Datenbankkonzepte in der Praxis

Pearson Education

ADDISON-WESLEY

Sönke Cordts

Datenbankkonzepte in der Praxis

Nach dem Standard SQL-99

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Abbildungen und Texten wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem
und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-1938-2

© 2002 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Hommer Design, Haar bei München
Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Christine Depta, Freising
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Kösel, Kempten
Printed in Germany

Inhaltsverzeichnis

1 Einleitung

Bis Ende der 60er Jahre wurde für Anwendungen das Dateisystem zum Speichern von Daten verwendet. Jede Anwendung eines Unternehmens speicherte in seinen eigenen Dateien die Daten, die es für die Anwendung benötigte. Dies führte zwangsläufig dazu, dass Daten mehrfach gehalten wurden. Jede Anwendung, die z.B. Kundendaten benötigte, speicherte diese in der Regel selbständig in einer eigenen Datei. Dadurch entstand ein regelrechtes »Datenchaos«, ein spezifischer Kunde des Unternehmens wurde mehrmals mit der gleichen Adresse in den Dateien gespeichert. Bei einer Änderung der Adresse mussten alle Dateien gleichzeitig angepasst werden.

Aufgrund dieser Unzulänglichkeit und des damit hervorgerufenen »Datenchaos«entstanden Ende der 60er Jahre die ersten Datenbanksysteme. Diese sollten zentral alle Daten eines Unternehmens in einer Datenbank speichern. Benötigte eine Anwendung nun Kundendaten, so musste sie sich an das Datenbanksystem wenden, um diese zu erhalten. Die Verwaltung der Daten wurde also dem Datenbanksystem übertragen, genau so wie eine Bank das Geld ihrer Kunden verwaltet.

Diese Datenbanksysteme (hierarchische und Netzwerk-Datenbanksysteme) hatten jedoch einen entscheidenden Nachteil, sie waren aufgrund ihrer Struktur in der Abfrage der Daten eingeschränkt. Eine Änderung dieser Struktur erforderte in der Regel einen erneuten Aufwand in der Programmierung. Noch heute sind, gerade im Großrechnerbereich, hierarchische und auch Netzwerk-Datenbanken im Einsatz. Das bekannteste hierarchische Datenbanksystem ist IMS von der Firma IBM.

1970 konnten Mitglieder des ACM (»Association for Computing Machinery«) im Datenbankjournal einen Artikel von Dr. E. F. Ted Codd mit dem Titel »A Relational Model of Data for Large Shared Data Banks«lesen. Dieser Artikel war die Grundlage für die Entwicklung der heute am häufigsten eingesetzten Datenbanksysteme (weltweit etwa 75%), den relationalen Datenbanksystemen. Das relationale Datenbankmodell hatte einen entscheidenden Vorteil: Änderungen an der Struktur der Datenbank bewirkten nicht zwangläufig auch Modifizierungen an den Anwendungsprogrammen.

Auf Basis des Artikels von Codd entwickelte die Firma IBM bis 1974 einen Prototypen eines relationalen Datenbanksystems, System/R. Um die Daten des Datenbanksystems abzufragen, wurde eine Datenbankabfragesprache entwickelt, die man SEQUEL nannte. Diese Sprache war der Vorläufer des heutigen SQL (Structured Query Language), das 1999 von den Normungsgremien des »American National Standard Institute« (ANSI) in einer dritten Fassung standardisiert wurde. Diese Fassung wird allgemein hin als SQL:1999 bzw. SQL-3 bezeichnet. SQL wird heute von allen gängigen relationalen Datenbankprodukten in verschiedenen Stufen unterstützt.

Die wesentlichen Aufgaben eines Datenbanksystems bestehen in der Verwaltung und Strukturierung von Daten. Jedoch gerade die zweite Aufgabe, die Strukturierung der

Daten, kann in der Regel nicht durch einen Computer vorgenommen werden, da es sich um eine analytische Aufgabe handelt. Die Strukturierung der Daten, also das Entwerfen eines Datenbankmodells, ist deshalb auch vielmehr eine Aufgabe, die ein Analytiker mit Papier und Stift vornimmt. Genau so wie ein Architekt einen »Bauplan« entwickeln muss, bevor er mit dem Bau eines Gebäudes beginnt, muss auch der Architekt einer Datenbank einen »Bauplan« entwickeln, der die Struktur der zukünftigen Datenbank widerspiegelt.

Durch die Verbreitung des Internets seit Anfang der 90er Jahre, die gekennzeichnet ist durch die Begriffe des »Datenchaos« bzw. der »Informationsflut«, kommt gerade der Strukturierung von Daten heute eine wichtige Rolle zu.

Neben dem Internet waren die Anfänge der 90er Jahre geprägt durch die damals neuen Paradigmen der objektorientierten Softwareentwicklung. Diese Entwicklung ist gekennzeichnet durch Begriffe wie Komponenten, »Business Objects« (BO), Mehrschichtenmodell etc. Der erste SQL-Standard von 1986 (SQL-1) berücksichtigt diese Programmierparadigmen nicht.

Vielmehr war SQL in der Anfangsphase u. a. auch als Abfragesprache für Endanwender gedacht, die durch Eingabe von SQL-Befehlen die gewünschten Daten gesucht haben. Der Zeit entsprechend war diese Vorgehensweise durchaus benutzerfreundlich. Mit dem Aufkommen grafischer Benutzeroberflächen wurden Anwender entsprechend anspruchsvoller. Heutzutage wird ein Anwender es sicher zu Recht als Zumutung empfinden, SQL-Befehle eingeben zu müssen, um z. B. die Rechnungsdaten eines Kunden zu erhalten.

Daher wird SQL heute vorwiegend in der Softwareentwicklung eingesetzt. Dem wird entsprechend den neuen Paradigmen in der Softwareentwicklung im zweiten und im dritten Standard von 1992 und 1999 (SQL-2 und SQL-3) Rechnung getragen. Hier wird die Leichtigkeit der Verwendung von SQL eher aus Entwicklersicht, denn aus Anwendersicht betrachtet.

Das vorliegende Buch hat das Ziel, einen praxisorientierten Ansatz zur Beantwortung der folgenden Fragen zu geben:

- Wie komme ich von der Realität zu einer Datenbank?
- Welche grundlegenden Sprachelemente enthält SQL?
- Welche neuen Konzepte von SQL werden heutzutage in der Anwendungsentwicklung verwendet?

Der Inhalt dieses Buches baut auf Vorlesungen auf, die ich an der Fachhochschule Westküste im Fachbereich Betriebswirtschaftslehre, Schwerpunkt Wirtschaftsinformatik, über Datenbanken gehalten habe.

Dementsprechend wendet es sich primär an Personen, die in der Praxis ein Datenbankmodell entwerfen sollen, aber auch an solche, die sich innerhalb der Anwendungsentwicklung mit SQL beschäftigen und hierfür eine solide Einführung benötigen.

