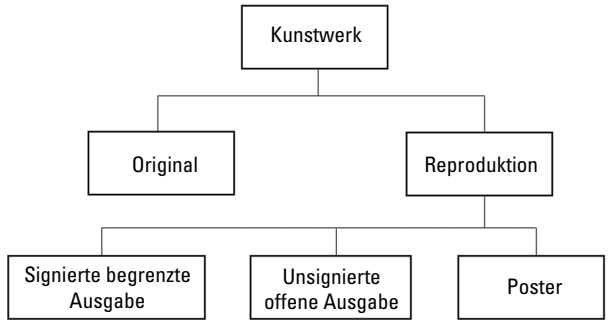


Abbildung 14.2: Hierarchie der `Kunstwerk`-Tabelle

Das Recht zur Vergabe von Rechten übertragen

Der DBA kann jedem Benutzer jedes Recht einräumen. Ein Besitzer eines Objekts kann allen anderen Benutzern alle Rechte einräumen, die sein Objekt betreffen. Benutzer, die auf diese Weise Rechte erhalten, können diese Rechte nicht an Dritte weitergeben. Diese Beschränkung unterstützt den DBA oder den Besitzer von Tabellen dabei, die Übersicht über die vergebenen Rechte zu behalten. Nur die Benutzer, die vom DBA oder vom Besitzer eines Objekts die entsprechenden Rechte erhalten haben, können auf die betreffenden Objekte zugreifen.

Aus Sicherheitsgründen ist es sinnvoll, die Möglichkeiten einer Person einzuschränken, Zugriffsrechte weiterzugeben. Es gibt jedoch häufig Situationen, in denen Benutzer die Möglichkeit haben müssen, ihr Berechtigungsrecht weiterzugeben. Die Arbeit darf nicht deshalb nicht weitergehen, weil jemand krank, in Urlaub oder in der Mittagspause ist.

Sie können *einigen* Benutzern das Recht gewähren, ihre Zugriffsrechte an bestimmte zuverlässige Ersatzleute weiterzugeben. Um einem Benutzer ein solches Delegationsrecht zuzuweisen, wird `GRANT` mit der Klausel `WITH GRANT OPTION` benutzt. Die folgende Anweisung zeigt, wie Sie diese Klausel einsetzen können:

```
GRANT UPDATE (BonusProzent )
   ON BonusRate
   TO VerkaufLtr
   WITH GRANT OPTION ;
```

Jetzt kann der Verkaufsleiter das `UPDATE`-Recht mit folgender Anweisung an seinen Assistenten weitergeben:

```
GRANT UPDATE (BonusProzent )
   ON BonusRate
   TO AssistVerkaufLtr ;
```

Wenn diese Anweisung ausgeführt worden ist, kann auch der Assistent des Verkaufsleiters die Spalte `BonusProzent` der Tabelle `BonusRate` ändern.

Natürlich müssen Sie einen Kompromiss zwischen Sicherheit und Bequemlichkeit schließen, wenn Sie die Vergabemöglichkeit von Zugriffsrechten delegieren. Der Besitzer der Tabelle `Bonusrate` verzichtet auf einen großen Teil seiner Kontrollmöglichkeiten, indem er dem Verkaufsleiter das `UPDATE`-Recht mit `WITH GRANT OPTION` einräumt. Man kann nur hoffen, dass der Verkaufsleiter sich seiner Verantwortung bewusst ist und sich sorgfältig überlegt, wem er dieses Recht gegebenenfalls weitergibt.

Rechte entziehen

Mit SQL können Sie Rechte nicht nur einräumen, sondern auch wieder entziehen. Die Aufgaben der Mitarbeiter ändern sich und damit auch die Daten, auf die diese Mitarbeiter zugreifen müssen. Manche Mitarbeiter verlassen die Firma und fangen bei der Konkurrenz an. Sie sollten dann unverzüglich alle Zugriffsrechte dieser Personen widerrufen.

Mit SQL können Sie mit der Anweisung `REVOKE` Zugriffsrechte wieder entfernen. Diese Anweisung funktioniert wie `GRANT`, kehrt aber ihre Wirkung genau um. Die Syntax von `REVOKE` sieht so aus:

```
REVOKE [GRANT OPTION FOR] Rechteliste
   ON Objekt
   FROM Benutzerliste [RESTRICT|CASCADE] ;
```

Sie können mit dieser Struktur bestimmte Rechte entziehen und andere bestehen lassen. Die Anweisungen `REVOKE` und `GRANT` unterscheiden sich hauptsächlich durch die Anwesenheit der beiden optionalen Schlüsselwörter `RESTRICT` und `CASCADE` bei `REVOKE`.

Angenommen, Sie haben einem Benutzer mit den Optionen `WITH GRANT OPTION` bestimmte Rechte eingeräumt. Dann können Sie dieses Recht mit `CASCADE` in der `REVOKE`-Anweisung später wieder widerrufen. Wenn Sie so ein Recht zurückziehen, heben Sie es nicht nur für den unmittelbar betroffenen Benutzer, sondern gleich auch für alle auf, an die er das Recht weitergegeben hatte.

Andererseits funktioniert die Anweisung `REVOKE` mit der Option `RESTRICT` nur dann, wenn der Rechte-Empfänger (englisch *Grantee*) die entsprechenden Rechte *nicht* weitergegeben hat. In diesem Fall gibt es beim Widerrufen von Rechten keinerlei Probleme. Wenn der Rechte-Empfänger die Rechte aber weitergegeben hat, widerruft die Anweisung `REVOKE` mit einer `RESTRICT`-Option nichts, sondern gibt einen Fehler zurück. Dies ist eine deutliche Warnung, dass Sie herausfinden müssen, wem Rechte von der Person eingeräumt worden sind, deren Rechte Sie widerrufen wollen. Sie können, müssen aber nicht, die Rechte dieser weiteren Person ebenfalls widerrufen.

Wenn Sie die Anweisung `REVOKE` mit der optionalen Klausel `GRANT OPTION FOR` benutzen, können Sie dem betroffenen Benutzer das Recht entziehen, seine Rechte zu delegieren. Dabei behält dieser Benutzer aber seine eigenen Rechte. Wenn Sie sowohl die Klausel `GRANT OPTION FOR` als auch das Schlüsselwort `CASCADE` verwenden, werden nicht nur alle Rechte widerrufen, die der betroffene Benutzer delegiert hat, sondern auch sein eigenes Recht zum Weiter-

geben dieser Rechte – so, als hätten Sie ihm dieses Recht nie eingeräumt. Wenn Sie sowohl die Klausel `GRANT OPTION FOR` als auch die Klausel `RESTRICT` benutzen, können zwei Dinge geschehen:

- ✔ Wenn der betroffene Benutzer keines der Rechte delegiert hat, die Sie wieder entziehen wollen, wird die Anweisung `REVOKE` ausgeführt und dem Benutzer das Recht zum Weitergeben von Rechten entzogen.
- ✔ Wenn der Benutzer jedoch bereits wenigstens eines der Rechte, die Sie wieder entziehen wollen, delegiert hat, wird die Anweisung `REVOKE` nicht ausgeführt und eine Fehlermeldung zurückgegeben.

Die Tatsache, dass Sie Rechte mit der Option `WITH GRANT OPTION` vergeben können, in Kombination mit der Tatsache, dass Sie Rechte selektiv widerrufen können, macht das Konzept der Systemsicherheit viel komplexer, als es dem ersten Anschein nach ist. So können beispielsweise mehrere berechtigte Personen ihre Rechte an ein und denselben Benutzer weitergeben. Wenn eine dieser Personen das Recht widerruft, verfügt der Benutzer immer noch darüber, weil es ihm ja auch von einer weiteren Person eingeräumt worden ist. Wenn ein Recht mit der Option `WITH GRANT OPTION` von einem Benutzer an einen anderen weitergegeben wird, entsteht eine *Kette von Abhängigkeiten*, in der die Rechte eines Benutzers von denen eines anderen Benutzers abhängen. Wenn Sie DBA oder Objektbesitzer sind, sollten Sie immer daran denken, dass ein Recht, das Sie mit `WITH GRANT OPTION` erteilen, an ganz unerwarteten Stellen wieder auftauchen kann. Es kann zu einer echten Herausforderung werden, den nicht berechtigten Benutzern ihre Privilegien wieder zu entziehen und sie im gleichen Atemzug berechtigten Benutzern zu belassen. Im Allgemeinen weisen die Klauseln `GRANT OPTION` und `CASCADE` zahlreiche Spitzfindigkeiten auf. Wenn Sie diese Klauseln benutzen wollen, sollten Sie sowohl den SQL-Standard als auch Ihre Produktdokumentation sehr sorgfältig lesen, damit Sie die Funktionsweise dieser Klauseln in Ihrem System genau verstehen.

Mit GRANT und REVOKE zusammen Zeit und Aufwand sparen

Wenn Sie mehreren Benutzern mehrere Rechte nur für bestimmte Spalten einer Tabelle einräumen wollen, müssen Sie viel tippen. Betrachten Sie das folgende Beispiel der Firma Zetec. Der Verkaufsdirektor möchte, dass jeder Mitarbeiter der Verkaufsabteilung alle Daten in der Tabelle `Kunde` sehen kann. Doch nur die Verkaufsleiter sollen die Möglichkeit haben, Zeilen einzufügen, zu ändern oder zu löschen – und *niemand* soll das Feld `KundID` ändern können. Die Verkaufsleiter heißen Tyson, Keith und David. Sie können ihre Rechte mit einer Reihe von `GRANT`-Anweisungen festlegen:

```
GRANT SELECT, INSERT, DELETE
   ON Kunde
   TO Tyson, Keith, David ;
GRANT UPDATE
   ON Kunde (Firma, KundAdresse, KundOrt,
      KundStaat, KundPLZ, KundTel, ModEbene)
   TO Tyson, Keith, David ;
GRANT SELECT
   ON Kunde
   TO Jenny, Valerie, Melody, Neil, Robert, Sam,
      Brandon, Michellet, Allison, Andrew,
      Scott, Michelleb, Jaime, Linleigh, Matthew, Amanda;
```

Damit sollte das Problem erledigt sein. Jeder verfügt jetzt über ein `SELECT`-Recht für die Tabelle `Kunde`. Die Verkaufsleiter verfügen über volle `INSERT`- und `DELETE`-Rechte für die Tabelle und sie haben außerdem das Recht, alle Spalten außer der Spalte `KundID` zu ändern.

Es gibt eine einfachere Methode, um dasselbe Ergebnis zu erzielen:

```
GRANT SELECT
   ON Kunde
   TO Verkäufer ;
GRANT INSERT, DELETE, UPDATE
   ON Kunde
   TO Managers;
REVOKE UPDATE
   ON Kunde (KundID)
   FROM Managers;
```

Wenn Sie die Rollen korrekt definiert haben, benötigen Sie in diesem Beispiel immer noch drei Anweisungen, um denselben Schutz wie im vorangegangenen Beispiel zu erzielen. Niemand kann die Daten in der Spalte `KundID` ändern und nur Tyson, Keith und David verfügen über `INSERT`-, `DELETE`- und `UPDATE`-Rechte. Die letzten drei Anweisungen sind jedoch viel kürzer als die des vorhergehenden Beispiels, weil Sie nicht immer wieder alle Benutzer der Verkaufsabteilung, nicht alle Manager und nicht alle Spalten der Tabelle aufführen müssen.

Daten schützen

15

In diesem Kapitel

- Die Datenbank vor Schaden bewahren
- Probleme verstehen, die beim gleichzeitigen Zugriff auf Daten auftreten können
- Probleme des gleichzeitigen Datenzugriffs mit SQL-Mechanismen lösen
- Schutzmaßnahmen mit `SET TRANSACTION` an den eigenen Bedarf anpassen
- Daten schützen, ohne die Arbeit lahmzulegen

Jeder hat schon mal von Murphys Gesetz gehört: »Alles, was schiefgehen kann, geht schief.« Wir lachen über dieses Pseudogesetz, weil meistens alles reibungslos verläuft. Manchmal sieht es so aus, als würden wir zu den wenigen Glücklichen gehören, für die die Grundgesetze des Universums nicht gelten. Wenn unerwartete Probleme auftreten, erkennen wir normalerweise die Ursache und handeln entsprechend.

In einer komplexen Struktur wächst das Potenzial für unerwartete Probleme in ungeahnte Höhen (ein Mathematiker würde sagen, dass es »ungefähr mit dem Quadrat der Komplexität ansteigt«). Das ist der Grund dafür, dass Software-Projekte fast immer zu spät beendet werden und voller Fehler stecken. Eine gewichtige DBMS-Anwendung für mehrere Benutzer stellt eine große, komplexe Struktur dar. Beim Arbeiten mit einem solchen System können viele Dinge schiefgehen. Es wurden Methoden entwickelt, um die Auswirkungen dieser Probleme so gering wie möglich zu halten, aber die Probleme können niemals ganz vermieden werden. Das sind gute Nachrichten – zumindest für die Leute, die mit der Wartung von Datenbanken ihr Geld verdienen, weil es wahrscheinlich nie möglich sein wird, diese Aufgabe zu automatisieren und sie dadurch brotlos zu machen. Dieses Kapitel handelt von den großen Dingen, die in einer Datenbank schiefgehen können, und von den Werkzeugen, die SQL zur Verfügung stellt, um mit diesen Problemen umzugehen.

Gefahren für die Datenintegrität

Der Cyberspace (zu dem auch Ihr Netzwerk gehört) mag ja ganz unterhaltsam sein – aber für die Daten, die in ihm »leben«, stellt er eine unsichere Umgebung dar. Daten können auf vielerlei Weise beschädigt werden. In Kapitel 5 beschreibe ich Probleme, die durch das Eingeben falscher Daten, Bedienungsfehler und absichtliche Beschädigungen entstehen können. Schlecht formulierte SQL-Anweisungen und falsch entworfene Anwendungen können Ihre Daten ebenfalls beschädigen, und es ist nicht schwer, sich vorzustellen, wie das passieren kann. Zwei Bedrohungen habe ich noch nicht erwähnt: die Plattforminstabilität und den Ausfall von Geräten. Ich gehe in diesem Kapitel kurz auf diese Gefahren ein, konzentriere mich

aber hauptsächlich auf die Probleme, die durch den gleichzeitigen Zugriff auf Daten entstehen können.

Plattforminstabilität

Plattforminstabilität zählt zu einer Kategorie von Problemen, die es überhaupt nicht geben sollte – aber leider sieht die Praxis anders aus. Probleme dieser Art treten meistens auf, wenn Sie Ihrem System eine oder mehrere neue, noch wenig bewährte Komponenten hinzufügen. Probleme können durch eine neue DBMS-Version, eine neue Version des Betriebssystems oder neue Hardware ausgelöst werden. Während Sie eine kritische Anwendung laufen lassen, treten Bedingungen oder Situationen ein, die es nie zuvor gegeben hat. Ihr System hängt sich auf und Ihre Daten sind beschädigt. Um das Problem zu beheben, können Sie zwar Ihren Computer und seine Erbauer insgeheim oder lautstark beschimpfen, aber eigentlich können Sie nur hoffen, dass Ihre letzte Datensicherung in Ordnung ist.

Führen Sie niemals wichtige Geschäftsaufgaben auf einem System mit Komponenten aus, die sich noch nicht in der Praxis bewährt haben. Widerstehen Sie der Versuchung, die Betaversion der neuesten, mit vielen versprochenen Zusatzfunktionen versehenen Version Ihres DBMS oder Betriebssystems zur Erledigung der täglichen Aufgaben einzusetzen, mit denen Sie Ihr Brot verdienen. Wenn Sie praktische Erfahrungen mit der neuen Version eines Software-Produkts sammeln müssen, benutzen Sie dafür einen Rechner, der von dem Netzwerk, mit dem Sie Ihre geschäftlichen Aufgaben erledigen, vollkommen isoliert ist.

Geräteausfall

Selbst bewährte, sehr zuverlässige Geräte fallen manchmal aus und schicken Ihre Daten ins Jenseits. Jedes Gerät nutzt sich mit der Zeit ab – sogar Computer, die kaum bewegliche Teile enthalten. Wenn ein solcher Geräteausfall eintritt, während Ihre Datenbank geöffnet und aktiv ist, können Sie Daten verlieren. Es kann sogar noch schlimmer sein: Sie verlieren Daten und bemerken es nicht. Solch ein Ausfall tritt früher oder später ein. Wenn Murphys Gesetz gerade an diesem Tag gilt, passiert dies natürlich zum ungünstigsten Zeitpunkt.

Eine Möglichkeit, sich gegen Geräteausfälle zu schützen, ist *Redundanz*. Machen Sie von allem eine Extrakopie. Halten Sie ein zweites Hardware-System als Ersatz bereit, das genauso wie Ihr Arbeitssystem konfiguriert ist. Legen Sie Sicherungskopien Ihrer Datenbank und Anwendungen an, die Sie im Bedarfsfall auf Ihre Ersatz-Hardware kopieren können. Die Kosten können Sie davon abhalten, alles zu duplizieren (wodurch auch die Kosten verdoppelt werden), aber achten Sie wenigstens darauf, dass Ihre Datenbank und Ihre Anwendungen häufig genug gesichert werden, damit Sie nach einem Ausfall nicht ein zweites Mal große Datenmengen eingeben müssen. Viele DBMS-Produkte verfügen über die Fähigkeit der Datenreplikation. Das ist alles schön und gut, hilft aber nur, wenn Sie Ihr System auch so konfigurieren, dass diese Fähigkeit auch tatsächlich genutzt wird.

Eine weitere Möglichkeit, die schlimmsten Folgen eines Geräteausfalls zu vermeiden, ist die sogenannte *Transaktionsverarbeitung*. Dieses wichtige Thema wird weiter hinten in diesem Kapitel behandelt. Eine *Transaktion* ist eine unteilbare Arbeitseinheit. Entweder wird die ge-

samte Transaktion ausgeführt oder gar kein Teil von ihr. Dieses Alles-oder-nichts sieht recht drastisch aus, aber die schlimmsten Probleme tauchen auf, wenn nur ein Teil einer Reihe zusammengehöriger Datenbankoperationen ausgeführt wird. Folglich sollten Sie alles dafür tun, dass Sie selbst dann keine Daten verlieren oder beschädigen, wenn die Maschine, auf der die Datenbank liegt, ausfällt.

Gleichzeitiger Datenzugriff

Stellen wir uns vor, dass Sie mit einer bewährten Hard- und Software arbeiten, dass Ihre Daten in Ordnung sind, dass Ihre Anwendung fehlerfrei ist und dass Ihre Ausrüstung zuverlässig funktioniert. Utopie, nicht wahr? Aber auch dann können noch Probleme auftreten, wenn mehrere Personen versuchen, gleichzeitig auf dieselben Datenbanktabellen zuzugreifen *(Parallelzugriff)* und die Computer dieser Benutzer darüber streiten, wer dabei das Vorrecht hat *(Konkurrenzsituation)*. Mehrbenutzer-Datenbanksysteme müssen effizient mit diesem Tohuwabohu umgehen können.

Interaktionsprobleme bei Transaktionen

Probleme, die sich aus Konkurrenzsituationen ergeben, können sogar in Anwendungen auftreten, die sonst scheinbar nicht anfällig für Fehler sind. Betrachten wir ein Beispiel: Angenommen, Sie schreiben eine Anwendung zur Auftragsverarbeitung, die mit vier Tabellen arbeitet: `AuftragsKopf`, `AuftragsPosition`, `Kunde` und `Artikel`. Die Tabellen sollen die folgenden Eigenschaften haben:

- ✔ Die Tabelle `AuftragsKopf` hat die Spalten `AuftragsNummer` als Primärschlüssel und `KundenNummer` als Fremdschlüssel, der auf die Tabelle `Kunde` verweist.
- ✔ Die Tabelle `AuftragsPosition` hat die Spalten `PosNummer` als Primärschlüssel, `Artikel Nummer` als Fremdschlüssel, der die Tabelle `Artikel` referenziert, und `Menge`.
- ✔ Die Tabelle `Artikel` hat die Spalten `ArtikelNummer` als Primärschlüssel, und es gibt auch noch ein Feld `Lagerbestand`.
- ✔ Alle drei Tabellen verfügen außerdem über weitere Spalten, die bei diesem Beispiel jedoch keine Rolle spielen.

Ihre Firmenpolitik schreibt vor, einen Auftrag komplett oder gar nicht abzuwickeln. Es gibt keine Teillieferungen und keine Rückstände. (Entspannen Sie sich – dies ist nur ein hypothetisches Beispiel!) Sie schreiben die Anwendung `Auftragsverarbeitung`, um alle eingehenden Aufträge wie folgt in der Tabelle `AuftragsKopf` abzuarbeiten: Die Anwendung stellt fest, ob eine Lieferung *aller* Auftragspositionen möglich ist. Falls dies der Fall ist, stellt sie die Papiere aus und verringert den `Lagerbestand` in der Tabelle `Artikel`. (Diese Aktion löscht gleichzeitig die betreffenden Einträge in den Tabellen `AuftragsKopf` und `Auftrags Position`.) Sie haben die Anwendung so geschrieben, dass sie Aufträge auf zwei Arten bearbeiten kann:

- ✔ Die erste Methode besteht darin, die `Artikel`-Zeilen zu verarbeiten, die den Zeilen in der Tabelle `Auftragsposition` entsprechen. Wenn `Lagerbestand` genügend groß ist, zieht die Anwendung die Auftragsmenge ab. Falls `Lagerbestand` nicht groß genug ist, stellt

die Anwendung den ursprünglichen Zustand wieder her, sie führt also ein sogenanntes Rollback der Transaktion durch, um alle Reduzierungen der Positionen dieses Auftrags wieder rückgängig zu machen.

✔ Die zweite Methode besteht darin, zunächst alle `Artikel`-Zeilen zu prüfen, die von den Auftragspositionen betroffen sind, und danach – wenn *alle* Lagerbestände groß genug sind – die Auftragspositionen vom Lagerbestand abzubuchen.

Normalerweise ist die erste Methode effizienter, wenn alle Artikel auf Lager sind; die zweite dagegen ist effizienter, wenn Artikel fehlen. Wenn Ihr Lagerbestand normalerweise ausreicht, um Aufträge ausliefern zu können, sollten Sie deshalb die erste Methode verwenden, andernfalls die zweite. Stellen Sie sich vor, diese hypothetische Anwendung würde auf einem Mehrbenutzersystem laufen, das die durch den gleichzeitigen Zugriff bedingten Konkurrenzsituationen nicht ausreichend kontrolliert. Dies würde zu Problemen führen, wie das folgende Beispiel zeigt:

1. **Ein Kunde spricht jemanden in Ihrer Auftragsannahme (Benutzer A) an, um zehn Bolzenscheider und fünf Schraubenschlüssel zu bestellen.**

2. **Benutzer A wickelt einen Auftrag nach der ersten Methode ab. Die erste Position des Auftrags sind zehn Stück von Position 1 (Bolzenschneider).**

   Vom Artikel-1 des Auftrags sind zehn Stück auf Lager, die von dem Auftrag komplett verbucht werden.

   Die Auftragsverarbeitung läuft und die Lagermenge der Bolzenschneider wird auf null reduziert. Jetzt wird es interessant. Ein weiterer Kunde kontaktiert Ihre Firma und spricht mit Benutzer B.

3. **Benutzer B versucht, den kleinen Auftrag über einen Bolzenschneider auszuführen – und stellt fest, dass Sie keine Bolzenschneider mehr auf Lager haben.**

   Der Auftrag wird zurückgerollt, weil er nicht ausgeführt werden kann.

4. **Inzwischen versucht Benutzer A, fünf Stück von Artikel-37 (Schraubenschlüssel) abzubuchen.**

   Unglücklicherweise sind von denen aber nur vier Stück am Lager. Der gesamte Auftrag von Benutzer A (einschließlich der Bolzenschneider) wird deshalb zurückgesetzt, weil er nicht komplett ausgeführt werden kann.

   Die Tabelle `Artikel` ist jetzt wieder in dem Zustand, in dem sie war, bevor beide Benutzer anfingen zu arbeiten. Keiner der beiden Aufträge ist erledigt worden, obwohl der zweite hätte abgewickelt werden können.

Mit Methode 2 fahren Sie leider auch nicht viel besser, wenn auch aus einem anderen Grund. Benutzer A könnte alle bestellten Artikel prüfen und feststellen, dass alle Artikel in ausreichender Menge auf Lager sind. Wenn jetzt Benutzer B mit einem Auftrag dazwischenkommt und einen der (von A) bestellten Artikel abbucht, *bevor* Benutzer A seine Abbuchungen ausführt, können die Abbuchungen in der Transaktion von Benutzer A scheitern.

Serialisierung beseitigt schädliche Interaktionen

Wenn die Transaktionen *seriell*, also nacheinander, statt parallel ausgeführt werden, tritt kein Konflikt ein. Wird im ersten Beispiel die erfolglose Transaktion von Benutzer A vor dem Beginn der Transaktion von Benutzer B vollständig abgeschlossen, bevor die Transaktion von Benutzer B startet, bewirkt die Funktion `ROLLBACK`, dass Benutzer B den einzelnen Bolzenschneider liefern kann. (Die Funktion `ROLLBACK` setzt die gesamte Transaktion zurück und widerruft sie somit.) Werden im zweiten Beispiel die Transaktionen nacheinander (seriell) ausgeführt, hat Benutzer B keine Gelegenheit, die Menge eines Artikels zu ändern, ehe nicht die Transaktion von Benutzer A – erfolgreich oder nicht – abgeschlossen ist und Benutzer B feststellen kann, welche Menge des Artikels auf Lager ist.

Wenn Transaktionen seriell ausgeführt werden, können sie sich nicht gegenseitig schaden. Die Ausführung paralleler Transaktionen wird als *serialisierbar* bezeichnet, wenn das Ergebnis dasselbe ist wie bei einer seriellen Ausführung der Transaktionen.

Die Serialisierung gleichzeitiger Transaktionen ist kein Allheilmittel. Bei der Wahl des Verfahrens müssen Sie das Leistungsverhalten gegen den Sicherheitsbedarf abwägen. Je stärker Sie die Transaktionen voneinander abschotten, desto länger dauert die Ausführung einer Funktion. Versuchen Sie, hier die richtige Balance zu finden, und sichern Sie Ihr System so weit wie nötig ab – aber nicht weiter. Eine zu strikte Kontrolle der gleichzeitigen Datenzugriffe kann das Leistungsverhalten des Systems in die Knie zwingen.

Die Gefahr der Verfälschung von Daten reduzieren

Sie können an mehreren Stellen ansetzen, um zu vermeiden, dass Ihre Daten durch ein Missgeschick oder eine unvorhergesehene Interaktion beschädigt werden. Einige dieser Vorsichtsmaßnahmen sind in Ihr DBMS eingebaut und können von Ihnen konfiguriert werden. Sie bewahren Sie wie ein Schutzengel vor Schaden. Und wie bei Schutzengeln bemerken Sie ihr Wirken wahrscheinlich nicht einmal. Ihr Datenbankadministrator (DBA) trifft bestimmte Vorsichtsmaßnahmen, von denen Sie wahrscheinlich ebenfalls nichts merken. Schließlich können Sie – als Entwickler – selbst einige Vorkehrungen treffen, wenn Sie den Code schreiben.

Wenn Sie sich eine Menge Kummer sparen wollen, sollten Sie versuchen, es sich zur Gewohnheit zu machen, automatisch an einigen einfachen Vorkehrungen festzuhalten, damit diese automatisch in Ihren Code oder in Ihre Arbeit mit der Datenbank einfließen:

- ✔ Arbeiten Sie mit SQL-Transaktionen.
- ✔ Isolieren Sie Transaktionen voneinander und achten Sie dabei auf den Ausgleich zwischen Leistungsverhalten und Schutz.
- ✔ Lernen Sie, wann und wie Sie mit Transaktionen arbeiten, Datenbankobjekte sperren und Sicherungen durchführen sollten.

Die Einzelheiten zu diesen Punkten folgen gleich.

Mit SQL-Transaktionen arbeiten

Transaktionen gehören zu den Hauptwerkzeugen von SQL, um die Integrität einer Datenbank zu bewahren. Eine *SQL-Transaktion* kapselt alle SQL-Anweisungen, die eine Auswirkung auf die Datenbank haben können. Eine SQL-Transaktion wird entweder mit einer Anweisung vom Typ `COMMIT` oder `ROLLBACK` abgeschlossen.

- ✔ Wenn die Transaktion mit einem `COMMIT` beendet wird, werden alle Anweisungen in der Transaktion im Schnellverfahren auf die Datenbank angewendet.
- ✔ Wenn die Transaktion mit einem `ROLLBACK` beendet wird, werden die Auswirkungen aller Anweisungen »zurückgerollt« (widerrufen), und die Datenbank wird in den Zustand zurückversetzt, den sie vor Beginn der Transaktion gehabt hat.

Bei der folgenden Behandlung dieses Themas benutze ich den Begriff *Anwendung*, um entweder die Ausführung eines Programms (in Java, C++, COBOL, C oder einer anderen Programmiersprache) oder eine Reihe von Aktionen zu bezeichnen, die in einer einzelnen Sitzung an einem Terminal eingegeben werden.

Eine Anwendung kann eine Reihe von SQL-Transaktionen umfassen. Die erste SQL-Transaktion beginnt, wenn die Anwendung beginnt; die letzte SQL-Transaktion endet, wenn die Anwendung endet. Jedes `COMMIT` oder `ROLLBACK`, das die Anwendung ausführt, beendet eine SQL-Transaktion und beginnt die nächste. So hat beispielsweise eine Anwendung mit drei SQL-Transaktionen die folgende Form:

```
Beginn der Anwendung
   Verschiedene SQL–Anweisungen (SQL–Transaktion–1)
COMMIT oder ROLLBACK
   Verschiedene SQL–Anweisungen (SQL–Transaktion–2)
COMMIT oder ROLLBACK
   Verschiedene SQL–Anweisungen (SQL–Transaktion–3)
Ende der Anwendung
```

Ich benutze die Bezeichnung *SQL-Transaktion*, weil die Anwendung auch andere Funktionen (zum Beispiel Netzwerkzugriffe) ausführen kann, die andere Arten von Transaktionen verwenden. Wenn ich im folgenden Text *Transaktionen* erwähne, meine ich damit *SQL-Transaktionen*.

Eine normale SQL-Transaktion hat einen Zugriffsmodus, der entweder den Wert `READ–WRITE` (lesen und schreiben) oder `READ–ONLY` (nur lesen) hat. Sie hat eine Isolierungsebene, die einen der Werte `SERIALIZABLE`, `REPEATABLE`, `READ COMMITTED` oder `READ UNCOMMITTED` annehmen kann. (Ich beschreibe diese Eigenschaften einer Transaktion weiter hinten in diesem Kapitel im Abschnitt *Isolierungsebenen*.) Die Standardeigenschaften lauten `READ–WRITE` und `SERIALIZABLE`. Alle anderen Eigenschaften müssen Sie mit der Anweisung `SET TRANSACTION` setzen, beispielsweise:

```
SET TRANSACTION READ ONLY ;
```

oder

```
SET TRANSACTION READ ONLY REPEATABLE READ ;
```

oder

```
SET TRANSACTION READ COMMITTED ;
```

Eine Anwendung kann mehrere `SET TRANSACTION`-Anweisungen enthalten, aber jede Transaktion darf nur eine davon enthalten – und diese muss die erste SQL-Anweisung sein, die in der Transaktion ausgeführt wird. Wenn Sie eine `SET TRANSACTION`-Anweisung verwenden wollen, führen Sie sie entweder zu Beginn der Anwendung oder nach einem `COMMIT` oder `ROLLBACK` aus.

Weil jede neue Transaktion nach einem `COMMIT` oder `ROLLBACK` automatisch wieder mit den Standardeigenschaften arbeitet, müssen Sie die Anweisung `SET TRANSACTION` am Beginn einer jeden Transaktion ausführen, wenn diese nicht mit Standardeigenschaften arbeiten soll.

Die Anweisung `SET TRANSACTION` kann mit der Option `DIAGNOSTICS SIZE` die maximale Anzahl der Fehlermeldungen festlegen, die das System bei der Ausführung der Anwendung speichern soll. (Eine solche zahlenmäßige Begrenzung ist notwendig, weil eine Implementierung während einer Anweisung mehr als einen Fehler entdecken kann.) Der Standardwert für diese Fehleranzahl ist implementierungsabhängig und fast immer groß genug.

Die Standardtransaktion

Die Standardtransaktion verfügt über Eigenschaften, die für die meisten Benutzer in fast allen Fällen ausreichend sind. Bei den wenigen Gelegenheiten, bei denen Sie mit anderen Transaktionseigenschaften arbeiten müssen, können Sie diese mit der Anweisung `SET TRANSACTION` ändern (siehe den vorherigen Abschnitt). Auf `SET TRANSACTION` gehe ich weiter hinten in diesem Kapitel ausführlicher ein.

Die Standardtransaktion geht von einigen impliziten Voraussetzungen aus:

- ✔ Die Datenbank ändert sich im Laufe der Zeit.
- ✔ Vorbeugen ist besser als Reparieren.

Die Standardtransaktion arbeitet im `READ–WRITE`-Modus, in dem Sie die Datenbank ändern können. Die Isolierungsebene ist auf `SERIALIZABLE` gesetzt – die höchstmögliche und damit auch sicherste Isolierungsebene. Die Standardgröße für den Fehlerspeicher (`DIAGNOSTICS SIZE`) ist von Ihrer Implementierung abhängig und geht aus der SQL-Dokumentation Ihres Systems hervor.

Isolierungsebenen

Ideal wäre es, wenn das System Ihre Transaktionen unabhängig voneinander ausführte – und zwar auch dann, wenn sie bei Ihnen gleichzeitig abliefen. Dieses Konzept wird auch als *Isolierung* bezeichnet. In der Praxis ist dieses Ideal leider nicht immer zu erreichen, weil das Leistungsverhalten zu sehr darunter leiden würde. Deshalb stellt sich die Frage: »Wie viel Isolierung wollen Sie haben und welchen Preis in Form von schlechterem Leistungsverhalten sind Sie bereit zu zahlen?«

Die schwächste Isolierungsebene: READ UNCOMMITTED

Die schwächste Isolierungsebene heißt `READ UNCOMMITTED` (unbestätigtes Lesen), mit der ein manchmal problematischer Dirty-Read möglich ist. Ein *Dirty-Read* (schmutziges Lesen) ist eine Situation, in der eine Änderung, die von einem Benutzer vorgenommen worden ist, von einem zweiten Benutzer gelesen werden kann, ehe der erste Benutzer die Änderung mit einem `COMMIT` endgültig abgeschlossen hat.

Dieses Problem tritt auf, wenn der erste Benutzer seine Transaktion abbricht und rückgängig macht. Die folgenden Operationen des zweiten Benutzers basieren jetzt auf einem falschen Wert. Das klassische Beispiel dafür ist eine Anwendung zur Lagerverwaltung. Im Abschnitt *Interaktionsprobleme bei Transaktionen* weiter vorne in diesem Kapitel beschreibe ich ein solches Szenario. Hier ist ein weiteres Beispiel: Ein Benutzer verringert einen Lagerbestand. Ein zweiter Benutzer liest den neuen (niedrigeren) Wert. Der erste Benutzer macht seine Transaktion rückgängig und stellt dabei den ursprünglichen Wert des Lagerbestands wieder her. In der Zwischenzeit bestellt der zweite Benutzer neue Ware, weil er davon ausgeht, dass der Bestand zu niedrig ist, und erreicht damit – wenn er Glück hat, nur – einen massiven Warenüberhang.

Arbeiten Sie nicht mit der Isolierungsebene `READ UNCOMMITTED`, außer Ihnen machen ungenaue Ergebnisse nichts aus.

Sie können `READ UNCOMMITTED` benutzen, wenn Sie nur ungefähre statistische Daten benötigen, beispielsweise:

- ✔ Die maximale Verzögerung beim Ausführen von Aufträgen
- ✔ Das Durchschnittsalter der Verkäufer, die ihre Quoten nicht erreichen
- ✔ Das Durchschnittsalter neuer Mitarbeiter

In vielen dieser Fälle reichen die ungefähren Informationen aus, und die zusätzlichen Kosten (in Form eines schlechteren Leistungsverhaltens), um die Parallelverarbeitung strenger zu regulieren und damit genauere Daten zu erhalten, sind den Aufwand nicht wert.

Die nächsthöhere Isolierungsebene: READ COMMITTED

Die nächsthöhere Isolierungsebene ist `READ COMMITTED` (bestätigtes Lesen). Eine Änderung einer anderen Transaktion ist für Ihre Transaktion erst sichtbar, wenn der andere Benutzer seine Transaktion mit `COMMIT` abgeschlossen hat. Dies ist besser als der vorangegangene Fall,

kann aber immer noch zu einem ernsten Problem führen – dem *nicht reproduzierbaren Lesevorgang*.

Um dieses Phänomen zu veranschaulichen, wollen wir zu dem Lagerbestandsbeispiel zurückkehren.

1. Benutzer A fragt die Datenbank ab, um herauszufinden, wie viel Stück eines bestimmten Artikels auf Lager sind. Die Zahl sei zehn.
2. Fast gleichzeitig startet und beendet Benutzer B eine Transaktion, durch die der Lagerbestand dieses Artikels um zehn Stück auf null reduziert wird.
3. Benutzer A, der gesehen hat, dass zehn Stück des Artikels auf Lager sind, versucht jetzt, fünf davon für einen Auftrag abzubuchen. Es sind jedoch keine fünf Stück mehr auf Lager, weil Benutzer B zugeschlagen hat.

Der ursprüngliche Lesevorgang der ersten Abfrage von Benutzer A ist nicht mehr reproduzierbar. Weil sich die Lagermenge geändert hat, sind alle Annahmen, die auf der ersten Abfrage basieren, nicht länger gültig.

Die Isolierungsebene REPEATABLE READ

Die Isolierungsebene `REPEATABLE READ` (wiederholbares Lesen) garantiert, dass das Problem des nicht wiederholbaren Lesens nicht auftreten kann. Bei dieser Isolierungsebene kann jedoch immer noch ein sogenannter *Phantom-Read* auftreten – ein Problem, das entsteht, wenn sich die Daten, die ein Benutzer liest, als Folge einer anderen Transaktion ändern (wobei die Änderungen aber nicht auf dem Bildschirm angezeigt werden), *während der Benutzer die Daten liest*.