Gerade das Thema der Anwendungsentwicklung soll hierbei ein Schwerpunkt bilden, da viele Erweiterungen im SQL-Standard hinzugekommen sind, die Programmierparadigmen wie »Business Objects« und Mehrschichtenmodell unterstützen. Dabei

gehe ich vorwiegend auf die SQL-Bestandteile »Stored Procedures«, »Trigger« und benutzerdefinierte Datentypen bzw. UDT (»User Defined Types«) ein. Gerade diese Bestandteile erscheinen mir sehr wichtig, da damit eine grundlegende Änderung der Programmierung von Datenbanken einhergeht und dieses Thema in der bisherigen Datenbank-Literatur nur »stiefmütterlich« oder gar nicht behandelt wird.

Da SQL-3 und auch SQL-2 nicht vollständig von den heutigen Datenbankprodukten unterstützt werden, wird nur auf solche Sprachelemente eingegangen, die in den gängigen relationalen Datenbankprodukten (Oracle, Informix, IBM DB/2, Sybase, Microsoft SQL Server, Borland InterbaseBase) enthalten sind.

Dieses Buch gliedert sich damit in folgende Kapitel:

- Kapitel 1 gibt eine kurze Einleitung und eine Übersicht über das vorliegende Buch.
- Kapitel 2 behandelt die Probleme, die ein Datenbanksystem beseitigen soll und gibt einen ersten Überblick über die Schritte, die von einem realen Problem zu einer Datenbank führen.
- Kapitel 3 und 4 gehen auf die Datenmodellierung des realen Problems ein. Hierbei werden das Entity-Relationship-Modell (ERM) und das Relationenmodell behandelt. In Kapitel 3 wird auch kurz auf unterschiedliche Notationen des Entity-Relationship-Modells und auf die Verwendung der »Unified Modeling Language« (UML) zur Datenmodellierung eingegangen.
- Kapitel 5 behandelt die Überprüfung bzw. Optimierung der Datenbankstruktur mit Hilfe der sogenannten Normalformen.
- Kapitel 6 gibt eine Einführung in SQL und beschreibt, wie man eine Datenbankstruktur anlegt bzw. wieder ändert und löscht.
- Kapitel 7 geht auf das Einfügen, Ändern und Löschen von Daten in SQL ein.
- Kapitel 8 behandelt einfache Abfragen in SQL.
- Kapitel 9 beschäftigt sich mit Abfragen, bezogen auf mehrere Tabellen.
- Kapitel 10 erläutert Abfragen innerhalb von Abfragen.
- Kapitel 11 beschäftigt sich mit Transaktionen.
- Die Kapitel 12 bis 14 behandeln die Sprachkonstrukte für benutzerdefinierte Datentypen (UDT), »Stored Procedures« und »Trigger«.

Zur Veranschaulichung der angesprochenen Themen wird den Leser das gesamte Buch hindurch eine fiktive Firma begleiten, die alle Phasen vom »Datenchaos« bis zur fertigen Datenbank durchläuft.

Jedes Kapitel beginnt mit Fragen, die anschließend beantwortet werden und endet mit einer Zusammenfassung sowie problemspezifischen Aufgaben.

Zur praktischen Durchführung der Aufgaben ab dem 6. Kapitel empfehle ich die Nutzung von Testversionen der Datenbanksoftware, die von allen gängigen Datenbankherstellern im Internet zum Download bereitgestellt werden und die in der Regel bis zu einem bestimmten Datum lauffähig sind (die Internetadressen hierzu finden Sie im Internet unter »*http://www.addison-wesley.de/DBMS*«).

Das Datenbanksystem mySQL, das in einer Version für Linux kostenlos erhältlich ist, kann ich nur eingeschränkt zur Lösung der Aufgaben empfehlen, da wesentliche Elemente eines relationalen Datenbanksystems fehlen (Transaktionen nur bedingt, Unterabfragen, »Stored Procedures« etc.).

Falls Sie noch Fragen, Anregungen oder Kommentare haben, können Sie mich unter folgender E-Mail-Adresse erreichen: *soenke.cordts@addison-wesley.de*.

2 Grundlagen von Datenbanken

In Kapitel 2 sollen folgende Fragen geklärt werden:

- Worin bestehen die Probleme herkömmlicher Datenspeicherung?
- Warum sollte ich mich mit Datenbanken beschäftigen?
- Wie komme ich von der Realität zu einem »Bauplan«für meine Datenbank?
- Wie setze ich den »Bauplan«so um, dass ich Daten in meiner Datenbank speichern und wiederfinden kann?

2.1 Motivation

Um die Probleme herkömmlicher Datenspeicherung zu verstehen, betrachten wir zunächst unsere fiktive Firma »KartoFinale«, deren Hauptgeschäftszweck im Verkauf und in der Abrechnung von Eintrittskarten besteht.

In der Firma »KartoFinale«arbeiten verschiedene Personen. Neben dem Geschäftsführer, Herrn Kowalski, existieren die Abteilungen Rechnungswesen, Vertrieb, Einkauf und Technik.

Zum Vertrieb gehören Frau Klug, Herr Klein, die Studenten Inga und Klaus, die für den Verkauf der Eintrittskarten zuständig sind und als Abteilungsleiterin Frau Kart.

Im Rechnungswesen ist Herr Wunder für das Schreiben der Rechnungen, die Buchhaltung usw. eigenverantwortlich zuständig. Schließlich gibt es noch Herrn Münze, der für die Personalabrechung verantwortlich ist.

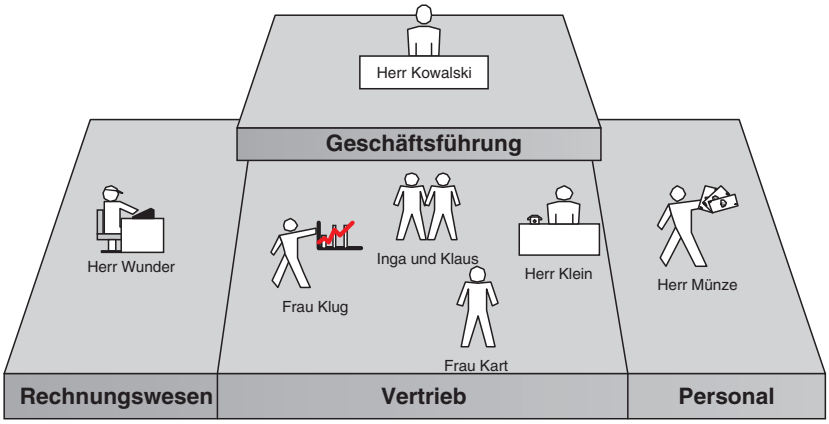


Abbildung 2.1: Das Unternehmen »KartoFinale«

Da Eintrittskarten telefonisch verkauft und versendet werden, muss Herr Wunder jeden Tag Rechnungen schreiben. Dazu verwendet er ein Textverarbeitungsprogramm. Um sich die Arbeit zu erleichtern, hat er sich eine Textdatei angelegt, in der er die Adressen der Kunden speichert.