Angenommen, Benutzer A setzt eine Anweisung ab, deren Suchbedingung (die `WHERE`- oder `HAVING`-Klausel) eine Reihe von Zeilen auswählt, und unmittelbar danach führt Benutzer B eine Operation aus, die die Daten in einigen dieser Zeilen ändert, und bestätigt sie. Diese Datenelemente entsprechen zwar der ursprünglichen Suchbedingung von Benutzer A, erfüllen die alten Voraussetzungen aber nicht mehr. Umgekehrt kann es jetzt andere Zeilen geben, die die Suchbedingung ursprünglich nicht erfüllt haben, was aber aufgrund der Änderungen mittlerweile der Fall ist. Benutzer A, dessen Transaktion noch aktiv ist, hat keine Ahnung von den Änderungen, weil sich die Anwendung so verhält, als wäre nichts geschehen. Er gibt eine weitere SQL-Anweisung mit derselben Suchbedingung wie beim ersten Mal ein und erwartet, dass dieselben Zeilen zurückgegeben werden. Stattdessen wird aber die zweite Operation auf anderen Zeilen als zuvor ausgeführt. Verlässliche Ergebnisse gehen in Rauch auf, weggeblasen durch Phantom-Read.

Die Isolierungsebene SERIALIZABLE

Die Isolierungsebene `SERIALIZABLE` ist nicht von den Problemen betroffen, von denen die drei anderen Ebenen heimgesucht werden. Auf dieser Ebene können gleichzeitige Transaktionen ablaufen, ohne sich gegenseitig zu beeinflussen, und die Ergebnisse sind dieselben, als wenn die Transaktionen seriell – eine nach der anderen und nicht parallel – abgelaufen wären. Wenn Sie mit dieser Isolierungsebene arbeiten, können Ihre Transaktionen zwar noch durch Hardware- oder Software-Probleme scheitern, aber Sie müssen sich wenigstens keine Sorgen

wegen der Richtigkeit der Ergebnisse machen, wenn Ihr System einwandfrei funktioniert.

Natürlich hat die höhere Zuverlässigkeit ihren Preis: Das Leistungsverhalten nimmt ab, weshalb Sie wieder vor dem Problem stehen, sich für etwas entscheiden zu müssen. Tabelle 15.1 zeigt die vier Isolierungsebenen und die Probleme, die sie lösen.

Isolierungsebene	Gelöste Probleme
`READ UNCOMMITTED`	Keine
`READ COMMITTED`	Dirty-Read
`REPEATABLE READ`	Dirty-Read nicht wiederholbarer Tabellen-Read
`SERIALIZABLE`	Dirty-Read nicht wiederholbarer Tabellen-Read Phantom-Read

Tabelle 15.1: Isolierungsebenen und gelöste Probleme

Anweisungen mit implizitem Transaktionsbeginn

Bei einigen SQL-Implementierungen müssen Sie eine Transaktion explizit mit einer Anweisung, zum Beispiel `BEGIN` oder `BEGIN TRAN`, beginnen. SQL verlangt dies nicht. Wenn im Moment keine Transaktion aktiv ist und Sie eine Anweisung absetzen, die eine Transaktion aufruft, startet SQL automatisch eine Standardtransaktion. `CREATE TABLE`, `SELECT` und `UPDATE` sind Beispiele für Anweisungen, die im Kontext einer Transaktion ausgeführt werden. Wenn Sie eine dieser Anweisungen ausführen, startet SQL automatisch eine Transaktion.

SET TRANSACTION

Gelegentlich kommt es sicherlich vor, dass Sie nicht mit den Standardeigenschaften einer Transaktion arbeiten wollen. Sie können diese unterschiedlichen Eigenschaften mit der Anweisung `SET TRANSACTION` angeben. Dies muss vor der ersten Anweisung geschehen, die eine Transaktion verlangt. `SET TRANSACTION` gibt Ihnen die Möglichkeit, den Modus, die Isolierungsebene und die Größe des Diagnosebereichs festzulegen.

Wenn Sie alle drei Parameter auf einen Schlag definieren wollen, können Sie das mit einer Anweisung wie der folgenden erledigen:

```
SET TRANSACTION
   READ ONLY,
   ISOLATION LEVEL READ UNCOMMITTED,
   DIAGNOSTICS SIZE 4 ;
```

Damit legen Sie fest, dass die Datenbank nicht geändert werden kann (`READ ONLY`) und Sie mit der niedrigsten und gefährlichsten Isolierungsebene (`READ UNCOMMITTED`) arbeiten wollen. Der Diagnosebereich soll die Größe vier haben. Mit dieser Anweisung belasten Sie die Ressourcen des Systems nur minimal.

Im Gegensatz dazu können Sie auch schreiben:

```
SET TRANSACTION
   READ WRITE,
   ISOLATION LEVEL SERIALIZABLE,
   DIAGNOSTICS SIZE 8 ;
```

Damit legen Sie fest, dass die Datenbank geändert werden darf und Sie mit der höchsten Isolierungsebene arbeiten wollen. Der Diagnosebereich soll die Größe acht haben. Damit belasten Sie die Ressourcen des Systems in höherem Maße. Abhängig von Ihrer Implementierung könnten diese Einstellungen mit denen einer Standardtransaktion übereinstimmen. Natürlich können Sie mit `SET TRANSACTION` auch andere Werte für die Isolierungsebene und den Diagnosebereich festlegen.

Setzen Sie die Isolierungsebene von Transaktionen so hoch wie nötig, aber nicht höher. Es mag vernünftig klingen, die Isolierungsebene aus Sicherheitsgründen immer auf `SERIALIZABLE` zu setzen, aber nicht alle Systeme kommen damit klar. Abhängig von Ihrer Implementierung und Ihren Aufgaben ist dies möglicherweise auch nicht immer erforderlich. Das Leistungsverhalten des Systems kann beträchtlich darunter leiden, wenn Sie immer diese Isolierungsebene wählen. Wenn Sie die Datenbank in Ihrer Transaktion nicht ändern wollen, sollten Sie den Modus beispielsweise auf `READ ONLY` setzen. Reservieren Sie keine Systemressourcen, die Sie nicht unbedingt benötigen.

COMMIT

Obwohl SQL kein Schlüsselwort kennt, um eine Transaktion explizit einzuleiten, gibt es zwei, um sie explizit zu beenden: `COMMIT` und `ROLLBACK`. Verwenden Sie `COMMIT`, wenn Sie mit Ihrer Transaktion zu einem Ende kommen und (gegebenenfalls) die Änderungen, die Sie an der Datenbank vorgenommen haben, dauerhaft speichern wollen. Sie können zusätzlich das optionale Schlüsselwort `WORK` (`COMMIT WORK`) angeben. Sollte es zu einem Fehler in Ihrer Datenbank kommen oder Ihr System ausfallen, während ein `COMMIT` ausgeführt wird, können Sie die Transaktion mit `ROLLBACK` rückgängig machen und von vorn beginnen.

ROLLBACK

Wenn Sie mit Ihrer Transaktion fertig sind, kann es passieren, dass Sie sich dazu entscheiden, die Änderungen an der Datenbank, zu denen es während der Transaktion gekommen ist, nicht dauerhaft werden zu lassen. Genau genommen wollen Sie die Datenbank in den Zustand zurückversetzen, den Sie hatten, bevor Sie mit der Transaktion anfingen. Setzen Sie zu diesem Zweck die Anweisung `ROLLBACK` ein. `ROLLBACK` dient also als Sicherheitsmechanismus.

Selbst wenn das System abstürzt, während ein `ROLLBACK` läuft, können Sie das `ROLLBACK` neu starten, wenn das System wiederhergestellt ist; `ROLLBACK` setzt seine Arbeit fort und versetzt die Datenbank wieder in den Zustand vor der Transaktion.

Datenbankobjekte sperren

Die Isolierungsebene, die entweder standardmäßig oder durch die Anweisung `SET TRANSACTION` eingestellt ist, teilt dem DBMS mit, wie streng es Ihre Arbeit von den Einflüssen der Arbeit anderer Benutzer isolieren soll. Der Hauptschutz, den Ihnen das DBMS vor schädlichen Transaktionen bietet, besteht darin, die Datenbankobjekte zu sperren, mit denen Ihre Anwendung arbeitet. Einige Beispiele hierzu sind:

- ✔ Die Tabellenzeile, auf die Sie zugreifen, wird gesperrt, um andere Benutzer daran zu hindern, auf diesen Datensatz gleichzeitig mit Ihnen zuzugreifen.
- ✔ Eine ganze Tabelle wird gesperrt, wenn Ihre Operation die gesamte Tabelle betrifft.
- ✔ Manchmal ist nur das Lesen der Daten erlaubt, nicht aber das Schreiben. Manchmal ist nur das Schreiben der Daten erlaubt, nicht aber das Lesen.

Jede Implementierung handhabt Sperren auf eigene Weise. Einige Implementierungen sind sicherer als andere, aber die meisten modernen Systeme schützen Ihre Daten vor den schlimmsten Problemen, die beim gleichzeitigen Zugriff auf Daten auftreten können.

Datensicherung

Die Datensicherung (englisch *Backup*) ist eine vorbeugende Schutzmaßnahme, die Ihr DBA regelmäßig ausführen sollte. Alle Komponenten Ihres Systems sollten regelmäßig gesichert werden, wobei das Sicherungsintervall von der Änderungshäufigkeit der Komponenten abhängt. Wenn Ihre Datenbank täglich geändert wird, sollte sie täglich gesichert werden. Auch Ihre Anwendungen, Formulare und Berichte ändern sich, wenn auch weniger häufig. Bei jeder Änderung sollte Ihr DBA die neuen Versionen ebenfalls sichern.

Arbeiten Sie mit mehreren Sicherungsgenerationen. Manchmal wird ein Schaden an der Datenbank erst einige Zeit später bemerkt. Dann müssen Sie möglicherweise auf eine ältere Version zurückgreifen, um eine korrekte Version der Datenbank wiederherzustellen.

Es gibt verschiedene Methoden, um Daten zu sichern:

- ✔ Erstellen Sie mit SQL Sicherungstabellen und kopieren Sie die Daten in diese Tabellen.
- ✔ Benutzen Sie einen von der Implementierung abhängigen Mechanismus, um die komplette Datenbank oder Teile davon zu sichern. Dieser Mechanismus ist im Allgemeinen viel bequemer und effizienter als die Verwendung von SQL.
- ✔ Auch Ihr System kann über einen Mechanismus verfügen, der alles – einschließlich der Datenbanken, Programme, Dokumente, Tabellenkalkulationen, Utilities und Computerspiele – auf einmal sichert. Falls dies der Fall ist, müssen Sie nur darauf achten, dass das Backup häufig genug erfolgt, um die Sicherheit Ihrer Daten zu gewährleisten.

Mit einer ACID-Datenbank arbeiten

Vielleicht haben Sie schon einmal gehört, dass Datenbankentwickler davon sprechen, dass Ihre Datenbanken *ACID* haben sollten. ACID ist die Abkürzung für *Atomarität* (englisch *Atomicity*), *Konsistenz* (englisch *Consistency*), *Isolierung* (englisch *Isolation*) und *Dauerhaftigkeit* (englisch *Durability*). Diese vier Eigenschaften sind notwendig, um eine Datenbank vor Beschädigung zu schützen:

- ✔ **Atomarität:** Datenbanktransaktionen sollten – im klassischen Sinn des Wortes – atomar sein: Die gesamte Transaktion wird als eine unteilbare Einheit behandelt. Entweder wird sie als Ganzes ausgeführt (`COMMIT`) oder die Datenbank wird wieder in den Anfangszustand vor dem Beginn der Transaktion zurückversetzt (`ROLLBACK`).
- ✔ **Konsistenz:** Seltsamerweise ist die Bedeutung von *Konsistenz* nicht konsistent. Sie unterscheidet sich von einer Anwendung zur nächsten. Wenn Sie beispielsweise in einer Bankanwendung einen Geldbetrag von einem Konto auf ein anderes überweisen, sollte der Saldo beider Konten zusammen denselben Wert haben wie zu Beginn der Transaktion. Bei einer anderen Anwendung kann Ihr Kriterium für Konsistenz ganz anders sein.
- ✔ **Isolierung:** Idealerweise sollten Datenbanktransaktionen vollkommen von anderen Transaktionen isoliert sein, die gleichzeitig ausgeführt werden. Wenn die Transaktionen serialisierbar sind, wird eine komplette Isolierung erzielt. Falls in Ihrem System Transaktionen möglichst schnell ausgeführt werden müssen, kann manchmal eine niedrigere Ebene der Isolierung das Leistungsverhalten verbessern.
- ✔ **Dauerhaftigkeit:** Nachdem eine Transaktion abgeschlossen (`COMMIT`) oder rückgängig gemacht (`ROLLBACK`) worden ist, sollten Sie sich darauf verlassen können, dass die Datenbank in einem einwandfreien Zustand ist: richtig bestückt, betriebssicher und mit aktuellen Daten gefüllt. Selbst wenn Ihr System nach einem `COMMIT` – aber bevor die Transaktion auf der Festplatte gespeichert ist – einen Hardware-Absturz erleidet, kann ein robustes DBMS garantieren, dass die Datenbank nach dem Wiederanlaufen in einen einwandfreien Zustand versetzt werden kann.

Speicherpunkte und Untertransaktionen

Idealerweise sollten Transaktionen atomar – also unteilbar im ursprünglichen Sinne – sein. Nun hat die moderne Physik gezeigt, dass auch Atome nicht unteilbar sind, und spätestens seit SQL:1999 sind auch Datenbanktransaktionen nicht wirklich unteilbar. Eine Transaktion lässt sich in mehrere *Untertransaktionen* zerlegen. Jede Untertransaktion wird mit einer `SAVEPOINT`-Anweisung beendet. Diese Anweisung wird zusammen mit der Anweisung `ROLLBACK` verwendet. Vor der Einführung von Speicherpunkten (englisch *Savepoints*; der Punkt in einem Programm, an dem die Anweisung `SAVEPOINT` wirksam wird) konnte die Anweisung `ROLLBACK` nur dazu verwendet werden, eine komplette Transaktion rückgängig zu machen. Jetzt kann die Anweisung dazu verwendet werden, eine Transaktion bis zu einem Speicherpunkt rückgängig zu machen. Welchen Sinn hat das?

Zugegebenermaßen besteht der Hauptzweck der Anweisung ROLLBACK darin, eine Beschädigung der Daten zu verhindern, wenn eine Transaktion durch einen Fehler unterbrochen wird. Natürlich macht es auch keinen Sinn, die Aktionen bis zu einem Speicherpunkt rückgängig zu machen, wenn ein Fehler während der Transaktion aufgetreten ist. Sie sollten die *komplette* Transaktion rückgängig machen, um die Datenbank in den Zustand vor dem Beginn der Transaktion zurückzubringen. Sie könnten auch Gründe dafür haben, nur einen Teil einer Transaktion rückgängig zu machen.

Angenommen, Sie führen eine komplexe Folge von Operationen mit Ihren Daten aus. Auf halbem Wege stellen Sie fest, dass Sie Ergebnisse erhalten, die sich als unproduktiv erweisen. Wenn Sie unmittelbar vor der betreffenden Teiloperation eine SAVEPOINT-Anweisung eingefügt hätten, könnten Sie jetzt zu diesem Speicherpunkt zurückkehren und eine andere Option ausprobieren. Vorausgesetzt, dass sich der Rest Ihres Codes vor dem Speicherpunkt wie gewünscht verhält, funktioniert dieser Ansatz besser, als die laufende Transaktion abzubrechen und ganz von vorn zu starten, nur um einen anderen Weg auszuprobieren.

Mit der folgenden Anweisung können Sie einen Speicherpunkt in den SQL-Code einfügen:

```
SAVEPOINT SpeicherpunktName ;
```

Mit dem folgenden Code können Sie die Ausführung bis zu einem bestimmten Speicherpunkt rückgängig machen:

```
ROLLBACK TO SAVEPOINT SpeicherpunktName;
```

Einige SQL-Implementierungen unterstützen die Anweisung SAVEPOINT möglicherweise nicht. Wenn dies auch für Ihre Implementierung gilt, können Sie SAVEPOINT nicht verwenden.

Einschränkungen innerhalb von Transaktionen

Um die Gültigkeit der Daten in Ihrer Datenbank zu gewährleisten, müssen Sie mehr tun, als nur darauf zu achten, dass die Datentypen der Daten stimmen. Einige Spalten dürfen beispielsweise keine Nullwerte enthalten, während die Werte anderer Spalten innerhalb eines bestimmten Wertebereichs liegen müssen. Diese Arten von Einschränkungen werden in Kapitel 5 beschrieben.

Einschränkungen spielen auch bei Transaktionen eine Rolle, weil es denkbar ist, dass sie Sie daran hindern, bestimmte Aufgaben auszuführen. Angenommen, Sie wollen Daten in eine Tabelle einfügen, die eine Spalte mit einer NOT NULL-Einschränkung enthält. Eine übliche Methode zum Hinzufügen eines Datensatzes zu einer Tabelle besteht darin, zunächst eine leere Zeile zur Tabelle hinzuzufügen und dann erst die Werte einzusetzen. Bei einer NOT NULL-Einschränkung einer Spalte ist dieses Verfahren jedoch nicht anwendbar, weil SQL es nicht zulässt, Zeilen mit Nullwerten zu einer Tabelle hinzuzufügen – und zwar selbst dann, wenn Sie die leeren Felder vor Bestätigung der Transaktion mit Leben füllen. Um dieses Problem zu lösen, gibt Ihnen SQL die Möglichkeit, Einschränkungen entweder als DEFERRABLE oder NOT DEFERRABLE (deutsch *aufschiebbar* beziehungsweise *nicht aufschiebbar*) zu bezeichnen.

Einschränkungen mit dem Attribut NOT DEFERRABLE werden sofort angewendet. Eine aufschiebbare Einschränkung kann wahlweise mit dem Anfangsattribut DEFERRED (deutsch *aufgeschoben*) oder IMMEDIATE (deutsch *sofort*) versehen werden. Wenn eine aufschiebbare Einschränkung auf IMMEDIATE gesetzt wird, funktioniert sie wie eine Einschränkung mit dem Attribut NOT DEFERRABLE. Sie wird sofort angewendet. Wenn eine aufschiebbare Einschränkung auf DEFERRED gesetzt wird, wird ihre Einhaltung nicht erzwungen.

Um leere Datensätze einzufügen oder andere Operationen auszuführen, die gegen Einschränkungen mit dem Attribut DEFERRABLE verstoßen, können Sie die folgende Anweisung benutzen:

```
SET CONSTRAINTS ALL DEFERRED ;
```

Damit werden alle DEFERRABLE-Einschränkungen in den Modus DEFERRED (hinausgeschoben) versetzt. Die NOT DEFERRABLE-Einschränkungen werden von der Anweisung nicht berührt. Wenn Sie alle Operationen ausgeführt haben, die gegen Einschränkungen verstoßen konnten, und wenn die Tabelle sich jetzt in einem Zustand befindet, der nicht mehr gegen diese Einschränkungen verstößt, können Sie die Einschränkungen problemlos wieder reaktivieren:

```
SET CONSTRAINTS ALL IMMEDIATE ;
```

Wenn Sie einen Fehler gemacht haben und Einschränkungen doch verletzt worden sind, erfahren Sie davon, sobald diese Anweisung ausgeführt worden ist.

Wenn Sie DEFERRED-Einschränkungen nicht explizit auf IMMEDIATE setzen, macht SQL dies automatisch für Sie, wenn Sie versuchen, Ihre Transaktion mit COMMIT zu bestätigen. Wenn zu diesem Zeitpunkt noch Einschränkungen verletzt werden, wird die Transaktion nicht bestätigt, sondern SQL meldet Ihnen einen Fehler.

Die Art und Weise, wie SQL mit Einschränkungen umgeht, schützt Sie einerseits davor, ungültige Daten einzugeben (oder keine Daten einzugeben, wo welche gefordert sind), und gibt Ihnen andererseits die Flexibilität, Einschränkungen temporär während einer aktiven Transaktion außer Kraft zu setzen.

Wir wollen ein Lohnabrechnungsbeispiel betrachten, um zu zeigen, warum die Möglichkeit wichtig ist, die Anwendung von Einschränkungen aufzuschieben.

Angenommen, eine Tabelle Mitarbeiter hat die Spalten MitarbNr, MitarbName, AbteilungNr und Gehalt. AbteilungNr ist ein Fremdschlüssel, der auf die Tabelle Abteilung verweist. Außerdem hat die Tabelle Abteilung die Spalten AbteilungNr und Abteilung Name, wobei AbteilungNr der Primärschlüssel ist.

Außerdem soll es eine Tabelle wie Abteilung geben, die zusätzlich über eine Spalte Gehaltssumme verfügt, die die Summe der Gehälter der Mitarbeiter einer jeden Abteilung enthält.

Wenn Ihr DBMS diese Standard-SQL-Funktionalität unterstützt, können Sie ein Äquivalent dieser Tabelle mit der folgenden Sicht erstellen:

```
CREATE VIEW Abt2 AS
   SELECT A.*, SUM(M.Gehalt) AS Gehaltssumme
      FROM Abteilung A, Mitarbeiter M
      WHERE A.AbteilungNr = M.AbteilungNr
      GROUP BY A.AbteilungNr ;
```

Sie können diese Sicht aber auch folgendermaßen definieren:

```
CREATE VIEW Abt3 AS
  SELECT A.*,
     (SELECT SUM(M.Gehalt)
       FROM Mitarbeiter M
       WHERE A.AbteilungNr = M.AbteilungNr) AS Gehaltssumme
     FROM Abteilung A ;
```

Nehmen Sie jetzt aber an, dass Sie aus Gründen der Effizienz die Summe (`SUM`) nicht jedes Mal neu berechnen wollen, wenn Sie auf `Abteilung.Gehaltssumme` verweisen. Stattdessen möchten Sie in der Tabelle `Abteilung` eine echte Spalte mit dem Namen `Gehaltssumme` verwenden und diese Spalte jedes Mal aktualisieren, wenn Sie ein `Gehalt` ändern.

Damit die Spalte `Gehalt` auch immer korrekte Werte enthält, fügen Sie eine Einschränkung in die Tabellendefinition ein:

```
CREATE TABLE Abteilung
x   (AbteilungNr CHAR(5),
   AbteilungName CHAR(20),
   Gehaltssumme DECIMAL(15,2),
   CHECK (Gehaltssumme = (SELECT SUM(Gehalt)
      FROM Mitarbeiter M
      WHERE M.AbteilungNr = Abteilung.AbteilungNr)));
```

Stellen Sie sich jetzt weiterhin vor, dass Sie das `Gehalt` von Mitarbeiter `123` um 100 Euro erhöhen wollen. Sie können dies mit der folgenden `UPDATE`-Anweisung tun:

```
UPDATE Mitarbeiter
   SET Gehalt = Gehalt + 100
   WHERE MitarbNr = '123' ;
```

Und Sie dürfen nicht vergessen, auch die folgende Anweisung auszuführen:

```
UPDATE Abteilung A
SET Gehaltssumme = Gehaltssumme + 100
WHERE A.AbteilungNr = (SELECT M.AbteilungNr
                          FROM Mitarbeiter M
                          WHERE M.MitarbNr = '123') ;
```

(Sie benutzen die Unterabfrage, um auf die `AbteilungNr` von Mitarbeiter `123` zu verweisen.)

Aber Sie haben ein Problem. Einschränkungen werden am Ende einer jeden Anweisung geprüft. Im Prinzip werden *alle* Einschränkungen geprüft. Praktisch prüft eine Implementierung jedoch nur die Einschränkungen, die mit Werten zu tun haben, die durch die Anweisung geändert werden.

Deshalb prüft die Implementierung nach der ersten der vorangegangenen UPDATE-Anweisungen alle Einschränkungen, die auf Werte verweisen, die durch die Anweisung geändert werden. Dazu zählt die Einschränkung, die in der Tabelle Abteilung definiert wird, weil diese Einschränkung auf die Spalte Gehalt der Tabelle Mitarbeiter verweist und die Anweisung UPDATE diese Spalte modifiziert. Nach der ersten UPDATE-Anweisung wird diese Einschränkung verletzt. Sie gehen davon aus, dass die Datenbank, ehe Sie die UPDATE-Anweisung ausführen, korrekt ist und jeder Wert in der Spalte Gehaltssumme der Tabelle Abteilung gleich der Summe der Werte von Gehalt in der entsprechenden Spalte der Tabelle Mitarbeiter ist. Dann erhöht die erste UPDATE-Anweisung einen Gehalt-Wert, und diese Gleichheit ist nicht mehr gegeben. Die zweite UPDATE-Anweisung korrigiert die Situation, und danach erfüllen die Datenbankwerte die Einschränkung wieder. Aber zwischen den beiden UPDATE-Anweisungen wird die Einschränkung verletzt.

Die Anweisung SET CONSTRAINTS DEFERRED hat den Zweck, alle Einschränkungen oder einige bestimmte Einschränkungen temporär außer Kraft zu setzen. Die Anwendung der Einschränkungen wird aufgeschoben, bis Sie entweder die Anweisung SET CONSTRAINTS ALL IMMEDIATE oder ein COMMIT oder ROLLBACK ausführen. Deshalb sollten Sie die beiden vorangegangenen UPDATE-Anweisungen durch zwei SET CONSTRAINTS-Anweisungen einschließen. Der entsprechende Code sieht so aus:

```
SET CONSTRAINTS DEFERRED ;
UPDATE Mitarbeiter
   SET Gehalt = Gehalt + 100
   WHERE MitarbNr = '123' ;
UPDATE Abteilung A
   SET Gehaltssumme = Gehaltssumme + 100
   WHERE A.AbteilungNr = (SELECT M.AbteilungNr
               FROM Mitarbeiter M
      WHERE M.MitarbNr = '123') ;
SET CONSTRAINTS IMMEDIATE ;
```

Diese Prozedur schiebt alle Einschränkungen auf. Wenn Sie eine neue Zeile in Abteilung einfügen, werden die Primärschlüssel nicht geprüft. Sie haben damit einen Schutz aufgehoben, auf den Sie möglicherweise nicht verzichten möchten. Deshalb ist es besser, die Einschränkung oder Einschränkungen festzulegen, die Sie aufschieben möchten. Um dies tun zu können, müssen Sie den Einschränkungen Namen geben, wenn Sie sie anlegen:

```
CREATE TABLE Abteilung
   (AbteilungNr CHAR(5),
   AbteilungName CHAR(20),
   Gehaltssumme DECIMAL(15,2),
   CONSTRAINT GehaltCon
```

```
   CHECK (Gehaltssumme = SELECT SUM(Gehalt)
                  FROM Mitarbeiter M WHERE
                  M.AbteilungNr = Abteilung.AbteilungNr)) ;
```

Dann können Sie die Einschränkungen individuell ansprechen:

```
SET CONSTRAINTS GehaltCon DEFFERRED;
UPDATE Mitarbeiter
   SET Gehalt = Gehalt + 100
   WHERE MitarbNr = '123' ;
UPDATE Abteilung A
   SET Gehaltssumme = Gehaltssumme + 100
   WHERE A.AbteilungNr = (SELECT M.AbteilungNr
           FROM Mitarbeiter M
           WHERE M.MitarbNr = '123') ;
SET CONSTRAINTS GehaltCon IMMEDIATE;
```

Wenn Sie in der Anweisung `CREATE` keinen Namen für eine Einschränkung festlegen, erzeugt SQL intern einen Namen. Dieser interne Name findet sich dann in den Tabellen mit den Schemainformationen (Katalogtabellen) wieder. Es ist aber unkomplizierter, wenn Sie den Namen explizit angeben.

Stellen Sie sich jetzt vor, dass Sie in der zweiten `UPDATE`-Anweisung aus Versehen eine Gehaltserhöhung von 1000 Euro angegeben haben. Dies ist während der `UPDATE`-Anweisung erlaubt, weil die Einschränkung aufgeschoben worden ist. Aber wenn Sie die Anweisung `SET CONSTRAINTS IMMEDIATE` ausführen, wird die entsprechende Einschränkung geprüft. Wenn dieser Vorgang fehlschlägt, meldet die Anweisung `SET CONSTRAINTS` eine Ausnahme. Wenn Sie statt der Anweisung `SET CONSTRAINTS GehaltCon IMMEDIATE` eine `COMMIT`-Anweisung ausführen und eine Verletzung der Einschränkung festgestellt wird, kommt es nicht zu einem `COMMIT`, sondern zu einem `ROLLBACK`.

Fazit: Sie können Einschränkungen somit nur *innerhalb* einer Transaktion aufschieben. Wenn die Transaktion entweder mit einem `ROLLBACK` oder einem `COMMIT` beendet wird, werden die Einschränkungen wieder in Kraft gesetzt und geprüft. Die Möglichkeit, Einschränkungen aufzuschieben, kann innerhalb einer Transaktion benutzt werden. Sie ist kein Mechanismus, mit dem Sie Daten in die Datenbank einfügen können, die gegen Einschränkungen verstoßen.

SQL in Anwendungen benutzen

16

In diesem Kapitel

- SQL in einer Anwendung benutzen
- SQL mit prozeduralen Sprachen kombinieren
- Zwischensprachliche Inkompatibilitäten vermeiden
- SQL in prozeduralen Code einbetten
- SQL-Module von prozeduralem Code aus aufrufen
- SQL von einem RAD-Werkzeug aus aufrufen

In den vorangegangenen Kapiteln dieses Buches habe ich SQL-Anweisungen meistens isoliert behandelt. Beispielsweise habe ich Fragen über Daten gestellt und dann SQL-Abfragen erstellt, um die Antworten zu erhalten. Auf diese Weise kann man zwar lernen, wie SQL funktioniert, aber nicht, wie es praktisch eingesetzt wird.

Obwohl die SQL-Syntax eine gewisse Ähnlichkeit mit normalem Englisch hat, ist die Sprache nicht leicht zu lernen. Die weitaus meisten Computerbenutzer beherrschen SQL nicht. Daran wird sich sehr wahrscheinlich auch in Zukunft nichts ändern, selbst wenn dieses Buch ein großer Erfolg wird. Wenn es um Datenbankabfragen geht, setzt sich der normale Benutzer wahrscheinlich nicht an sein Terminal, um ein SELECT einzugeben und so die Antwort zu ermitteln. Die typischen SQL-Anwender sind Systemanalytiker und Anwendungsentwickler, die ihren Lebensunterhalt nicht damit verdienen, ad hoc Datenbankabfragen an einem Terminal einzugeben. Sie sind es, die Anwendungen entwickeln, die diese Abfragen ausführen.

Wenn Sie eine bestimmte Operation wiederholt ausführen wollen, sollten Sie sie nicht jedes Mal neu an der Konsole eingeben müssen. Schreiben Sie für diesen Zweck eine Anwendung und führen Sie sie dann so oft aus, wie Sie wollen. SQL kann Teil einer Anwendung sein, funktioniert in diesem Fall aber etwas anders als im interaktiven Modus.

SQL in einer Anwendung

In Kapitel 2 wird SQL als unvollständige Programmiersprache vorgestellt. Um SQL in einer Anwendung verwenden zu können, müssen Sie es mit einer *prozeduralen* Sprache, zum Beispiel Visual Basic, Java, C, C++, C# oder COBOL kombinieren. Aufgrund seiner Struktur hat SQL einige Stärken und Schwächen. Prozedurale Sprachen sind anders strukturiert und haben *andere* Stärken und Schwächen.

Glücklicherweise gleichen die Stärken von SQL viele Schwächen der prozeduralen Sprachen aus und umgekehrt. Wenn Sie die beiden kombinieren, sind Sie in der Lage, leistungsstarke Anwendungen zu entwickeln. In jüngster Zeit sind objektorientierte Werkzeuge zur schnellen Anwendungsentwicklung mit grafischen Benutzeroberflächen, sogenannte *RAD-Werkzeuge (Rapid Application Development)*, zum Beispiel Microsoft Visual Studio und das Open-Source-Produkt Eclipse, auf den Markt gekommen. Diese Produkte betten SQL-Code in Anwendungen ein, die durch die Bearbeitung von Objekten statt durch das Schreiben von prozeduralem Code entwickelt werden.

Nach dem Sternchen Ausschau halten

In den vorangegangenen Kapiteln habe ich bei der Behandlung von interaktivem SQL das Sternchen (*) als Abkürzung für »alle Spalten in der Tabelle« benutzt. Wenn eine Tabelle viele Spalten enthält, können Sie mit dem Sternchen viel Schreibarbeit sparen. Wenn Sie SQL in einer Anwendung benutzen, ist damit jedoch eine Gefahr verbunden. Nachdem Ihre Anwendung fertiggestellt worden ist, fügen Sie oder jemand anders Spalten zu einer Tabelle hinzu oder löschen Spalten. Dies ändert natürlich die Bedeutung von »alle Spalten«. Wenn Ihre Anwendung mit einem Sternchen arbeitet, können Sie plötzlich andere Ergebnisse erhalten, als Sie ursprünglich beabsichtigt haben.

Werden solche Veränderungen an Tabellen vorgenommen, wirkt sich dies erst dann auf die bestehenden Programme aus, wenn sie neu kompiliert werden müssen, um einen Fehler zu beheben oder Änderungen zu übernehmen – was erst Monate nach der Änderung geschehen kann. In diesem Fall verarbeitet die Anweisung mit dem Platzhalterzeichen (*) auch alle neuen Spalten. Dies kann dazu führen, dass die Anwendung trotz Fehlerbehebung (oder anderer Änderungen am Programmcode) nicht mehr korrekt arbeitet – eine Situation, die einen Debugging-Albtraum auslösen kann.

Aus Sicherheitsgründen sollten Sie deshalb in einer Anwendung alle Spalten explizit mit dem Namen statt mit einem Sternchen angeben.

Stärken und Schwächen von SQL

Die Stärke von SQL liegt im Abrufen von Daten. Wenn wichtige Informationen irgendwo in einer Datenbank mit einer oder mehreren Tabellen vergraben sind, liefert Ihnen SQL die Werkzeuge, mit denen Sie diese Informationen ausgraben können. Sie brauchen die Reihenfolge der Zeilen oder Spalten in einer Tabelle nicht zu kennen, weil SQL nicht mit einzelnen Zeilen oder Spalten arbeitet. Die Transaktionsverarbeitung von SQL sorgt dafür, dass Ihre Operationen nicht von anderen Benutzern beeinflusst werden, die gleichzeitig mit Ihnen auf dieselbe Tabelle zugreifen.

Eine Hauptschwäche von SQL ist seine rudimentäre Benutzeroberfläche. Es enthält keine Einrichtungen zur Formatierung von Bildschirmausgabe und Berichten. Die Oberfläche nimmt Befehlszeilen über die Tastatur entgegen und sendet die Ergebnisse zeilenweise wieder an das Terminal zurück.

Manchmal ist die Stärke in einem Zusammenhang eine Schwäche in einem anderen. Eine Stärke von SQL ist die Fähigkeit, auf einmal mit einer kompletten Tabelle umgehen zu können. Dabei spielt es keine Rolle, ob die Tabelle eine, hundert oder hunderttausend Zeilen enthält; eine einzige SELECT-Anweisung kann die gewünschten Daten herbeiholen. SQL kann nicht direkt auf einzelne Zeilen einer Tabelle zugreifen, was manchmal jedoch notwendig ist. In solchen Fällen können Sie auf einen SQL-Cursor zurückgreifen (siehe Kapitel 19) oder eine prozedurale Host-Sprache verwenden.

Stärken und Schwächen prozeduraler Sprachen

Im Gegensatz zu SQL sind prozedurale Sprachen entworfen worden, um mit genau einer Zeile zu arbeiten. Dadurch können Entwickler genau steuern, wie eine Tabelle verarbeitet wird. Dieser hohe Grad an Kontrolle ist eine Stärke der prozeduralen Sprachen. Diese Möglichkeit ist jedoch mit einer Schwäche verbunden: Ein Anwendungsentwickler muss genau wissen, wie die Daten in den Tabellen gespeichert werden. Die Reihenfolge der Spalten und Zeilen ist wichtig und muss berücksichtigt werden.

Weil Anwendungen in prozeduralen Sprachen Schritt für Schritt ausgeführt werden, eignen sie sich gut, um benutzerfreundliche Bildschirmformulare für die Dateneingabe und die Datenausgabe zu erstellen. Weiterhin können Sie anspruchsvolle Berichte mit einem hoch entwickelten Layout erstellen.

Probleme bei der Kombination von SQL mit prozeduralen Sprachen

Wenn man SQL mit prozeduralen Sprachen kombiniert, sollte man versuchen, die Schwächen der einen Sprache durch die Stärken der anderen auszugleichen. So wertvoll eine solche Kombination auch sein mag, es müssen doch einige Hürden gemeistert werden, bevor sie in die Praxis umgesetzt werden kann.

Die verschiedenen Arbeitsweisen

Ein großes Problem bei der Kombination von SQL mit einer prozeduralen Sprache besteht darin, dass SQL tabellenorientiert, prozedurale Sprachen hingegen zeilenorientiert arbeiten. Manchmal können Sie dieses Problem einfach dadurch lösen, dass Sie tabellenorientierte und zeilenorientierte Operationen getrennt mit dem passenden Werkzeug ausführen.

Wenn Sie aber in einer Tabelle bedingungsabhängig nach Datensätzen suchen und die gefundenen dann weiterverarbeiten wollen, können Probleme auftreten. Dieser Vorgang verlangt sowohl die Abfragestärke von SQL als auch die Verzweigungsmöglichkeiten einer prozeduralen Sprache. Wenn Sie SQL an strategischen Stellen in ein Programm einer prozeduralen Sprache einbetten, können Sie beide Stärken kombinieren. (Sie finden mehr Informationen zu diesem Thema weiter hinten in diesem Kapitel im Abschnitt *Eingebettetes SQL*.)

Inkompatibilität von Datentypen

Ein weiteres Hindernis bei der reibungslosen Integration von SQL in eine prozedurale Sprache besteht darin, dass sich die Datentypen von SQL und den wichtigsten prozeduralen Sprachen unterscheiden. Dieser Umstand sollte Sie nicht überraschen, weil die Datentypen, die Sie

für eine prozedurale Sprache festlegen, in einer anderen prozeduralen Sprache nicht benutzt werden können.