In der Abteilung »Vertrieb« hat Herr Klein von seiner Vorgesetzten, Frau Kart, den Auftrag bekommen, die Kunden per Postversand über die aktuellen Veranstaltungen zu benachrichtigen. Kunden mit Kindern soll ein besonderer Rabatt gewährt werden, dementsprechend sollen Kunden mit Kindern natürlich einen anderen Werbebrief erhalten.

Da Herr Klein weiß, dass Herr Wunder die Kundendaten in einer Textdatei gespeichert hat, holt er sich diese von Herrn Klein und liest sie in sein Tabellenkalkulationsprogramm ein. Danach ergänzt er die Tabelle um ein Feld für die Namen der Kinder.

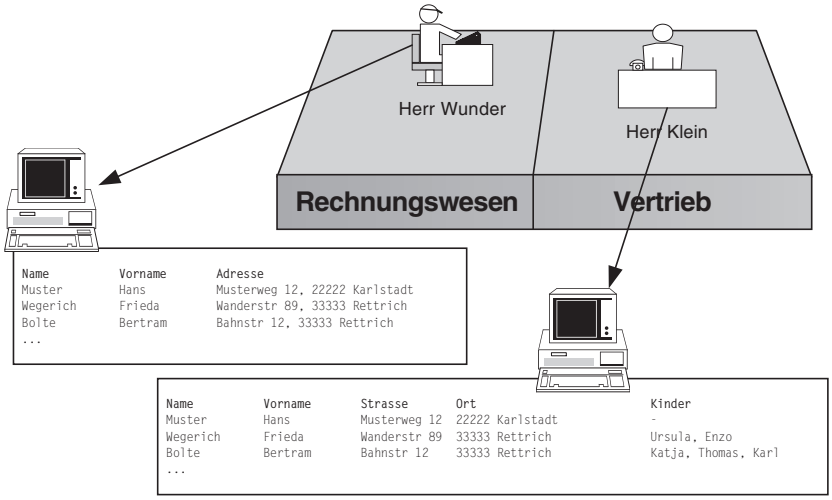


Abbildung 2.2: Kundendaten mehrfach dezentral gespeichert

Inzwischen sind drei Tage vergangen und bei Herrn Wunder klingelt das Telefon. Herr Bolte, ein wichtiger Kunde von »KartoFinale« teilt Herrn Wunder mit, dass sich seine Adresse geändert hat. Herr Wunder ändert die Adressdaten, vergisst aber, Herrn Klein davon zu benachrichtigen.

Ebenso stellt Herr Klein bei der Durchsicht seiner Adressdaten fest, dass der Name von Frau Wegerich falsch geschrieben ist, entsprechend korrigiert er den Namen in Wiegerich ohne seinerseits Herrn Wunder davon zu benachrichtigen.

Mit der Zeit treten mehrere solcher Fälle auf: Kunden werden neu in die Adressdateien aufgenommen bzw. gelöscht, Adressänderungen finden statt, ohne dass diese zwischen den beiden Abteilungen abgeglichen werden.

Nach zwei Wochen beauftragt Frau Kart Herrn Klein erneut, eine Werbebriefkampagne zu starten. Da Herr Klein seine eigenen Adressdaten hat, verwendet er diese, um die

Briefe zu versenden. Diesmal kommen jedoch Briefe zurück, da Adressen nicht mehr stimmen. Neue Kunden, die ausschließlich in der Abteilung »Rechnungswesen« in die Adressdatei aufgenommen wurden, werden gar nicht benachrichtigt.

Um dieses Problem zu lösen, wird in einem Meeting zwischen Frau Kart, Herrn Klein und Herrn Wunder entschieden, dass nur noch eine Adressdatei verwendet werden soll. Frau Kart beauftragt Herrn Klein, die beiden Dateien abzugleichen, so dass nur noch eine Datei verwendet wird. Neben den bisherigen Adressdaten sollen zusätzlich auch die Rechnungsdaten in dieser Datei hinterlegt werden.

Nach zwei Tagen hat Herr Klein die Unstimmigkeiten beseitigt und legt die Datei auf einem Computer ab, auf den sowohl Herr Klein, als auch Herr Wunder über das firmeninterne Netzwerk zugreifen können.

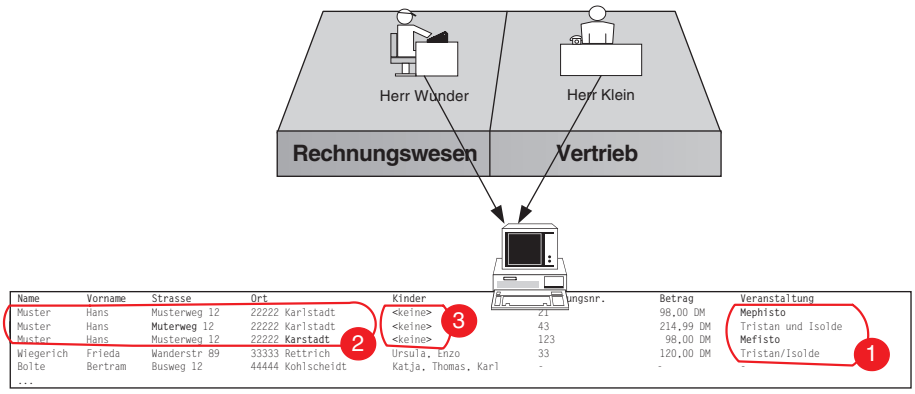


Abbildung 2.3: Kundendaten zentral gespeichert

Dabei sind Herrn Klein einige Fehler unterlaufen, so hat er gleiche Veranstaltungen unterschiedlich bezeichnet (1). Pro Rechnung hat Herr Klein eine Zeile in der Datei vorgesehen. Hat ein Kunde mehrere Rechnungen erhalten, so wie Herr Muster, so muss die Adresse in jeder Zeile wiederholt werden. Auch hierbei sind Herrn Klein Fehler unterlaufen (2).

Am nächsten Tag schreibt Herr Wunder eine Rechnung für Frau Wiegerich. Dazu öffnet er die neue Adressdatei. Zur gleichen Zeit öffnet auch Herr Klein die Datei, da er die Daten noch einmal überprüfen und ggf. korrigieren möchte.

Nachdem Herr Wunder die Rechnungsdaten für Frau Wiegerich eingegeben hat, speichert er die Datei. Herr Klein bekommt davon nichts mit, da er die Adressdatei ja vorher schon geöffnet hatte. Also korrigiert er die Fehler und speichert die Datei danach. Dabei überschreibt er unwissentlich die von Herrn Wunder neu eingegebenen Rechnungsdaten.

Herr Klein hat sich in der Zwischenzeit mit der Serienbrieffunktionalität seines Textverarbeitungsprogrammes auseinandergesetzt. Damit kann er nun Serienbriefe an Adressaten ohne Kinder erstellen, abhängig davon, ob im Feld zur Angabe der Kinder der Wert »<keine>« steht (3). Inzwischen ist jedoch Herr Wunder auf die Idee gekom-

men, für das Nichtvorhandensein von Kindern einen Bindestrich zu verwenden. Der Serienbrief von Herrn Klein für seine Werbekampagne wird damit nicht mehr korrekt erstellt.