Egal wo man hinschaut, aber Datentypen sind nun einmal nicht sprachübergreifend standardisiert. In den SQL-Versionen vor SQL-92 war die Datentyp-Inkompatibilität ein ziemlich großes Problem. Mit SQL-92 wurde die Anweisung `CAST` eingeführt, um dieses Problem zu lösen. Kapitel 9 beschreibt, wie Sie mit `CAST` die Datentypen von Datenelementen einer prozeduralen Sprache in SQL-kompatible Datentypen umwandeln können, solange die Datenelemente selbst mit dem neuen Datentyp kompatibel sind.

SQL in prozedurale Sprachen einbinden

Obwohl es bei der Integration von SQL in prozedurale Sprachen viele Schwierigkeiten gibt, ist dies doch durchaus möglich. Oft *müssen* Sie sogar SQL in eine prozedurale Sprache integrieren, um das gewünschte Ergebnis zu erhalten. Glücklicherweise existiert eine Vielzahl von Möglichkeiten, um SQL in eine prozedurale Sprache einzubinden. Drei dieser Methoden – eingebettetes SQL, die SQL-Modulsprache und RAD-Werkzeuge – werde ich in den folgenden Abschnitten genauer behandeln.

Eingebettetes SQL

Die gebräuchlichste Methode, SQL mit prozeduralen Sprachen zu kombinieren, wird als *eingebettetes SQL* bezeichnet. Wenn Sie wissen wollen, wie eingebettetes SQL arbeitet, schauen Sie sich den Namen an. Er sagt es bereits: SQL-Anweisungen werden an der Stelle in ein prozedurales Programm eingefügt, an der sie benötigt werden.

Sie können sich sicherlich vorstellen, dass solche SQL-Anweisungen, die plötzlich mitten in einem C-Programm auftauchen, eine Herausforderung für einen Compiler sind, der auf so etwas nicht vorbereitet ist. Deshalb werden Programme mit eingebettetem SQL-Code normalerweise von einem *Präprozessor* verarbeitet, ehe sie kompiliert oder interpretiert werden. Der Präprozessor wird durch sogenannte `EXEC SQL`-Direktiven darauf hingewiesen, dass der folgende Code in SQL geschrieben ist.

Schauen Sie sich als Beispiel für eingebettetes SQL ein Programm an, das in Oracles Version der Sprache C, Pro*C, geschrieben ist. Das Programm, das auf die Mitarbeitertabelle einer Firma zugreift, fragt den Benutzer nach dem Namen eines Mitarbeiters und zeigt dann sein Gehalt und seine Provision an. Danach fordert es den Benutzer auf, neue Daten für das Gehalt und die Provision einzugeben, und aktualisiert dann die Mitarbeitertabelle:

```
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR Benutzer[20];
   VARCHAR Kennwort[20];
   VARCHAR Mitarbeitername[10];
   FLOAT Gehalt, Provision;
   SHORT GehaltInd, ProvisionInd;
```

```
EXEC SQL END DECLARE SECTION;
main()
{
   int sret;          /* Return-Code */
   /* Log in */
   strcpy(Benutzer.arr,"FRED");  /* Benutzernamen kopieren */
   Benutzer.len=strlen(Benutzer.arr);
   strcpy(Kennwort.arr,"TOWER");   /* Kennwort kopieren */
   Kennwort.len=strlen(Kennwort.arr);
   EXEC SQL WHENEVER SQLERROR STOP;
   EXEC SQL WHENEVER NOT FOUND STOP;
EXEC SQL CONNECT :Benutzer;
printf("Mit Benutzer %s verbunden. \n",Benutzer.arr);
   printf("Geben Sie den Namen des Mitarbeiters ein:  ");
   scanf("%s",Mitarbeitername.arr);
   Mitarbeitername.len=strlen(Mitarbeitername.arr);
   EXEC SQL SELECT GEHALT,COMM INTO :Gehalt,:Provision
               FROM MITARBEITER
               WHERE ENAME=:Mitarbeitername;
   printf("Employee: %s Gehalt: %6.2f Provision: %6.2f \n",
           Mitarbeitername.arr, Gehalt, Provision);
   printf("Geben Sie das neue Gehalt ein: ");
   sret=scanf("%f",&Gehalt);
   Gehalt_ind = 0;
   if (sret == EOF !! sret == 0) /* Indikator setzen */
        GehaltInd =-1; /* Indikator für NULL setzen */
   printf("Geben Sie die neue Provision ein: ");
   sret=scanf("%f",&provision);
   ProvisionInd = 0;            /* Indikator setzen*/
   if (sret == EOF !! sret == 0)
        ProvisionInd=-1; /* Indikator auf NULL setzen */
   EXEC SQL UPDATE EMPLOY
               SET GEHALT=:Gehalt:Gehalt_ind
               SET COMM=:Provision:Provision_ind
               WHERE ENAME=:Mitarbeitername;
printf("Mitarbeiter %s geändert. \n",Mitarbeitername.arr);
   EXEC SQL COMMIT WORK;
   exit(0);
}
```

Sie müssen kein C-Experte sein, um Zweck und Ablauf dieses Programms zu verstehen. Hier habe ich schrittweise zusammengefasst, wie die Anweisungen arbeiten:

1. SQL deklariert Host-Variablen.
2. Der C-Code kontrolliert den Anmeldevorgang des Benutzers.
3. SQL richtet die Fehlerbehandlung ein und stellt die Verbindung zur Datenbank her.

4. Der C-Code fragt den Benutzer nach dem Namen eines Mitarbeiters und legt ihn in einer Variablen ab.
5. Eine SQL-Anweisung vom Typ `SELECT` liest das Gehalt und die Provision dieses Mitarbeiters aus der Mitarbeitertabelle und legt die Daten in den Host-Variablen `:Gehalt` und `:Provision` ab.
6. C übernimmt die Ablaufsteuerung wieder und zeigt den Namen des Mitarbeiters, das Gehalt und die Provision an und fragt nach neuen Werten für Gehalt und Provision. Es prüft, ob eine Eingabe gemacht worden ist, und setzt einen Indikator, falls das nicht der Fall gewesen ist.
7. SQL aktualisiert die Datenbank mit den neuen Werten.
8. Der C-Code zeigt an, dass der Vorgang abgeschlossen wurde.
9. SQL führt ein `COMMIT` aus, und C beendet das Programm.

Weil Sie einen Präprozessor verwenden, können Sie Befehle zweier Sprachen mischen. Der Präprozessor trennt die SQL-Anweisungen von den Befehlen der Host-Sprache, indem die SQL-Anweisungen in eine separate externe Routine geschrieben werden. Jede SQL-Anweisung wird in der Host-Sprache durch einen Aufruf (`CALL`) der entsprechenden externen Routine ersetzt. Danach kann das Programm kompiliert werden.

Wie der SQL-Teil an die Datenbank weitergeleitet wird, ist von Ihrer Implementierung abhängig. Als Anwendungsentwickler brauchen Sie sich darüber keine Gedanken zu machen. Der Präprozessor kümmert sich darum. Sie *sollten* jedoch auf einige Dinge achten, die es bei interaktivem SQL nicht gibt – etwa Host-Variablen und inkompatible Datentypen.

Host-Variablen deklarieren

Zwischen dem Programm, das in der Host-Sprache vorliegt, und den SQL-Segmenten müssen Informationen ausgetauscht werden. Zu diesem Zweck werden sogenannte *Host-Variablen* verwendet. Damit SQL die Host-Variablen erkennt, müssen sie deklariert werden, bevor Sie sie einsetzen können. Die Deklaration erfolgt in einem Deklarationssegment, das vor dem Programmsegment steht. Das Deklarationssegment wird durch die folgende Direktive eingeleitet:

```
EXEC SQL BEGIN DECLARE SECTION ;
```

Das Ende des Deklarationssegments wird folgendermaßen gekennzeichnet:

```
EXEC SQL END DECLARE SECTION ;
```

Vor jeder SQL-Anweisung muss die SQL-Direktive `EXEC` stehen. Das Ende eines SQL-Segments kann, muss aber nicht durch eine abschließende Direktive gekennzeichnet sein. In COBOL lautet diese abschließende Direktive "`END-EXEC`", in C ist es ein Semikolon.

Datentypen umwandeln

In Abhängigkeit von der Kompatibilität der Datentypen, die von der Host-Sprache und von SQL unterstützt werden, kann es notwendig sein, bestimmte Typen mit einem `CAST` umzuwandeln. Sie können Host-Variablen benutzen, die im Deklarationsabschnitt (`DECLARE SECTION`) deklariert worden sind. Denken Sie daran, vor den Namen einer Host-Variablen einen Doppelpunkt (`:`) zu setzen, wenn Sie ihn in SQL-Anweisungen benutzen:

```
INSERT INTO Lebensmittel
   (Lebensmittelname, Kalorien, Eiweiß, Fett, Kohlenhydrate)
   VALUES
   (:Lebensmittelname, :Kalorien, :Eiweiß, :Fett,
    :Kohlenhydrate)
```

Die SQL-Modulsprache

Die *SQL-Modulsprache* stellt eine weitere Methode dar, um SQL mit einer prozeduralen Programmiersprache zu benutzen. Dabei fügen Sie alle SQL-Anweisungen explizit in ein separates SQL-Modul ein.

Ein SQL-Modul ist einfach eine Liste mit SQL-Anweisungen. Jede SQL-Anweisung wird in eine *SQL-Prozedur* eingeschlossen und durch einen Prozedurnamen sowie die Anzahl und Typen der Parameter eingeleitet.

Jede SQL-Prozedur enthält eine einzige SQL-Anweisung. SQL-Prozeduren werden von dem Host-Programm an den Stellen aufgerufen, an denen eine SQL-Anweisung ausgeführt werden soll. SQL-Prozeduren werden wie Unterprogramme der Host-Sprache aufgerufen.

Im Wesentlichen sind ein SQL-Modul und das zugehörige Host-Programm nichts anderes als die manuell codierte Version des Ergebnisses, das der SQL-Präprozessor aus eingebetteten SQL-Anweisungen ableitet.

Man setzt viel häufiger eingebetteten SQL-Code als die Modulsprache ein. Die meisten Anbieter stellen zwar eine Form von Modulsprache zur Verfügung, heben dies aber nicht in ihrer Dokumentation hervor. Die Modulsprache hat mehrere Vorteile:

- ✔ **SQL-Programmierer müssen keine Experten in einer prozeduralen Sprache sein.** Weil der SQL-Code vollkommen von der prozeduralen Sprache getrennt ist, können Sie die besten verfügbaren SQL-Programmierer engagieren, um Ihre SQL-Module zu entwickeln, und zwar unabhängig davon, ob die Programmierer Ihre prozedurale Sprache beherrschen oder nicht. Sie können sogar die Entscheidung aufschieben, welche prozedurale Sprache Sie benutzen wollen, bis Ihre SQL-Module fertig geschrieben und getestet sind.
- ✔ **Sie können die besten verfügbaren Programmierer in Ihrer prozeduralen Sprache engagieren, selbst wenn diese nichts über SQL wissen.** Es leuchtet ein, dass sich Ihre Experten auf dem Gebiet der prozeduralen Sprachen nicht mit SQL abzuquälen brauchen, wenn Ihre SQL-Experten keine Profis im Bereich der prozeduralen Sprache sein müssen.

✔ **SQL wird nicht mit dem prozeduralen Code vermischt, weshalb der Debugger Ihrer prozeduralen Sprache problemlos funktioniert.** Sie sparen dadurch beträchtliche Entwicklungszeit.

Um es noch einmal zu wiederholen: Was aus der einen Perspektive wie ein Vorteil aussieht, kann sich aus einer anderen nachteilig auswirken. Weil die SQL-Module von dem prozeduralen Code getrennt werden, können Sie die Ablauflogik des Programms nicht so leicht nachvollziehen wie bei eingebettetem SQL.

Moduldeklarationen

Die Syntax für die Deklaration eines Moduls lautet:

```
MODULE [Modulname]
   [NAMES ARE Zeichensatzname]
   LANGUAGE {ADA | C | COBOL | FORTRAN | MUMPS | PASCAL | PLI | SQL}
   [SCHEMA Schemaname]
   [AUTHORIZATION AutorisierungsID]
   [Deklarationen temporärer Tabellen...]
   [Deklarationen von Cursorn...]
   [Deklarationen dynamischer Cursor...]
   Prozeduren...
```

Die eckigen Klammern deuten an, dass der Modulname optional ist. Dennoch sollten Sie Module benennen, um mögliche Verwechslungen zu vermeiden.

Die optionale Klausel `NAMES ARE` gibt einen Zeichensatz an. Ohne diese Klausel wird der Standard-SQL-Zeichensatz Ihrer Implementierung benutzt. Die Klausel `LANGUAGE` teilt dem Modul mit, von welcher Sprache aus es aufgerufen wird. Der Compiler muss die aufrufende Sprache kennen, um den Code entsprechend der Konventionen für Prozedur- und Funktionsaufrufe der jeweiligen Sprache generieren zu können, damit die SQL-Anweisungen für das aufrufende Programm wie Unterprogramme der Host-Sprache aussehen.

Die Klauseln `SCHEMA` und `AUTHORIZATION` sind zwar beide als optional gekennzeichnet, aber Sie müssen wenigstens eine von beiden festlegen. Die Klausel `SCHEMA` gibt das Standardschema an, und die Klausel `AUTHORIZATION` legt den *Autorisierungsbezeichner* fest, der angibt, über welche Rechte Sie verfügen. Wenn Sie die Klausel weglassen, leitet das DBMS die Rechte des Moduls aus den Rechten ab, die mit Ihrer Sitzung verbunden sind. Wenn Sie nicht über das Recht verfügen, die Operation ablaufen zu lassen, die in Ihrer Prozedur festgelegt ist, wird Ihre Prozedur nicht ausgeführt.

Falls Ihre Prozedur temporäre Tabellen benötigt, deklarieren Sie diese in der entsprechenden Klausel. Außerdem müssen Sie Cursor und dynamische Cursor vor den Prozeduren deklarieren, die damit arbeiten. Es ist zulässig, einen Cursor nach dem Start einer Prozedur zu deklarieren, wenn sie den Cursor nicht benutzt. Diese Vorgehensweise kann sinnvoll sein, wenn ein Cursor erst in späteren Prozeduren benutzt werden soll. Cursor werden detailliert in Kapitel 19 behandelt.

Modulprozeduren

In dem Modul folgen nach den Deklarationen die Prozeduren, die funktionellen Teile des Moduls. Eine Prozedur eines SQL-Moduls besteht aus einem Namen, aus Parameterdeklarationen und aus einer ausführbaren SQL-Anweisung. Ein prozedurales Programm ruft die Prozedur mit ihrem Namen auf und übergibt dabei mittels der deklarierten Parameter Werte an die Funktion. Die Syntax einer Prozedur lautet:

```
PROCEDURE Prozedurname
   (Parameterdeklaration [, Parameterdeklaration ]... )
   SQL–Anweisung ;
   [SQL–Anweisungen] ;
```

Die Parameterdeklaration hat die Form

```
Parametername Datentyp
```

oder

```
SQLSTATE
```

Ein Parameter kann ein Eingabeparameter, ein Ausgabeparameter oder beides sein. `SQLSTATE` ist ein Statusparameter, der zur Übermittlung von Fehlermeldungen dient. Wenn Sie sich tiefer mit Parametern beschäftigen wollen, lesen Sie in Kapitel 21 nach.

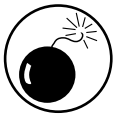
Objektorientierte RAD-Werkzeuge

Mit modernen RAD-Werkzeugen können Sie ausgefeilte Anwendungen entwickeln, ohne eine einzige Codezeile in C++, C#, Python, Java oder einer anderen prozeduralen Sprache schreiben zu müssen. Stattdessen wählen Sie Objekte aus einer Bibliothek und positionieren sie an passenden Stellen auf dem Bildschirm.

Es gibt eine Reihe von Standardobjekten, die bestimmte Eigenschaften haben, mit bestimmten, objektspezifischen Ereignissen verbunden sind und eine Reihe bestimmter Methoden ausführen können. *Methoden* sind Prozeduren, die in einer prozeduralen Sprache speziell für ein bestimmtes Objekt geschrieben worden sind. Es ist jedoch möglich, nützliche Anwendungen zu entwickeln, ohne dafür Methoden zu schreiben.

Obwohl Sie komplexe Anwendungen ohne eine prozedurale Sprache entwickeln können, werden Sie wahrscheinlich früher oder später SQL brauchen. SQL verfügt über eine Bandbreite an Ausdrucksmöglichkeiten, die im Objektparadigma nur sehr schwer oder nicht nachempfunden werden können. Deshalb verfügen komplette RAD-Werkzeuge über einen Mechanismus, um SQL-Befehle in objektorientierte Anwendungen zu integrieren. *Microsoft Visual Studio* ist ein Beispiel für eine solche objektorientierte Entwicklungsumgebung mit SQL-Fähigkeit. *Microsoft Access* ist eine weitere Entwicklungsumgebung für Anwendungen, in der Sie SQL zusammen mit der prozeduralen Sprache VBA verwenden können.

Kapitel 4 zeigt, wie Sie Datenbanktabellen mit Access erstellen. Damit sind die Fähigkeiten von Access jedoch noch nicht erschöpft. Access ist ein Werkzeug, und sein primärer Zweck besteht darin, Anwendungen zu entwickeln, die Daten in Datenbanktabellen verarbeiten. Der Entwickler positioniert Objekte auf Formularen und passt dann die Objekte an seine Anwendung an, indem er deren Eigenschaften abändert, ihre Reaktion auf bestimmte Ereignisse festlegt und möglicherweise einige Methoden schreibt. Sie können die Formulare und Objekte mit VBA-Code bearbeiten. Dieser Code wiederum kann eingebettetes SQL enthalten.



Auch wenn RAD-Werkzeuge, beispielsweise Access, in kurzer Zeit qualitativ hochwertige Anwendungen liefern können, sind sie normalerweise auf eine spezielle oder eine kleine Anzahl von Plattformen beschränkt. Access beispielsweise läuft nur unter Microsoft Windows. Denken Sie daran, wenn Sie Ihre Anwendung auf andere Plattformen portieren müssen.

RAD-Werkzeuge wie Access markieren den Anfang einer Entwicklung, bei der der relationale und der objektorientierte Datenbankentwurf zusammenwachsen. Die strukturellen Stärken des relationalen Entwurfs und von SQL werden dabei überleben. Sie werden durch die schnellen und verhältnismäßig fehlerfreien Entwicklungswerkzeuge ergänzt werden, die ihren Ursprung in der objektorientierten Programmierung haben.

SQL mit Microsoft Access verwenden

Die Zielgruppe von Microsoft Access sind Personen, die ohne große Programmierkenntnisse relativ einfache Anwendungen entwerfen wollen. Wenn diese Beschreibung auf Sie zutrifft, sollten Sie sich *Access für Dummies* (erschienen bei Wiley-VCH) als Referenz ins Regal packen. Sowohl die prozedurale Sprache VBA *(Visual Basic für Anwendungen)* als auch SQL sind zwar Bestandteile von Access, aber weder in der Werbung noch in der Dokumentation wird besonders darauf verwiesen. Seien Sie sich aber darüber im Klaren, dass SQL nicht vollständig in Access implementiert ist und dass Sie beinahe über die detektivischen Kenntnisse eines Sherlock Holmes verfügen müssen, um es zu finden.

In Kapitel 3 gehe ich auf die drei SQL-Komponenten Data Definition Language, Data Manipulation Language und Data Control Language ein. Die Teilmenge von SQL, die Sie in Access finden, richtet nur die Data Manipulation Language ein. Das Anlegen Ihrer Tabellen müssen Sie mit dem RAD-Werkzeug erledigen, das ich in Kapitel 4 beschreibe. Dasselbe gilt für Sicherheitsfunktionen, mit denen ich mich in Kapitel 14 beschäftige.

Um einen Blick auf Access-SQL werfen zu können, müssen wir uns durch die Hintertür einschleichen. Lassen Sie uns als Beispiel eine Datenbank der fiktiven Oregon Lunar Society nehmen. Diese nicht kommerzielle Forschungsgesellschaft hat verschiedene Teams, von denen eines das Mondbasis-Forschungsteam (MBFT) ist. Es tauchte die Frage auf, zu welchen wissenschaftlichen Themen die Mitglieder des Teams bereits Abhandlungen geschrieben haben. Mit dem Abfragegenerator von Access ist eine Abfrage formuliert worden, um die gewünschten Daten zu erhalten. Diese Abfrage, die in Abbildung 16.1 dargestellt wird, holt sich mit den Zwischentabellen `AutorForschung` und `AutorAbhandlung` die Daten aus den Tabellen

Forschungsteams, Autoren und Abhandlungen. Die Zwischentabellen werden benötigt, um die Viele-zu-viele-Beziehungen aufzubrechen.

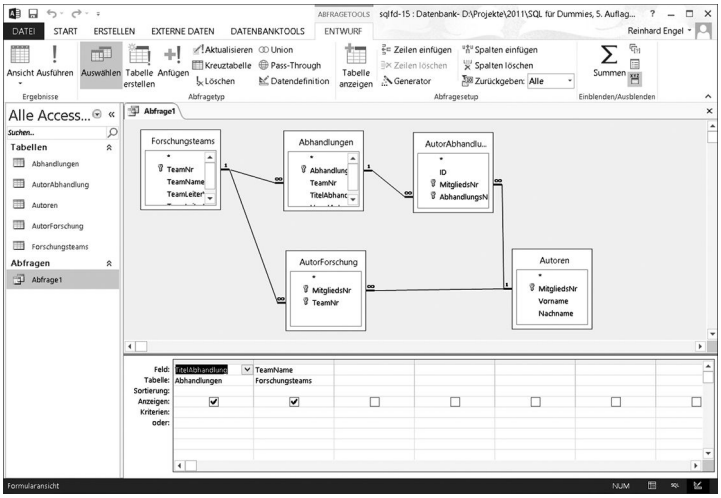


Abbildung 16.1: Die Abfrage MBFT–Abhandlungen in der Entwurfsansicht

Wenn Sie auf weitere Ansichten der Datenbank zugreifen wollen, klicken Sie auf das Symbol Ansicht in der linken oberen Ecke. Dort finden Sie andere Sichten auf die Datenbank, darunter auch die SQL-Ansicht (siehe Abbildung 16.2).

Wenn Sie auf das Untermenü SQL-Ansicht klicken, öffnet sich das Fenster zur Bearbeitung von SQL-Anweisungen und zeigt die Anweisung, die Access per QBE *(Query by Example)* auf der Grundlage der Auswahl im grafischen Teil des Abfragegenerators erstellt hat (siehe Abbildung 16.3).

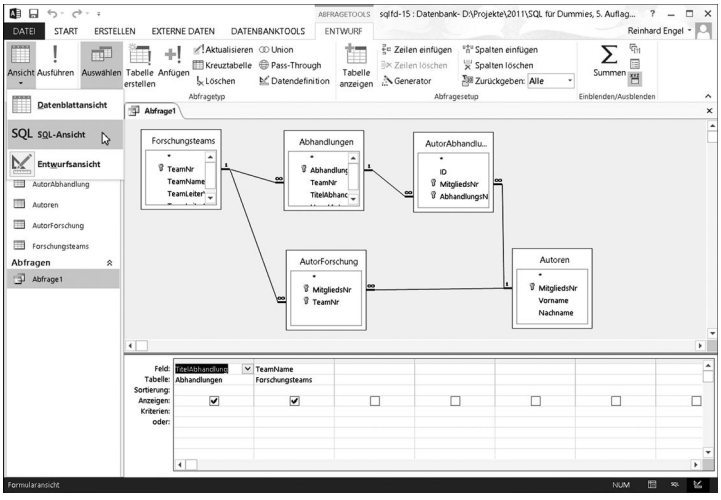


Abbildung 16.2: Eine der Optionen ist die SQL-Ansicht.

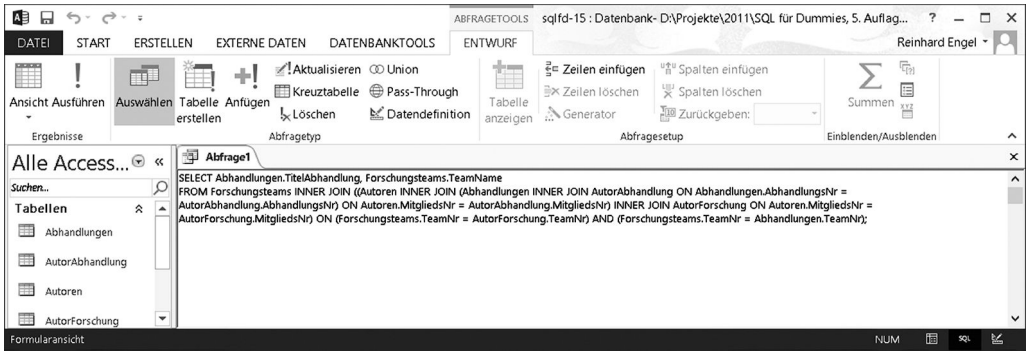


Abbildung 16.3: Eine SQL-Anweisung, die die Namen aller Abhandlungen abruft, die von Mitgliedern des MBFT geschrieben worden sind

Diese SQL-Anweisung, die in Abbildung 16.3 dargestellt ist, wird zur Datenbankengine gesendet. Die Datenbankengine, die direkt mit der Datenbank zu tun hat, versteht nur SQL. Jede Information, die in der Umgebung des Abfragegenerators eingegeben wird, muss in SQL übersetzt werden, bevor sie zur Verarbeitung an die Datenbankengine gesendet wird.

Vielleicht fällt Ihnen auf, dass sich die Syntax der SQL-Anweisung aus Abbildung 16.3 etwas vom ANSI-ISO-SQL-Standard unterscheidet. Nehmen Sie sich den Ausspruch »Wenn du in Rom bist, benimm dich wie ein Römer« zu Herzen. Wenn Sie mit Access arbeiten, verwenden Sie den Access-Dialekt von SQL und hinterfragen ihn nicht lange. Dieser Rat gilt übrigens für alle Umgebungen, mit denen Sie arbeiten. Alle SQL-Implementierungen unterscheiden sich auf die eine oder andere Art vom Standard.

Wenn Sie eine neue Abfrage in Access-SQL schreiben wollen (eine, die noch nicht per QBE mit der grafischen Oberfläche erstellt worden ist), löschen Sie im SQL-Bearbeitungsfenster einfach eine bestehende und schreiben Sie eine neue `SELECT`-Anweisung. Klicken Sie in der Symbolleiste auf das Symbol mit dem Ausrufezeichen, um die neue Abfrage ablaufen zu lassen. Das Ergebnis wird dann in der Datenblattansicht dargestellt.

Teil V

SQL in der Praxis

In diesem Teil ...

- ✔ Mit ODBC arbeiten
- ✔ Mit JDBC arbeiten
- ✔ Mit XML-Daten arbeiten

Datenzugriffe mit ODBC und JDBC

17

In diesem Kapitel

- Was ODBC ist
- Die Komponenten von ODBC
- ODBC in einer Client/Server-Umgebung benutzen
- ODBC im Internet benutzen
- ODBC in einem Intranet benutzen
- JDBC verwenden

In den vergangenen Jahren wurden immer mehr Computer miteinander verbunden, sowohl innerhalb von Unternehmen als auch zwischen ihnen. Im Zuge dieser Entwicklung stieg der Bedarf, Datenbankinformationen über Netzwerke austauschen zu können. Das Haupthindernis für den freien Austausch von Informationen über Netzwerke sind die Inkompatibilitäten zwischen den Betriebssystemen und Anwendungen auf den verschiedenen Rechnerplattformen. Ein wichtiger Schritt hin zur Beseitigung dieser Inkompatibilität war die Erfindung und Weiterentwicklung von SQL.

Leider ist »Standard-SQL« nicht gleich »Standard-SQL«. Selbst DBMS-Anbieter, die behaupten, den internationalen SQL-Standard einzuhalten, haben ihre Implementierungen erweitert, weshalb diese mit den Erweiterungen der Implementierungen anderer Anbieter inkompatibel sind. Die Anbieter würden nur sehr ungerne auf ihre Erweiterungen verzichten, weil ihre Kunden diese in ihre Anwendungen eingebaut haben und von ihnen abhängig geworden sind. Deshalb ist eine andere Methode erforderlich, die die Kommunikation zwischen Datenbanken ermöglicht und von den Anbietern nicht verlangt, ihre Implementierungen auf den kleinsten gemeinsamen Nenner zu reduzieren. Diese andere Methode heißt ODBC (Open DataBase Connectivity).

ODBC

ODBC ist eine Standardschnittstelle zwischen einer Datenbank und einer Anwendung, die auf die Daten in der Datenbank zugreift. Ein solcher Standard ermöglicht es jedem Anwendungs-Frontend mit SQL, auf ein beliebiges Datenbank-Backend zuzugreifen. Die einzige Bedingung dafür ist, dass sowohl das Frontend als auch das Backend dem ODBC-Standard entsprechen. Die aktuelle Version des Standards ist ODBC 4.0.

Eine Anwendung greift auf eine Datenbank zu, indem sie einen *Treiber* (den ODBC-Treiber) benutzt, der als Schnittstelle zwischen einer Anwendung und einer Datenbank entwickelt worden ist. Das Frontend des Treibers, das sich auf der Anwendungsseite befindet, muss dem ODBC-Standard strikt entsprechen. Es sieht für die Anwendung immer gleich aus, unabhän-

gig davon, welche Backend-Datenbank angesprochen wird. Das Backend des Treibers ist an die spezielle Datenbank angepasst, die es adressiert. Mit dieser Architektur brauchen Anwendungen nicht an eine spezielle Datenbank angepasst zu werden, ja, sie brauchen nicht einmal zu wissen, welche Backend-Datenbank die Daten verwaltet, mit denen sie arbeiten. Der Treiber verbirgt die Unterschiede zwischen den verschiedenen Backends.

Die ODBC-Schnittstelle

Die *ODBC-Schnittstelle* besteht im Wesentlichen aus einer Reihe von Definitionen, die als Standard akzeptiert werden. Die Definitionen decken alle Punkte ab, die benötigt werden, um die Kommunikation zwischen einer Anwendung und der Datenbank mit ihren Daten herzustellen. Die Definitionen umfassen folgende Bereiche:

- ✔ Eine Bibliothek mit Funktionsaufrufen
- ✔ Standard-SQL-Syntax
- ✔ Standard-SQL-Datentypen
- ✔ Ein Standardprotokoll für die Verbindung zu einer Datenbankengine
- ✔ Standardfehlercodes

Die ODBC-Funktionsaufrufe stellen die Mittel bereit, um die Verbindung zu einer Backend-Datenbankengine herzustellen. Sie führen SQL-Anweisungen aus und senden die Ergebnisse zurück an die Anwendung.

Um eine Datenbankoperation auszuführen, übergeben Sie die entsprechende SQL-Anweisung als Argument eines ODBC-Funktionsaufrufs. Solange Sie die von ODBC vorgegebene Standard-SQL-Syntax benutzen, funktioniert die Operation – und zwar unabhängig davon, welche Datenbankengine Backend-seitig arbeitet.

Die Komponenten von ODBC

Die ODBC-Schnittstelle besteht aus vier funktionellen Komponenten, die auch als Schichten (englisch *Layer*) bezeichnet werden. Jede Komponente trägt einen Teil dazu bei, ODBC so flexibel zu machen, dass es die Kommunikation – für den Benutzer nicht sichtbar – von einem beliebigen kompatiblen Frontend zu einem beliebigen kompatiblen Backend ermöglicht. Zwischen dem Benutzer und den Daten, mit denen der Benutzer arbeitet, liegen die folgenden vier Schichten der ODBC-Schnittstelle:

- ✔ **Anwendung:** Der Teil der ODBC-Schnittstelle, der am dichtesten am Benutzer ist, ist die Anwendung. Natürlich enthalten auch Systeme, die nicht mit ODBC arbeiten, eine Anwendung. Dennoch ist es sinnvoll, die Anwendung als Teil der ODBC-Schnittstelle zu betrachten. Die Anwendung muss »wissen«, dass sie mittels ODBC mit ihrer Datenquelle kommuniziert. Sie muss sich reibungslos mit dem ODBC-Treibermanager verbinden und den ODBC-Standard strikt einhalten.

- ✔ **Treibermanager:** Der Treibermanager ist eine DLL (*Dynamic Link Library*), die im Allgemeinen von Microsoft bereitgestellt wird. Sie lädt die entsprechenden Treiber für die Datenquelle oder die Datenquellen des Systems und leitet die Funktionsaufrufe, die von der Anwendung kommen, über den passenden Treiber an die entsprechende Datenquelle weiter. Außerdem führt der Treibermanager einige ODBC-Funktionsaufrufe direkt aus und entdeckt und behandelt einige Arten von Fehlern. Obwohl der ODBC-Standard von Microsoft entwickelt wurde, wird er heute universell akzeptiert, sogar von Open-Source-Hardlinern.
- ✔ **Treiber-DLL:** Weil sich die Datenquellen (zum Teil *sehr* stark) voneinander unterscheiden können, benötigen Sie eine Methode, um die Standard-ODBC-Funktionsaufrufe in die »Muttersprache« der jeweiligen Datenquelle zu übersetzen. Diese Übersetzung ist Aufgabe der Treiber-DLL. Jede Treiber-DLL nimmt Funktionsaufrufe über die Standard-ODBC-Schnittstelle entgegen und übersetzt sie dann in den Code, den die zur DLL gehörende Datenquelle verarbeiten kann. Wenn die Datenquelle ein Ergebnis zurückgibt, übersetzt der Treiber dieses umgekehrt in das Standard-ODBC-Ergebnisformat. Der Treiber ist das Schlüsselelement, mit dem ODBC-kompatible Anwendungen die Struktur und den Inhalt von ODBC-kompatiblen Datenquellen bearbeiten können.
- ✔ **Datenquelle:** Die Datenquellen können sehr verschiedene Formen haben. Es kann sich um ein relationales DBMS mit einer Datenbank handeln, die auf demselben Computer wie die Anwendung liegt. Die Datenquelle kann sich auch auf einem entfernten Computer befinden. Es kann sich aber auch um eine ISAM-Datei (*Indexed Sequential Access Method* – indexsequenzielle Zugriffsmethode) ohne DBMS handeln, die entweder auf dem lokalen oder auf einem entfernten Computer gespeichert ist. Die Datenquelle kann ein Netzwerk umfassen oder auch nicht. Jede der vielen möglichen Formen von Datenquellen erfordert einen eigenen Treiber.

ODBC in einer Client/Server-Umgebung

In einem Client/Server-System wird die Schnittstelle zwischen dem Client und dem Server als *API* (*Application Programming Interface* – Schnittstelle für die Anwendungsprogrammierung) bezeichnet. Eine API kann entweder anbieterspezifisch sein oder dem Standard entsprechen. Eine *anbieterspezifische (proprietäre)* API ist eine API, bei der die Client-Komponente der Schnittstelle speziell auf das Arbeiten mit einem bestimmten Backend auf dem Server abgestimmt worden ist. Der eigentliche Code, der eine solche firmenspezifische Schnittstelle bildet, wird als *nativer Treiber* bezeichnet. Ein nativer Treiber ist im Hinblick auf die Zusammenarbeit zwischen einem speziellen Frontend-Client und einer dazugehörigen Backend-Datenquelle optimiert. Deshalb erfolgt die Übermittlung von Befehlen und Informationen mit nativen Treibern besonders schnell und effizient.

Falls Ihr Client/Server-System immer auf dieselbe Datenquelle zugreift und Sie sicher sind, dass es nie auf die Daten anderer Datenquellen zugreifen wird, sollten Sie den nativen Treiber Ihres DBMS verwenden. Wenn das System dagegen jetzt oder in Zukunft auf verschiedene Datenquellen zugreift oder zugreifen wird, sollten Sie die ODBC-Schnittstelle benutzen. Das erspart Ihnen später sehr viel Arbeit beim Umschreiben der Anwendungen.

Auch ODBC-Treiber sind im Hinblick auf das Arbeiten mit speziellen Backend-Datenquellen optimiert, aber sie verfügen alle über dieselbe zum Treibermanager führende Frontend-Schnittstelle. Deshalb ist es leicht zu verstehen, dass ein Treiber, der nicht für ein bestimmtes Frontend optimiert wurde, nicht so schnell ist wie ein nativer Treiber, der speziell für dieses Frontend entwickelt worden ist. Bei der ersten Generation von ODBC-Treibern wurde vor allem ihr schlechtes Zeitverhalten im Vergleich zu nativen Treibern kritisiert. Neuere Benchmarks haben jedoch gezeigt, dass ODBC-4.0-Treiber durchaus mit nativen Treibern mithalten können. Die Technik ist inzwischen so ausgereift, dass es nicht mehr notwendig ist, die Vorteile der Standardisierung einer Verbesserung des Leistungsverhaltens zu opfern.

ODBC und das Internet

Datenbankoperationen über das Internet unterscheiden sich in mehreren wichtigen Punkten von Datenbankoperationen auf einem Client/Server-System. Der offensichtlichste Unterschied aus der Sicht des Benutzers ist die Client-Komponente des Systems, die die Benutzeroberfläche enthält. Bei einem Client/Server-System gehört die Benutzeroberfläche zu einer Anwendung, die mit der Datenquelle auf dem Server über ODBC-kompatible SQL-Anweisungen kommuniziert. Im World Wide Web besteht die Client-Komponente des Systems aus einem Webbrowser, der mit der Datenquelle auf dem Server mit Standard-*HTTP (HyperText Transfer Protocol)* kommuniziert.

Jeder Benutzer mit einer geeigneten Client-End-Software (und den entsprechenden Rechten) kann auf Daten zugreifen, die im Web verfügbar sind. Das bedeutet, dass Sie auf Ihrem Arbeitscomputer eine Anwendung erstellen und später mit einem Mobilgerät darauf zugreifen können. Abbildung 17.1 zeigt einen Vergleich eines Client/Server-Systems mit einem webbasierten System.