Frau Kart möchte alle Mitarbeiter darüber benachrichtigen, dass auf dem zentralen Computer die Adressen der Kunden gespeichert sind. Frau Kart weiß, dass Herr Münze von der Personalabteilung alle Adressen der Mitarbeiter in einer Datei gespeichert hat. Sie bittet Herrn Münze deshalb, seine Datei auf dem zentralen Computer abzulegen, damit sowohl Herr Münze als auch andere Mitarbeiter auf die Mitarbeiteradressen zugreifen können. Da jedoch auch das Gehalt der jeweiligen Mitarbeiter in dieser Datei steht, kann Herr Münze der Bitte von Frau Kart nicht nachkommen.

Fassen wir noch einmal die Probleme zusammen, die im obigen Szenario aufgetreten sind:

- Daten werden »mehrfach« dezentral gespeichert (Datenredundanz).
- Dadurch »laufen« die Daten auseinander und sind nicht mehr konsistent (Datenintegrität).
- Auch bei zentraler Speicherung werden Daten trotzdem noch »mehrfach« gespeichert (Datenredundanz),
  Beispiel: pro Rechnung werden die Adressdaten zusätzlich gespeichert.
- Änderungen des Datenformates bzw. der Semantik der Daten bewirken, dass bestimmte Programme nicht mehr korrekt funktionieren (Datenabhängigkeit),
  Beispiel: Serienbrief.
- Bei gleichzeitigem Zugriff besteht die Gefahr des gegenseitigen Überschreibens von Daten (Mehrbenutzerbetrieb, Synchronisation).
- Unterschiedliche Sichten auf Daten (Datensicht),
  Beispiel: Abteilung Personal und Marketing.
- Nicht jeder darf alle Daten sehen (Datensicherheit),
  Beispiel: Gehaltsdaten.

Unser erstes Problem war relativ einfach zu beseitigen: Wir mussten unsere Daten zentral für alle Mitarbeiter zugreifbar machen.

Alle Probleme, die danach auftraten, bestanden in der unzureichenden Kontrollaufsicht unserer Datei (Um alle Aufgaben korrekt zu erledigen, müsste in unserem Beispiel ein einziger Mitarbeiter alleine für die Datei zuständig sein, dort Änderungen vornehmen, Auskünfte geben, Rechnungen schreiben usw.)

Es ist deshalb notwendig, unsere Daten unter die »Obhut« eines eigenen Softwareprogrammes zu geben, das die Daten verwaltet.

Diese Software wird im Allgemeinen als Datenbank-Management-System (DBMS) bezeichnet. Die Aufgaben des DBMS bestehen in der Beseitigung der oben aufgeführten Probleme.

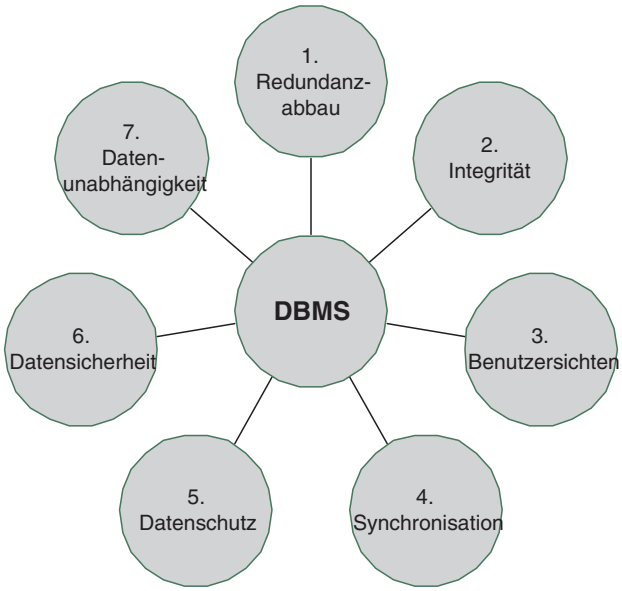


Abbildung 2.4: Aufgaben eines Datenbank-Management-Systems

Unter einem Datenbank-Management-System (DBMS) versteht man also eine Software zur Verwaltung von Datenbanken. Eine Datenbank (DB) wiederum stellt einen zusammengehörigen Datenbestand dar (z.B. alle Daten der Abteilung Personal oder alle Adressdaten eines Unternehmens). Alle Datenbanken zusammen bilden die gesamte Datenbasis eines Datenbank-Management-Systems.

Datenbank-Management-System und Datenbanken bezeichnet man entsprechend als Datenbanksystem (DBS).

Datenbank-Management-Systeme sind heutzutage allgegenwärtig, auch wenn man als Anwender nicht unmittelbar mit ihnen in Berührung kommt. Heben Sie an einem Bankterminal Geld ab, so steht dahinter immer ein Datenbank-Management-System, das die Kundendaten abruft, Überprüfungen vornimmt und den neuen Kontostand speichert. Egal, ob Geld am Bankterminal abgehoben wird oder ob per Online-Banking Geld bei einer Bank überwiesen wird, ein DBMS nimmt diese Transaktion vor und steuert den Ablauf der Daten.

Ob Dienstleistungen durchgeführt oder Produkte gefertigt werden, in der Regel werden die dabei anfallenden Daten in Datenbanken gespeichert und durch das DBMS verwaltet.

1998 umfasste der Umsatz mit Datenbank-Management-Systemen 7,1 Mrd. US $ (vgl. hierzu [Panny99]). Diese Summe umfasst den reinen Verkaufsumsatz von Datenbank-Management-Systemen. Darin sind nicht enthalten: Betrieb des DBMS, Beratung, Programmierung usw. Diese Zahl macht deutlich, welche zentrale Rolle Datenbank-Management-Systeme gerade im kommerziellen Bereich der Informationstechnologie spielen.

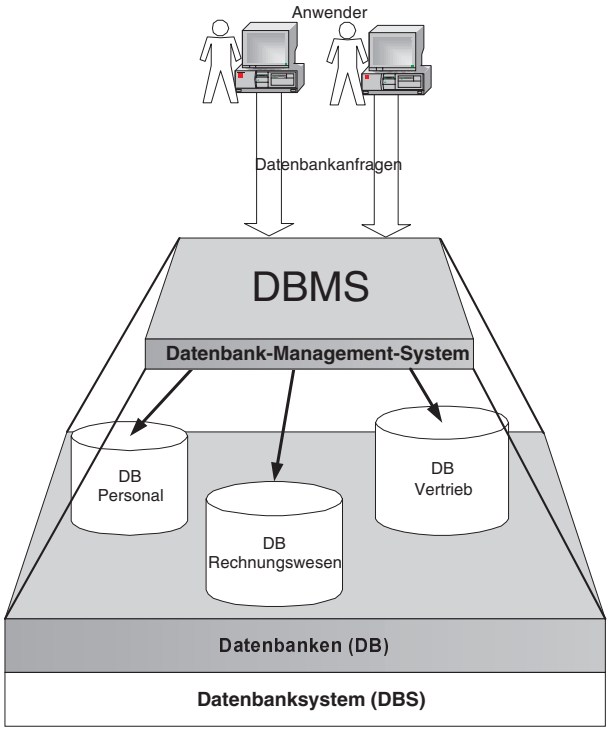


Abbildung 2.5: DBMS, DB und DBS

2.2 Von der Realität zum »Bauplan«

Ein Architekt erstellt vor dem Bau eines Gebäudes in der Regel einen Projektplan, in dem er festlegt, welche Tätigkeiten in welcher Reihenfolge zu erfolgen haben. Kernpunkt des Projektplanes ist die Bestimmung des eigentlichen Zwecks des Gebäudes.