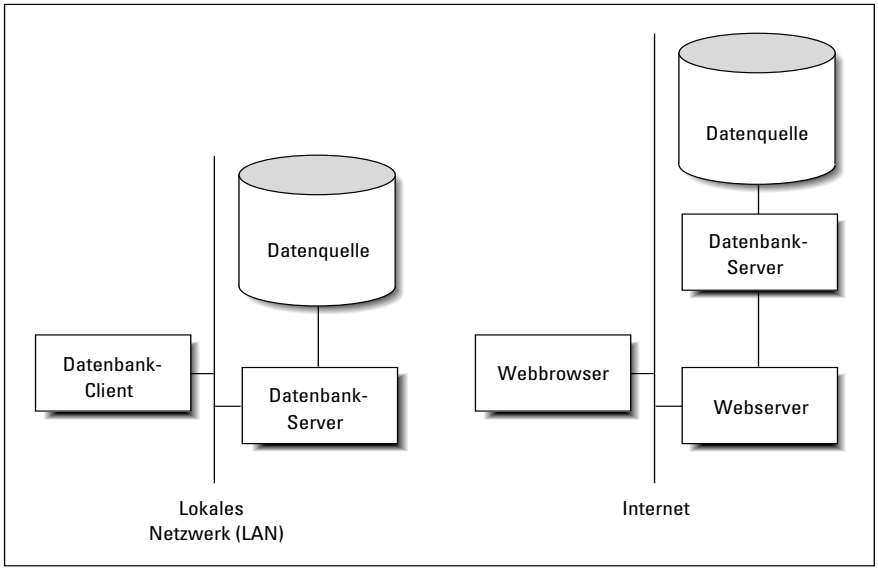


Abbildung 17.1: Vergleich eines Client/Server-Systems mit einem webbasierten System

Server-Erweiterungen

In einem webbasierten System erfolgt die Kommunikation zwischen dem Browser auf dem Client-Rechner und dem Webserver auf dem Server-Rechner mittels HTTP. Eine Systemkomponente mit der Bezeichnung *Server-Erweiterung* übersetzt Anweisungen, die über das Netzwerk eingehen, in ODBC-kompatiblen SQL-Code. Dieser wird dann vom Datenbank-Server interpretiert und ausgeführt, der seinerseits direkt mit der Datenquelle arbeitet. In umgekehrter Richtung wird das Ergebnis einer Abfrage von der Datenquelle über den Datenbank-Server zur Server-Erweiterung gesendet, die es in eine Form übersetzt, die der Webserver verarbeiten kann. Die Ergebnisse werden dann über das Web zum Webbrowser auf dem Client-Rechner geschickt, wo es auf dem Bildschirm angezeigt wird. Abbildung 17.2 zeigt die Architektur eines solchen Systems.

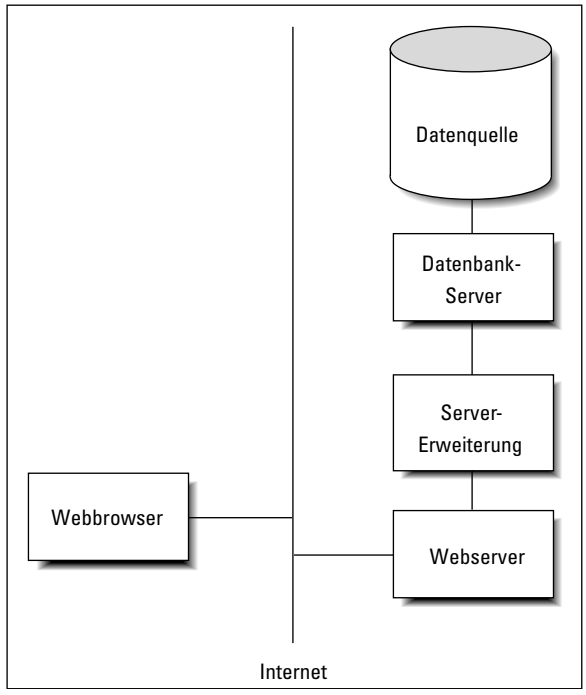


Abbildung 17.2: Webbasiertes Datenbanksystem mit Server-Erweiterung

Client-Erweiterungen

Anwendungen wie Microsoft Access 2013 wurden entwickelt, um mit Daten zu arbeiten, die lokal auf dem Rechner des Anwenders, auf einem Server in einem lokalen Wide Area Network (LAN oder WAN) oder im Internet in der Cloud gespeichert sind. Die Cloud von Microsoft ist SkyDrive. Es ist auch möglich, nur mit einem Webbrowser auf eine Anwendung in der Cloud zuzugreifen.

Webbrowser sind dafür entwickelt und optimiert worden, eine leicht verständliche und einfach zu bedienende Schnittstelle zu Websites aller Art bereitzustellen. Die gebräuchlichsten Browser, Mozilla Firefox, Microsoft Internet Explorer, Opera, Apple Safari und Google Chrome, sind weder dafür geschaffen noch optimiert worden, um als Datenbank-Frontend zu dienen. Damit Benutzer über das Internet sinnvoll mit einer Datenbank interagieren können, benötigt die Client-Seite des Systems eine Funktionalität, die die Browser nicht zur Verfügung stellen. Um diese Lücke zu füllen, wurden mehrere Arten von *Client-Erweiterungen* entwickelt. Diese Erweiterungen umfassen Helper-Anwendungen, ActiveX-Steuerelemente, Java-Applets und Skripte. Die Erweiterungen kommunizieren mit dem Server über HTML, das die Sprache des Webs ist. Jeder HTML-Code, der mit Datenbankzugriffen zu tun hat, wird von der Server-Erweiterung in ODBC-kompatibles SQL übersetzt, ehe er an die Datenquelle weitergeleitet wird.

ActiveX-Steuerelemente

Microsofts ActiveX-Steuerelemente arbeiten mit Microsofts Internet Explorer zusammen, der weltweit wohl der bekannteste Browser ist, obwohl er in den letzten Jahren erhebliche Marktanteile an Google Chrome und Mozilla Firefox verloren hat.

Skripte

Skripte sind die flexibelsten Werkzeuge, um Client-Erweiterungen zu erstellen. Mit einer der neuen Skripting-Sprachen, zum Beispiel das allgegenwärtige JavaScript oder VBScript von Microsoft, haben Sie die maximale Kontrolle über das Geschehen auf der Client-Seite. Sie können damit Gültigkeitsprüfungen für Eingabefelder durchführen und ungültige Einträge korrigieren oder ablehnen, ohne auf das Web zugreifen zu müssen. Dadurch können Sie viel Zeit sparen und den Verkehr im Web so reduzieren, dass auch andere Benutzer einen Gewinn davon haben. Skripte werden wie Java-Applets in eine HTML-Seite eingebettet und ausgeführt, wenn der Benutzer mit dieser Seite arbeitet.

ODBC und Intranets

Ein *Intranet* ist ein lokales Netzwerk (LAN) oder ein Wide Area Network (WAN), das wie eine einfachere Version des Internets funktioniert. Weil sich ein Intranet vollständig innerhalb eines Unternehmens befindet, benötigen Sie keine komplexen Sicherheitsmaßnahmen wie Firewalls. Alle Werkzeuge für die Anwendungsentwicklung für das World Wide Web funktionieren genauso gut als Entwicklungswerkzeuge für Intranet-Anwendungen. ODBC arbeitet in einem Intranet in derselben Weise wie im Internet. Wenn Sie mit mehreren verschiedenen Datenquellen arbeiten, können Clients, die die Webbrowser und die entsprechenden Client- und Servererweiterungen benutzen, mit diesen Datenquellen kommunizieren. Dabei wird ODBC-kompatibler SQL-Code über die HTML- und ODBC-Stufen an den Treiber weitergeleitet, der den Code in die native Befehlssprache der Datenbank übersetzt, die ihn schließlich ausführt.

JDBC

JDBC *(Java DataBase Connectivity)* ähnelt ODBC, unterscheidet sich aber davon in einigen wichtigen Punkten. Auf einen Unterschied weist schon der Name hin. JDBC ist eine Datenbankschnittstelle, die für das Client-Programm immer gleich aussieht – unabhängig von der Datenquelle, die sich auf dem Server (Backend) befindet. Der Unterschied besteht darin, dass JDBC erwartet, dass die Client-Anwendung in der Sprache Java und nicht in einer anderen Sprache wie C++ oder Visual Basic geschrieben ist. Ein weiterer Unterschied besteht darin, dass sowohl Java als auch JDBC von Anfang an für den Einsatz im World Wide Web oder in einem Intranet entworfen worden sind.

Java ist eine Programmiersprache, die C++ ähnelt und von Sun Microsystems speziell für die Entwicklung von Clientanwendungen im Web entwickelt worden ist. Sobald über das Web eine Verbindung zwischen einem Server und einem Client hergestellt worden ist, wird das entsprechende Java-Applet auf den Client heruntergeladen, wo es anfängt zu laufen. Das Applet, das in eine HTML-Seite eingebettet ist, sorgt für die datenbankspezifischen Funktionalitäten, die der Client für einen flexiblen Zugriff auf die Serverdaten benötigt. Abbildung 17.3 zeigt schematisch eine Web-Datenbankanwendung und ein Java-Applet, das auf einem Client läuft.

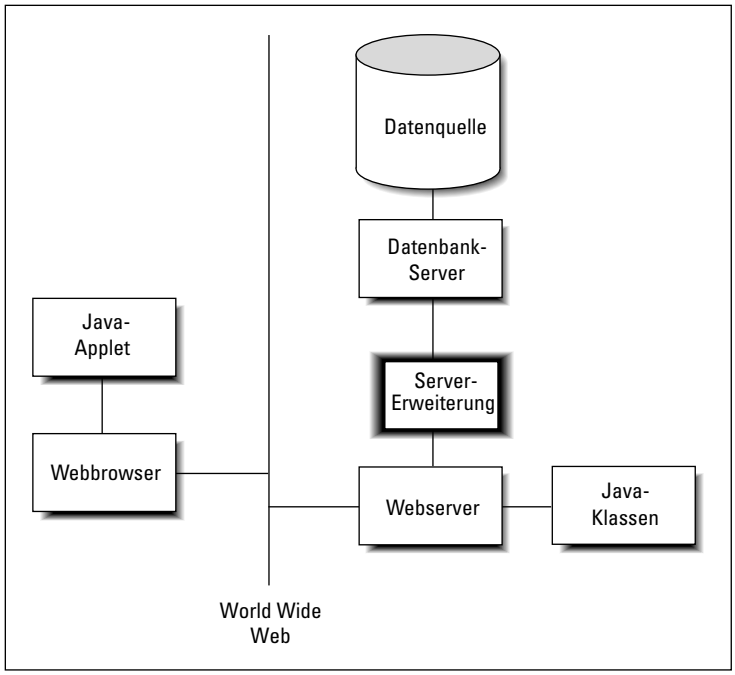


Abbildung 17.3: Eine Web-Datenbankanwendung, die ein Java-Applet verwendet

Ein *Applet* ist eine kleine Anwendung, die auf einem Server gespeichert ist. Wenn ein Client über das Web eine Verbindung zu diesem Server herstellt, wird das Applet heruntergeladen und auf dem Client-Computer ausgeführt. Java-Applets sind speziell für die Ausführung in

einer sogenannten *Sandbox* konzipiert, einem wohldefinierten und isolierten Bereich im Speicher des Client-Computers, der für die Ausführung von Applets reserviert wird. Das Applet kann nichts außerhalb der Sandbox beeinflussen. Diese Architektur soll den Client-Rechner vor potenziell feindlichen Applets schützen, die vertrauliche Daten stehlen oder Schaden anrichten wollen.

Ein großer Vorteil von Java-Applets ist, dass sie eigentlich zu jeder Zeit aktuell sind. Weil die Applets immer zur Laufzeit vom Server heruntergeladen und nicht auf dem Client gehalten werden, hat der Client die Garantie, dass er ausschließlich die neueste Version eines Applets verwendet.

Wenn Sie für die Wartung des Servers Ihres Unternehmens verantwortlich sind, müssen Sie sich keine Gedanken darüber machen, dass einige Ihrer Clients mit der Serveranwendung nicht mehr kompatibel sind, wenn Sie diese aktualisieren. Achten Sie nur darauf, dass Ihr herunterladbares Java-Applet mit der neuen Serverkonfiguration kompatibel ist, und solange die Webbrowser so konfiguriert sind, dass Java aktiviert ist, werden auch alle Browser wieder kompatibel. Java ist eine voll funktionsfähige Programmiersprache, und Sie sind problemlos in der Lage, stabile Anwendungen in Java zu schreiben, die in einer Client/Server-Umgebung auf Datenbanken zugreifen können. Wenn Sie eine Java-Anwendung auf diese Weise benutzen, erfolgt der Datenbankzugriff über JDBC. Die Anwendung ist dann einer Datenbankanwendung vergleichbar, die in C++ geschrieben ist und über ODBC auf die Daten zugreift. Der große Unterschied liegt darin, dass Java komplett anders arbeitet als C++, wenn es um das Internet (oder ein Intranet) geht.

Wenn sich das System, für das Sie sich interessieren, im Web befindet, liegen ganz andere Arbeitsbedingungen vor als in einem Client/Server-System. Die Client-Seite einer Anwendung, die über das Internet arbeitet, besteht aus einem Browser mit minimalen rechnerbetonten Fähigkeiten. Diese Fähigkeiten müssen erweitert werden, um eine ernsthafte Arbeit mit Datenbanken zu ermöglichen. Diese Fähigkeiten werden von Java-Applets zur Verfügung gestellt.

Sie sind einer gewissen Gefahr ausgesetzt, wenn Sie etwas von einem Server herunterladen, von dem Sie nicht wissen, ob er vertrauenswürdig ist. Wenn Sie ein Java-Applet herunterladen, ist diese Gefahr erheblich geringer, wird aber nicht vollkommen beseitigt. Seien Sie vorsichtig, wenn Sie ausführbaren Code von einem fragwürdigen Server auf Ihren Rechner herunterladen.

JDBC übergibt, wie ODBC, SQL-Anweisungen einer Frontend-Anwendung (Applet), die auf einem Client ausgeführt wird, an die Datenquelle auf dem Backend. Sie dient auch dazu, die Ergebnisse oder Fehlermeldungen von der Datenquelle zurück an die Anwendung zu geben. Der Nutzen, der sich aus der Arbeit mit JDBC ergibt, besteht darin, dass sich der Applet-Autor auf die Standard-JDBC-Schnittstelle stützen kann, ohne sich darum kümmern zu müssen, welche Datenbank sich am Backend befindet. JDBC führt alle Umwandlungen durch, die für eine genaue Zwei-Wege-Kommunikation erforderlich sind. Obwohl JDBC für die Arbeit im Web konzipiert worden ist, funktioniert es auch in Client/Server-Umgebungen, bei denen eine in Java geschriebene Anwendung per JDBC-Schnittstelle mit einem Datenbank-Backend kommuniziert.

SQL und XML

18

In diesem Kapitel

- SQL mit XML verwenden
- XML, Datenbanken und das Internet

Seit SQL:2008 unterstützt ISO/IEC-Standard-SQL XML. XML-Dateien (E*X*tensible *M*arkup *L*anguage, erweiterbare Auszeichnungssprache) haben sich als überall akzeptierter Standard für den Datenaustausch zwischen verschiedenen Plattformen durchgesetzt. Mit XML spielt es keine Rolle, ob die Person, mit der Sie Daten austauschen, über eine andere Anwendungsumgebung, ein anderes Betriebssystem oder sogar eine andere Hardware verfügt. XML kann eine »Datenbrücke« bilden.

Was XML mit SQL zu tun hat

XML ist wie HTML eine Auszeichnungssprache, was bedeutet, dass es sich hierbei um keine richtige Programmiersprache wie C++ oder Java handelt. XML ist noch nicht einmal eine Datenuntersprache wie SQL. Aber anders als diese Sprachen konzentriert sich XML auf den Inhalt der von ihm verwalteten Daten. Dort, wo HTML in einem Dokument lediglich Text und Grafiken formatiert, versieht XML den Dokumentinhalt mit einer Struktur. XML ist nicht für die Formatierung der Daten zuständig. Zu diesem Zweck wird XML um ein *Stylesheet* erweitert, das, wie HTML, Formatierungen auf ein XML-Dokument anwendet.

Die Struktur eines XML-Dokuments wird von seinem XML-Schema zur Verfügung gestellt, das ein Beispiel für sogenannte *Metadaten* (Daten, die Daten beschreiben) darstellt. Ein XML-Schema beschreibt, wo und in welcher Reihenfolge Elemente in einem Dokument stehen. Es kann auch den Datentyp eines Elements beschreiben und die Werte einschränken, die ein Typ annehmen darf.

SQL und XML bieten zwei verschiedene Verfahren an, um Daten so zu strukturieren, dass ausgewählte Informationen gespeichert und abgerufen werden können:

- ✔ SQL ist ein exzellentes Hilfsmittel für die Arbeit mit numerischen und Textdaten, die mit Datentypen kategorisiert werden können und eine klar definierte Größe besitzen.

  SQL ist entwickelt worden, um Daten zu verwalten und zu verarbeiten, die in relationalen Datenbanken gespeichert sind.

- ✔ XML kann besser mit formlosen Daten umgehen, die sich nicht einfach in Kategorien einordnen lassen.

  Der Hauptgrund für die Entwicklung von XML war der Wunsch, einen allgemeingültigen Standard für den Datentransfer zwischen verschiedenen Computern und für die Anzeige dieser Daten im Internet zur Verfügung zu stellen.

Die Stärken und Ziele von SQL und XML ergänzen sich. Jede Sprache beherrscht ihr Terrain und verbündet sich mit der jeweils anderen Sprache, um den Benutzern jederzeit die benötigten Informationen zu liefern.

Der XML-Datentyp

XML wurde mit SQL:2003 als neuer Datentyp eingeführt. Dies bedeutet, dass Implementierungen, die den Standard befolgen, XML-Daten direkt speichern und verarbeiten können, ohne sie zunächst von einem der anderen SQL-Datentypen in das XML-Format umwandeln zu müssen.

Der `XML`-Datentyp einschließlich seiner Untertypen verhält sich wie ein benutzerdefinierter Datentyp (UDT), obwohl er ein integraler Bestandteil einer jeden SQL-Implementierung ist, die ihn unterstützt. Die Untertypen sind:

- ✔ `XML (DOCUMENT (UNTYPED))`
- ✔ `XML (DOCUMENT (ANY))`
- ✔ `XML (DOCUMENT (XMLSCHEMA))`
- ✔ `XML (CONTENT (UNTYPED))`
- ✔ `XML (CONTENT (ANY))`
- ✔ `XML (CONTENT (XMLSCHEMA))`
- ✔ `XML (SEQUENCE)`

Der Datentyp `XML` bringt SQL und XML näher zusammen, da er es Anwendungen ermöglicht, SQL-Operationen auf XML-Inhalten und XML-Operationen auf SQL-Inhalten auszuführen. Sie können eine Spalte vom Datentyp `XML` mit der `WHERE`-Klausel zusammen mit Spalten, die von einem der Datentypen sind, die in Kapitel 2 beschrieben werden, in einer Verknüpfungsoperation einsetzen. Ihr DBMS ermittelt dann die optimale Vorgehensweise, die zur Ausführung der Abfrage erforderlich ist, und führt diese dann auch aus.

Wann der XML-Datentyp verwendet werden sollte

Ob Sie Daten im XML-Format speichern sollten, hängt davon ab, was Sie mit diesen Daten vorhaben. Nachfolgend finden Sie einige Situationen beschrieben, in denen es sinnvoll ist, Daten im XML-Format abzulegen:

- ✔ Sie möchten einen vollständigen Datenblock speichern und später wieder abrufen.
- ✔ Sie möchten ein vollständiges XML-Dokument abfragen. Einige Implementierungen haben den Operator `EXTRACT` erweitert, damit Sie die gewünschten Inhalte aus einem XML-Dokument herausholen können.
- ✔ Sie benötigen in SQL-Anweisungen die strikte Typisierung von Daten. Der Einsatz des Datentyps `XML` gewährleistet, dass Daten gültige XML-Werte und nicht einfach irgendwelche Textfolgen sind.

- ✔ Sie möchten die Kompatibilität mit zukünftigen Speichersystemen gewährleisten, die gegenwärtig noch nicht spezifiziert sind und vorhandene Datentypen wie `CHARACTER LARGE OBJECT` (oder `CLOB`) möglicherweise nicht unterstützen werden. (Das zweite Kapitel hält detaillierte Informationen zum Datentyp `CLOB` bereit.)
- ✔ Sie möchten für zukünftige Optimierungen bereit sein, die nur den `XML`-Datentyp unterstützen werden.

Das folgende Beispiel zeigt Ihnen, wie Sie `XML`-Typen einsetzen könnten:

```
CREATE TABLE Kunde (
     KundenName       CHARACTER(30)     NOT NULL,
     Adresse1         CHARACTER(30),
     Adresse2         CHARACTER(30),
     PLZ              CHARACTER(25),
     Ort              CHARACTER(2),
     Land             CHARACTER(10),
     Telefon          CHARACTER(13),
     Fax              CHARACTER(13),
     KontaktPerson    CHARACTER(30),
     Kommentare       XML(SEQUENCE) ) ;
```

Diese SQL-Anweisung speichert in der Spalte `Kommentare` der Tabelle `Kunde` ein XML-Dokument. Dieses Dokument könnte so aussehen:

```
<Kommentare>
   <Kommentar>
      <KommentarNr>1</KommentarNr>
      <Nachricht>Ist VetLab in der Lage, auch Pinguinblut
       zu analysieren?</Nachricht>
      <AntwortErforderlich>Ja</AntwortErforderlich>
   </Kommentar>
   <Kommentar>
      <KommentarNr>2</KommentarNr>
      <Nachricht>Vielen Dank für die schnelle Bearbeitung der
       Blutprobe unseres Seeleoparden.</Nachricht>
      <AntwortErforderlich>Nein</AntwortErforderlich>
   </Kommentar>
</Kommentare>
```

Wann der Datentyp XML nicht verwendet werden sollte

Nur weil der SQL-Standard es zulässt, den `XML`-Typ zu verwenden, heißt das noch lange nicht, dass das immer sinnvoll ist. Die Daten der meisten relationalen Datenbanken sind heute in ihrem aktuellen Format besser aufgehoben als im XML-Format. Nachfolgend finden Sie einige Beispiele, die Ihnen verraten, wann Sie den Datentyp `XML` besser nicht verwenden sollten:

- ✔ Wenn sich die Daten natürlich in eine relationale Struktur mit Tabellen, Zeilen und Spalten gliedern lassen.
- ✔ Wenn Sie Teile des Dokuments und nicht das Dokument als Ganzes aktualisieren müssen.

SQL in XML und XML in SQL konvertieren

Um Daten zwischen SQL-Datenbanken und XML-Dokumenten austauschen zu können, müssen die verschiedenen Elemente einer SQL-Datenbank in die entsprechenden Elemente eines XML-Dokuments übersetzt werden und umgekehrt. In den folgenden Abschnitten beschreibe ich, welche Elemente übersetzt werden müssen.

Zeichensätze konvertieren

Welche Zeichensätze von SQL unterstützt werden, ist von der jeweiligen Implementierung abhängig. Dies bedeutet, dass DB2 von IBM andere Zeichensätze unterstützen kann als Microsofts SQL Server. SQL Server wiederum kann mit Zeichensätzen arbeiten, die Oracle nicht zur Verfügung stellt. Wenngleich die gebräuchlichsten Zeichensätze von fast allen Implementierungen unterstützt werden, kann ein etwas aus der Reihe fallender Zeichensatz die Migration Ihrer Datenbank und Anwendung von einer RDBMS-Plattform zur nächsten erschweren.

XML kennt keine Kompatibilitätsprobleme mit Zeichensätzen – es unterstützt nur einen: Unicode. Für den Austausch von Daten zwischen einer SQL-Implementierung und XML ist dies eine gute Voraussetzung. Alle RDBMS-Anbieter kennen ein Verfahren zur Übersetzung von Zeichenfolgen eines beliebigen Zeichensatzes in Unicode und umgekehrt. XML unterstützt glücklicherweise nicht mehrere Zeichensätze. Die Anbieter hätten andernfalls ein großes Viele-zu-viele-Problem, da sie unzählige Konvertierungsmechanismen zur Verfügung stellen müssten.

Bezeichner konvertieren

XML reglementiert strenger als SQL die Zeichen, die in Bezeichnern zulässig sind. Zeichen, die in SQL erlaubt, aber in XML nicht zulässig sind, müssen zunächst übersetzt werden, bevor sie Teil eines XML-Dokuments werden können. SQL unterstützt Bezeichner mit Begrenzungszeichen. Dies bedeutet, dass alle Arten seltsamer Zeichen, zum Beispiel %, $ und &, erlaubt sind, solange sie in doppelte Anführungszeichen eingeschlossen werden. Solche Zeichen sind jedoch in XML nicht zulässig. Darüber hinaus sind XML-Namen, die mit den Zeichen *XML* beginnen, in jedem Fall reserviert und dürfen nicht ungestraft verwendet werden. SQL-Bezeichner, die mit diesen Buchstaben beginnen, müssen geändert werden.

Ein Übersetzungsverfahren, auf das man sich geeinigt hat, überbrückt die Kluft zwischen SQL- und XML-Bezeichnern. Von SQL zu XML werden alle SQL-Bezeichner in Unicode konvertiert. SQL-Bezeichner, die zulässige XML-Namen sind, bleiben dabei unverändert. Zeichen in SQL-Bezeichnern, die in XML-Namen nicht erlaubt sind, werden durch einen hexadezimalen Code ersetzt, der entweder in der Form "_xNNNN_" vorliegt oder das Format "xNNNNNNNN"

hat, wobei das `N` einer hexadezimalen Ziffer in Großbuchstaben entspricht. Der Unterstrich "_" wird beispielsweise als "_x005F_" und der Punkt als "_x003A_" dargestellt. Wenn ein SQL-Bezeichner mit den Buchstaben *x*, *m* und *l* beginnt, wird allen Instanzen der Code "_xFFFF_" vorangestellt.

Die Konvertierung von XML zu SQL ist sehr viel einfacher. Die Zeichen eines XML-Namens müssen lediglich nach der Sequenz "_xNNNN_" oder "_xNNNNNNNN_" durchsucht werden. Wenn Sie eine solche Zeichenfolge entdecken, ersetzen Sie sie durch das Zeichen, das dem jeweiligen Unicode entspricht. Wenn ein XML-Name mit den Zeichen "_xFFFF_" beginnt, ignorieren Sie sie einfach.

Wenn Sie diese einfachen Regeln befolgen, können Sie einen SQL-Bezeichner problemlos in einen XML-Namen und wieder in einen SQL-Bezeichner konvertieren. Diese glückliche Situation trifft leider nicht zu, wenn es darum geht, einen XML-Namen in einen SQL-Bezeichner und zurück zu konvertieren.

Datentypen konvertieren

Der SQL-Standard verlangt, dass ein SQL-Datentyp in einen XML-Schemadatentyp konvertiert werden muss, der die größte Ähnlichkeit zu seinem SQL-Pendant aufweist. Mit »größte Ähnlichkeit« ist gemeint, dass alle vom SQL-Typ akzeptierten Werte auch vom XML-Schematyp anerkannt werden. Außerdem sollte der XML-Typ bestenfalls keine, aber zumindest möglichst wenige Werte akzeptieren, die auch für den SQL-Typ nicht zulässig sind. XML-Facetten, zum Beispiel `maxInclusive` und `minInclusive`, können die für einen XML-Schematyp zulässigen Werte so einschränken, dass diese vom entsprechenden SQL-Typ akzeptiert werden. Wenn beispielsweise eine SQL-Implementierung einen `INTEGER`-Typ auf Werte zwischen -2.157.483.648 und 2.157.483.647 beschränkt, kann in XML der Wert `maxInclusive` auf 2.157.483.647 und der Wert `minInclusive` auf -2.157.483.648 gesetzt werden. Hier ist ein Beispiel für eine solche Konvertierung:

```
<xsd:simpleType>
   <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="2157483647"/>
      <xsd:minInclusive value="–2157483648"/>
      <xsd:annotation>
         <sqlxml:sqltype name="INTEGER"/>
      </xsd:annotation>
   </xsd:restriction>
</xsd:simpleType>
```

Im Anmerkungsbereich (`annotation`) befinden sich Informationen zur Definition des SQL-Datentyps, die von XML nicht benötigt werden, später aber nützlich sein können, wenn dieses Dokument wieder in SQL konvertiert wird.

Tabellen konvertieren

Sie können eine Tabelle in ein XML-Dokument konvertieren. Sie verfügen außerdem über die Möglichkeit, alle Tabellen eines Schemas oder eines Kataloges zu konvertieren. Benutzerrechte werden auch bei der Konvertierung beachtet. Ein Benutzer, der über ein `SELECT`-Recht für nur einige Tabellenspalten verfügt, kann auch nur diese Spalten in ein XML-Dokument konvertieren. Die Konvertierung erzeugt eigentlich zwei Dokumente. Das erste Dokument enthält die Tabellendaten und im zweiten Dokument ist das XML-Schema enthalten, das das erste Dokument beschreibt. Hier ist ein Beispiel für die Konvertierung einer SQL-Tabelle in ein XML-Dokument:

```
<Kunde>
   <zeile>
      <Vorname>Abe</Vorname>
      <Nachname>Abelson</Nachname>
      <Ort>Springfield</Ort>
      <Vorwahl>714</Vorwahl>
      <Rufnummer>5551111</Rufnummer>
   </zeile>
   <zeile>
      <Vorname>Bill</Vorname>
      <Nachname>Bailey</Nachname>
      <Ort>Decatur</Ort>
      <Vorwahl>714</Vorwahl>
      <Rufnummer>5552222</Rufnummer>
   </zeile>
...
</Kunde>
```

Das Stammelement des Dokuments trägt den Namen der Tabelle. Jede Tabellenzeile befindet sich innerhalb eines `<zeile>`-Elements, und jedes `<zeile>`-Element enthält mehrere Spaltenelemente, die dieselben Namen wie die entsprechenden Spalten in der Quelltabelle haben. Jedes Spaltenelement enthält einen Datenwert.

Mit Nullwerten umgehen

Da SQL-Daten aus Nullwerten bestehen können, müssen Sie darüber nachdenken, wie diese in einem XML-Dokument dargestellt werden sollen. Sie können einen Nullwert mit `nil` darstellen oder ganz auf das entsprechende Element verzichten. Wenn Sie sich für die Option `nil` entscheiden, kennzeichnet das Attribut `xsi:nil="true"` die Spaltenelemente, die Nullwerte enthalten. Das Attribut kann wie folgt verwendet werden:

```
<zeile>
      <Vorname>Bill</Vorname>
      <Nachname>Bailey</Nachname>
      <Ort xsi:nil="true" />
      <Vorwahl>714</Vorwahl>
      <Rufnummer>5552222</Rufnummer>
</zeile>
```

Möchten Sie stattdessen auf das Element, das den Nullwert enthält, verzichten, können Sie die folgende Lösung benutzen:

```
<zeile>
      <Vorname>Bill</Vorname>
      <Nachname>Bailey</Nachname>
      <Vorwahl>714</Vorwahl>
      <Rufnummer>5552222</Rufnummer>
</zeile>
```

In diesem Fall fehlt die Zeile, die den Nullwert enthält.

Das XML-Schema erzeugen

Wird SQL in XML konvertiert, enthält das erste erzeugte Dokument die Daten, das zweite die Schemainformationen. Schauen Sie sich als Beispiel das Schema für das Dokument `Kunde` an, das weiter oben im Abschnitt *Tabellen konvertieren* vorgestellt wird:

```
<xsd:schema>
   <xsd:simpleType name="CHAR_15">
      <xsd:restriction base="xsd:string">
         <xsd:length value = "15"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="CHAR_25">
      <xsd:restriction base="xsd:string">
         <xsd:length value = "25"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="CHAR_3">
      <xsd:restriction base="xsd:string">
         <xsd:length value = "3"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="CHAR_8">
      <xsd:restriction base="xsd:string">
         <xsd:length value = "8"/>
      </xsd:restriction>
   </xsd:simpleType>
```

```
   <xsd:sequence>
      <xsd:element name="Vorname" type="CHAR_15"/>
      <xsd:element name="Nachname" type="CHAR_25"/>
      <xsd:element
         name="Ort" type="CHAR_25 nillable="true"/>
      <xsd:element
         name="Vorwahl" type="CHAR_3" nillable="true"/>
      <xsd:element
         name="Rufnummer" type="CHAR_8" nillable="true"/>
   </xsd:sequence>

</xsd:schema>
```

Dieses Schema ist für den Einsatz von `nil` zur Darstellung von Nullwerten geeignet. Wenn Sie anstelle der Option `nil` auf Spalten mit Nullwerten verzichten möchten, benötigen Sie eine etwas andere Elementdefinition, zum Beispiel die folgende:

```
<xsd:element
name="Ort" type="CHAR_25 minOccurs="0"/>
```

SQL-Funktionen, die mit XML-Daten arbeiten

Der SQL-Standard definiert eine Reihe von Operatoren, Funktionen und Pseudofunktionen, die ein XML-Ergebnis liefern, wenn sie auf eine SQL-Datenbank angewendet werden, und ein Ergebnis produzieren, das das Standard-SQL-Format hat, wenn sie auf XML-Daten angewendet werden. Zu diesen Funktionen gehören `XMLELEMENT`, `XMLFOREST`, `XMLGEN`, `XMLCONCAT` und `XMLAGG`. Ich beschreibe in den folgenden Abschnitten kurz diese und ein paar weitere Funktionen, die häufig verwendet werden, um Daten im Web zu veröffentlichen. Einige dieser Funktionen stützen sich stark auf XQuery, eine neue Abfragesprache, die speziell für die Abfrage von XML-Daten entwickelt worden ist. XQuery ist ein eigenständiges Thema, das den Rahmen dieses Buchs überschreitet.

XMLDOCUMENT

Der Operator `XMLDOCUMENT` übernimmt einen XML-Wert als Input und gibt einen anderen XML-Wert als Output zurück. Der neue XML-Wert ist ein Dokument-Knoten, der nach den Regeln des berechneten Dokument-Konstruktors in XQuery konstruiert ist.

XMLELEMENT

Der Operator `XMLELEMENT` übersetzt einen relationalen Wert in ein XML-Element. Sie können ihn in einer `SELECT`-Anweisung benutzen, um XML-Daten aus einer SQL-Datenbank herauszuholen und im Web zu veröffentlichen. Dazu ein Beispiel:

```
SELECT k.Nachname
   XMLELEMENT ( NAME "Ort", k.Ort ) AS "Ergebnis"
FROM Kunde k
WHERE Nachname="Abelson" ;
```

Das Ergebnis, das diese Anweisung zurückgibt, lautet:

Nachname	Ergebnis
Abelson	`<Ort>Springfield</Ort>`

XMLFOREST

Der Operator `XMLFOREST` produziert aus einer Liste mit relationalen Werten einen »Wald« (englisch *Forest*) von XML-Elementen. Jeder Wert eines Operators erzeugt ein neues Element, wie das folgende Beispiel zeigt:

```
SELECT k.Nachname
   XMLFOREST (k.Ort,
              k.Vorwahl,
              k.Rufnummer ) AS "Ergebnis"
FROM Kunde k
WHERE Nachname="Abelson" OR Nachname="Bailey" ;
```

Diese Anweisung erzeugt die folgende Ausgabe:

Nachname	Ergebnis

Abelson	`<Ort>Springfield</Ort>`
	`<Vorwahl>714</Vorwahl>`
	`<Rufnummer>5551111</Rufnummer>`

Bailey	`<Ort>Decatur</Ort>`
	`<Vorwahl>714</Vorwahl>`
	`<Rufnummer>5552222</Rufnummer>`

XMLCONCAT

`XMLCONCAT` bietet eine alternative Möglichkeit, um einen »Wald« aus XML-Elementen zu erstellen. Dazu verkettet es seine XML-Argumente wie im folgenden Beispiel:

```
SELECT k.Nachname,
   XMLCONCAT(
      XMLELEMENT ( NAME "vor", k.Vorname,
      XMLELEMENT ( NAME "nach", k.Nachname)
      ) AS "Ergebnis"
FROM Kunde k ;
```

Das Ergebnis:

Nachname	Ergebnis
Abelson	<vor>Abe</vor> <nach>Abelson</nach>
Bailey	<vor>Bill</vor> <nach>Bailey</nach>

XMLAGG

Die Aggregatfunktion `XMLAGG` benutzt XML-Dokumente oder Teile von XML-Dokumenten als Eingabe und erzeugt daraus in `GROUP BY`-Abfragen ein einziges XML-Dokument. Die Aggregation enthält einen »Wald« aus XML-Elementen. Das folgende Beispiel illustriert dies:

```
SELECT XMLELEMENT
   ( NAME "Ort",
      XMLATTRIBUTES ( k.Ort AS "name" ) ,
      XMLAGG (XMLELEMENT ( NAME "nach" k.Nachname )
             )
   ) AS "OrtsListe"
FROM Kunde k
GROUP BY Ort ;
```

Wenn diese Abfrage für die Tabelle `Kunde` ausgeführt wird, erhalten Sie das folgende Ergebnis:

```
OrtsListe
<Ort name="Decatur">
   <nach>Bailey</nach>
</Ort>
<Ort name="Philo">
   <nach>Stetson</nach>
   <nach>Stetson</nach>
   <nach>Wood</nach>
</Ort
<Ort name="Springfield">
   <nach>Abelson</nach>
</Ort>
```

XMLCOMMENT

Die Funktion `XMLCOMMENT` gibt einer Anwendung die Möglichkeit, einen XML-Kommentar zu erstellen. Ihre Syntax lautet:

```
XMLCOMMENT ( 'Inhalt Kommentar'
     [RETURNING
          { CONTENT | SEQUENCE } ] )
```

Zum Beispiel:

```
XMLCOMMENT ('Jede Nacht um 2:00 Uhr die Datenbank sichern!')
```

erstellt einen XML-Kommentar wie diesen:

```
<!-- Jede Nacht um 2:00 Uhr die Datenbank sichern! -->
```

XMLPARSE

Die Funktion `XMLPARSE` erzeugt einen XML-Wert, indem sie eine unbestätigte Analyse einer Zeichenfolge durchführt. Sie können sie wie in diesem Beispiel verwenden:

```
XMLPARSE (DOCUMENT '   TOLLE ARBEIT!   '
   PRESERVE WHITESPACE )
```

Dieser Code erzeugt einen XML-Wert, der entweder vom Typ `XML(UNTYPED DOCUMENT)` oder vom Typ `XML(ANY DOCUMENT)` ist, wobei die Art des Untertyps von Ihrer Implementierung abhängt.