Ebenso wie der Architekt muss auch der Datenbank-Designer gemeinsam mit dem Kunden zuerst den Zweck der zu erstellenden Datenbank und zukünftiger Anwendungssoftware ermitteln, denn die Zweckbestimmung legt letztendlich fest, welche Daten als relevant anzusehen sind. So ist es sicherlich möglich, in einer Datenbank die Lieblingsspeise eines Kunden zu speichern. Diese Information ist für das eigentliche Kerngeschäft eines Unternehmens jedoch in der Regel nicht relevant, es sei denn, es handelt sich um ein Feinschmecker-Restaurant o.ä.

Neben der Zweckbestimmung erstellt der Datenbankarchitekt einen Projektplan, in dem er festhält, was, wann, in welcher Reihenfolge getan, überprüft und eventuell noch einmal überarbeitet werden muss (Phasenkonzept).

Nach dem Projektplan benötigt der Architekt einen »Bauplan« für das Gebäude. Dieser »Bauplan« dient zum einen als Diskussionsgrundlage mit dem Kunden, zum anderen

als Basis zum Bau des Gebäudes. Der Architekt erstellt deshalb einen Grundriss und die jeweiligen Außenansichten. Der Kunde kann dann entsprechend Änderungswünsche äußern, die von dem Architekten wieder übernommen werden, bis man sich schließlich auf einen gemeinsamen Bauplan einigt.

Ein Datenbank-Designer steht vor dem gleichen Problem. Er muss dem Kunden leicht verständlich einen »Bauplan« seiner späteren Datenbankstruktur vorstellen. Im Gegensatz zum Architekten, der immer wieder die gleiche Tätigkeit durchzuführen hat, nämlich den Bau eines Gebäudes, muss sich ein Datenbank-Designer immer wieder fachlich in die verschiedensten Tätigkeiten eines Unternehmens einarbeiten, für das eine Datenbank erstellt werden soll. Bevor er also einen »Bauplan« entwirft, führt er Gespräche mit den zukünftigen Endanwendern, sichtet Dokumente des Unternehmens, ermittelt Informationen anhand von Fragebögen u.ä.

Ziel des Datenbank-Designers in dieser ersten Phase ist es also, das Geschäftsumfeld des Kunden mit den jeweiligen Funktionen und anfallenden Daten kennenzulernen.

Danach kann er anfangen, einen ersten »Bauplan« für das zukünftige System zu erstellen. Er bildet damit die Realität in einem sogenannten Datenmodell ab. Dazu identifiziert er die Daten der relevanten Geschäftsobjekte, die für das Geschäftsumfeld relevant sind, und deren Beziehungen untereinander. Nehmen wir z.B. das einfache Geschäftsumfeld »Kunde kauft Auto«: Hier stellen »Kunde« und »Auto« Geschäftsobjekte dar und »kauft« beschreibt die Beziehung zwischen diesen beiden Geschäftsobjekten.

Dieser erste Entwurf eines Datenmodells dient dem Datenbank-Designer nun als Grundlage zur Diskussion mit dem Kunden, um fachliche Verständnisprobleme oder fehlende Daten bzw. Beziehungen zu ermitteln. In Zusammenarbeit mit dem Kunden wird dieses Datenmodell für alle relevanten Geschäftsobjekte weiter verfeinert. Der Datenbank-Designer entwirft dieses Datenmodell auf einem Blatt Papier. Dabei könnte der Datenbank-Designer eigene grafische Symbole verwenden, die er dem Kunden erläutert, bevor über das Datenmodell diskutiert wird, genau so wie ein Bauherr die Symbole auf einem Grundriss verstehen muss.

Beim Datenbankentwurf hat sich jedoch eine bestimmte grafische Notation bzw. Modellierung etabliert: das Entity-Relationship-Modell (ERM). Das ERM wird heutzutage in der Regel für den Entwurf eines Datenmodells verwendet und hat sich in der Praxis durchgesetzt, da es ohne viel Einarbeitungsaufwand relativ schnell verstanden wird. Gerade dieser Punkt ist wichtig, um sich bei den Gesprächen mit dem Kunden auf die fachspezifischen Inhalte konzentrieren zu können und Missverständnisse bei den grafischen Notationen zu vermeiden.

So ist die aufwendige Notation der Grund, weshalb sich die Modellierungssprache »Unified Modeling Language« (UML) bisher noch nicht in der Modellierung von Datenbankentwürfen als Grundlage für Kundengespräche durchgesetzt hat.

Im Gegensatz zur UML hat das ERM wiederum den Nachteil, dass es nicht standardisiert wurde. Dadurch existieren zur Zeit verschiedene »Dialekte«. Auf das ERM und die verschiedenen »Dialekte« (Chen, Barker) und ansatzweise auf die UML wird in Kapitel 3 ausführlich eingegangen.

Das so im ersten Schritt erstellte Datenmodell, auch konzeptionelles Modell genannt (siehe Abb. 2.6), muss also so einfach sein, dass es als Grundlage der Gespräche mit dem Kunden dienen kann. Zum anderen muss es aber auch detailliert genug sein, damit es vom Datenbank-Designer physisch in Datenbankstrukturen umgewandelt werden kann.

2.3 Vom »Bauplan« zur Datenbank

Das konzeptionelle Datenmodell, das zur Diskussion mit dem Kunden dient, ist unabhängig von technischen Implementierungen. Es dient dazu, zu beschreiben, was mit welchen Daten später auf dem Computer umgesetzt werden soll.

So wie ein Architekt beim Bau eines Gebäudes für die Handwerker einen eigenen Plan erstellt, muss auch der Datenbank-Designer einen eigenen Plan zur Umsetzung des konzeptionellen Datenmodells erstellen. Nach dem Entwurf des konzeptionellen Datenmodells muss also ein weiterer »Bauplan«, das so genannte logische Datenmodell, erstellt werden, das die Umsetzung des konzeptionellen Modells in eine physische Datenstruktur unterstützt. Das logische Datenmodell legt fest, wie die im konzeptionellen Modell definierten Geschäftsobjekte und Beziehungen elektronisch abgebildet werden können. In relationalen Datenbanksystemen erfolgt die Speicherung von Geschäftsobjekten und Beziehungen in Form von Tabellen. Eine Zeile innerhalb der Tabelle stellt dabei ein reales Objekt, eine Spalte die Eigenschaft des realen Objektes dar. Man spricht bei diesem logischen Datenmodell dementsprechend vom Relationenmodell. Nachdem dieses entwickelt wurde, kann mit Hilfe der relationalen Datenbanksprache SQL die Struktur der Datenbank physikalisch auf dem Computer erstellt werden (siehe Abb. 2.6).

Der Entwurf des logischen Datenmodells auf Basis des Relationenmodells wird ausführlich in Kapitel 4 beschrieben, die Erstellung der Datenbankstruktur mit SQL in Kapitel 9.

Da Benutzer unterschiedliche Sichten auf die Daten haben, wird im nächsten Schritt das so genannte externe Datenmodell festgelegt. Hierbei wird zum einen bestimmt, welche Benutzer bzw. welche Benutzergruppen, welche Berechtigungen auf bestimmte Informationsinhalte erhalten. Zum anderen werden Sichten auf die Daten erzeugt in Abhängigkeit von der Funktion des jeweiligen Anwendungsprogramms.

Schließlich wird in einem physischen Datenmodell festgelegt, welche Zugriffsmechanismen und Speicherstrukturen zur Effizienzsteigerung verwendet werden können. Da das physische Datenmodell stark abhängig ist vom jeweils verwendeten RDBMS, wird hier nicht weiter darauf eingegangen. Hierzu sollte man die Handbücher der entsprechenden Datenbankprodukte zu Rate ziehen bzw. Datenbankbücher, die auf Speicherstrukturen und Zugriffsmechanismen detailliert eingehen (siehe hierzu die Literaturliste).

2.4 Zusammenfassung

In diesem Kapitel haben wir zunächst die Probleme kennen gelernt, die auftreten, sofern man Daten über ein Dateisystem speichert. Die Gründe für diese Probleme liegen vorwiegend in der mangelnden Verwaltung der Daten durch eine übergeordnete Instanz. Diese übergeordnete Instanz stellt ein so genanntes Datenbank-Management-System (DBMS) dar, das die Daten unter seiner »Obhut« verwaltet. Genau wie der Manager einer Firma plant und steuert, kontrolliert und organisiert das DBMS die Daten. Hierdurch können Redundanzen vermieden werden, die Integrität, Sicherheit und der Schutz der Daten sichergestellt werden.

Ein Problem kann jedoch nicht durch ein DBMS beseitigt werden: Redundanzen aufgrund der Struktur, in der die Daten gespeichert wurden.

Probleme dieser Art lassen sich durch eine analytische Tätigkeit, den Datenbankentwurf, vermeiden. Der Datenbankentwurf, also die Erstellung eines »Bauplanes« aufgrund der Realität, erfolgt in mehreren Schritten. Zunächst wird festgelegt, zu welchem Zweck eine Datenbank entworfen werden soll. Danach wird in Absprache mit dem Kunden ein konzeptionelles Datenmodell erstellt.

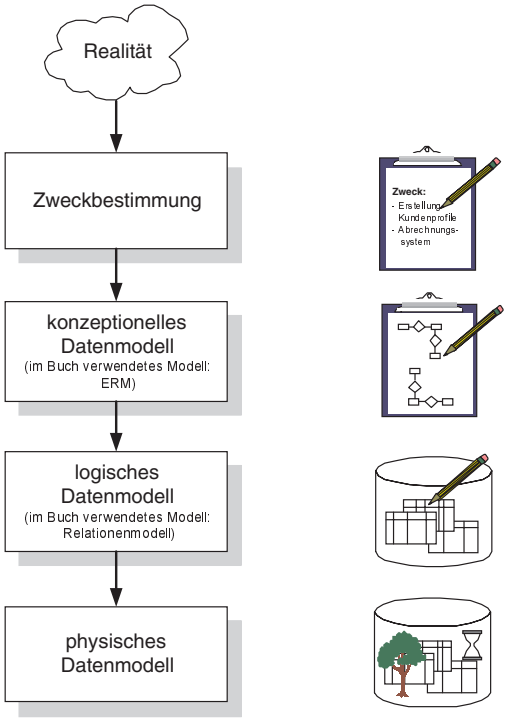


Abbildung 2.6: Schritte zur Erstellung einer Datenbank

Nachdem das konzeptionelle Datenmodell so weit fortgeschritten ist, dass es dem Geschäftsumfeld des Kunden entspricht, entwickelt der Datenbank-Designer das logische Datenmodell. Dieses dient dem Datenbank-Designer als Grundlage zur Erstellung einer physischen Datenstruktur. Bei dem Entwurf einer relationalen Datenbank wird hier entsprechend das Relationenmodell verwendet.

Schließlich wird das Relationenmodell physikalisch mit der Datenbanksprache SQL in eine physische Datenbankstruktur umgesetzt.

Im folgenden Kapitel wollen wir uns mit Phasenkonzepten und der Erstellung des konzeptionellen Datenmodells beschäftigen.

2.5 Aufgaben

Wiederholungsfragen

1) Was versteht man unter einem Datenbank-Management-System (DBMS)?
2) Was ist ein Datenbanksystem (DBS)?
3) Was ist eine Datenbank (DB)?
4) Welche Aufgaben erfüllt ein DBMS?
5) Was versteht man unter »Redundanz von Daten«?
6) Was versteht man unter »Datenunabhängigkeit«?
7) Was ist mit »Datenintegrität« gemeint?
8) Worin besteht die Hauptaufgabe des Datenbank-Designers?
9) Welche Datenmodellierungstechniken gibt es?

Übungen

1) Herr Klein von der Firma »KartoFinale« hat seine Datei mit den Kundendaten noch einmal geändert. Um nicht für jede Rechnung die gesamten Kundendaten zu wiederholen, hat er zwei Dateien erstellt, eine Datei für die Kundendaten und eine weitere Datei mit den Rechnungsdaten. Die Dateien sehen wie in der folgenden Abbildung aus.

   Um die Rechnung dem jeweiligen Kunden zuzuordnen, hat Herr Klein den Vor- und Nachnamen mit in die Rechnungsdatei übernommen. Über den Vor- und Nachnamen kann er in der Kundendatei dann die vollständige Adresse des Kunden ermitteln.

   Beschreiben Sie mögliche Probleme dieser Lösung!

Kundendatei

```
Name          Vorname      Strasse        Ort                    Kinder
Muster        Hans         Musterweg 12 22222 Karlstadt        <keine>
Wiegerich     Frieda       Wanderstr 89 33333 Rettrich         Ursula, Enzo
Bolte         Bertram      Busweg 12    44444 Kohlscheidt      Katja, Thomas, Karl
...
```

Rechnungsdatei

```
Name          Vorname      Rechnungsnr. Betrag        Veranstaltung
Muster        Hans         21             98,00 DM    Mephisto

Muster        Hans         43           214,99 DM     Tristan und Isolde
Muster        Hans         123            98,00 DM    Mephisto
Wiegerich     Frieda       33           120,00 DM     Tristan und Isolde
...
```

3 Datenbankentwurf – Von der Realität zum »Bauplan«

In Kapitel 3 sollen folgende Fragen geklärt werden:

- Wie sieht ein Projektplan (Phasenkonzept) zum Entwurf einer Datenbank bzw. eines Anwendungssystems aus?
- Wie werden die Anforderungen an ein zu entwickelndes Anwendungssystem bzw. die Datenbankstruktur ermittelt?
- Wie werden die ermittelten Anforderungen formal dargestellt (»Bauplan«), um mit dem Kunden über den Entwurf zu diskutieren?
- Welche formalen Entwurfsmethoden haben sich in der Praxis etabliert?
- Welche formalen Entwurfsmethoden werden zukünftig in der Praxis eingesetzt?