XMLPI

Die Funktion `XMLPI` gibt Anwendungen die Möglichkeit, XML zu erzeugen, das Anweisungen verarbeiten kann. Die Syntax dieser Funktion sieht so aus:

```
XMLPI NAME Ziel
     [ , Zeichenfolge ]
     [RETURNING
          { CONTENT | SEQUENCE } ] )
```

Der Platzhalter *`Ziel`* identifiziert das Ziel der verarbeitenden Anweisung. Der Ausdruck *`Zeichenfolge`* steht für den Inhalt von PI. Diese Funktion erzeugt einen XML-Kommentar in der Form

```
<? Ziel Zeichenfolge ?>
```

XMLQUERY

Die Funktion XMLQUERY wertet einen XQuery-Ausdruck aus und übergibt das Ergebnis an die SQL-Anwendung. Die Syntax von XMLQUERY lautet:

```
XMLQUERY ( XQuery-Ausdruck
     [ PASSING { By REF | BY VALUE }
       ArgumentListe ]
     RETURNING { CONTENT | SEQUENCE }
     { BY REF | BY VALUE } )
```

Hier kommt ein Beispiel für den Einsatz von XMLQUERY:

```
SELECT maxTore,
     XMLQUERY (
          'for $Tore in
               /Spieler/Tore
          where /Spieler/Nachname = $var1
          return $Tore'
          PASSING BY VALUE
               'Mantle' AS var1,
          RETURNING SEQUENCE BY VALUE )
FROM ErsteLiga
```

XMLCAST

Die Funktion XMLCAST arbeitet so ähnlich wie eine normale SQL CAST-Funktion, wobei sie aber ein paar zusätzliche Einschränkungen kennt. XMLCAST gibt einer Anwendung die Möglichkeit, einen Wert von einem XML-Typ in einen anderen XML-Typ oder einen SQL-Typ umzuwandeln. Auch der Weg von SQL zu XML funktioniert. Zu den Einschränkungen, die für XMLCAST gelten, gehören:

- ✔ Mindestens einer der beiden Typen, die umgewandelt werden sollen, muss ein XML-Typ sein.
- ✔ Der Datentyp darf auf der SQL-Seite nicht zu den Auflistungs-, Zeilen- und Verweistypen gehören und darf auch kein strukturierter Typ sein.
- ✔ Es können nur Werte eines XML-Typs oder des SQL-Typs NULL in XML(UNTYPED DOCUMENT) oder XML(ANY DOCUMENT) umgewandelt werden.

Ein Beispiel:

```
XMLCAST ( Kunde.KundeName AS XML(UNTYPED CONTENT)
```

Die Funktion XMLCAST wird wie eine normale SQL CAST-Funktion verwendet. Der einzige Grund für ein spezielles Schlüsselwort liegt in den Einschränkungen, die ich in der Auflistung zusammengefasst habe.

Prädikate

Prädikate geben als Wert entweder *wahr* oder *falsch* zurück. Es gibt einige Prädikate, die speziell für XML zum SQL-Standard hinzugefügt worden sind.

DOCUMENT

Das Prädikat `DOCUMENT` soll festlegen, ob ein XML-Wert ein XML-Dokument ist. Es prüft, ob ein XML-Wert eine Instanz von `XML(ANY DOCUMENT)` oder `XML(UNTYPED DOCUMENT)` ist. Seine Syntax lautet:

```
XML-Wert IS [NOT]
   [ANY | UNTYPED] DOCUMENT
```

Wenn der Ausdruck wahr wird, gibt das Prädikat `TRUE` zurück, ansonsten gibt es `FALSE` zurück. Wenn der XML-Wert leer ist, gibt das Prädikat den Wert `UNKNOWN` zurück. Wenn Sie weder `ANY` noch `UNTYPED` vorgeben, wird der Standard, `ANY`, verwendet.

CONTENT

Das Prädikat `CONTENT` soll festlegen, ob ein XML-Wert eine Instanz von `XML(ANY CONTENT)` oder `XML(UNTYPED CONTENT)` ist. Seine Syntax lautet:

```
XML-Wert IS [NOT]
   [ANY | UNTYPED] CONTENT
```

Wenn Sie weder `ANY` noch `UNTYPED` vorgeben, wird der Standard, `ANY`, verwendet.

XMLEXISTS

Sie können dieses Prädikat, wie es der Name schon vermuten lässt, dazu verwenden, herauszufinden, ob ein Wert existiert. Die Syntax von `XMLEXISTS` lautet:

```
XMLEXISTS ( XQuery-Ausdruck
   [ ArgumentListe ])
```

Der XQuery-Ausdruck wird mit der Argumentliste ausgewertet. Wenn der Wert, der von XQuery abgefragt wird, ein SQL-Nullwert ist, ist das Ergebnis des Prädikats unbekannt. Wenn die Auswertung eine leere XQuery-Sequenz zurückgibt, liefert das Prädikat den Wert `FALSE`, anderenfalls `TRUE`. Sie können dieses Prädikat dazu nutzen, um herauszufinden, ob ein XML-Dokument einen bestimmten Inhalt hat, bevor Sie einen Teil dieses Inhalts in einem Ausdruck verwenden.

VALID

Das Prädikat `VALID` wird dazu verwendet, einen Wert auszuwerten, um herauszufinden, ob er im Inhalt eines registrierten XML-Schemas gültig ist. Die Syntax dieses Prädikats ist viel komplexer, als das bei den meisten anderen Prädikaten der Fall ist:

```
xmlWert IS [NOT] VALID
[gültige Identitätseinschränkung]
[gültige Zugehörigkeitsklausel]
```

Das Prädikat prüft, ob *xmlWert* zu einem der fünf XML-Typen XML(SEQUENCE), XML(ANY CONTENT), XML(UNTYPED CONTENT), XML(ANY DOCUMENT) oder XML(UNTYPED DOCUMENT) gehört. Zusätzlich kann es optional prüfen, ob die Gültigkeit des XML-Werts von einer Identitätseinschränkung abhängt oder ob es in Bezug auf ein bestimmtes XML-Schema gültig ist.

Für die Syntax der Option *Identitätseinschränkung* gibt es vier Möglichkeiten:

- ✔ WITHOUT IDENTITY CONSTRAINTS: Wenn *Identitätseinschränkung* nicht angegeben wird, wird WITHOUT IDENTITY CONSTRAINTS als Standard verwendet. Wenn DOCUMENT eingesetzt wird, handelt das Prädikat wie eine Kombination des Prädikats DOCUMENT und des Prädikats VALID mit gesetztem WITH IDENTITY CONSTRAINTS GLOBAL.
- ✔ WITH IDENTITY CONSTRAINTS GLOBAL: Diese Komponente der Syntax sagt aus, dass der Wert nicht nur gegen das XML-Schema, sondern auch gegen die XML-Regeln für ID/IDREF-Beziehungen geprüft wird.

ID und IDREF sind XML-Attributtypen, die Elemente eines Dokuments identifizieren.

- ✔ WITH IDENTITY CONSTRAINTS LOCAL: Diese Komponente der Syntax sagt aus, dass der Wert anhand des XML-Schemas, nicht aber anhand der XML-Regeln für ID/IDREF oder die XML-Schemaregeln für Identitätseinschränkungen geprüft wird.
- ✔ DOCUMENT: Diese Komponente der Syntax sagt aus, dass der XML-Werteausdruck ein Dokument ist, wenn WITH IDENTITY CONSTRAINTS GLOBAL mit *gültige Zugehörigkeitsklausel* gilt. Diese Klausel identifiziert das Schema, gegen das der Wert geprüft wird.

XML-Daten in SQL-Tabellen umwandeln

Bis jetzt haben wir uns, wenn es um die Beziehung zwischen SQL und XML geht, schwerpunktmäßig damit beschäftigt, SQL-Daten in XML umzuwandeln, um auf die Daten im Internet zugreifen zu können. Die neueste Erweiterung des SQL-Standards spricht das umgekehrte Problem an: die Umwandlung von XML-Daten in SQL-Tabellen, damit die Daten schnell und einfach mit Standard-SQL-Anweisungen abgefragt werden können. Die Pseudofunktion XMLTABLE führt diese Operation aus. Ihre Syntax lautet:

```
XMLTABLE ( [DeklarationNamensraum,]
XQuery–Ausdruck
[PASSING ArgumentListe]
COLUMNS XMLtbl–SpaltenDefinitionen
```

Für *ArgumentListe* gilt:

```
WerteAusdruck AS Identifizierer
```

Bei *XMLtbl-SpaltenDefinitionen* handelt es sich um eine kommaseparierte Liste von Spaltendefinitionen, die Folgendes enthalten kann:

```
SpaltenName FOR ORDINALITY
```

und/oder:

```
SpaltenName DatenTyp
[BY REF | BY VALUE]
[StandardKlausel]
[PATH XQuery-Ausdruck]
```

Das nächste Beispiel zeigt Ihnen, wie Sie XMLTABLE einsetzen können, um Daten eines XML-Dokuments in eine SQL-Pseudotabelle einzufügen. Eine Pseudotabelle ist nicht dauerhaft, verhält sich aber ansonsten wie eine normale SQL-Tabelle. Wenn Sie sie dauerhaft machen wollen, erledigen Sie das mit der Anweisung CREATE TABLE.

```
SELECT kundenTelefon.*
FROM
   KundenXML ,
   XMLTABLE(
      'for $m in
         $col/Kunde
      return
         $m'
   PASSING KundenXML.Kunde AS "col"
   COLUMNS
      "KundeName" CHARACTER (30) PATH 'KundeName' ,
      "Telefon" CHARACTER (13) PATH 'Telefon'’
   ) AS kundenTelefon
```

Wenn Sie diese Anweisung ablaufen lassen, erhalten Sie das folgende Ergebnis:

```
KundeName                    Telefon
---------------------------- -------------
Abe Abelson                  (0714)5551111
Bill Bailey                  (0714)5552222
Chuck Wood                   (0714)5553333

(3 rows in kundenTelefon)
```

Nicht vordefinierte Datentypen in XML abbilden

Zu den nicht vordefinierten Datentypen von SQL zählen Domänen, spezifische benutzerdefinierte Datentypen (UDT, *User Defined Type*), Zeilen, Arrays und Multisets. Sie können jeden dieser Typen als XML-Daten mit dem jeweils richtigen XML-Code abbilden. Die nächsten Abschnitte zeigen Beispiele für die Konvertierung dieser Datentypen.

Domänen

Um eine SQL-Domäne in XML abzubilden, müssen Sie zunächst über eine Domäne verfügen. Erstellen Sie für dieses Beispiel eine Domäne. Verwenden Sie dazu die Anweisung `CREATE DOMAIN`:

```
CREATE DOMAIN WestküsteUSA AS CHAR(2)
   CHECK (Staat IN ('CA', 'OR', 'WA', 'AK')) ;
```

Erstellen Sie nun eine Tabelle, die diese Domäne verwendet:

```
CREATE TABLE WestRegion (
   Kundenname          Character(20)          NOT NULL,
   Staat               WestküsteUSA           NOT NULL
   ) ;
```

Dies ist das XML-Schema zur Abbildung der Domäne als XML-Daten:

```
<xsd:simpleType>
   Name='DOMAIN.Verkäufe.WestküsteUSA'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='DOMAIN'
            schemaName='Verkäufe'
            typeName='WestküsteUSA'
            mappedType='CHAR_2'
            final='true'/>
      <xsd:appinfo>
   </xsd:annotation>

   <xsd:restriction base='CHAR_2'/>

</xsd:simpleType>
```

Die Anwendung dieser Umwandlung führt zu folgendem XML-Dokument:

```
<WestRegion>
   <zeile>
   ...
   <Staat>AK</Staat>
   ...
   </zeile>
   ...
</WestRegion>
```

Distinct UDT (Spezifischer benutzerdefinierter Datentyp)

Ein spezifischer benutzerdefinierter Datentyp gleicht einer Domäne, ist jedoch strenger typisiert. Ein Beispiel:

```
CREATE TYPE WestküsteUSA AS Character(2) FINAL ;
```

Das XML-Schema für die Konvertierung dieses Typs in XML sieht so aus:

```
<xsd:simpleType>
   Name='UDT.Verkaeufe.WestküsteUSA'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='DISTINCT'
            schemaName='Verkäufe'
            typeName='WestküsteUSA'
            mappedType='CHAR_2'
            final='true'/>
      <xsd:appinfo>
   </xsd:annotation>

   <xsd:restriction base='CHAR_2'/>

</xsd:simpleType>
```

Das Element, das auf diese Weise erzeugt wird, ist identisch mit dem, das für die vorangegangene Domäne erstellt wurde.

Row (Zeile)

Der Datentyp `ROW` ermöglicht es Ihnen, die Informationen einer ganzen Zeile in einem einzelnen Feld einer Tabellenzeile abzulegen. Sie können den `ROW`-Typ wie im folgenden Beispiel als Teil der Tabellendefinition erstellen:

```
CREATE TABLE Kontaktinfo (
   Name          CHARACTER (30)
   Phone         ROW (Zuhause CHAR (13), Arbeit CHAR (13))
) ;
```

Mit dem folgenden Schema können Sie diesen Typ in XML abbilden:

```
<xsd:complexType Name='ROW.1'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='ROW'>
            <sqlxml:field name='Zuhause'
               mappedType='CHAR_13'/>
            <sqlxml:field name='Arbeit'
               mappedType='CHAR_13'/>
         </sqlxml:sqltype>
      <xsd:appinfo>
   </xsd:annotation>

   <xsd:sequence>
      <xsd:element Name='Zuhause' nillable='true'
         Type='CHAR_13'/>
      <xsd:element Name='Arbeit' nillable='true'
         Type='CHAR_13'/>
   </xsd:sequence>

</xsd:complexType>
```

Diese Konvertierung könnte die folgenden XML-Daten für eine Spalte generieren:

```
<Rufnummer>
   <Zuhause>(0888)5551111</Zuhause>
   <Arbeit>(0888)5551212</Arbeit>
</Rufnummer>
```

Array

Sie können mehr als ein Element in einem einzigen Feld speichern, wenn Sie anstelle des Datentyps `ROW` den Datentyp `ARRAY` benutzen. In der Tabelle `Kontaktinfo` des vorherigen Beispiels könnten Sie die Rufnummer als Array deklarieren und anschließend ein XML-Schema erstellen, um das Datenfeld in XML-Code umzuwandeln.

```
CREATE TABLE Kontaktinfo (
   Name             CHARACTER(30),
   Rufnummer        CHARACTER(13) ARRAY[4]
) ;
```

Nun erfolgt die Umwandlung des Datentyps in XML. Dies geschieht mit folgendem Schema:

```
<xsd:complexType Name='ARRAY_4.CHAR_13'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='ARRAY'
                         maxElements='4'
                         mappedElementType='CHAR_13'/>
      </xsd:appinfo>
   </xsd:annotation>

   <xsd:sequence>
      <xsd:element Name='element'
      minOccurs='0' maxOccurs='4'
      nillable='true' type='CHAR_13'/>
   </xsd:sequence>

</xsd:complexType>
```

Das Ergebnis der Umwandlung könnte so aussehen:

```
<Rufnummer>
   <element>(0888)5551111</element>
   <element>xsi:nil='true'/>
   <element>(0888)5553434</element>
</Rufnummer>
```

Das Element im Array, das `xsi:nil='true'` enthält, zeigt an, dass die zweite Rufnummer in der Quelltabelle einen Nullwert enthält.

Multiset

Die Rufnummer des vorherigen Beispiels könnte auch in einem Multiset gespeichert werden. Um ein Multiset abzubilden, gehen Sie wie folgt vor:

```
CREATE TABLE Kontaktinfo (
   Name             CHARACTER (30),
   Rufnummer        CHARACTER (13) MULTISET
) ;
```

Nun erfolgt die Umwandlung des Datentyps in XML. Dies geschieht mit folgendem Schema:

```
<xsd:complexType Name='MULTISET.CHAR_13'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='MULTISET'
                         mappedElementType='CHAR_13'/>
      </xsd:appinfo>
   </xsd:annotation>

   <xsd:sequence>
      <xsd:element Name='element'
      minOccurs='0' maxOccurs='unbounded'
      nillable='true' type='CHAR_13'/>
   </xsd:sequence>

</xsd:complexType>
```

Das Ergebnis der Umwandlung könnte so aussehen:

```
<Rufnummer>
   <element>(0888)5551111</element>
   <element>xsi:nil='true'/>
   <element>(0888)5553434</element>
</Rufnummer>
```

Die Hochzeit von SQL und XML

SQL stellt einen weltweiten Standard für eine hochstrukturierte Speicherung von Daten zur Verfügung. Diese Struktur ermöglicht es Benutzern, extrem große Datenspeicher zu verwalten und Daten aus diesen Speichern sehr effizient abzurufen. XML ist zu einem De-facto-Standard für den Austausch von Daten zwischen inkompatiblen Systemen, insbesondere über das Internet, geworden. Indem Sie diese beiden mächtigen Methoden zusammenbringen, steigern Sie den Wert von beiden. SQL kann nun mit Daten umgehen, die nicht in die strikten relationalen Vorgaben passen, die ursprünglich von Dr. Codd definiert worden waren. XML kann Daten effizient aus SQL übernehmen oder an SQL übergeben. Das Ergebnis sind leichter zugängliche Daten, die einfacher gemeinsam zu nutzen sind. Und im Grunde ist »gemeinsam« ja das Wesen jeder Hochzeit.

Teil VI

SQL für Fortgeschrittene

In diesem Teil ...

- ✔ Cursor erstellen
- ✔ Zusammengesetzte Anweisungen erstellen
- ✔ Fehler behandeln
- ✔ Trigger einsetzen

Cursor

19

In diesem Kapitel

- Die Reichweite eines Cursors mit der Anweisung DECLARE festlegen
- Einen Cursor öffnen
- Daten satzweise lesen
- Einen Cursor schließen

Eine zentrale Inkompatibilität zwischen SQL und den gebräuchlichen Sprachen zur Anwendungsentwicklung besteht darin, dass SQL ganze Datenmengen auf einmal abarbeitet, während prozedurale Sprachen Tabellen datensatzweise bearbeiten. Ein *Cursor* stattet SQL mit der Fähigkeit aus, einzelne Zeilen einer Tabelle zu lesen, zu ändern oder zu löschen, damit Sie SQL in Verbindung mit einer Anwendung verwenden können, die in einer allgemeinen Sprache geschrieben ist.

Ein Cursor funktioniert wie ein Zeiger, der auf eine bestimmte Zeile gerichtet ist. Wenn ein Cursor aktiv ist, können Sie die Anweisungen SELECT, UPDATE oder DELETE auf die Zeile anwenden, auf die er zeigt.

Cursors sind wertvoll, wenn Sie auf bestimmte Zeilen einer Tabelle zugreifen wollen, um deren Inhalt zu prüfen und verschiedene Aktionen in Abhängigkeit von diesem Inhalt auszuführen. SQL kann eine solche Folge von Operationen nicht selbst ausführen. SQL kann zwar die Zeilen abrufen, aber prozedurale Sprachen sind besser dazu geeignet, Entscheidungen zu treffen, die auf dem Inhalt einzelner Felder basieren. Mit einem Cursor kann SQL die Daten einer Tabelle zeilenweise abrufen und dann die Zeilen einzeln zur Weiterverarbeitung an den prozeduralen Code übergeben. Wenn Sie den SQL-Code in einer Schleife unterbringen, können Sie eine komplette Tabelle zeilenweise abarbeiten.

Der folgende Pseudocode zeigt, wie der Programmablauf bei eingebettetem SQL üblicherweise aussieht:

```
EXEC SQL DECLARE CURSOR Anweisung
EXEC SQL OPEN Anweisung
Testen, ob Tabellenende erreicht wurde
   Prozeduraler Code
   EXEC SQL FETCH
   Prozeduraler Code
   Testen, ob Tabellenende erreicht wurde
Schleifenende
EXEC SQL CLOSE Befehl
Prozeduraler Code
```

Die SQL-Anweisungen in diesem Listing lauten DECLARE, OPEN, FETCH und CLOSE. In den folgenden Abschnitten werde ich jede dieser Anweisungen im Detail behandeln.

Wenn Sie die gewünschte Operation mit normalen SQL-Anweisungen (eine Datenmenge auf einmal) ausführen können, sollten Sie dieses Verfahren wählen. Wollen Sie das gewünschte Ergebnis nicht nur mit SQL erzielen, sollten Sie einen Cursor deklarieren, um die Zeilen einzeln abzurufen und an die Host-Sprache Ihres Systems zu übergeben.

Einen Cursor deklarieren

Ehe Sie einen Cursor benutzen können, müssen Sie dem DBMS bekannt geben, dass es ihn gibt. Sie erledigen dies mit der Anweisung DECLARE CURSOR, die keine Aktion auslöst, sondern dem DBMS nur den Namen des Cursors mitteilt und angibt, welche Abfrage er verwenden soll. Die Anweisung DECLARE CURSOR hat die folgende Syntax:

```
DECLARE CursorName [<Cursor-Sensitivität>][<Cursor-Beweglichkeit>]
CURSOR [<Cursor-Beibehaltung>] [<Ergebnisrückgabe>]
FOR Abfrageausdruck
   [ORDER BY Order-by-Ausdruck]
   [FOR Aktualisierbarkeitsausdruck] ;
```

Anmerkung: Der Name identifiziert einen Cursor eindeutig und muss sich deshalb im aktuellen Modul oder in der aktuellen Kompilationseinheit von anderen Cursorn unterscheiden.

Um Ihre Anwendung lesbarer zu machen, sollten Sie dem Cursor einen aussagekräftigen Namen geben. Leiten Sie den Namen von den Daten ab, die die Abfrage abrufen soll, oder bringen Sie die Operation zum Ausdruck, die Ihr prozeduraler Code mit den Daten ausführt.

Zu den Merkmalen, die Sie angeben müssen, wenn Sie einen Cursor deklarieren, gehören:

- ✔ **Cursor-Empfindlichkeit:** Wählen Sie SENSITIVE, INSENSITIVE oder ASENSITIVE.
- ✔ **Cursor-Scrollbarkeit:** Wählen Sie SCROLL oder NO SCROLL.
- ✔ **Cursor-Beibehaltung:** Wählen Sie WITH HOLD oder WITHOUT HOLD.
- ✔ **Cursor-Ergebnisrückgabe:** Wählen Sie WITH RETURN oder WITHOUT RETURN.

Der Abfrageausdruck

Der *Abfrageausdruck* kann aus einer beliebigen gültigen SELECT-Anweisung bestehen. Die Datensätze, die diese Anweisung abruft, werden vom Cursor einzeln durchlaufen. Diese Datensätze definieren die Reichweite des Cursors.

Die Abfrage wird noch nicht ausgeführt, wenn die Anweisung DECLARE CURSOR gelesen wird. Sie können erst dann Daten erhalten, wenn die Anweisung OPEN ausgeführt wird. Die zeilenweise Bearbeitung der Daten beginnt nach dem Eintritt in die Schleife, die die Anweisung FETCH einschließt.

Die Klausel ORDER BY

Vielleicht möchten Sie die abgerufenen Daten, je nachdem, was Ihr prozeduraler Code damit vorhat, in einer bestimmten Reihenfolge abarbeiten. Sie können die Daten mit der optionalen Klausel ORDER BY sortieren, bevor Sie sie verarbeiten. Die Klausel hat die folgende Syntax:

```
ORDER BY Sortierfolge [ , Sortierfolge ]...
```

Sie können mehrere Sortierfolgen festlegen, die jeweils die folgende Syntax haben:

```
( Spaltenname )[ COLLATE BY CollationName ] [ ASC | DESC ]
```

Der Name der Spalte, die Sie zum Sortieren verwenden wollen, muss in der SELECT-Liste des Abfrageausdrucks enthalten sein. Sie können keine Spalten verwenden, die zwar in der Tabelle, aber nicht in der SELECT-Liste der Abfrage stehen. Angenommen, Sie möchten für einige Datensätze der Tabelle Kunde eine Operation ausführen, was von SQL nicht unterstützt wird. Dann können Sie eine DECLARE CURSOR-Anweisung wie diese benutzen:

```
DECLARE kunde1 CURSOR FOR
   SELECT KundenID, Vorname, Nachname, Stadt, Land, Telefon
      FROM Kunde
   ORDER BY Land, Nachname, Vorname ;
```

In diesem Beispiel gibt die Anweisung SELECT Datensätze zurück, die zuerst nach Land, dann nach Nachname und zuletzt nach Vorname sortiert werden. Die Anweisung findet die Kunden in Irland vor den Kunden in Italien. Innerhalb eines Landes werden die Kunden nach ihrem Nachnamen sortiert. Beispielsweise steht *Aaron* vor *Abbott*. Wenn mehrere Kunden denselben Nachnamen haben, werden sie nach ihrem Vornamen sortiert. *George Aaron* steht vor *Henry Aaron*.

Mussten Sie jemals vierzig Kopien eines zwanzigseitigen Dokuments auf einem Fotokopierer ohne Sortierer erstellen? Was für ein Aufwand! Sie müssen auf dem Tisch zwanzig Haufen bilden und dann vierzigmal an diesen Haufen entlanggehen, um auf jeden Haufen jeweils ein Blatt zu legen. Dieser Vorgang wird als *Sortierfolge* (englisch *Collation*) bezeichnet. Etwas Ähnliches gibt es auch in SQL.

Eine SQL-*Sortierfolge (Collation)* ist ein Satz von Regeln, der festlegt, wie Zeichenfolgen in einem bestimmten Zeichensatz miteinander verglichen werden. Jeder Zeichensatz hat eine Standardfolge für das Sortieren von Zeichenfolgen. Sie können aber einer Spalte eine andere Sortierfolge als die Standardfolge zuweisen. Dies erledigen Sie mit der optionalen Klausel COLLATE BY. Wahrscheinlich unterstützt Ihre Implementierung mehrere der allgemein üblichen Sortierfolgen. Sie können eine auswählen und dann mit den Schlüsselwörtern ASC und DESC festlegen, ob die Sortierung *aufsteigend* oder *absteigend* erfolgen soll.

In einer DECLARE CURSOR-Anweisung können Sie außerdem eine berechnete Spalte definieren, die in den zugrunde liegenden Tabellen nicht existiert. Eine solche Spalte können Sie nicht in der Klausel ORDER BY verwenden. Mit dem Schlüsselwort AS können Sie einer solchen Spalte in dem Abfrageausdruck der Anweisung DECLARE CURSOR einen Namen geben, der es Ihnen später möglich macht, diese Spalte zu identifizieren. Betrachten Sie das folgende Beispiel:

```
DECLARE Einnahme CURSOR FOR
   SELECT Modell, Einheiten, Preis,
         Einheiten * Preis AS Umsatz
      FROM Rechnungsposition
   ORDER BY Modell, Umsatz DESC ;
```

Dieses Beispiel benutzt die Standardsortierfolge, weil die Klausel ORDER BY keine COLLATE BY-Klausel enthält. Beachten Sie, dass die vierte Spalte in der SELECT-Liste aus der zweiten und dritten Spalte berechnet wird. In meinem Beispiel sortiert die Klausel ORDER BY die Daten zuerst nach Modell und dann nach Umsatz. Das Schlüsselwort DESC legt fest, dass die Sortierung von Umsatz absteigend ist; die Rechnungspositionen mit dem höchsten Umsatz pro Modell werden zuerst verarbeitet.

Die standardmäßige Sortierfolge in der Klausel ORDER BY ist aufsteigend. Wenn eine Sortieranweisung eine DESC-Sortierung umfasst und die folgende Sortierung ebenfalls absteigend sein soll, müssen Sie diese Sortierung auch als DESC festlegen. Zum Beispiel:

```
ORDER BY A, B DESC, C, D, E, F
```

ist gleichbedeutend mit:

```
ORDER BY A ASC, B DESC, C ASC, D ASC, E ASC, F ASC
```

Die Klausel FOR UPDATE

Manchmal wollen Sie vielleicht Zeilen ändern oder löschen, auf die Sie mit einem Cursor zugreifen. Bei anderen Gelegenheiten wollen Sie Änderungen oder Löschvorgänge verhindern. SQL lässt Sie diese Fälle kontrollieren, indem es Ihnen die Aktualisierbarkeitsklausel der Anweisung DECLARE CURSOR zur Verfügung stellt. Wenn Sie verhindern wollen, dass Zeilen innerhalb der Reichweite des Cursors geändert oder gelöscht werden, geben Sie folgende Klausel ein:

```
FOR READ ONLY
```

Um Änderungen auf bestimmte Spalten zu beschränken, geben Sie Folgendes ein:

```
FOR UPDATE OF Spaltenname [, Spaltenname ]...
```

Alle aufgeführten Spalten müssen im Abfrageausdruck von DECLARE CURSOR vorkommen. Wenn die Cursor-Deklaration keine Aktualisierbarkeitsklausel enthält, lassen sich standardmäßig alle Spalten des Abfrageausdrucks, auf die der Cursor zeigt, mit der Anweisung UPDATE ändern, und DELETE löscht diese Zeile.

Sensitivität

Der Abfrageausdruck in DECLARE CURSOR ermittelt die Datensätze, die in der Reichweite des Cursors liegen. Dabei können folgende Probleme auftreten: Was passiert, wenn eine Abfrage in Ihrem Programm zwischen den Befehlen OPEN und CLOSE den Inhalt einiger Datensätze so ändert, dass diese die Abfrage nicht mehr erfüllen? Was passiert, wenn eine Abfrage einige dieser Datensätze komplett löscht? Verarbeitet der Cursor dann alle ursprünglich ausgewählten Zeilen oder erkennt er die neue Situation und ignoriert die Zeilen, die die Abfrage nicht mehr erfüllen oder die gelöscht worden sind?

Eine normale SQL-Anweisung wie UPDATE, INSERT oder DELETE betrifft eine Menge von Zeilen einer Datenbanktabelle (oder die gesamte Tabelle). Während eine solche Abfrage läuft, schützt der Transaktionsmechanismus von SQL die Daten vor Einflüssen anderer Anweisungen, die gleichzeitig auf dieselben Daten zugreifen. Wenn Sie dagegen mit einem Cursor arbeiten, öffnen Sie schädlichen Interaktionen Tür und Tor. Das Risiko besteht von dem Moment an, an dem Sie einen Cursor öffnen, bis zu dem Moment, an dem Sie ihn wieder schließen. Wenn Sie einen Cursor öffnen, mit der Verarbeitung einer Tabelle beginnen und dann einen zweiten Cursor öffnen, während der erste noch aktiv ist, können die Aktionen, die Sie mit dem zweiten Cursor ausführen, die Daten verändern, die der erste Cursor sieht.

Wenn Sie Daten in Spalten, die zu einem DECLARE CURSOR-Abfrageausdruck gehören, ändern, nachdem einige, aber noch nicht alle Zeilen der Abfrage verarbeitet worden sind, lösen Sie ein regelrechtes Chaos aus. Ihre Ergebnisse werden wahrscheinlich inkonsistent und irreführend sein. Um dieses Problem zu vermeiden, können Sie Ihren Cursor mit dem Schlüsselwort INSENSITIVE in der Cursor-Deklaration gegen alle Änderungsanweisungen in seiner Reichweite unempfindlich machen. Solange Ihr Cursor geöffnet ist, ist er dann unempfindlich gegen Tabellenänderungen, die Zeilen in seiner Reichweite betreffen. Ein Cursor kann nicht gleichzeitig unempfindlich und aktualisierbar sein. Ein unempfindlicher Cursor kann immer nur lesen.

Stellen Sie sich beispielsweise vor, dass Sie diese Abfragen schreiben:

```
DECLARE C1 CURSOR FOR SELECT * FROM Mitarbeiter
   ORDER BY Gehalt ;
DECLARE C2 CURSOR FOR SELECT * FROM Mitarbeiter
   FOR UPDATE OF Gehalt ;
```

Wenn Sie jetzt beide Cursor öffnen, einige Datensätze mit C1 lesen und dann ein Gehalt mit C2 erhöhen, kann dies dazu führen, dass Sie einen Datensatz, den Sie bereits mit C1 gelesen haben, später noch einmal mit C1 abrufen.

Die besonderen wechselseitigen Beeinflussungen, die bei mehreren geöffneten Cursorn und Mengenoperationen möglich sind, gehören zu den Problemen des gleichzeitigen (konkurrierenden) Datenzugriffs, die durch die Isolierung von Transaktionen vermieden werden. Wenn Sie so arbeiten, erzeugen Sie Ihre Probleme selbst. Denken Sie deshalb daran: Arbeiten Sie nicht mit mehreren geöffneten Cursorn. In Kapitel 15 finden Sie Informationen zur Isolierung von Transaktionen.

Die Cursor-Sensitivität ist standardmäßig auf `ASENSITIVE` eingestellt. Die Bedeutung von `ASENSITIVE` ist von der jeweiligen SQL-Implementierung abhängig. `ASENSITIVE` kann, je nachdem, mit welcher Implementierung Sie arbeiten, empfindlich (`SENSITIVE`) oder unempfindlich (`INSENSITIVE`) bedeuten. Lesen Sie das Handbuch zu Ihrem System, um eine Antwort auf diese Frage zu erhalten.

Scrollbarkeit

Scrollbarkeit (englisch *scrollability*) ist eine Fähigkeit, Cursor in einer Ergebnismenge zu verschieben. Mit dem Schlüsselwort `SCROLL` können Sie dem Cursor in der Deklaration die Fähigkeit geben, in beliebiger Reihenfolge auf die Zeilen zuzugreifen. Dabei wird die Bewegung des Cursors durch die Anweisung `FETCH` gesteuert. Ich beschreibe `FETCH` weiter hinten in diesem Kapitel.

Einen Cursor öffnen

Die Anweisung `DECLARE CURSOR` legt zwar die Datensätze fest, die die Reichweite des Cursors bilden sollen, führt aber keine Aktion aus; denn `DECLARE` ist eine Deklaration und keine ausführbare Anweisung. Der Cursor wird erst mit der Anweisung `OPEN` erstellt:

```
OPEN CursorName;
```

Der Cursor, den ich bei der Klausel `ORDER BY` benutzt habe, wird folgendermaßen geöffnet:

```
DECLARE Einnahme CURSOR FOR
   SELECT Modell, Einheiten, Preis,
         Einheiten * Preis AS Umsatz
      FROM Rechnungsposition
   ORDER BY Modell, Umsatz DESC ;
OPEN Einnahme ;
```

Sie können Datensätze erst dann mit dem Cursor abfragen, wenn Sie den Cursor geöffnet haben. Wenn Sie einen Cursor öffnen, werden die Werte der Variablen festgesetzt, auf die in der Anweisung `DECLARE CURSOR` verwiesen wird. Dasselbe gilt für Datums- und Zeit-Funktionen, die in der Deklaration verwendet werden. Betrachten Sie das folgende Beispiel:

```
DECLARE CURSOR C1 FOR SELECT * FROM Aufträge
   WHERE Aufträge.Kunde = :name
      AND FälligDatum < CURRENT_DATE ;
Name := 'Acme Co';     /* Ein Befehl der Host-Sprache */
OPEN C1;
Name := 'Omega Inc.';  /* Ein weiterer Host-Befehl */
...
UPDATE Aufträge SET Fällig_Datum = CURRENT_DATE;
```

Die Anweisung `OPEN` legt die Werte aller Variablen, auf die in der Cursor-Deklaration verwiesen wird, und alle Datums- und Zeitfunktionen fest. Deshalb hat die zweite Zuweisung der Namensvariablen (`Name := 'Omega Inc.'`) keine Auswirkung auf die Datensätze, die der Cursor abruft. (Dieser Wert von `Name` wird beim nächsten Öffnen von `C1` verwendet.) Selbst wenn die Anweisung `OPEN` eine Minute vor Mitternacht und die Anweisung `UPDATE` eine Minute nach Mitternacht ausgeführt werden, enthält `CURRENT_DATE` in der `UPDATE`-Anweisung den Wert, den die Funktion beim Ausführen von `OPEN` hat. Dies gilt selbst dann, wenn `DECLARE CURSOR` nicht auf diese Datums- und Zeitfunktion verweist.

Werte (von Datums- und Zeitangaben) fixieren

Wie ich im Abschnitt *Einen Cursor öffnen* beschreibe, fixiert die `OPEN`-Anweisung den Wert aller Variablen, auf die in der Cursordeklaration verwiesen wird. Die Anweisung macht nicht nur die Inhalte von Variablen, sondern auch Datums- und Zeitfunktionen quasi statisch. Ein ähnlicher Vorgang findet bei `SET`-Operationen statt. Betrachten Sie folgendes Beispiel:

```
UPDATE Aufträge
SET WiedervorlageDatum = CURRENT_DATE WHERE....;
```

Angenommen, Sie verwalten sehr viele Aufträge. Sie beginnen um eine Minute vor Mitternacht mit der Ausführung dieser Anweisung. Um Mitternacht läuft die Anweisung immer noch und sie wird erst um fünf Minuten nach Mitternacht beendet. Das spielt keine Rolle. Wenn eine Anweisung einen Verweis auf `CURRENT_DATE` (oder `TIME` oder `TIMESTAMP`) enthält, wird der Wert auf das Datum und/oder die Zeit gesetzt, das oder die beim Beginn der Anweisung galt, womit alle Datensätze in der Tabelle `Aufträge` dasselbe `WiedervorlageDatum` erhalten. Auf ähnliche Weise arbeitet eine Anweisung, die auf `TIMESTAMP` verweist, nämlich nur mit einem einzigen Zeitstempelwert – egal, wie lange die Anweisung läuft.

Diese Regel hat einige interessante Auswirkungen. Betrachten Sie beispielsweise folgende Anweisung:

```
UPDATE Mitarbeiter SET Key=CURRENT_TIMESTAMP;
```

Falls Sie erwarten, dass diese Anweisung jedem Mitarbeiter in der Spalte Key einen eindeutigen Wert zuweist, werden Sie enttäuscht sein: Der Wert ist in allen Datensätzen gleich. Sie müssen sich ein anderes Verfahren ausdenken, um einen eindeutigen Schlüssel zu erzeugen.