3.1 Motivation

Als Einstieg betrachten wir wieder den Verkauf von Eintrittskarten bei der Firma »KartoFinale« Wir haben im zweiten Kapitel erfahren, dass durch das Speichern der Kunden- und Rechnungsdaten zentral in einer einzigen Datei verschiedene Probleme auftraten und diese nur unzureichend gelöst werden konnten. Als vorbildliche Abteilungsleiterin trägt Frau Kart deshalb dieses Problem bei der Geschäftsführung vor. Der Geschäftsführer, Herr Kowalski, lässt sich die Probleme erklären und kommt zu dem Entschluss eine Unternehmensberatung zu beauftragen, ein eigenes Abrechnungssystem für »KartoFinale«zu entwickeln. Sein Nachbar, Herr Warner, ist zufälligerweise Geschäftsführer der Unternehmensberatung »Software Consult AG«, die sich auf die Analyse und Konzeption von Anwendungssystemen für den Dienstleistungsbereich spezialisiert hat.

Also beauftragt Herr Kowalski seinen Nachbarn, Herrn Warner, zur Erstellung eines Konzeptes für ein EDV-gestütztes Abrechnungssystem. Herr Warner, der sich Herrn Kowalski als Nachbar verpflichtet fühlt, setzt seinen besten Mitarbeiter, Herrn Dr. Fleissig für diese Aufgabe ein.

Vor einem ersten Gespräch mit Herrn Kowalski, lässt sich Herr Fleissig eine Broschüre und andere Informationen über die Firma »KartoFinale« zusenden, um einen ersten Eindruck von den Geschäftsprozessen der Firma zu erhalten. Er erstellt einen ersten Fragenkatalog, den er am nächsten Tag mit Herrn Kowalski klären will. Dabei unterscheidet er zwischen fachlichen und rein projektspezifischen Fragen.

Fachliche Fragen sind u.a.:

- Wie sieht der Ablauf des Verkaufes von Eintrittskarten aus?
- Welche Kundengruppen existieren beim Verkauf?

- Wird nur an Privatkunden verkauft?
- Werden auch Abonnementkarten verkauft?
- Wie erfolgt die Bezahlung?
- Aus welchen Abteilungen besteht die Firma »KartoFinale«? u.a.

Projektspezifische Fragen betreffen dagegen den Ablauf der Erstellung des EDV-gestützten Abrechnungssystems.

- Gibt es einen vorgegebenen Einsatztermin für das Abrechnungssystem?
- Welcher Mitarbeiter von »KartoFinale« ist Ansprechpartner in organisatorischen Fragen?
- Bis wann soll das Konzept für das Abrechnungssystem spätestens fertiggestellt sein? u.a.

Nach dem Gespräch am nächsten Tag mit Herrn Kowalski hat Herr Fleissig einen ersten Überblick über die Geschäftstätigkeiten von »KartoFinale« erhalten. Um einzelne Geschäftsprozesse detailliert zu verstehen, vereinbart er Gespräche mit Frau Kart als Vertriebsleiterin und Herrn Wunder als Leiter des Rechnungswesens. Vor den Gesprächen erhält er von Herrn Kowalski noch Formulare, die bei der Abwicklung der Verkaufes verwendet werden und Exemplare von Rechnungen und Mahnungen an die Kunden.

Im Gespräch mit Frau Kart erfährt Herr Fleissig, dass Eintrittskarten ausschließlich per Telefon von Kunden bestellt werden, eine direkte Vorverkaufsstelle sei nicht vorhanden und auch nicht geplant. Allerdings gibt es Überlegungen, Eintrittskarten auch über das Internet zu verkaufen. Herr Fleissig bittet Frau Kart zu erläutern, wie der bisherige Ablauf der Kartenbestellung aussieht. Frau Kart erzählt: »Ruft ein Kunde bei uns an, so wird er von Inga oder Klaus, die die eingehenden Telefonate entgegennehmen, zunächst nach einer Kundennummer gefragt. Handelt es sich um einen neuen Kunden, so wird ein Formular mit den Adressdaten und der gewünschten Zahlungsweise ausgefüllt. Ist er bereits Kunde, so wird in der Adressdatei auf unserem zentralen Computer nach seiner Kundennummer gesucht und die Adressdaten auf ein Bestellformular übertragen. In das Bestellformular trägt Inga oder Klaus dann die gewünschten Eintrittskarten ein. Hierzu wird die Bezeichnung der Veranstaltung, der Vorstellungstermin, sowie die Spielstätte und die Anzahl der gewünschten Karten eingetragen.

Danach gibt Inga bzw. Klaus das Bestellformular an Herrn Klein weiter. Herr Klein ergänzt das Formular um die Vorstellungsnummer, die er aus einem Katalog aller verfügbaren Vorstellungen manuell heraussucht. Anhand der Vorstellungsnummer geht er zu unserem Tresor und sucht die gewünschten Eintrittskarten heraus und gibt sie zusammen mit dem Bestellformular an Herrn Wunder vom Rechnungswesen weiter.«

Im darauf folgenden Gespräch mit Herrn Wunder erfährt Herr Fleissig, was danach mit dem Bestellformular passiert. Herr Wunder erzählt: »Ich nehme das Bestellformular und suche in der zentralen Kundendatei nach dem Kunden, um die Adressdaten in die zu erstellende Rechnung zu übernehmen. Die Rechnung schreibe ich in meinem Textverarbeitungsprogramm. Entsprechend führe ich die Einzelposten mit den Eintrittskarten auf und berechne eine Gesamtsumme mit Mehrwertsteuer. Abhängig von der Gesamtsumme und davon, ob es sich um einen langjährigen Kunden handelt,

rechne ich schließlich noch einen Rabatt von 2% bis 4% heraus. Danach drucke ich die Rechnung zweifach aus und stecke ein Exemplar zusammen mit den Eintrittskarten in einen Briefumschlag, der dann an den Kunden versendet wird. Das zweite Exemplar geht schließlich an unseren Steuerberater, der extern die Finanzbuchhaltung für »KartoFinale« durchführt«.

Nach den Gesprächen erstellt Herr Fleissig mit Bleistift und Papier zunächst einen groben Entwurf über den Ablauf dieses Geschäftsprozesses, um grafisch einen integrierten Überblick zu erhalten. Dabei bemerkt er, dass noch Informationen fehlen. So hat Frau Kart noch nicht erklärt, was mit dem Kundenformular für neue Kunden geschieht. Herr Fleissig geht davon aus, dass Inga bzw. Klaus diese in die zentrale Kundendatei selbständig eintragen.