Wenn die Anweisung OPEN die Datums- und Zeitwerte für alle Anweisungen fixiert, die auf den Cursor verweisen, behandelt sie damit alle diese Anweisungen wie eine erweiterte Anweisung.

Daten aus einer einzelnen Zeile abrufen

Ein Cursor wird in drei Schritten verarbeitet.

1. Die Anweisung DECLARE CURSOR legt den Namen und die Reichweite eines Cursors fest.
2. Die Anweisung OPEN sammelt die Zeilen ein, die durch den Abfrageausdruck von DECLARE CURSOR ausgewählt werden.
3. Die Anweisung FETCH ruft diese Daten dann ab.

Der Cursor zeigt entweder auf eine Zeile innerhalb seiner Reichweite, auf den leeren Raum unmittelbar vor der ersten Zeile, auf den leeren Raum zwischen zwei Zeilen oder auf den leeren Raum unmittelbar hinter der letzten Zeile. Mit der Orientierungsklausel können Sie in der FETCH-Anweisung angeben, wohin der Cursor zeigen soll.

Syntax

Die Syntax für die Anweisung FETCH lautet:

```
FETCH [[Orientierung] FROM] CursorName
   INTO Zielspezifikation [, Zielspezifikation ]...
```

Es gibt sieben Orientierungen:

- ✔ NEXT
- ✔ PRIOR
- ✔ FIRST
- ✔ LAST
- ✔ ABSOLUTE
- ✔ RELATIVE
- ✔ <einfache Wertspezifikation>

Die Standardoption lautet NEXT. In den SQL-Versionen vor SQL-92 war dies die *einzige* verfügbare Orientierung. Diese Orientierung bewegt den Cursor von seiner aktuellen Position zur nächsten Zeile der Datenmenge, die vom Abfrageausdruck zusammengestellt worden ist. Wenn der Cursor vor der ersten Zeile steht, bewegt er sich zur ersten Zeile. Wenn er auf die Zeile *n* zeigt, bewegt er sich zur Zeile *n+1*. Wenn der Cursor auf die letzte Zeile in der Rückgabemenge zeigt, bewegt er sich hinter diese Zeile und in der Systemvariablen SQLSTATE wird vermerkt, dass keine Daten vorhanden sind. (In Kapitel 21 finden Sie detailliertere Informationen über SQLSTATE und andere Fehlerbehandlungsmöglichkeiten von SQL.)

Die Zielspezifikationen sind entweder Host-Variablen oder Parameter, je nachdem, ob der Cursor von eingebettetem SQL-Code oder von der Modulsprache benutzt wird. Die Anzahl und die Typen der Zielspezifikationen müssen mit der Anzahl und den Typen der Spalten im Abfrageausdruck von DECLARE CURSOR übereinstimmen. Wenn Sie beispielsweise mit eingebettetem SQL-Code arbeiten und fünf Werte aus einer Tabellenzeile herauslesen, muss es fünf Host-Variablen passenden Typs geben, um diese Werte aufzunehmen.

Die Orientierung eines scrollbaren Cursors

Weil der SQL-Cursor scrollbar ist, gibt es zusätzlich zu NEXT noch weitere Bewegungsrichtungen. Mit der Option PRIOR bewegen Sie den Cursor zur unmittelbar vorangehenden Zeile, mit FIRST setzen Sie ihn auf die erste und mit LAST auf die letzte Zeile.

Bei den Optionen ABSOLUTE und RELATIVE müssen Sie zusätzlich eine Ganzzahl angeben. Beispielsweise setzt FETCH ABSOLUTE 7 den Cursor auf die siebte Zeile, vom Anfang der Datenmenge an gerechnet. FETCH RELATIVE 7 bewegt den Cursor von seiner jetzigen Position aus sieben Zeilen weiter zum Ende der Datenmenge hin. Bei FETCH RELATIVE 0 bleibt der Cursor an seiner gegenwärtigen Position stehen.

FETCH RELATIVE 1 hat dieselbe Wirkung wie FETCH NEXT. FETCH RELATIVE –1 hat dieselbe Wirkung wie FETCH PRIOR. FETCH ABSOLUTE 1 springt zur ersten Zeile in der Menge, FETCH ABSOLUTE 2 springt zur zweiten Zeile und so weiter. FETCH ABSOLUTE –1 geht zur letzten Zeile in der Menge, FETCH ABSOLUTE –2 zur vorletzten Zeile und so weiter. FETCH ABSOLUTE 0 gibt den Fehlercode zurück, der Aufschluss darüber gibt, dass keine Daten gelesen wurden. Dasselbe gilt für FETCH ABSOLUTE 17, wenn die Ergebnisdatenmenge nur aus 16 Zeilen besteht. FETCH <einfache Wertspezifikation> gibt die Zeile zurück, die Sie mit dem einfachen Wert spezifiziert haben.

Cursor-Zeilen löschen oder ändern

Sie können die Zeile, auf die der Cursor aktuell zeigt, ändern oder löschen. Die Löschanweisung DELETE hat folgende Syntax:

```
DELETE FROM Tabellenname
   WHERE CURRENT OF CursorName ;
```

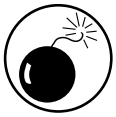
Wenn der Cursor nicht auf eine Zeile zeigt, erzeugt diese Anweisung eine Fehlerbedingung, und es werden keine Daten gelöscht.

Die Änderungsanweisung UPDATE hat folgende Syntax:

```
UPDATE Tabellenname
   SET Spaltenname = Wert [, Spaltenname = Wert]...
   WHERE CURRENT OF CursorName ;
```

Der Wert, den Sie den angegebenen Spalten zuweisen, muss entweder aus einem Ausdruck oder aus dem Schlüsselwort DEFAULT bestehen. Wenn der Versuch einer Änderung einen Fehler zurückgibt, werden die Daten nicht geändert.

Einen Cursor schließen



Wenn Sie den Cursor nicht mehr benötigen, sollten Sie ihn sofort schließen. Einen Cursor geöffnet zu lassen, könnte zu Schäden an Ihren Daten führen. Außerdem belegen geöffnete Cursor Systemressourcen.

Wenn Sie einen Cursor schließen, der unempfindlich gegen Änderungen war, und ihn dann wieder öffnen, zeigt er danach die Änderungen an.

Sie können den Cursor Einnahme, den ich weiter oben in diesem Kapitel angelegt habe, mit einer einfachen Anweisung wie dieser wieder schließen:

```
CLOSE Einnahme ;
```

Prozedurale Möglichkeiten mit dauerhaft gespeicherten Modulen schaffen

20

In diesem Kapitel

- Zusammengesetzte Anweisungen mit Atomarität, Cursorn, Variablen und Bedingungen ausstatten
- Anweisungen zur Ablaufsteuerung regulieren
- Schleifen ausführen, die Schleifen ausführen, die Schleifen ausführen
- Gespeicherte Prozeduren und gespeicherte Funktionen abrufen und benutzen
- Rechte zuweisen, gespeicherte Module erstellen und anwenden

Einige führende Praktiker der Datenbanktechnologie arbeiten seit Jahren an der Standardisierung von SQL. Selbst unmittelbar, nachdem ein Standard verabschiedet und von der weltweiten Gemeinschaft der Datenbankanwender akzeptiert worden ist, verlangsamt sich die Arbeit am nächsten Standard nicht. Zwischen der Freigabe von SQL-92 und der Freigabe der ersten Komponenten von SQL:1999 lagen sieben Jahre. Dies war jedoch keine Ruhezeit. In diesem Zeitabschnitt veröffentlichten das ANSI und die ISO einen Zusatz zu SQL-92 namens *SQL-92/PSM* (*Persistent Stored Modules*). Dabei handelt es sich um Module, die dauerhaft (englisch *persistent*) gespeichert werden und zur Verfügung stehen. Dieser Zusatz bildete die Grundlage für den gleichnamigen Teil von SQL:1999. SQL/PSM definiert eine Reihe von Anweisungen, die SQL um Strukturen zur Ablaufsteuerung erweitern, die mit den entsprechenden Strukturen kompletter Programmiersprachen vergleichbar sind. Damit erhalten Sie die Möglichkeit, SQL zu verwenden, um Aufgaben auszuführen, für die Programmierer bis dahin zwingend andere Werkzeuge einsetzen mussten. Sie können sich vielleicht vorstellen, wie das Leben in der Zeit der Höhlenmenschen von 1992 ausgesehen hat, als man ständig zwischen SQL und der prozeduralen Host-Sprache hin und her wechseln musste, um einfach nur seine Arbeit zu erledigen.

Zusammengesetzte Anweisungen

In diesem Buch wird immer wieder betont, dass SQL eine nichtprozedurale Sprache ist, die mit Datenmengen anstatt mit einzelnen Datensätzen arbeitet. In Anbetracht der Funktionen, die in diesem Kapitel behandelt werden, gilt diese Aussage nicht mehr so uneingeschränkt wie früher. Obwohl SQL immer noch mit Datenmengen arbeitet, hat es einige prozedurale Eigenschaften bekommen.

Das uralte SQL (definiert durch SQL-92) folgt nicht dem prozeduralen Modell, bei dem Anweisungen schrittweise nacheinander ausgeführt werden mussten, um das gewünschte Ergebnis zu erzielen. Die früheren SQL-Anweisungen sind somit eigenständige Einheiten gewesen – eventuell eingebettet in ein C++- oder Visual-Basic-Programm. Bei diesen frühen Versionen von SQL war das Absetzen einer Abfrage oder das Ausführen anderer Operationen in Form einer Reihe von SQL-Anweisungen ein entmutigender Vorgang. Diese »komplizierten« Aktivitäten wurden sofort durch Einbrüche im Leistungsverhalten Ihres Netzwerks bestraft. SQL:1999 und die späteren Versionen lassen zusammengesetzte Anweisungen zu, die zwar aus einzelnen Anweisungen bestehen, aber als Einheit ausgeführt werden und damit Verstopfungen im Netz vermeiden.

Alle Anweisungen, die in einer zusammengesetzten Anweisung zusammengefasst sind, werden durch das Schlüsselwort `BEGIN` am Anfang der Anweisung und das Schlüsselwort `END` am Ende der Anweisung eingeschlossen. Um beispielsweise Daten in mehrere miteinander verknüpfte Tabellen einzufügen, wird eine Syntax verwendet, die dem folgenden Beispiel ähnelt:

```
void main {
   EXEC SQL
   BEGIN
      INSERT INTO Teilnehmer (TeilnehmerID, Vorname, Nachname)
         VALUES (:sid, :svorname, :snachname) ;
      INSERT INTO Kurse (KursID, Kurs, TeilnehmerID)
         VALUES (:kid, :kkurs, :sid) ;
      INSERT INTO Gebühren (TeilnehmerID, Kurs, Gebühr)
         VALUES (:sid, :kkurs, :kgebühr)
   END ;
/* SQLSTATE auf Fehler prüfen */
}
```

Dieses Fragment eines C-Programms enthält eine eingebettete zusammengesetzte SQL-Anweisung. Der Kommentar mit `SQLSTATE` hat mit der Fehlerbearbeitung zu tun. Falls die zusammengesetzte Anweisung nicht erfolgreich ausgeführt wird, wird dem Statusparameter `SQLSTATE` ein Fehlercode zugewiesen. Der Kommentar soll Sie daran erinnern, dass in einem echten Programm an dieser Stelle der Code zur Fehlerbehandlung eingefügt werden muss. Fehlerbehandlung wird in Kapitel 21 ausführlich beschrieben.

Atomarität

Zusammengesetzte Anweisungen führen eine Fehlermöglichkeit ein, die bei einfachen SQL-Anweisungen nicht existiert. Eine einfache SQL-Anweisung wird entweder erfolgreich abgeschlossen oder nicht. Falls sie nicht erfolgreich abgeschlossen wird, bleibt die Datenbank unverändert. Dies ist aber nicht notwendigerweise der Fall, wenn eine zusammengesetzte Anweisung einen Fehler hervorruft.

Schauen Sie sich noch einmal das Beispiel aus dem letzten Abschnitt an. Was passiert, wenn die beiden `INSERT`-Anweisungen in den Tabellen `Teilnehmer` und `Kurse` erfolgreich ausgeführt werden, aber das `INSERT` in der Tabelle `Gebühren` wegen einer Störung durch einen an-

deren Benutzer scheiterte? Ein Teilnehmer wäre für einen Kurs eingeschrieben, würde darüber aber keine Gebührenrechnung erhalten. Diese Art von Fehler wäre für die Finanzen eines Weiterbildungsinstituts nicht sehr vorteilhaft.

In diesem Szenario fehlt das Konzept der *Atomarität*. Eine atomare Anweisung ist unteilbar – sie wird entweder vollständig oder gar nicht ausgeführt. Einfache SQL-Anweisungen sind von Natur aus atomar, aber zusammengesetzte SQL-Anweisungen sind es nicht. Sie können jedoch eine zusammengesetzte SQL-Anweisung atomar machen, indem Sie ihr das entsprechende Attribut zuweisen. In dem folgenden Beispiel ist die zusammengesetzte SQL-Anweisung sicher, weil sie über das Attribut der Atomarität verfügt:

```
void main {
   EXEC SQL
   BEGIN ATOMIC
      INSERT INTO Teilnehmer (TeilnehmerID, Vorname, Nachname)
         VALUES (:sid, :svorname, :snachname) ;
      INSERT INTO Kurse (KursID, Kurs, TeilnehmerID)
         VALUES (:kid, :kkurs, :sid) ;
      INSERT INTO Gebühren (TeilnehmerID, Kurs, Gebühr)
         VALUES (:sid, :kkurs, :kgebühr)
   END ;
/* SQLSTATE auf Fehler prüfen */
}
```

Durch das Schlüsselwort `ATOMIC` nach dem Schlüsselwort `BEGIN` können Sie dafür sorgen, dass entweder die komplette Anweisung ausgeführt wird oder dass – falls ein Fehler auftritt – die komplette Anweisung rückgängig gemacht wird, wodurch sich die Datenbank wieder in dem Zustand befindet, den sie vor der Ausführung der Anweisung hatte. Die Atomarität wird in Kapitel 15 im Zusammenhang mit Transaktionen ausführlich behandelt.

Sie können feststellen, ob eine Anweisung erfolgreich ausgeführt worden ist. Im Abschnitt *Zustand (Condition)* später in diesem Kapitel finden Sie mehr darüber.

Variablen

Komplette Computersprachen wie C oder Basic bieten schon immer *Variablen* an, unter SQL gibt es so etwas erst seit der Einführung von SQL/PSM. Eine Variable ist ein Symbol, das einen Wert eines jeden beliebigen Datentyps annehmen kann. Sie können eine Variable innerhalb einer zusammengesetzten Anweisung deklarieren, ihr einen Wert zuweisen und mit dieser Variablen arbeiten.

Wenn eine zusammengesetzte Anweisung beendet wird, werden alle Variablen zerstört, die in ihr deklariert worden sind. Deshalb sind Variablen, die in SQL in zusammengesetzten Anweisungen deklariert werden, auf diese Anweisungen beschränkt *(lokal)*.

Ein Beispiel für den Einsatz von Variablen sieht so aus:

```
BEGIN
   DECLARE Bonus NUMERIC ;
   SELECT Gehalt
   INTO Bonus
   FROM Mitarbeiter
   WHERE Funktion = 'Präsident' ;
END;
```

Cursor

Innerhalb einer zusammengesetzten Anweisung können Sie einen *Cursor* deklarieren. Sie können mit einem Cursor die Daten einer Tabelle zeilenweise verarbeiten (Einzelheiten zu diesem Thema finden Sie in Kapitel 19). In einer zusammengesetzten Anweisung können Sie einen Cursor deklarieren, verwenden und dann wieder vergessen, weil der Cursor zerstört wird, wenn die zusammengesetzte Anweisung ausgeführt worden ist. Das folgende Beispiel zeigt die Verwendung eines solchen Cursors:

```
BEGIN
   DECLARE ipokandidat CHARACTER(30) ;
   DECLARE cursor1 CURSOR FOR
           SELECT Firma
           FROM Biotech ;
   OPEN cursor1 ;
   FETCH cursor1 INTO ipokandidat ;
   CLOSE cursor1 ;
END;
```

Zustand (Condition)

Wenn man sagt, jemand sei in keinem besonders guten Zustand, heißt das in der Regel, dass mit dieser Person etwas nicht in Ordnung ist. Wenn es jemandem gesundheitlich ganz schlecht geht, sprechen wir vielleicht sogar davon, dass er in einem kritischen Zustand ist. Ähnlich sprechen Programmierer vom Zustand einer SQL-Anweisung. Die Ausführung einer SQL-Anweisung führt zu einem erfolgreichen Ergebnis, einem fragwürdigen Ergebnis oder einem zweifelsfrei fehlerhaften Ergebnis. Jedes dieser möglichen Ergebnisse stimmt mit einem *Zustand* überein.

Jedes Mal, wenn eine zusammengesetzte SQL-Anweisung ausgeführt wird, weist der Datenbankserver dem Statusparameter `SQLSTATE` einen Wert zu. `SQLSTATE` besteht aus fünf Zeichen. Der Wert von `SQLSTATE` zeigt an, ob die vorangegangene SQL-Anweisung erfolgreich ausgeführt worden ist. Falls die Anweisung nicht erfolgreich ausgeführt worden ist, liefert der Wert von `SQLSTATE` Informationen über den Fehler.

Die ersten beiden Zeichen von SQLSTATE (der sogenannte *Klassenwert*) zeigen die wichtigsten Informationen an: Wurde die vorangegangene SQL-Anweisung erfolgreich ausgeführt, wird ein Ergebnis zurückgegeben, das möglicherweise fraglich ist oder ausweist, dass ein Fehler aufgetreten ist. Tabelle 20.1 zeigt die vier möglichen Ergebnisse.

Klasse	Beschreibung	Einzelheiten
00	Erfolgreiche Beendigung	Die Anweisung wurde erfolgreich beendet.
01	Warnung	Während der Ausführung der Anweisung ist zwar etwas Ungewöhnliches passiert, das DBMS ist aber nicht in der Lage, mitzuteilen, ob es sich dabei um einen Fehler handelt oder nicht. Überprüfen Sie sorgfältig die letzte SQL-Anweisung, ob sie sauber ausgeführt worden ist.
02	Nicht gefunden	Es wurden als Ergebnis der Ausführung der Anweisung keine Daten zurückgegeben. Je nachdem, was Sie mit der Anweisung vorhatten, können das gute Nachrichten sein oder auch nicht. Sie können auf eine leere Ergebnistabelle hoffen.
andere Zeichen	Ausnahme	Die beiden Zeichen des Klassencodes und die drei Zeichen des Unterklassencodes bilden zusammen die fünf Zeichen von SQLSTAE. Sie liefern Ihnen einen Hinweis darauf, um welchen Fehler es sich handeln könnte.

Tabelle 20.1: Klassenwerte von SQLSTATE

Mit Zuständen umgehen

Sie können Ihr Programm veranlassen, nach der Ausführung einer SQL-Anweisung den Wert von SQLSTATE zu überprüfen. Was machen Sie jetzt mit dem Wissen, das Sie dadurch gewinnen?

- ✔ **Wenn Sie den Klassencode 00 ermitteln, sollten Sie wahrscheinlich gar nichts tun.** Sie machen einfach wie geplant weiter.
- ✔ **Bei einem Klassencode von 01 oder 02 können Sie besondere Aktionen ausführen – oder auch nicht.** Wenn Sie die Meldung »Warnung« oder »Nicht gefunden« erwartet haben, können Sie das Programm wahrscheinlich normal fortsetzen. Falls Sie jedoch keinen dieser Klassencodes erwartet haben, sollten Sie die Programmausführung zu einer Prozedur verzweigen lassen, die sich speziell mit außergewöhnlichen, aber nicht ganz unerwarteten Warnungen oder Nicht-gefunden-Meldungen beschäftigt.
- ✔ **Falls Sie einen anderen Klassencode erhalten, dann ist etwas nicht in Ordnung.** Sie sollten in diesem Fall zu einer Ausnahmefehlerbehandlung verzweigen. Die spezielle Prozedur, die Sie für diesen Zweck auswählen, hängt von dem Inhalt der drei Zeichen des Klassenuntercodes sowie von den beiden Zeichen des Klassencodes von SQLSTATE ab. Falls mehrere verschiedene Ausnahmen möglich sind, sollten Sie für jeden Ausnahmefehler eine separate Prozedur schreiben, weil verschiedene Fehler oft unterschiedliche Maßnahmen erfordern. Einige Fehler können korrigierbar sein oder umgangen werden. Andere können sehr schwer sein und einen Abbruch der Anwendung erforderlich machen.

Ausnahme-Handler deklarieren

Sie können in einer zusammengesetzten Anweisung einen Ausnahme-Handler deklarieren. Um einen Ausnahme-Handler zu erstellen, müssen Sie zunächst den Zustand (den Fehler) deklarieren, auf den der Handler reagieren soll. Bei diesem Zustand kann es sich um eine Ausnahme oder einfach um eine Bedingung handeln, die den Wert *wahr* angenommen hat. Tabelle 20.2 zeigt die möglichen Bedingungen und beschreibt kurz, wie die Ausnahmen verursacht werden können.

Ausnahme	Beschreibung
`SQLSTATE VALUE 'xxyyy'`	Spezieller `SQLSTATE`-Wert
`SQLEXCEPTION`	`SQLSTATE`-Klasse außer `00`, `01` oder `02`
`SQLWARNING`	`SQLSTATE`-Klasse `01`
`NOT FOUND`	`SQLSTATE`-Klasse `02`

Tabelle 20.2: Ausnahmen, die in einem Ausnahme-Handler festgelegt werden können

Das folgende Beispiel zeigt eine Ausnahmedeklaration:

```
BEGIN
   DECLARE EinschränkungsVerletzung CONDITION
      FOR SQLSTATE VALUE '23000' ;
END ;
```

Dieses Beispiel ist nicht realistisch, weil sowohl die SQL-Anweisung, die die Ausnahme verursachen kann, als auch der Handler, der zur Bearbeitung der Ausnahme aufgerufen wird, normalerweise ebenfalls in die `BEGIN...END`-Struktur eingeschlossen werden.

Handler-Aktionen und Handler-Folgen

Falls ein bestimmter Zustand eintritt, der einen Handler aufruft, wird die Aktion ausgeführt, die in dem Handler festgelegt ist. Bei dieser Aktion handelt es sich um eine SQL-Anweisung, die auch zusammengesetzt sein kann. Wenn die Handler-Aktion erfolgreich abgeschlossen wird, wird die Handler-Folge ausgeführt. Es gibt drei mögliche Handler-Folgen:

- ✔ **`CONTINUE`**. Die Ausführung wird unmittelbar nach der Anweisung fortgesetzt, die den Aufruf des Handlers verursacht hat.
- ✔ **`EXIT`**. Die Ausführung wird nach der zusammengesetzten Anweisung fortgesetzt, die den Handler enthält.
- ✔ **`UNDO`**. Die Arbeit der vorangegangenen Anweisungen in der zusammengesetzten Anweisung wird rückgängig gemacht. Dann wird die Ausführung nach der Anweisung fortgesetzt, die den Handler enthält.

Falls der Handler das Problem korrigieren kann, das seinen Aufruf verursacht hat, könnte die `CONTINUE`-Folge passend sein. Die `EXIT`-Folge kann angebracht sein, wenn der Handler das Problem nicht beseitigt, aber die Änderungen in der zusammengesetzten Anweisung nicht

rückgängig gemacht werden müssen. Die UNDO-Folge sollte verwendet werden, wenn Sie die Datenbank in den Zustand zurückversetzen möchten, den sie vor dem Beginn der zusammengesetzten Anweisung gehabt hat. Betrachten Sie das folgende Beispiel:

```
BEGIN ATOMIC
   DECLARE EinschränkungsVerletzung CONDITION
      FOR SQLSTATE VALUE '23000' ;
   DECLARE UNDO HANDLER
      FOR EinschränkungsVerletzung RESIGNAL ;
   INSERT INTO Teilnehmer (TeilnehmerID, Vorname, Nachname)
      VALUES (:sid, :svorname, :snachname) ;
   INSERT INTO Kurse (KursID, Kurs, TeilnehmerID)
      VALUES (:kid, :kkurs, :sid) ;
END ;
```

Wenn eine der beiden INSERT-Anweisungen eine Einschränkungsverletzung verursacht, indem sie beispielsweise einen Datensatz mit einem Primärschlüssel einfügt, der bereits in der Tabelle enthalten ist, dann nimmt SQLSTATE den Wert '23000' an und setzt damit EinschränkungsVerletzung auf den Wert *wahr*. Diese Aktion veranlasst den Handler, die Änderungen der Tabellen, die von einer der INSERT-Anweisungen durchgeführt worden sind, rückgängig zu machen (UNDO). Die Anweisung RESIGNAL gibt die Kontrolle an die Prozedur zurück, die die aktuell ausgeführte Prozedur aufgerufen hat.

Wenn beide INSERT-Anweisungen erfolgreich ausgeführt werden, wird die Ausführung mit der Anweisung fortgesetzt, die nach dem Schlüsselwort END steht.

Das Schlüsselwort ATOMIC ist erforderlich, wenn die Folge des Handlers UNDO ist. Für die Folgen CONTINUE oder EXIT ist dies nicht erforderlich.

Zustände, die nicht verarbeitet werden

Betrachten wir in dem vorangegangenen Beispiel die folgende Möglichkeit: Was passiert, wenn eine Ausnahme eingetreten ist, die nicht den SQLSTATE-Wert '23000' zurückgibt? Es liegt definitiv ein Fehler vor, aber Ihr Ausnahme-Handler kann nicht damit umgehen. Was passiert jetzt?

Weil die aktuelle Prozedur nicht weiß, was sie tun soll, wird ein RESIGNAL ausgeführt. Damit wird das Problem an die nächsthöhere Kontrollebene weitergereicht. Falls das Problem dort ebenfalls nicht bearbeitet wird, geht es immer weiter nach oben, bis es entweder bearbeitet wird oder in der Hauptroutine eine Fehlerbedingung auslöst.

Damit möchte ich betonen, dass Sie, falls eine SQL-Anweisung Ausnahmen verursachen kann, für alle möglichen Fälle Ausnahme-Handler schreiben sollten. Falls Sie dies nicht tun, wird es später schwieriger, die Quelle von Problemen zu identifizieren, die unvermeidlich auftreten werden.

Zuweisung

Mit SQL/PSM erhält SQL endlich eine Funktion, über die selbst die einfachsten prozeduralen Sprachen seit ihrer Erfindung verfügen: die Fähigkeit, einer Variablen einen Wert zuzuweisen. Im Wesentlichen hat eine Zuweisungsanweisung die folgende Form:

```
SET zielVar = ausdruck ;
```

Dabei steht `zielVar` für einen Variablennamen und `ausdruck` für einen Ausdruck. Einige Beispiele:

```
SET vvorname = 'Brandon' ;
SET vflaeche = 3.1416 * :radius * :radius ;
SET vhiggsmass = NULL ;
```

Anweisungen zur Ablaufsteuerung

Seit der ursprünglichen Formulierung des SQL-86-Standards war das Fehlen von Anweisungen zur Ablaufsteuerung einer der Hauptgründe dafür, dass SQL nicht prozedural eingesetzt wurde. Bis SQL/PSM zu einem Bestandteil des SQL-Standards wurde, war es nicht möglich, aus einer streng sequenziellen Ausführungsabfolge auszubrechen, ohne auf eine Host-Sprache wie C oder BASIC zurückzugreifen. SQL/PSM führt die traditionellen Strukturen zur Ablaufsteuerung ein, die in anderen Sprachen vorhanden sind, und ermöglicht es SQL-Programmen damit, Funktionen auszuführen, ohne zwischen verschiedenen Sprachen wechseln zu müssen.

IF ... THEN ... ELSE ... END IF

Die wohl grundlegendste Anweisung zur Ablaufsteuerung ist die Anweisung `IF ... THEN ... ELSE ... END IF`. Grob übersetzt bedeutet diese Anweisung: Falls (`IF`) eine Bedingung wahr ist, dann (`THEN`) führe die Anweisungen aus, die dem Schlüsselwort `THEN` folgen. Andernfalls führe die Anweisungen aus, die dem Schlüsselwort `ELSE` folgen. Ein Beispiel:

```
IF
   vvorname = 'Brandon'
THEN
   UPDATE Studenten
   SET Vorname = 'Brandon'
      WHERE StudentID = 314159 ;
ELSE
   DELETE FROM Studenten
      WHERE StudentID = 314159 ;
END IF
```

Wenn in diesem Beispiel die Variable vvorname den Wert 'Brandon' enthält, dann wird der Datensatz des Studenten 314159 aktualisiert, wobei das Feld Vorname den Wert 'Brandon' erhält. Wenn die Variable vvorname einen anderen Wert als 'Brandon' enthält, wird der Datensatz des Studenten 314159 aus der Tabelle Studenten gelöscht.

Die Anweisung IF ... THEN ... ELSE ...END IF eignet sich ausgezeichnet für Situationen, in denen Sie in Abhängigkeit von einer Bedingung eine von zwei möglichen Aktionen ausführen wollen. Bisweilen müssen Sie jedoch unter mehr als zwei möglichen Optionen wählen. In solchen Fällen ist eine CASE-Anweisung oft die bessere Wahl.

CASE ... END CASE

CASE-Anweisungen haben zwei verschiedene Formen: die einfache CASE-Anweisung und die suchende CASE-Anweisung. Beide Arten lassen es zu, dass Sie auf der Grundlage von Bedingungswerten unterschiedliche Ausführungspfade einschlagen.

Einfache CASE-Anweisung

Eine einfache CASE-Anweisung wertet eine einzelne Bedingung aus. Auf der Grundlage des Werts der Bedingung kann die Ausführung unterschiedlich verzweigen. Ein Beispiel:

```
CASE vhauptfach
   WHEN 'Informatik'
   THEN INSERT INTO Hacker (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   WHEN 'Sportmedizin'
   THEN INSERT INTO Cracks (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   WHEN 'Philosophie'
   THEN INSERT INTO Skeptiker (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   ELSE INSERT INTO Namenlos (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
END CASE
```

Die ELSE-Klausel verarbeitet alles, was nicht unter die ausdrücklich genannten Kategorien in den THEN-Klauseln fällt.

Die ELSE-Klausel ist optional. Wenn sie jedoch nicht angegeben und die Bedingung der CASE-Anweisung nicht in einer der THEN-Klauseln verarbeitet wird, löst SQL eine Ausnahme aus.

Suchende CASE-Anweisung

Eine suchende CASE-Anweisung ähnelt einer einfachen CASE-Anweisung, aber sie wertet nicht nur eine, sondern mehrere Bedingungen aus. Ein Beispiel:

```
CASE
   WHEN vhauptfach
      IN ('Informatik', 'Elektronik')
      THEN INSERT INTO Hacker (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   WHEN vclub
      IN ('Amateurfunk', 'Raumfahrt', 'Computer')
      THEN INSERT INTO Hacker (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   WHEN vhauptfach
      IN ('Sportmedizin', 'Fitness')
      THEN INSERT INTO Cracks (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
   ELSE
      INSERT INTO Dichter (StudentID, Vorname, Nachname)
            VALUES (:sid, :svorname, :snachname) ;
END CASE
```

Sie vermeiden eine Ausnahme, indem Sie alle Studenten, die keine Hacker sind, in die Tabelle `Dichter` einfügen. Dies ist sicherlich nicht in allen Fällen in Ordnung, aber zur Lösung des Problems könnten Sie einige weitere `WHEN`-Klauseln hinzufügen.

LOOP ... ENDLOOP

Mit der `LOOP`-Anweisung können Sie eine Folge von SQL-Anweisungen mehrfach ausführen. Nachdem die letzte SQL-Anweisung ausgeführt worden ist, die die `LOOP ... ENDLOOP`-Anweisung enthält, springt die Ablaufsteuerung zur ersten Anweisung in der Schleife zurück und durchläuft die eingeschlossenen Anweisungen ein weiteres Mal. Die Syntax sieht folgendermaßen aus:

```
SET vzähler = 0 ;
LOOP
   SET vzähler = vzähler + 1 ;
   INSERT INTO Asteroiden (AsteroidID)
      VALUES (vzähler) ;
END LOOP
```

Dieses Code-Fragment füllt Ihre Tabelle `Asteroiden` mit eindeutigen Bezeichnern. Sie können später anhand Ihrer Beobachtungen weitere Einzelheiten über die Asteroiden einfügen, wenn Sie sie mit Ihrem Teleskop entdeckt haben.

Das Codefragment des vorangegangenen Beispiels enthält ein kleines Problem: Es handelt sich dabei um eine Endlosschleife. Es ist keine Vorsorge dafür getroffen worden, die Schleife wieder zu verlassen, wodurch sie so lange Zeilen in die Tabelle `Asteroiden` einfügt, bis das DBMS den gesamten verfügbaren Speicherplatz mit Datensätzen gefüllt hat. Wenn Sie Glück haben, löst das DBMS dann eine Ausnahme aus. Wenn Sie Pech haben, stürzt das System einfach ab.

Damit der LOOP-Befehl praktisch eingesetzt werden kann, müssen Sie über eine Möglichkeit verfügen, die Schleife zu verlassen, bevor eine Ausnahme ausgelöst wird. Dies ist der Zweck der LEAVE-Anweisung.

LEAVE

Die Anweisung LEAVE funktioniert genauso, wie Sie es vielleicht erwarten. Wenn die Ausführung auf eine LEAVE-Anweisung stößt, die in eine benannte Anweisung eingebettet ist, macht sie mit der nächsten Anweisung weiter, die auf die benannte Anweisung folgt. Ein Beispiel:

```
AsteroidVorlader:
SET vzähler = 0 ;
LOOP
   SET vzähler = vzähler + 1 ;
   IF vzähler > 10000
      THEN
         LEAVE AsteroidVorlader ;
   END IF ;
   INSERT INTO Asteroiden (AsteroidID)
      VALUES (vzähler) ;
END LOOP AsteroidVorlader
```

Dieser Code fügt zehntausend sequenziell nummerierte Datensätze in die Tabelle ASTEROIDEN ein und verlässt dann die Schleife.

WHILE ... DO ... END WHILE

Die Anweisung WHILE stellt eine weitere Methode zur Verfügung, um eine Reihe von SQL-Anweisungen mehrfach auszuführen. Wenn eine benannte Bedingung wahr ist, wird die WHILE-Schleife ausgeführt. Wenn die Bedingung falsch wird, wird die Schleife abgebrochen. Ein Beispiel:

```
AsteroidVorlader2:
SET vzähler = 0 ;
WHILE
   vzähler < 10000 DO
      SET vzähler = vzähler + 1 ;
      INSERT INTO Asteroiden (AsteroidID)
         VALUES (vzähler) ;
END WHILE AsteroidVorlader2
```

Dieser Code führt genau dieselbe Aufgabe aus wie AsteroidVorlader aus dem vorangegangenen Abschnitt. Dies hier ist ein weiteres Beispiel für die schon bereits erwähnte Tatsache, dass SQL im Allgemeinen mehrere Wege kennt, um eine Aufgabe auszuführen. Sie können die Methode wählen, mit der Sie am besten zurechtkommen – vorausgesetzt, dass Ihre Implementierung beide Methoden unterstützt.

REPEAT ... UNTIL ... END REPEAT

Die REPEAT-Schleife verhält sich sehr ähnlich wie die WHILE-Schleife, außer dass die Bedingung nicht vor, sondern nach den eingebetteten Anweisungen geprüft wird:

```
AsteroidVorlader3:
SET vzähler = 0 ;
REPEAT
   SET vzähler = vzähler + 1 ;
   INSERT INTO Asteroiden (AsteroidID)
      VALUES (vzähler) ;
   UNTIL X = 10000
END REPEAT AsteroidVorlader3
```

Obwohl Sie dieselbe Operation auf drei verschiedenen Wegen durchführen können (mit LOOP, WHILE und REPEAT), ist eine dieser Strukturen in manchen Situationen deutlich besser geeignet als die beiden anderen. Deshalb sollten Sie alle drei Methoden kennen, damit Sie das jeweils beste Werkzeug für eine Aufgabe auswählen können.

FOR ... DO ... END FOR

Die FOR-Schleife von SQL deklariert und öffnet einen Cursor, ruft seine Zeilen ab, führt den Körper der FOR-Anweisungen einmal für jede Zeile aus und schließt dann den Cursor wieder. Diese Schleife ermöglicht es, Aufgaben komplett in SQL zu lösen, anstatt zu diesem Zweck auf eine Host-Sprache auszuweichen. Wenn Ihre Implementierung die SQL-Schleife FOR unterstützt, können Sie sie als einfache Alternative zum Arbeiten mit Cursorn einsetzen (siehe Kapitel 19). Ein Beispiel:

```
FOR vzähler AS Curs1 CURSOR FOR
   SELECT AsteroidID FROM Asteroiden
DO
   UPDATE Asteroiden SET Beschreibung = 'Eisenstein'
      WHERE CURRENT OF Curs1 ;
END FOR
```

In diesem Beispiel aktualisieren Sie jede Zeile der Tabelle Asteroiden, indem Sie 'Eisenstein' in das Feld Beschreibung einfügen. Dies ist eine wirklich schnelle Methode, um die Zusammensetzung von Asteroiden zu erkennen, aber es kann Probleme mit der Genauigkeit der Tabelle geben. Vielleicht sollten Sie die Spektralsignaturen der Asteroiden prüfen und dann ihren Typ einzeln eingeben.

ITERATE

Mit der Anweisung ITERATE können Sie den Ablauf der Ausführung in einer wiederholenden (iterierenden) SQL-Anweisung ändern. Die wiederholenden SQL-Anweisungen sind LOOP, WHILE, REPEAT und FOR. Wenn die Iterationsbedingung der wiederholenden SQL-Anweisungen wahr oder nicht angegeben ist, beginnt die nächste Iteration der Schleife sofort nach der

Ausführung der ITERATE-Anweisung. Wenn die Iterationsbedingung der wiederholenden SQL-Anweisung falsch oder unbekannt ist, wird die Iteration nach der Ausführung von ITERATE abgebrochen. Ein Beispiel:

```
AsteroidVorlader4:
SET vzähler = 0 ;
WHILE
   vzähler < 10000 DO
      SET vzähler = vzähler + 1 ;
      INSERT INTO Asteroiden (AsteroidID)
         VALUES (vzähler) ;
      ITERATE AsteroidVorlader4 ;
      SET vpreload = 'FERTIG' ;
END WHILE AsteroidVorlader4
```

Die Ausführung springt bei jedem Schleifendurchlauf unmittelbar nach der ITERATE-Anweisung zum Beginn der WHILE-Anweisung zurück, bis der vzähler den Wert 9999 hat. In dieser Iteration wird vzähler auf 10000 erhöht, INSERT wird ausgeführt, die ITERATE-Anweisung beendet die Iteration, vpreload wird auf 'FERTIG' gesetzt, und die Ausführung wird mit der nächsten Anweisung nach der Schleife fortgesetzt.

Gespeicherte Prozeduren

Gespeicherte Prozeduren (englisch *Stored Procedures*) liegen in der Datenbank auf dem Server und werden nicht auf dem Client ausgeführt – auf dem vor SQL/PSM alle Prozeduren zu finden waren. Sobald Sie eine gespeicherte Prozedur definiert haben, können Sie sie mit einer CALL-Anweisung aufrufen. Dadurch, dass die Prozedur auf dem Server und nicht auf dem Client liegt, wird die Netzlast verringert und damit das Leistungsverhalten verbessert. Der einzige Verkehr zwischen dem Client und dem Server besteht aus der CALL-Anweisung. Sie können eine Prozedur auf folgende Weise erstellen:

```
EXEC SQL
   CREATE PROCEDURE Ergebnisvergleich
      ( IN Ergebnis CHAR(3),
        OUT Gewinner CHAR(5) )
   BEGIN ATOMIC
      CASE Ergebnis
         WHEN '1–0' THEN
            SET Gewinner = 'weiß' ;
         WHEN '0–1' THEN
            SET Gewinner = 'schwarz' ;
         ELSE
            SET Gewinner = 'Remis' ;
      END CASE
   END ;
```

Wenn Sie eine gespeicherte Prozedur wie diese erstellt haben, können Sie sie mit einer CALL-Anweisung wie der folgenden aufrufen:

```
CALL Ergebnisvergleich
   ('Kasparov', 'Karpov', '1–0', Gewinner) ;
```

Die ersten drei Argumente sind Eingabeparameter, die an die Prozedur Ergebnisvergleich übergeben werden. Das vierte Argument ist der Ausgabeparameter, den Ergebnisvergleich verwendet, um das Ergebnis an die aufrufende Routine zurückzugeben. In diesem Fall gibt sie 'weiß' zurück.

In SQL:2011 wurden einige Erweiterungen der gespeicherten Prozeduren eingeführt. Erstens wurden benannte Argumente eingeführt. Das folgende Beispiel entspricht dem vorhergehenden Aufruf mit benannten Argumenten:

```
CALL Ergebnisvergleich (Gewinner => :Outcome,Ergebnis =>'1–0');
```

Weil die Argumente benannt sind, können sie in beliebiger Reihenfolge angegeben werden, ohne dass sie verwechselt werden können.

Zweitens wurden Standardeingabeargumente eingeführt. Sie können für einen Inputparameter ein Standardargument spezifizieren. Wenn Sie danach in der CALL-Anweisung keinen Inputwert angeben, wird der Standardwert angenommen. (Natürlich sollten Sie dies nur tun, wenn Sie tatsächlich den Standardwert an die Prozedur übergeben wollen.)

Ein Anwendungsbeispiel:

```
EXEC SQL
   CREATE PROCEDURE Ergebnisvergleich
( INscore CHAR (3)DEFAULT '1–0',
        OUT result CHAR (10) )
   BEGIN ATOMIC
      CASE score
         WHEN '1–0' THEN
            SET Gewinner = 'weiß' ;
         WHEN '0–1' THEN
            SET Gewinner = 'schwarz' ;
         ELSE
            SET Gewinner = 'Remis' ;
      END CASE
END ;
```

Jetzt können Sie diese Prozedur mit dem Standardwert wie folgt aufrufen:

```
CALL Ergebnisvergleich (:Outcome) ;
```

Natürlich sollten Sie dies nur tun, wenn Sie tatsächlich den Standardwert an die Prozedur übergeben wollen.

Gespeicherte Funktionen

Eine gespeicherte Funktion (englisch *Stored Function*) ähnelt in vieler Hinsicht einer gespeicherten Prozedur. Zusammen werden beide als *gespeicherte Routinen* bezeichnet. Sie unterscheiden sich durch verschiedene Aspekte, unter anderem darin, wie sie aufgerufen werden. Eine gespeicherte Prozedur wird mit einer `CALL`-Anweisung aufgerufen, während eine gespeicherte Funktion mit einem Funktionsaufruf aktiviert wird, der ein Argument einer SQL-Anweisung ersetzen kann. Das folgende Beispiel zeigt die Definition einer Funktion und ein Beispiel für einen Aufruf dieser Funktion:

```
CREATE FUNCTION Kaufhistorie (KundeID)
   RETURNS CHAR VARYING (200)

   BEGIN
      DECLARE Kauf CHAR VARYING (200)
         DEFAULT " ;
      FOR x AS SELECT *
               FROM Transaktionen t
               WHERE t.KundenID = KundeID
      DO
         IF a <> "
            THEN SET Kauf = Kauf || ', ' ;
         END IF ;
         SET Kauf = Kauf || t.Beschreibung ;
      END FOR
      RETURN Kauf ;
   END ;
```

Diese Funktionsdefinition erstellt eine durch Kommata getrennte Liste der Käufe eines Kunden, der in der Tabelle `Transaktionen` eine bestimmte Kundennummer besitzt. Die folgende `UPDATE`-Anweisung enthält einen Funktionsaufruf von `Kaufhistorie`, der die neueste Kaufhistorie des Kunden mit der Kundennummer 314259 in den dazugehörigen Datensatz der `Kunden`-Tabelle einfügt:

```
SET KundeID = 314259 ;
UPDATE Kunden
   SET historie = Kaufhistorie (KundeID)
   WHERE KundeID = 314259 ;
```

Rechte

Die verschiedenen Rechte, die Sie Benutzern einräumen können, werden in Kapitel 14 beschrieben. Der Datenbankbesitzer kann anderen Benutzern die folgenden Rechte einräumen:

- ✔ Das Recht, Zeilen aus einer Tabelle zu löschen (`DELETE`)
- ✔ Das Recht, Zeilen in eine Tabelle einzufügen (`INSERT`)

- ✔ Das Recht, Zeilen einer Tabelle zu aktualisieren (UPDATE)
- ✔ Das Recht, eine Tabelle zu erstellen, die auf eine andere Tabelle verweist (REFERENCES)
- ✔ Das Recht, eine Domäne zu benutzen (USAGE)

SQL/PSM fügt ein weiteres Recht hinzu, das einem Benutzer eingeräumt werden kann: das Recht, etwas auszuführen (EXECUTE). Zwei Beispiele:

```
GRANT EXECUTE ON Ergebnisvergleich TO Turnierleiter ;
GRANT EXECUTE ON Kaufhistorie TO Verkaufsleiter ;
```

Die erste Anweisung räumt dem Turnierleiter eines Schachwettkampfs das Recht ein, die Prozedur `Ergebnisvergleich` auszuführen. Aufgrund der zweiten Anweisung hat der Verkaufsleiter das Recht, die Funktion `Kaufhistorie` auszuführen. Benutzer, die nicht über das EXECUTE-Recht für eine Routine verfügen, können diese nicht verwenden.

Gespeicherte Module

Ein gespeichertes Modul (englisch *Stored Module*) kann mehrere Routinen (Prozeduren und/oder Funktionen) enthalten, die von SQL aus aufgerufen werden können. Jeder Benutzer, der über das EXECUTE-Recht für ein Modul verfügt, kann auf alle Routinen des Moduls zugreifen. Rechte für Routinen innerhalb eines Moduls können nicht individuell vergeben werden. Der folgende Code ist ein Beispiel für ein gespeichertes Modul:

```
CREATE MODULE mod1

   PROCEDURE Ergebnisvergleich
      ( IN Ergebnis CHAR (3),
        OUT Gewinner CHAR (5) )
   BEGIN ATOMIC
      CASE Ergebnis
         WHEN '1–0' THEN
            SET Gewinner = 'weiß' ;
         WHEN '0–1' THEN
            SET Gewinner = 'schwarz' ;
         ELSE
            SET Gewinner = 'Remis' ;
      END CASE
   END ;
```

```
   FUNCTION Kaufhistorie (CustID)
   RETURNS CHAR VARYING (200)
   BEGIN
      DECLARE Kauf CHAR VARYING (200)
         DEFAULT " ;
      FOR x AS SELECT *
               FROM Transaktionen t
               WHERE t.KundeID = CustID
      DO
         IF a <> "
            THEN SET Kauf = Kauf || ', ' ;
         END IF ;
         SET Kauf = Kauf || t.beschreibung ;
      END FOR
      RETURN Kauf ;
   END ;

END MODULE ;
```

Die beiden Routinen in diesem Modul (eine Prozedur und eine Funktion) haben nicht viel gemeinsam – das ist auch gar nicht notwendig. Sie fassen in einem einzigen Modul verwandte Routinen zusammen oder Sie packen alle Routinen, die Sie wahrscheinlich benutzen werden, in ein einziges Modul, und zwar unabhängig davon, ob sie etwas gemeinsam haben oder nicht.

Fehlerbehandlung

21

In diesem Kapitel

- Fehlersituationen melden
- Zum Fehlerbehandlungscode verzweigen
- Die Art eines Fehlers genau bestimmen
- Feststellen, welches DBMS einen Fehler erzeugt hat

Wäre es nicht großartig, wenn jede Anwendung, die Sie geschrieben haben, immer perfekt funktionierte? Ja – und wäre es nicht genauso toll, wenn Sie den Jackpot im Lotto gewinnen würden? Leider sind beide Ereignisse etwa gleich unwahrscheinlich. Fehler der einen oder anderen Art sind unvermeidbar, und deshalb ist es hilfreich, ihre Ursachen zu kennen. SQL verfügt über einen Mechanismus, um Ihnen Fehlerinformationen zu liefern: den *Statusparameter* (oder die *Host-Variable*) SQLSTATE. Je nach Inhalt dieser Variablen können Sie verschiedene Wege einschlagen, um den Fehler zu beheben.

So gibt Ihnen beispielsweise die Direktive WHENEVER die Möglichkeit, immer dann eine vorher festgelegte Aktion auszuführen, wenn eine bestimmte Bedingung eintritt (wenn zum Beispiel SQLSTATE nicht den Wert null hat). Außerdem können Sie im Diagnosebereich detaillierte Statusinformationen über die SQL-Anweisung finden, die Sie gerade ausgeführt haben. In diesem Kapitel erkläre ich diese hilfreichen Möglichkeiten zur Fehlerbehandlung und beschreibe, wie Sie sie benutzen können.

SQLSTATE

SQLSTATE macht Angaben über eine Vielzahl von anormalen Zuständen einer Datenbank und ihrer Module. SQLSTATE besteht aus einer fünfstelligen Zeichenfolge, die nur die Großbuchstaben A bis Z sowie die Ziffern 0 bis 9 enthalten darf. Die fünfstellige Zeichenfolge ist in zwei Komponenten aufgeteilt: einen zweistelligen Klassencode und einen dreistelligen Unterklassencode. Abbildung 21.1 illustriert das Layout von SQLSTATE.

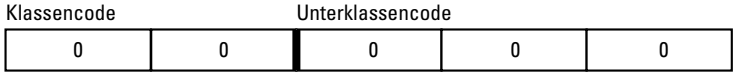


Abbildung 21.1: Layout des Statusparameters SQLSTATE

Der SQL-Standard legt alle Klassencodes fest, die mit den Buchstaben A bis H oder den Ziffern 0 bis 4 beginnen. Deshalb ist die Bedeutung dieser Klassencodes in allen Implementierungen gleich. Klassencodes, die mit den Buchstaben I bis Z oder den Ziffern 5 bis 9 beginnen, können dagegen von den Unternehmen definiert werden, die SQL-Implementierungen (Daten-

bankverwaltungssysteme) herstellen. Diese Codes werden den Unternehmen überlassen, weil die SQL-Spezifikation nicht alle Situationen voraussehen kann, die bei einer Implementierung auftreten können. Die Anbieter von Datenbankverwaltungssystemen sollten die spezifischen Klassencodes so wenig wie möglich benutzen, um Portierungsprobleme von einem System zu einem anderen zu vermeiden. Idealerweise sollten die Firmen hauptsächlich die Standardcodes verwenden und nur unter sehr ungewöhnlichen Umständen auf die eigenen, firmenspezifischen Codes zurückgreifen.

Ich behandle `SQLSTATE` ausführlich in Kapitel 21, weshalb ich mich hier auf eine kurze Wiederholung beschränken kann. Der Klassencode `00` zeigt die erfolgreiche Ausführung einer Anweisung an. Der Klassencode `01` bedeutet, dass die Anweisung zwar erfolgreich ausgeführt worden ist, aber eine Warnung erzeugt hat. Der Klassencode `02` weist darauf hin, dass keine Daten zurückgegeben worden sind. Jeder andere `SQLSTATE`-Klassencode als `00`, `01` oder `02` gibt an, dass die Anweisung nicht erfolgreich ausgeführt worden ist.

Weil `SQLSTATE` nach jeder SQL-Operation aktualisiert wird, können Sie seinen Wert jedes Mal überprüfen, wenn eine Anweisung ausgeführt worden ist. Wenn `SQLSTATE` den Wert `00000` (erfolgreiche Ausführung) enthält, fahren Sie mit der nächsten Operation fort. Wenn er einen anderen Wert enthält, sollten Sie aus der normalen Ablaufroutine Ihres Codes aussteigen und sich um die Angelegenheit (etwa in Form einer Fehlerroutine) kümmern. Der Klassencode und der Unterklassencode in `SQLSTATE` legen fest, welche Aktion Sie ausführen sollten.

Um `SQLSTATE` in einem SQL-Modul (siehe Kapitel 16) zu benutzen, setzen Sie in der Definition Ihrer Prozedur einen Verweis wie in dem folgenden Beispiel ein:

```
PROCEDURE NahrungsmittelEinfügen
   (SQLSTATE, :nahrungsmittel CHAR (20), :kalorien SMALLINT,
      :eiweiß DECIMAL (5,1), :fett DECIMAL (5,1),
      :kohlenhydrate DECIMAL (5,1))
INSERT INTO NahrungsmittelTabelle
   (Nahrungsmittel, Kalorien, Eiweiß, Fett, Kohlenhydrate)
   VALUES
   (:nahrungsmittel, :kalorien, :eiweiß, :fett,
    :kohlenhydrate) ;
```

An der passenden Stelle Ihres prozeduralen Programms stellen Sie Werte für die Parameter bereit (indem Sie sie gegebenenfalls vom Benutzer abfragen) und rufen dann die Prozedur auf. Die Syntax ist sprachabhängig, sieht aber ungefähr folgendermaßen aus:

```
nahrungsmittel = "Okra, gekocht" ;
kalorien = 29 ;
eiweiß = 2.0 ;
fett = 0.3 ;
kohlenhydrate = 6.0 ;
NUTRIENT(state, nahrungsmittel, kalorien,
   eiweiß, fett, kohlenhydrate);
```

Der Status von SQLSTATE wird in der Variablen state zurückgegeben. Ihr Programm kann diese Variable abfragen und eine Aktion ausführen, die dem Wert der Variablen entspricht.

Die Klausel WHENEVER

Welchen Sinn macht es zu wissen, dass eine SQL-Operation nicht erfolgreich ausgeführt worden ist, wenn Sie nichts daran ändern können? Wenn ein Fehler eintritt, soll Ihre Anwendung nicht so weiterlaufen, als sei alles in Ordnung. Sie müssen in der Lage sein, den Fehler zu erkennen und eine Gegenmaßnahme zu ergreifen. Wenn Sie den Fehler nicht beseitigen können, sollten Sie wenigstens den Benutzer über das Problem informieren und die Anwendung sauber beenden. Die Direktive WHENEVER dient in SQL der Ausnahmebehandlung.

Die Direktive WHENEVER ist eigentlich eine Deklaration und findet sich deshalb im Deklarationsabschnitt Ihrer SQL-Anwendung vor dem ausführbaren SQL-Code wieder. Die Syntax lautet:

```
WHENEVER Bedingung Aktion ;
```

Bedingung kann entweder SQLERROR oder NOT FOUND (nicht gefunden) lauten. Die Aktion kann entweder CONTINUE (fortfahren) oder GOTO Adresse lauten. SQLERROR ist *wahr*, wenn SQLSTATE einen anderen Klassencode als 00, 01 oder 02 enthält. NOT FOUND ist *wahr*, wenn SQLSTATE den Wert 02000 enthält.

Falls Aktion den Wert CONTINUE enthält, passiert nichts Besonderes, und die Ausführung wird normal fortgesetzt. Falls Aktion den Wert GOTO Adresse (oder GO TO Adresse) enthält, verzweigt die Ausführung im Programm zur angegebenen Adresse. An der Verzweigungsadresse können Sie SQLSTATE mit bedingten Befehlen untersuchen und in Abhängigkeit von den gefundenen Werten unterschiedliche Aktionen ausführen. Zwei Beispiele dafür:

```
WHENEVER SQLERROR GO TO FehlerAbfang ;
```

oder

```
WHENEVER NOT FOUND CONTINUE ;
```

Die Option GO TO ist einfach ein Makro: Die *Implementierung* (das heißt der eingebaute Präcompiler der Sprache) fügt hinter der Anweisung EXEC SQL folgenden Test ein:

```
IF SQLSTATE <> '00000'
   AND SQLSTATE <> '00001'
   AND SQLSTATE <> '00002'
THEN GOTO FehlerAbfang ;
```

Die Option CONTINUE ist im Wesentlichen eine NO-OP-Anweisung, die sagt: »Ignoriere diese Anweisung.«

Diagnosebereiche

Auch wenn SQLSTATE Sie darüber informiert, warum eine bestimmte Anweisung nicht ausgeführt werden konnte, ist diese Information ziemlich knapp gehalten. Deshalb stellt SQL in Diagnosebereichen zusätzliche Statusinformationen zur Verfügung.

Viele dieser Diagnosebereiche werden im LIFO-Verfahren (Last-In-First-Out) verwaltet. Dies bedeutet, dass sich Informationen über den zuletzt aufgetretenen Fehler zuoberst im Diagnosebereich befinden, während Informationen über ältere Fehler weiter unten in der Liste zu suchen sind. Diese zusätzlichen Statusinformationen sind besonders dann hilfreich, wenn die Ausführung einer einzelnen SQL-Anweisung mehrere Warnungen hervorruft, denen eine Fehlermeldung folgt. Während SQLSTATE nur das Auftreten eines einzigen Fehlers meldet, hat der Diagnosebereich die Kapazität, um mehrere (hoffentlich alle) Fehler zu speichern.

Der Diagnosebereich ist eine Datenstruktur, die vom DBMS verwaltet wird und aus zwei Komponenten besteht:

- ✔ **Kopf:** Der Kopf enthält allgemeine Informationen über die letzte ausgeführte SQL-Anweisung.
- ✔ **Detailbereich:** Der Detailbereich enthält Informationen über jeden Code (entweder Fehler, Warnung oder Erfolg), den die Anweisung erzeugt hat.

Der Kopf des Diagnosebereichs

In der Anweisung SET TRANSACTION (siehe Kapitel 15) können Sie die Klausel DIAGNOSTICS SIZE verwenden. Bei der Größe, die Sie mit DIAGNOSTIC SIZE angeben, handelt es sich um die Anzahl der Detailbereiche, die für Statusinformationen reserviert werden. Wenn Sie in Ihrer SET TRANSACTION-Anweisung keine DIAGNOSTICS SIZE-Klausel aufnehmen, weist Ihr DBMS automatisch eine Standardanzahl von Detailbereichen zu.

Der Kopfbereich enthält diverse Elemente, die in Tabelle 21.1 zusammengefasst sind.

Eintrag	Datentyp
NUMBER	Genauer numerischer Wert ohne Nachkommastellen
ROW_COUNT	Genauer numerischer Wert ohne Nachkommastellen
COMMAND_FUNCTION	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
COMMAND_FUNCTION_CODE	Genauer numerischer Wert ohne Nachkommastellen
DYNAMIC_FUNCTION	VARCHAR (implementierungsabhängig definierte maximale Länge)
DYNAMIC_FUNCTION_CODE	Genauer numerischer Wert ohne Nachkommastellen
MORE	Genauer numerischer Wert ohne Nachkommastellen
TRANSACTIONS_COMMITTED	Genauer numerischer Wert ohne Nachkommastellen
TRANSACTIONS_ROLLED_BACK	Genauer numerischer Wert ohne Nachkommastellen
TRANSACTION_ACTIVE	Genauer numerischer Wert ohne Nachkommastellen

Tabelle 21.1: Der Kopf des Diagnosebereichs

Im Einzelnen lassen sich die Elemente der Tabelle wie folgt beschreiben:

- ✔ Das Feld NUMBER gibt die Anzahl der Detailbereiche an, die mit diagnostischen Informationen über die aktuelle Ausnahme gefüllt sind.
- ✔ Das Feld ROW_COUNT enthält die Anzahl an Zeilen, die von der vorherigen SQL-Anweisung betroffen sind, sofern es sich dabei um ein INSERT, UPDATE oder DELETE gehandelt hat.
- ✔ Das Feld COMMAND_FUNCTION beschreibt die SQL-Anweisung, die gerade ausgeführt worden ist.
- ✔ Das Feld COMMAND_FUNCTION_CODE gibt die Codenummer für die SQL-Anweisung an, die gerade ausgeführt worden ist. Jede Kommandofunktion besitzt einen eigenen numerischen Code.
- ✔ Das Feld DYNAMIC_FUNCTION enthält eine dynamische SQL-Anweisung.
- ✔ Das Feld DYNAMIC_FUNCTION_CODE enthält einen numerischen Code, der mit der dynamischen SQL-Anweisung korrespondiert.
- ✔ Das Feld MORE enthält entweder 'Y' oder 'N'. 'Y' zeigt an, dass mehr Statuseinträge vorhanden sind, als der Detailbereich aufnehmen kann. 'N' zeigt an, dass sich alle Statusdatensätze, die erzeugt worden sind, im Detailbereich befinden. Abhängig von Ihrer Implementierung können Sie möglicherweise die Anzahl der Datensätze, die der Detailbereich aufnehmen kann, mit der Anweisung SET TRANSACTION erhöhen.
- ✔ Das Feld TRANSACTIONS_COMMITTED enthält die Anzahl an Transaktionen, die erfolgreich abgeschlossen worden sind.
- ✔ Das Feld TRANSACTIONS_ROLLED_BACK enthält die Anzahl an Transaktionen, die rückgängig gemacht worden sind.
- ✔ Das Feld TRANSACTION_ACTIVE enthält eine '1', wenn gerade eine Transaktion aktiv ist, andernfalls eine '0'. Eine Transaktion wird als aktiv bezeichnet, wenn ein Cursor geöffnet ist oder das DBMS auf einen zurückgestellten Parameter wartet.

Der Detailbereich des Diagnosebereichs

Die Detailbereiche enthalten Informationen über die einzelnen Fehler, Warnungen oder den Erfolgsstatus. Jeder Detailbereich enthält 28 Einträge (siehe Tabelle 21.2).

CONDITION_NUMBER enthält die laufende Nummer des Detailbereichs. Wenn eine Abfrage fünf Statuseinträge erzeugt, die in fünf Detailbereiche eingetragen werden, ist CONDITION_NUMBER des fünften Detailbereichs 5. Wenn Sie auf einen bestimmten Detailbereich zugreifen wollen, um ihn genau zu untersuchen, setzen Sie die Anweisung GET DIAGNOSTICS (die weiter hinten in diesem Kapitel im Abschnitt *Die Informationen auswerten, die von SQLSTATE zurückgegeben werden* beschrieben wird) mit der entsprechenden CONDITION_NUMBER ein. RETURNED_SQLSTATE enthält den Wert von SQLSTATE, der dazu geführt hat, dass dieser Detailbereich gefüllt worden ist.

Eintrag	Datentyp
CONDITION_NUMBER	Genauer numerischer Wert ohne Nachkommastellen
RETURNED_SQLSTATE	CHAR (6)
MESSAGE_TEXT	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
MESSAGE_LENGTH	Genauer numerischer Wert ohne Nachkommastellen
MESSAGE_OCTET_ LENGTH	Genauer numerischer Wert ohne Nachkommastellen
CLASS_ORIGIN	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
SUBCLASS_ORIGIN	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CONNECTION_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
SERVER_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CONSTRAINT_ CATALOG	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CONSTRAINT_SCHEMA	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CONSTRAINT_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CATALOG_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
SCHEMA_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
TABLE_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
COLUMN_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CURSOR_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
CONDITION_ IDENTIFIER	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
PARAMETER_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
PARAMETER_ ORDINAL_POSITION	Genauer numerischer Wert ohne Nachkommastellen
PARAMETER_MODE	Genauer numerischer Wert ohne Nachkommastellen
ROUTINE_CATALOG	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
ROUTINE_SCHEMA	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
ROUTINE_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
SPECIFIC_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
TRIGGER_CATALOG	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
TRIGGER_SCHEMA	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)
TRIGGER_NAME	VARCHAR (>=128) (implementierungsabhängig definierte maximale Länge)

Tabelle 21.2: Der Diagnose-Detailbereich

CLASS_ORIGIN teilt Ihnen die Quelle des Klassencodewerts mit, der in SQLSTATE zurückgegeben wurde. Wenn der Wert durch den SQL-Standard definiert ist, enthält CLASS_ORIGIN den Eintrag 'ISO 9075'. Falls Ihre DBMS-Implementierung den Wert definiert, enthält

CLASS_ORIGIN eine Zeichenfolge, die die Herkunft Ihres DBMS angibt. SUBCLASS_ORIGIN teilt Ihnen die Quelle des Unterklassen-Codewerts mit, der in SQLSTATE zurückgegeben wird.

CLASS_ORIGIN ist wichtig. Wenn SQLSTATE beispielsweise einen Wert '22012' enthält, können Sie daran erkennen, dass er zu den standardmäßigen SQLSTATE-Werten gehört, die in allen SQL-Implementierungen dasselbe bedeuten. Wenn SQLSTATE jedoch einen Wert wie beispielsweise '22500' enthält, teilen Ihnen die ersten beiden Ziffern mit, dass es sich um einen Wert des Standards handelt, der einen Datenfehler anzeigt. Die letzten drei Zeichen gehören dagegen zum implementierungsspezifischen Bereich. Wenn SQLSTATE dagegen den Wert '900001' enthält, liegt der Wert vollständig im implementierungsspezifischen Bereich. SQLSTATE-Werte im implementierungsspezifischen Bereich können bei unterschiedlichen Implementierungen verschiedene Bedeutungen haben, selbst wenn der Fehlercode bei allen Implementierungen der gleiche ist.

Wie finden Sie jetzt die genaue Bedeutung von '22500' oder '900001' heraus? Sie müssen in der Dokumentation des Anbieters Ihrer SQL-Implementierung nachschlagen. Welcher Anbieter? Wenn Sie mit CONNECT arbeiten, sind Sie möglicherweise mit verschiedenen Produkten verbunden. Um festzustellen, welches Produkt die Ausnahmebedingung hervorgerufen hat, schauen Sie sich CLASS_ORIGIN und SUBCLASS_ORIGIN an. Sie finden dort Werte, die jede Implementierung eindeutig identifizieren. Sie können CLASS_ORIGIN und SUBCLASS_ORIGIN abfragen, ob sie Anbieter angeben, für die Sie über SQLSTATE-Fehlerlisten verfügen. Die Werte in CLASS_ORIGIN und SUBCLASS_ORIGIN sind zwar implementierungsspezifisch, sollten aber selbsterklärende Firmennamen sein.

Falls es sich bei dem gemeldeten Fehler um einen Verstoß gegen eine Einschränkung (englisch *Constraint*) handelt, geben die Einträge CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA und CONSTRAINT_NAME die Einschränkung an, gegen die verstoßen wurde.

Beispiel für Verstöße gegen Einschränkungen

Die Informationen über Verstöße gegen Einschränkungen sind wahrscheinlich die wichtigsten Informationen, die GET DIAGNOSTICS liefert. Betrachten Sie die folgende Tabelle Mitarbeiter:

```
CREATE TABLE Mitarbeiter
(ID CHAR(5) CONSTRAINT MitPK PRIMARY KEY,
Gehalt DEC(8,2) CONSTRAINT MitGehalt CHECK Gehalt > 0,
Abtlg CHAR(5) CONSTRAINT MitAbt
   REFERENCES Abteilung) ;
```

Und zusätzlich die Tabelle Abteilung:

```
CREATE TABLE Abteilung
   (AbtNr CHAR(5),
   Budget DEC(12,2) CONSTRAINT AbtBudget
   CHECK(Budget >= SELECT SUM(Gehalt) FROM Mitarbeiter
         WHERE Mitarbeiter.Abtlg=Abteilung.AbtNr),
...);
```

Und schauen Sie sich jetzt noch folgenden INSERT an:

```
INSERT INTO Mitarbeiter VALUES(:id_var, :geh_var, :abt_var);
```

Angenommen, Sie erhalten einen SQLSTATE von '23000'. Sie schlagen in Ihrer SQL-Dokumentation nach und finden den Hinweis, dass die Anweisung einen Verstoß gegen die Integritätseinschränkung *(integrity constraint violation)* bewirkt hat. Was nun? Der Wert, den SQLSTATE liefert, kann Folgendes bedeuten:

- ✔ **Der Wert in id_var ist ein Duplikat einer bestehenden ID.** Sie haben gegen die Einschränkung PRIMARY KEY verstoßen.
- ✔ **Der Wert in geh_var ist negativ.** Sie haben gegen die Einschränkung CHECK von Gehalt verstoßen.
- ✔ **Der Wert in abt_var ist für keine Zeile von Abteilung ein gültiger Schlüsselwert.** Sie haben gegen die Einschränkung REFERENCES von Abteilung verstoßen.
- ✔ **Der Wert in geh_var ist so groß, dass die Summe aller Angestelltengehälter größer als das Budget der Abteilung ist.** Sie haben gegen die Einschränkung CHECK der Spalte Budget von Abteilung verstoßen. (Denken Sie daran, dass beim Ändern einer Datenbank nicht nur die Einschränkungen in der unmittelbar beteiligten Tabelle, sondern alle betroffenen Einschränkungen geprüft werden.)

Unter normalen Umständen müssten Sie einen großen Testaufwand betreiben, um den Fehler in der INSERT-Anweisung zu finden. Leichter geht es mit GET DIAGNOSTICS:

```
DECLARE ConstNameVar CHAR(18) ;
GET DIAGNOSTICS EXCEPTION 1
   ConstNameVar = CONSTRAINT_NAME ;
```

Angenommen, SQLSTATE enthält den Wert '23000', dann setzt dieses GET DIAGNOSTICS in ConstNameVar den Wert 'MitPK', 'MitGehalt', 'MitAbt' oder 'AbtBudget'. In der Praxis sollten Sie außerdem CONSTRAINT_SCHEMA und CONSTRAINT_CATALOG abfragen, um die Einschränkung in CONSTRAINT_NAME eindeutig zu identifizieren.

Einer Tabelle Einschränkungen hinzufügen

Dieser Einsatz von GET DIAGNOSTICS – das Herausfinden, gegen welche von mehreren Einschränkungen verstoßen wurde – ist besonders in dem Fall wichtig, in dem mit ALTER TABLE Einschränkungen hinzugefügt werden, die beim Schreiben des Programms noch nicht bestanden haben.

```
ALTER TABLE Mitarbeiter
   ADD CONSTRAINT GehaltLimit CHECK(Gehalt < 200000) ;
```

Wenn Sie eine Zeile in Mitarbeiter einfügen oder die Spalte Gehalt von Mitarbeiter ändern, wird ein SQLSTATE von '23000' gemeldet, wenn Gehalt größer als 200.000 Euro ist. Sie können Ihre INSERT-Anweisung so programmieren, dass Sie eine hilfreiche Meldung wie Invalid INSERT: Violated constraint GehaltLimit erhalten, wenn es zu einem

SQLSTATE von '23000' kommt und Sie den Namen der verantwortlichen Einschränkung nicht kennen, den GET DIAGNOSTICS zurückgibt. (Meldungen von SQL-Servern werden – implementierungsabhängig – in Englisch oder in Deutsch angezeigt, wundern Sie sich also nicht über den Text.)

Die Informationen auswerten, die von SQLSTATE zurückgegeben werden

CONNECTION_NAME und ENVIRONMENT_NAME identifizieren die Verbindung und die Umgebung, mit denen Sie verbunden sind, wenn die SQL-Anweisung ausgeführt wird.

Falls sich die Meldung auf eine Tabellenoperation bezieht, identifizieren CATALOG_NAME, SCHEMA_NAME und TABLE_NAME die Tabelle. COLUMN_NAME gibt die Spalte innerhalb der Tabelle an, die die Meldung ausgelöst hat. Wenn die Situation mit einem Cursor zu tun hat, enthält CURSOR_NAME dessen Namen.

Manchmal erzeugt das DBMS einen Text in natürlicher Sprache, um eine Fehlersituation zu erklären. Für diese Art von Informationen wird MESSAGE_TEXT verwendet. Die Inhalte dieses Eintrags hängen von der Implementierung ab, da sie der SQL-Standard nicht ausdrücklich festlegt. Falls MESSAGE_TEXT Informationen enthält, finden Sie die Anzahl der Zeichen dieser Informationen in MESSAGE_LENGTH und die Länge der Informationen in Oktetten in MESSAGE_OCTET_LENGTH. Wenn die Meldung aus normalen ASCII-Zeichen besteht, ist MESSAGE_LENGTH gleich MESSAGE_OCTET_LENGTH. Ist dagegen die Meldung in Kanji oder einer anderen Schrift abgefasst, deren Zeichen mehr als ein Oktett zur Darstellung benötigen, unterscheiden sich MESSAGE_LENGTH und MESSAGE_OCTET_LENGTH. Um diagnostische Informationen aus dem Kopf des Diagnosebereichs abzufragen, gehen Sie so vor:

```
GET DIAGNOSTICS status1 = item1 [, status2 = item2]...
```

statusN ist eine Host-Variable oder ein Parameter; itemN kann eines der Schlüsselwörter NUMBER, MORE, COMMAND_FUNCTION, DYNAMIC_FUNCTION oder ROW_COUNT sein.

Um diagnostische Einzelheiten des Diagnosebereichs abzufragen, verwenden Sie folgende Syntax:

```
GET DIAGNOSTICS EXCEPTION condition-number
   status1 = item1 [, status2 = item2]...
```

Auch hier ist statusN eine Host-Variable oder ein Parameter und itemN eines der 28 Schlüsselwörter für Detaileinträge, die in Tabelle 21.2 aufgeführt sind. condition-number steht für (Überraschung!) CONDITION_NUMBER des Detailbereichseintrags.

Ausnahmen handhaben

Wenn SQLSTATE einen anderen Wert als 00000, 00001 oder 00002 enthält, liegt eine Ausnahmesituation vor, auf die Sie folgendermaßen reagieren können:

- ✔ Sie geben die Kontrolle an die übergeordnete Prozedur zurück, die die auslösende Unterprozedur aufgerufen hat, die den Fehler bewirkt.

- ✔ Sie verwenden eine WHENEVER-Klausel, um zu einer Fehlerbehandlungsroutine zu verzweigen oder um eine andere Aktion auszuführen (diese Klausel wird weiter vorn in diesem Kapitel beschrieben).
- ✔ Sie behandeln die Ausnahme mit einer *zusammengesetzten* SQL-Anweisung dort, wo der Fehler auftritt. Eine zusammengesetzte SQL-Anweisung besteht aus einer oder mehreren einfachen SQL-Anweisungen, die durch die Schlüsselwörter BEGIN und END eingeschlossen sind. Sie finden in Kapitel 20 eine Beschreibung der zusammengesetzten SQL-Anweisungen.

Der folgende Code zeigt ein Beispiel für die Ausnahmebehandlung mit einer zusammengesetzten SQL-Anweisung:

```
BEGIN
   DECLARE WertNichtImBereich EXCEPTION FOR SQLSTATE '73003' ;
   INSERT INTO Nahrungsmittel
      (KALORIEN)
      VALUES
      (:kal) ;
   SIGNAL WertNichtImBereich ;
   MESSAGE 'Den neuen Kalorienwert verarbeiten.'
   EXCEPTION
      WHEN WertNichtImBereich THEN
         MESSAGE 'Den Kalorienwertfehler behandeln.' ;
      WHEN OTHERS THEN
         RESIGNAL ;
END
```

Mit einer oder mehreren DECLARE-Anweisungen können Sie speziellen SQLSTATE-Werten, die Sie möglicherweise während der Ausführung des Codes erwarten, Namen geben. Die Anweisung INSERT gehört zu den Anweisungen, die einen Fehler auslösen könnten. Wenn der Wert von :kal den Höchstwert für ein SMALLINT-Datenelement überschreitet, wird SQLSTATE auf 73003 gesetzt. Die Anweisung SIGNAL zeigt eine Ausnahmesituation an. Sie löscht den oberen Diagnosebereich. Sie weist dem Feld RETURNED_SQLSTATE des Diagnosebereichs den SQLSTATE für die benannte Ausnahme zu. Falls keine Ausnahme eingetreten ist, wird die Gruppe von Anweisungen ausgeführt, die durch die Anweisung MESSAGE 'Den neuen Kalorienwert verarbeiten.' vertreten wird. Falls jedoch eine Ausnahme aufgetreten ist, wird diese Gruppe von Anweisungen übersprungen und EXCEPTION ausgeführt.

Falls es sich bei der Ausnahme um eine WertNichtImBereich-Ausnahme handelt, wird die Gruppe von Anweisungen ausgeführt, die durch die Anweisung MESSAGE 'Den Kalorien wertfehler behandeln.' vertreten wird. Wenn es sich um eine andere Ausnahme handelt, wird RESIGNAL ausgeführt.

Trigger

22

In diesem Kapitel

- Trigger erstellen
- Überlegungen zur Auslösung von Triggern
- Einen Trigger ausführen
- Mehrere Trigger auslösen

Bei der Ausführung einer Datenbankanwendung können Situationen auftreten, in denen eine Aktion oder eine Folge von Aktionen in Abhängigkeit von einer anderen Aktion ausgeführt werden soll. In gewissem Sinne löst diese erste Aktion die Ausführung der folgenden Aktionen aus. SQL stellt für diese Fähigkeit den *Trigger*-Mechanismus (englisch *trigger* = deutsch *Auslöser*, *Abzug*) zur Verfügung.

Wahrscheinlich kennt jeder Trigger als den Teil von Feuerwaffen, mit dem diese ausgelöst werden. Allgemeiner ausgedrückt ist ein Trigger eine Aktion, die das Eintreten eines anderen Ereignisses auslöst. In SQL wird das Wort *Trigger* in dieser allgemeineren Bedeutung verwendet. Eine SQL-Anweisung, die als Trigger funktioniert, löst die Ausführung einer anderen SQL-Anweisung (die getriggerte Anweisung) aus.

Einige Anwendungen von Triggern

Trigger lassen sich in verschiedenen Situationen sinnvoll einsetzen. Ein Beispiel ist die Ausführung einer Protokollierungsfunktion. Bestimmte Aktionen, die für die Integrität einer Datenbank wesentlich sind, wie etwa das Einfügen, Bearbeiten oder Löschen einer Tabellenzeile, können das Einfügen eines Eintrags in einem Protokoll auslösen, um diese Aktion zu dokumentieren. Protokolleinträge können nicht nur festhalten, welche Aktion ausgeführt worden ist, sondern auch, wann und von wem.

Trigger können auch verwendet werden, die Konsistenz einer Datenbank zu bewahren. Bei einer Anwendung zur Auftragserfassung kann etwa eine Bestellung eines bestimmten Artikels eine Anweisung auslösen, die den Status dieses Produkts in der Lagerbestandstabelle von *verfügbar* in *reserviert* ändern. Ähnlich kann das Löschen einer Zeile in der Tabelle der Bestellungen eine Anweisung auslösen, die den Status des Produkts von *reserviert* in *verfügbar* ändert.

Trigger bieten mehr Flexibilität als von den vorhergehenden Beispielen illustriert wird. Das ausgelöste Element muss keine SQL-Anweisung sein. Es kann sich auch um eine Prozedur der Host-Sprache handeln, die eine Operation in der Außenwelt ausführt, etwa indem sie ein Fließband anhält oder einen Roboter veranlasst, ein kaltes Bier aus dem Kühlschrank zu holen.

Einen Trigger erstellen

Ein Trigger wird mit der Anweisung `CREATE TRIGGER` erstellt. Danach wartet der Trigger darauf, dass sein auslösendes Ereignis eintritt. Ist dies der Fall, wird der Trigger ausgelöst.

Die Syntax der Anweisung `CREATE TRIGGER` ist recht umfangreich; doch man kann sie in verständliche Teile zerlegen. Zunächst die Syntax insgesamt:

```
CREATE TRIGGER Trigger_Name
   Trigger_Aktionszeit Trigger_Ereignis
   ON Tabellenname
   [REFERENCING Alte_oder_neue_Wert_Alias_Liste]
   Ausgelöste_Aktion
```

Der `Trigger_Name` ist der eindeutige Bezeichner für diesen Trigger. Die `Trigger_Aktionszeit` ist der Zeitpunkt, an dem die getriggerte Aktion ausgeführt werden soll: entweder `BEFORE` oder `AFTER` dem Trigger-Ereignis. Die Tatsache, dass eine getriggerte Aktion *vor* dem Ereignis ausgeführt werden kann, durch das sie erst ausgelöst wird, scheint etwas bizarr zu sein; in manchen Situationen kann diese Fähigkeit sehr nützlich sein (und auch ohne Zeitreise realisiert werden). Weil die Datenbank-Engine weiß, dass sie auf ein Trigger-Ereignis reagieren wird, bevor sie die entsprechende Aktion tatsächlich ausführt, kann sie das getriggerte Ereignis vor der Ausführung einschieben, wenn die `Trigger_Aktionszeit` mit dem Wert `BEFORE` spezifiziert worden ist.

Drei mögliche Trigger-Ereignisse können einen Trigger auslösen: die Ausführung einer `INSERT`-Anweisung, einer `DELETE`-Anweisung oder einer `UPDATE`-Anweisung. Diese drei Anweisungen können den Inhalt einer Datenbanktabelle ändern. Deshalb kann der Trigger ausgelöst werden, wenn eine oder mehrere Zeilen in die Subjekttabelle eingefügt werden, wenn eine oder mehrere Zeilen aus der Subjekttabelle gelöscht werden oder wenn eine oder mehrere Spalten in einer oder in mehreren Zeilen der Subjekttabelle geändert werden. `ON Tabellen_Name` bezieht sich natürlich auf die Tabelle, für die ein `INSERT`, `DELETE` oder `UPDATE` spezifiziert worden ist.

Anweisungs- und Zeilen-Trigger

Die `Getriggerte_Aktion` aus dem vorhergehenden Beispiel hat die folgende Syntax:

```
[ FOR EACH { ROW | STATEMENT } ]
   WHEN <linkes Oberelement >
        <Suchbedingung>
        <rechtes Oberelement>
   <getriggerte SQL–Anweisung>
```

Sie können spezifizieren, wie sich der Trigger verhalten soll:

- ✔ **Zeilentrigger:** Der Trigger wird ausgelöst, wenn das Programm bei der Ausführung auf eine INSERT-, DELETE- oder UPDATE-Anweisung stößt, die das Trigger-Ereignis definiert.
- ✔ **Anweisungs-Trigger:** Der Trigger wird mehrfach ausgelöst, einmal für jede Zeile in der Subjekttabelle, die von dem Trigger-Ereignis betroffen ist.

Die eckigen Klammern zeigen an, dass die FOR EACH-Klausel optional ist. Dennoch muss sich der Trigger so oder so verhalten. Ist keine FOR EACH-Klausel angegeben, ist das Standardverhalten FOR EACH STATEMENT.

Wenn ein Trigger ausgelöst wird

Mit der Suchbedingung in der WHEN-Klausel können Sie Umstände spezifizieren, unter denen ein Trigger ausgelöst wird. Sie spezifizieren ein Prädikat, und wenn das Prädikat wahr ist, wird der Trigger ausgelöst; ist es falsch, wird er nicht ausgelöst. Diese Fähigkeit vergrößert den Nutzen von Triggern erheblich. Sie können einen Trigger spezifizieren, der nur ausgelöst wird, nachdem ein bestimmter Schwellenwert überschritten worden ist oder wenn eine andere Bedingung definiert werden kann, die entweder *True* oder *False* ist.

Die getriggerte SQL-Anweisung

Eine getriggerte SQL-Anweisung kann aus einer einzelnen SQL-Anweisung oder einer Folge von SQL-Anweisungen bestehen, die nacheinander ausgeführt werden. Im Fall einer einzelnen SQL-Anweisung ist die getriggerte SQL-Anweisung nur eine gewöhnliche SQL-Anweisung. Doch bei einer Folge von SQL-Anweisungen müssen Sie die Atomarität garantieren, um zu gewährleisten, dass die Operation nicht mittendrin unterbrochen und die Datenbank in einem unerwünschten Zustand hinterlassen wird. Sie können dies mit einem BEGIN–END-Block machen, der das Schlüsselwort ATOMIC enthält:

```
BEGIN ATOMIC
     { SQL Anweisung 1 }
     { SQL Anweisung 2 }
     ...
     { SQL Anweisung n }
END
```

Ein Beispiel für eine Trigger-Definition

Angenommen, der Personalchef eines Unternehmens will informiert werden, wenn ein Regionalleiter einen neuen Mitarbeiter einstellt. Der folgende Trigger liefert die gewünschten Informationen:

```
CREATE TRIGGER NeuerMitarbeiter
   BEFORE INSERT ON Mitarbeiter
   FOR EACH STATEMENT
     BEGIN ATOMIC
       CALL sendmail ('Personalchef')
       INSERT INTO logtable
         VALUES ('NEUEMITARBEITER', CURRENT_USER, CURRENT_TIMESTAMP) ;
     END;
```

Wenn eine neue Zeile in die `NeuerMitarbeiter`-Tabelle eingefügt wird, wird eine E-Mail an den Personalchef gesendet. Sie enthält Detaildaten sowie den Anmeldenamen der Person, die die Zeile eingefügt hat, und den zugehörigen Zeitstempel. Zwecks Nachverfolgbarkeit werden die Daten in einer Protokolldatei gespeichert.

Eine Folge von Triggern auslösen

Wahrscheinlich können Sie ein Problem bei der Arbeitsweise von Triggern erkennen. Angenommen, Sie erstellen einen Trigger, der die Ausführung einer SQL-Anweisung auslöst, die auf eine Tabelle angewendet werden soll, nachdem vorher eine andere SQL-Anweisung ausgeführt worden ist. Was passiert, wenn die getriggerte Anweisung selbst einen zweiten Trigger auslöst? Dieser zweite Trigger sorgt dafür, dass eine dritte SQL-Anweisung ausgeführt wird, die auf eine zweite Tabelle angewendet wird, wodurch selbst wiederum noch ein weiterer Trigger ausgelöst werden kann, der eine dritte Tabelle beeinflusst. Wie kann man da den Überblick behalten? SQL handhabt diese Art der Trigger-Auslösung im Stil eines Maschinengewehrs mit sogenannten *Trigger-Ausführungskontexten* (englisch *trigger execution context)*.

Eine Folge von `INSERT`-, `DELETE`- und `UPDATE`-Operationen kann durch Verschachtelung der Kontexte ausgeführt werden, in denen sie auftreten. Wenn ein Trigger ausgelöst wird, wird ein Ausführungskontext erstellt. Es kann immer nur ein Ausführungskontext aktiv sein. In diesem Kontext kann eine SQL-Anweisung ausgeführt werden, die einen zweiten Trigger auslöst. An diesem Punkt wird der vorhandene Ausführungskontext mit einer Operation zeitweilig unterbrochen, die damit vergleichbar ist, einen Wert auf einem Stack abzulegen. Es wird ein neuer Ausführungskontext erstellt, der dem zweiten Trigger entspricht; und seine Operation wird ausgeführt. Es gibt kein willkürliches Limit für die Tiefe der möglichen Verschachtelungen. Wenn eine Operation abgeschlossen ist, wird ihr Ausführungskontext zerstört; der nächsthöhere Ausführungskontext wird »vom Stack genommen« und reaktiviert. Dieser Prozess wird fortgesetzt, bis alle Aktionen abgeschlossen und alle Ausführungskontexte zerstört worden sind.

Alte Werte und neue Werte referenzieren

Die Anweisung CREATE TRIGGER enthält einen Teil, den ich noch nicht beschrieben habe: REFERENCING Alte_oder_neue_Werte_Alias_Liste, eine optionale Phrase. Damit können Sie einen Alias oder einen Korrelationsnamen erstellen, der Werte in der Subjekttabelle des Triggers referenziert. Nachdem Sie einen Korrelationsnamen für neue Werte oder einen Alias für neue Tabelleninhalte erstellt haben, können Sie dann die Werte referenzieren, die nach einer INSERT- oder UPDATE-Operation existieren. Ähnlich können Sie, nachdem Sie einen Korrelationsnamen für alte Werte oder einen Alias für alte Tabelleninhalte erstellt haben, dann die Werte referenzieren, die vor einer UPDATE- oder DELETE-Operation in der Subjekttabelle existiert haben.

Die Alte_oder_neue_Werte_Alias_Liste in der CREATE TRIGGER-Syntax kann eine oder mehrere der folgenden Phrasen enthalten:

```
OLD [ ROW ] [ AS ] <Alte Werte Korrelationsname>
```

oder

```
NEW [ ROW ] [ AS ] <Neue Werte Korrelationsname>
```

oder

```
OLD TABLE [ AS ] <Alte Werte Tabellen-Alias>
```

oder

```
NEW TABLE [ AS ] <Neue Werte Tabellen-Alias>
```

Die Tabellen-Aliasse sind Bezeichner für Transition-Tabellen, die nicht persistent (dauerhaft) sind, sondern die nur existieren, um die Referenzierungsoperation zu vereinfachen. Erwartungsgemäß dürfen NEW ROW und NEW TABLE nicht für einen DELETE-Trigger spezifiziert werden; desgleichen dürfen OLD ROW und OLD TABLE nicht für einen INSERT-Trigger spezifiziert werden. Nachdem Sie eine Zeile oder Tabelle gelöscht haben, gibt es keinen neuen Wert. Ähnlich dürfen OLD ROW und OLD TABLE nicht für einen INSERT-Trigger spezifiziert werden. Es gibt keine alten Werte, die man referenzieren könnte.

Bei einem Trigger auf Zeilenebene können Sie die Werte in der zu ändernden oder zu löschenden Zeile mit einem Alte Werte Korrelationsname referenzieren, indem Sie eine SQL-Anweisung für die Zeile in der Form triggern, wie sie existierte, bevor sie von der Anweisung bearbeitet oder gelöscht wurde. Ähnlich können Sie mit einem Alte Werte Tabellen-Alias auf die Werte in der gesamten Tabelle in der Form zugreifen, wie sie existierte, bevor die Aktion der getriggerten SQL-Anweisung wirksam wurde.

Weder OLD TABLE noch NEW TABLE darf mit einem BEFORE-Trigger verwendet werden. Es ist zu wahrscheinlich, dass die Transition-Tabellen, die mit OLD TABLE oder NEW TABLE erstellt werden, von den Aktionen beeinflusst werden, die von der getriggerten SQL-Anweisung ausgelöst werden. Um dieses potenzielle Problem zu vermeiden, dürfen OLD TABLE und NEW TABLE nicht mit einem BEFORE-Trigger verwendet werden.

Mehrere Trigger für eine einzelne Tabelle auslösen

Das letzte Thema, das ich in diesem Kapitel vorstellen möchte, ist der Fall, in dem mehrere Trigger erstellt werden, die alle die Ausführung einer SQL-Anweisung anstoßen, die dieselbe Tabelle bearbeiten. Alle diese Trigger werden aktiviert und sind bereit zu feuern. Doch welcher Trigger wird zuerst ausgelöst, wenn das Trigger-Ereignis eintritt? Das Problem wird durch eine Design-Entscheidung gelöst. Der Trigger, der zuerst erstellt wird, wird zuerst ausgelöst; dann folgt der zweite; dann der dritte und so weiter. So werden mögliche Mehrdeutigkeiten vermieden, und die Anweisungen werden geordnet ausgeführt.

Teil VII

Der Top-Ten-Teil

Besuchen Sie uns auf www.facebook.de/fuerdummies!

In diesem Teil ...

- ✔ Häufige Fehler
- ✔ Schnelle Datenabfrage

Zehn häufige Fehler

23

In diesem Kapitel

- Davon ausgehen, dass die Kunden wissen, was sie brauchen
- Sich keine Gedanken über den Umfang eines Projekts machen
- Nur technische Faktoren berücksichtigen
- Auf keinen Fall einen Benutzer um Rückmeldungen bitten
- Nur die liebste Entwicklungsumgebung oder Systemarchitektur benutzen
- Datenbanktabellen unabhängig voneinander entwerfen
- Die Design-Reviews, Betatests und Dokumentation vernachlässigen

Wenn Sie dieses Buch lesen, müssen Sie ein Interesse daran haben, relationale Datenbanksysteme zu entwerfen. Niemand studiert SQL nur aus Spaß an der Sache. Sie setzen SQL ein, um Datenbankanwendungen zu erstellen. Doch dazu brauchen Sie eine Datenbank. Leider scheitern viele Projekte bereits, bevor die erste Zeile der Anwendung geschrieben worden ist. Wenn Sie die Datenbank nicht richtig definieren, ist Ihre Anwendung zum Scheitern verurteilt – egal, wie gut Sie sie auch schreiben. Hier sind zehn Fehler, die häufig beim Erstellen einer Datenbank gemacht werden und die Sie tunlichst vermeiden sollten.

Annehmen, dass die Kunden wissen, was sie brauchen

Kunden beauftragen Sie im Allgemeinen dann, ein Datenbanksystem zu entwerfen, wenn sie ihre Daten mit ihrer normalen Arbeitsweise nicht mehr in der gewünschten Form abrufen können. Oft glauben sie zu wissen, worin das Problem besteht und wie es gelöst werden kann. Sie stellen sich vor, dass alles, was sie tun müssen, ist, Ihnen zu sagen, was *Sie* tun müssen.

Kunden genau das zu geben, was sie haben wollen, führt normalerweise garantiert zu einer Katastrophe. Die meisten Benutzer (und ihre Vorgesetzten) verfügen weder über das Wissen noch die Fähigkeiten, um das Problem wirklich zu erkennen, und haben damit kaum eine Chance, die beste Lösung zu ermitteln.

Es ist Ihre Aufgabe, den Kunden taktvoll davon zu überzeugen, dass Sie der Fachmann (oder die Fachfrau) für Systemanalyse und Systementwurf sind und dass Sie das Problem gründlich analysieren müssen, um seine wirkliche Ursache herauszufinden. Normalerweise verbirgt sich diese unter den vordergründigen Symptomen.

Den Umfang des Projekts ignorieren

Zu Beginn eines Entwicklungsprojekts sagt Ihnen Ihr Kunde, was die neue Anwendung leisten soll. Leider vergisst er dabei immer, Ihnen einige Punkte mitzuteilen. Im Laufe des Projekts tauchen diese dann immer wieder als neue Anforderungen auf und werden in das Projekt aufgenommen. Wenn Ihr Honorar auf Projektbasis und nicht auf Stundenbasis berechnet wird, kann diese Zunahme des Projektumfangs aus einem vorher profitablen Projekt ein Verlustgeschäft machen. Achten Sie darauf, dass alles, was Sie liefern sollen, vor dem Start des Projekts schriftlich festgelegt wird.

Nur technische Faktoren berücksichtigen

Anwendungsentwickler betrachten potenzielle Projekte oft nur unter dem Aspekt der technischen Machbarkeit und schätzen Zeit und Aufwand ausschließlich unter diesem Gesichtspunkt. Doch auch andere Faktoren, wie etwa Kostenobergrenzen, Verfügbarkeit von Ressourcen, Zeitrahmen und Firmenpolitik, können ein Projekt erheblich beeinflussen. Diese Faktoren können aus einem technisch machbaren Projekt einen Albtraum machen. Deshalb müssen Sie auch alle wesentlichen nicht technischen Faktoren kennen, bevor Sie mit einem Entwicklungsprojekt beginnen. Vielleicht stellen Sie fest, dass es sinnlos ist, weiterzumachen. Es ist besser, diese Einsicht am Anfang eines Projekts und nicht erst dann zu gewinnen, wenn Sie bereits einige Mühe dafür aufgewendet haben.

Nicht um Feedback bitten

Ihr erster Impuls mag darin bestehen, auf die Manager zu hören, die Sie engagiert haben. Denn schließlich haben die Benutzer selbst nicht die entsprechende Statur und stellen Ihnen letztendlich auch nicht Ihren Scheck aus. Andererseits gibt es gute Gründe, auch nicht auf die Manager zu hören. Denn normalerweise haben sie keine Ahnung davon, was die Benutzer wirklich benötigen. Stopp! Ignorieren Sie nicht jedermann. Gehen Sie nicht automatisch davon aus, dass Sie besser als Ihre Kunden und deren Mitarbeiter wissen, was eine Datenbank machen und wie sie arbeiten soll. Zwar haben Sachbearbeiter in der Datenerfassung keinen großen Einfluss auf ein Unternehmen und viele leitende Angestellte wissen nur wenig von der Tätigkeit der ihnen unterstellten Sachbearbeiter, doch wenn Sie sich von beiden Gruppen abkapseln, werden Sie mit ziemlicher Sicherheit ein System entwickeln, das Probleme löst, die niemand hat. Sie können sowohl von den Managern als auch von den Benutzern viel lernen, wenn Sie die richtigen Fragen stellen.

Immer Ihre liebste Entwicklungsumgebung benutzen

Wahrscheinlich haben Sie Monate oder sogar Jahre damit zugebracht, die Benutzung eines bestimmten Datenbanksystems oder einer bestimmten Entwicklungsumgebung zu erlernen. Ihre Lieblingsumgebung – egal, welche – hat ihre Stärken und Schwächen. Gelegentlich stehen Sie vor einer Entwicklungsaufgabe, die gerade in einem Bereich viel fordert, in dem Ihre

Entwicklungsumgebung schwach ist. Anstatt nun eine suboptimale Behelfslösung zusammenzuhacken, sollten Sie in einem solchen Fall in den sauren Apfel beißen. Sie haben zwei Optionen: Entweder erklimmen Sie die Lernkurve eines passenderen Werkzeugs und wenden es an oder Sie teilen Ihrem Kunden offen mit, dass sein Problem am besten mit einem Werkzeug gelöst werden kann, für das Sie kein Experte sind. Schlagen Sie vor, jemanden zu engagieren, der mit diesem Werkzeug sofort produktiv arbeiten kann. Ein professionelles Verhalten dieser Art erhöht den Respekt Ihres Kunden vor Ihnen. (Wenn Sie allerdings Angestellter einer Firma sind und nicht selbstständig arbeiten, kann dies leider auch dazu führen, dass Sie entlassen werden. In solch einem Fall sollten Sie sich für Alternative eins entscheiden.)

Immer Ihre liebste Systemarchitektur benutzen

Niemand kann Fachmann für alles sein. Datenbankverwaltungssysteme, die mit Datenfernverarbeitung zu tun haben, unterscheiden sich von Systemen, bei denen es um gemeinsam genutzte Ressourcen in Client/Server-Umgebungen geht. Die ein oder zwei Umgebungen, die Sie als Experte beherrschen, sind möglicherweise nicht die besten, um die anstehende Aufgabe zu lösen. Wählen Sie trotzdem die beste Architektur, selbst wenn Sie dadurch diesen Auftrag verlieren. Einen Auftrag zu verlieren ist besser, als ein System zu produzieren, das die Anforderungen des Kunden nicht erfüllt.

Datenbanktabellen unabhängig voneinander entwerfen

Wenn Sie Datenobjekte und ihre Beziehungen falsch identifizieren, führt dies zu Datenbanktabellen, die fehlerhafte Daten produzieren und die Gültigkeit aller Ergebnisse gefährden. Um eine solide Datenbank zu entwerfen, müssen Sie die gesamte Struktur der Datenobjekte berücksichtigen und sorgfältig herausfinden, in welchen Beziehungen sie zueinander stehen. Normalerweise gibt es nicht nur *eine* richtige Lösung. Sie müssen herausfinden, welche Variante die passende ist, und dabei sowohl die aktuellen als auch die zukünftigen Anforderungen Ihres Kunden berücksichtigen.

Design-Reviews ignorieren

Niemand ist perfekt. Selbst die besten Designer und Entwickler können wichtige Punkte übersehen, die aus einer anderen Perspektive sofort sichtbar sind. Wenn Ihr Entwurf einem Design-Review, also einer formellen Überprüfung unterworfen werden soll, werden Sie wahrscheinlich disziplinierter arbeiten und so zahlreiche potenzielle Probleme vermeiden. Lassen Sie Ihren Entwurf von einem kompetenten Profi begutachten und zeigen Sie sie auch Ihrem Kunden.

Betatests überspringen

Alle Datenbankanwendungen, die komplex genug sind, um wirklich nützlich zu sein, sind auch komplex genug, um Fehler zu enthalten. Selbst wenn Sie den Entwurf auf jede denkbare Weise testen, werden Sie nicht alle möglichen Fehlerbedingungen entdecken. Bei Betatests lassen Sie Personen mit der Anwendung arbeiten, die Ihren Entwurf nicht kennen. Wahrscheinlich stoßen diese dabei auf Probleme, mit denen Sie nie zu tun hatten, weil Sie die Anwendung einfach zu gut kennen. Kennen Ihre Tester zwar die Daten, nicht aber die Datenbank, nutzen sie die Anwendung wahrscheinlich eher wie im alltäglichen Einsatz; deshalb können sie eher Abfragen finden, deren Ausführung sehr lange dauert. So können Sie Fehler oder Engpässe beseitigen, bevor das Produkt offiziell freigegeben wird.

Keine Dokumentation erstellen

Meinen Sie wirklich, Ihre Anwendung sei so perfekt, dass sich niemand mehr darum kümmern muss? Nur eines ist in dieser Welt sicher: der Wandel. In sechs Monaten werden Sie nicht mehr wissen, warum Sie den Entwurf so gestaltet haben, wie Sie es getan haben, es sei denn, Sie haben sorgfältig dokumentiert, was Sie getan haben und warum Sie es getan haben. Wenn Sie in eine andere Abteilung versetzt werden oder im Lotto gewinnen und sich zur Ruhe setzen, hat Ihr Nachfolger kaum eine Chance, Ihre Arbeit an neue Anforderungen anzupassen. Ohne Dokumentation muss er Ihr Produkt möglicherweise komplett verschrotten und ganz von vorne anfangen.

Dokumentieren Sie Ihre Arbeit nicht nur ausreichend, sondern dokumentieren Sie sie mehr als sorgfältig. Geben Sie mehr Einzelheiten an, als eigentlich ausreichend wäre. Wenn Sie sich nach sechs oder acht Monaten Abwesenheit wieder mit dem Projekt befassen müssen, werden Sie froh sein, dass Sie sie detailliert dokumentiert haben.

Zehn Tipps für Abfragen

24

In diesem Kapitel

- Die Datenbankstruktur auf Gültigkeit prüfen
- Testdatenbanken benutzen
- Verknüpfungsabfragen analysieren
- Abfragen mit Unterabfragen sorgfältig untersuchen
- `GROUP BY` bei `SET`-Funktionen benutzen
- Einschränkungen der Klausel `GROUP BY` beachten
- Klammern in Ausdrücken verwenden
- Die Datenbank durch Rechte schützen
- Die Datenbank regelmäßig sichern
- Fehler voraussehen und beheben

Eine Datenbank kann eine Schatzkammer für Informationen sein, aber wie die Schätze der Piraten der Karibik sind die Daten, die Sie sehen wollen, wahrscheinlich tief vergraben und vor Ihren Blicken verborgen. Die SQL-Anweisung `SELECT` ist das Werkzeug, mit dem Sie diese verborgenen Informationen ausgraben können. Selbst wenn Sie eine genaue Vorstellung von den Daten haben, die Sie abrufen wollen, ist das Übersetzen dieser Vorstellung in SQL oft eine Herausforderung. Wenn Ihre Formulierung auch nur ein wenig falsch ist, können Sie falsche Ergebnisse erhalten. Möglicherweise weichen sie nur wenig von Ihren Erwartungen ab und führen Sie deshalb in die Irre. Um diese Gefahr zu verringern, sollten Sie die folgenden zehn Grundsätze beachten.

Prüfen Sie die Datenbankstruktur

Wenn Sie Daten aus einer Datenbank abrufen und die Ergebnisse nicht plausibel sind, überprüfen Sie den Datenbankentwurf. Viele Datenbanken haben eine miserable Struktur. Wenn Sie mit einer davon arbeiten, sollten Sie zunächst die Struktur ändern, ehe Sie anderweitig auf Abhilfen sinnen. Denken Sie daran, ein guter Datenbankentwurf ist eine Voraussetzung für die Integrität.

Testen Sie Abfragen mit einer Testdatenbank

Erstellen Sie eine Testdatenbank, die dieselbe Struktur wie die Produktionsdatenbank hat, aber nur wenige repräsentative Zeilen in den Tabellen enthält. Wählen Sie die Daten so, dass Sie vorher wissen, wie das Ergebnis der Abfrage aussehen wird. Führen Sie die Abfrage mit den Testdaten aus und prüfen Sie, ob das Ergebnis mit Ihren Erwartungen übereinstimmt. Falls dies nicht der Fall ist, formulieren Sie die Abfrage um. Wenn die Abfrage sauber formuliert ist, Sie aber immer noch falsche Ergebnisse bekommen, müssen Sie Ihre Datenbank möglicherweise neu strukturieren.

Konstruieren Sie mehrere Gruppen von Testdaten und beachten Sie dabei auch ausgefallene Fälle, wie leere Tabellen und Grenzwerte der zulässigen Datenbereiche. Denken Sie auch an sehr unwahrscheinliche Szenarien, und prüfen Sie, ob sich die Datenbank auch dann noch korrekt verhält. Wenn Sie unwahrscheinliche Fälle prüfen, lernen Sie auch eher seltene Fälle besser kennen.

Prüfen Sie Verknüpfungsabfragen doppelt

Verknüpfungen widersprechen oft der Intuition. Enthält Ihre Abfrage eine Verknüpfung, prüfen Sie, ob sie sich erwartungskonform verhält, bevor Sie sie mit `WHERE`-Klauseln oder anderen Faktoren komplizierter machen.

Prüfen Sie Abfragen mit einer Unterabfrage dreifach

Abfragen mit Unterabfragen rufen erst Daten aus einer Tabelle und dann, in Abhängigkeit von diesem Ergebnis, Daten aus einer zweiten Tabelle ab. Deshalb können Sie hier leicht Fehler machen. Prüfen Sie, ob das innere `SELECT` wirklich die Daten abruft, die das äußere `SELECT` benötigt, um das gewünschte Endergebnis zu produzieren. Bei zwei oder mehr Ebenen von Unterabfragen müssen Sie noch vorsichtiger sein.

Daten mit GROUP BY summieren

Angenommen, Sie hätten eine Tabelle (`Bundesliga`) mit den Spielernamen (`Spieler`), der Mannschaft (`Mannschaft`) und der Anzahl der Tore (`Tore`), die jeder Spieler in der Bundesliga geschossen hat. Dann können Sie die Gesamtzahl der Tore, die jede Mannschaft geschossen hat, mit folgender Abfrage ermitteln:

```
SELECT Mannschaft, SUM (Tore)
   FROM Bundesliga
   GROUP BY Mannschaft ;
```

Diese Abfrage listet alle Mannschaften und die Gesamtzahl aller Tore auf, die die Spieler jeder Mannschaft geschossen haben.

Beachten Sie die Einschränkungen der Klausel GROUP BY

Angenommen, Sie wollten eine Liste der Torjäger in der Bundesliga erstellen. Betrachten Sie folgende Abfrage:

```
SELECT Spieler, Mannschaft, Tore
   FROM Bundesliga
   WHERE Tore >= 20
   GROUP BY Mannschaft ;
```

Bei den meisten Implementierungen liefert diese Abfrage einen Fehler. Im Allgemeinen dürfen in der `SELECT`-Liste nur Spalten aufgeführt werden, die auch zum Gruppieren oder in einer Mengenfunktion benutzt werden. Die folgende Abfrage ist korrekt:

```
SELECT Spieler, Mannschaft, Tore
   FROM Bundesliga
   WHERE Tore >= 20
   GROUP BY Mannschaft, Spieler, Tore ;
```

Weil alle Spalten, die Sie anzeigen wollen, in der Klausel `GROUP BY` aufgeführt sind, liefert die Abfrage das gewünschte Ergebnis und sortiert die gelieferten Daten nach den Spalten `Mannschaft`, `Spieler` und `Tore`.

Benutzen Sie bei AND, OR und NOT Klammern

Wenn Sie in einem Ausdruck `AND` und `OR` gleichzeitig verwenden, verarbeitet SQL den Ausdruck nicht immer in der Reihenfolge, die Sie erwarten. Benutzen Sie in komplexen Ausdrücken Klammern, um das gewünschte Ergebnis zu bekommen. Der geringe zusätzliche Eingabeaufwand ist ein kleiner Preis für korrektere Ergebnisse.

Klammern helfen Ihnen auch dabei, dass das Schlüsselwort `NOT` auf genau den Ausdruck angewendet wird, auf den es angewendet werden soll.

Überwachen Sie Abfragerechte

Viele Anwender benutzen nicht die Sicherheitsfunktionen, die in ihrem DBMS verfügbar sind. Sie wollen mit diesem Thema nichts zu tun haben, weil sie davon ausgehen, dass Missbrauch und zweckfremde Verwendung von Daten etwas ist, das nur anderen passiert. Warten Sie nicht, bis Sie eines Besseren belehrt werden. Entwerfen Sie für alle Datenbanken, die für Sie von Wert sind, ein Sicherheitskonzept und richten Sie es ein.

Sichern Sie Ihre Datenbanken regelmäßig

Es ist extrem schwer, Daten wiederherzustellen, wenn Ihre Festplatte durch einen Spannungsstoß, ein Feuer, ein Erdbeben oder eine andere Katastrophe zerstört worden ist. (Und denken Sie daran, dass Computer manchmal »nur so« ihren Geist aufgeben.) Sichern Sie Ihre Daten regelmäßig und bewahren Sie die Sicherungsmedien an einer sicheren Stelle auf.

Was eine »sichere Stelle« ausmacht, ist davon abhängig, wie wichtig Ihre Daten sind. Sie können Ihre Sicherungskopien in einem feuerfesten Safe in dem Raum aufbewahren, in dem sich auch Ihr Computer befindet. Dieser Ort kann sich auch in einem anderen Gebäude befinden. Es könnte aber auch ein stabiler Bunker unter einem Berg sein, der einem atomaren Angriff standhalten kann. Sie entscheiden, welchen Grad an Sicherheit Ihre Daten benötigen.

Bauen Sie eine Fehlerbehandlung ein

Egal, ob Sie Abfragen an der Konsole ausführen oder in eine Anwendung einbetten – gelegentlich meldet SQL einen Fehler, anstatt das gewünschte Ergebnis zu liefern. An der Konsole können Sie aufgrund der Fehlermeldung entscheiden, was als Nächstes zu unternehmen ist. In einer Anwendung hingegen ist die Situation anders. Wahrscheinlich weiß der Benutzer der Anwendung nicht, was zu tun ist. Fügen Sie deshalb in Ihre Anwendungen eine ausführliche Fehlerbehandlung ein, um jeden vorstellbaren Fehler abzudecken, der auftreten könnte. Es ist sehr aufwendig, eine Fehlerbehandlung zu programmieren, aber das ist besser, als den Benutzer hilflos vor einem eingefrorenen Bildschirm sitzen zu lassen.

SQL:2011 Reservierte Wörter

```
ABS
ALL
ALLOCATE
ALTER
AND
ANY
ARE
ARRAY
ARRAY_AGG
AS
ASENSITIVE
ASYMMETRIC
AT
ATOMIC
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIGINT
BINARY
BLOB
BOOLEAN
BOTH
BY
CALL
CALLED
CARDINALITY
CASCADED
CASE
CAST
CEIL
CEILING
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLOB
CLOSE
COALESCE
COLLATE
COLLECT
COLUMN
COMMIT
CONDITION
CONNECT
CONSTRAINT
CONVERT
CORR
CORRESPONDING
COUNT
COVAR_POP
COVAR_SAMP
CREATE
CROSS
CUBE
CUME_DIST
CURRENT
CURRENT_CATALOG
CURRENT_DATE
CURRENT_DEFAULT_
  TRANSFORM_GROUP
CURRENT_PATH
CURRENT_ROLE
CURRENT_SCHEMA
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSFORM_
  GROUP_FOR_TYPE
CURRENT_USER
CURSOR
CYCLE
DATE
DAY
DAYS
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DELETE
DENSE_RANK
DEREF
DESCRIBE
DETERMINISTIC
DISCONNECT
DISTINCT
DOUBLE
DROP
DYNAMIC
EACH
ELEMENT
ELSE
END
END-EXEC
ESCAPE
EVERY
EXCEPT
EXEC
EXECUTE
EXISTS
EXP
EXTERNAL
EXTRACT
FALSE
FETCH
FILTER
FLOAT
FLOOR
FOR
FOREVER
FOREIGN
FREE
FROM
FULL
FUNCTION
FUSION
GET
GLOBAL
GRANT
GROUP
GROUPING
HAVING
HOLD
HOUR
HOURS
IDENTITY
IN
INDICATOR
INNER
INOUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTERSECTION
INTERVAL
INTO
IS
JOIN
KEEP
LAG
LANGUAGE
LARGE
LAST_VALUE
LATERAL
LEAD
```

LEADING
LEFT
LIKE
LIKE_REGEX
LN
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOWER
MATCH
MAX
MAX_CARDINALITY
MEMBER
MERGE
METHOD
MIN
MINUTE
MINUTES
MOD
MODIFIES
MODULE
MONTH
MULTISET
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NIL
NO
NONE
NORMALIZE
NOT
NTH_VALUE
NTILE
NULL
NULLIF
NUMERIC
OCCURRENCES_REGEX
OCTET_LENGTH
OF
OFFSET
OLD
ON
ONLY
OPEN
OR
ORDER
OUT
OUTER
OVER
OVERLAPS
OVERLAY
PARAMETER
PARTITION
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
POSITION
POSITION_REGEX
POWER
PRECISION
PREPARE
PRIMARY
PROCEDURE
RANGE
RANK
READS
REAL
RECURSIVE
REF
REFERENCES
REFERENCING
REGR_AVGX
REGR_AVGY
REGR_COUNT
REGR_INTERCEPT
REGR_R2
REGR_SLOPE
REGR_SXX
REGR_SXY
REGR_SYY
RELEASE
RESULT
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROLLUP
ROW
ROW_NUMBER
ROWS
SAVEPOINT
SCOPE
SCROLL
SEARCH
SECOND
SECONDS
SELECT
SENSITIVE
SESSION_USER
SET
SIMILAR
SMALLINT
SOME
SPECIFIC
SPECIFICTYPE
SQL
SQLEXCEPTION
SQLSTATE
SQLWARNING
SQRT
START
STATIC
STDDEV_POP
STDDEV_SAMP
SUBMULTISET
SUBSTRING
SUBSTRING_
REGEX
SUM
SYMMETRIC
SYSTEM
SYSTEM_USER
TABLE
TABLESAMPLE
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRAILING
TRANSLATE
TRANSLATE_REGEX
TRANSLATION
TREAT
TRIGGER
TRUNCATE
TRIM
TRIM_ARRAY
TRUE
UESCAPE
UNION
UNIQUE
UNKNOWN
UNNEST
UPDATE
UPPER
USER
USING
VALUE
VALUES
VAR_POP
VAR_SAMP
VARBINARY
VARCHAR
VARYING
VERSION
VERSIONING
VERSIONS
WHEN
WHENEVER
WHERE
WIDTH_BUCKET
WINDOW
WITH
WITHIN
WITHOUT
YEAR
YEARS

Stichwortverzeichnis

E

S

T

U

V

W

X

Y

Z