Danach gibt Herr Fleissig den ersten Grobentwurf zur Überprüfung noch einmal Frau Kart und Herrn Wunder und erfährt, dass neue Kundendaten nicht von Inga oder Klaus, sondern von Herrn Klein in die Adressdatei übertragen werden. Entsprechend korrigiert er seine Grafik. Herr Fleissig verwendet seine eigene Notation zur grafischen Darstellung des Geschäftsprozesses, da der Entwurf nur als vorläufige Skizze dienen soll, um einen ersten Überblick über diesen Geschäftsprozess zu erhalten.

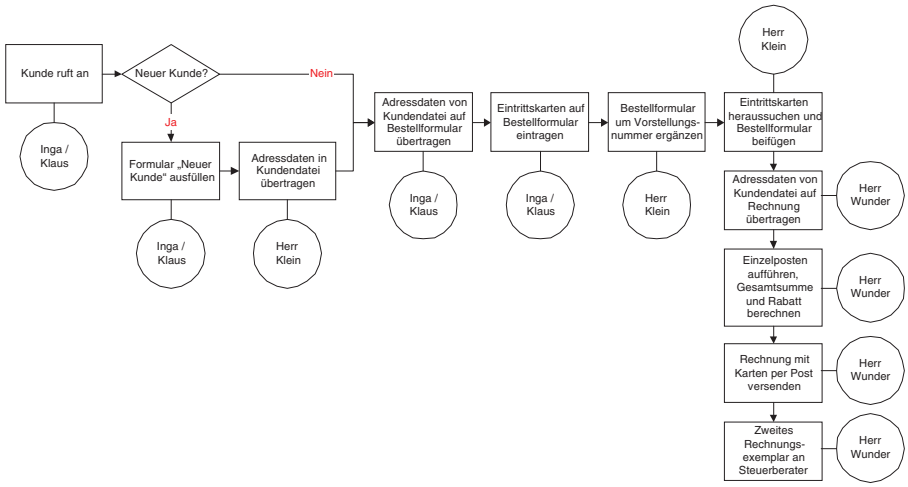


Abbildung 3.1: Erster Entwurf des Geschäftsprozesses »Kartenbestellung«

Nach den Gesprächen erhält Herr Fleissig noch die Adressdatei der Kunden. Wieder zurück in der Firma, beginnt Herr Fleissig, den Zweck des zukünftigen Abrechnungssystems zu beschreiben. Das zu entwickelnde Abrechnungssystem soll den Bestellvorgang von der Bestellannahme bis zum Versand der Eintrittskarten sowie Teile des Rechnungswesens, nämlich Fakturierung und Mahnwesen beinhalten.

Danach stellt Herr Fleissig einen ersten Projektplan auf, der aus vier Phasen besteht. In der ersten Phase, die sechs Wochen dauern soll, werden in Zusammenarbeit mit dem Kunden die Anforderungen an das zu entwickelnde Abrechnungssystem definiert. Als Ergebnis erhält man ein konzeptionelles Modell, das losgelöst ist von Hardware und

Software. In der zweiten Phase wird das eigentliche Abrechnungssystem entworfen, als Ergebnis entsteht ein in sich logisches Modell, das auf den Computer übertragen werden kann. Auf Grundlage der zweiten Phase wird in der dritten Phase das logische Modell auf dem Computer umgesetzt. Als Ergebnis entsteht das Abrechnungssystem, das getestet und schließlich an den Kunden ausgeliefert wird. In der vierten Phase wird dann das Abrechnungssystem vom Kunden bewertet, noch etwaige Änderungen vorgenommen und ein Support- und Wartungsplan erstellt.

Herr Fleissig beginnt also in der ersten Phase auf Basis der bisher geführten Gespräche das konzeptionelle Datenmodell zu erstellen. Er analysiert, welche Geschäftsobjekte in dem Geschäftsprozess auftreten:

- Mitarbeiter
- Abteilung
- Kunde
- Eintrittskarte
- Rechnung
- Einzelposten

Danach überlegt er sich, welche Beziehungen zwischen diesen Geschäftsobjekten bestehen:

- Mitarbeiter gehört zu Abteilung
- Kunde bestellt Eintrittskarten
- Kunde erhält Rechnung
- Rechnung besteht aus Einzelposten

Schließlich zeichnet er das Ganze grafisch als »Entity-Relationship-Modell« (ERM) auf.

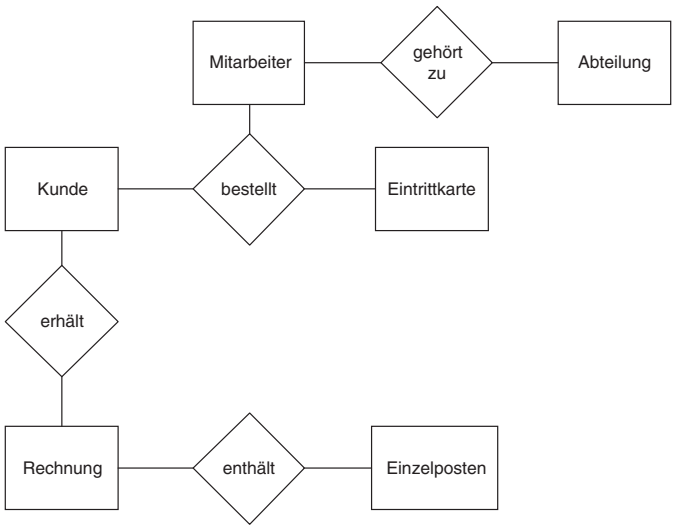


Abbildung 3.2: Erstes konzeptionelles Datenbankmodell

Am Beispiel der Firma »KartoFinale« haben wir bis jetzt den Ablauf eines Projektes vom Projektanstoß über das Kennenlernen des Geschäftsumfeldes, das Erstellen eines Projektplanes bis hin zum ersten Entwurf eines Datenmodells kennen gelernt. Diese einzelnen Schritte eines Projektablaufs werden im Folgenden detailliert erläutert.

3.2 Projektplan versus Phasenkonzept

Vor der eigentlichen Entwicklung eines Anwendungssystems, stellt man einen Plan auf, in dem die Tätigkeiten, die zur Erstellung des Anwendungssystems notwendig sind, aufgeführt werden. Jeder Tätigkeit wird das zu liefernde Ergebnis zugeordnet. Man teilt also die Entwicklung des Anwendungssystems in verschiedene Schritte bzw. Phasen ein.

Ein Phasenkonzept ist ein Vorschlag für einen Entwicklungsprozess zur erfolgreichen Durchführung eines Softwareprojektes. In der Literatur existieren verschiedene Phasenkonzepte, die einen Anhaltspunkt geben sollen, wie der eigene Projektplan aussehen könnte. Die Konzeptvorschläge unterscheiden sich dabei nur marginal. Lediglich bei der Wiederholhäufigkeit einzelner Phasen, falls die angestrebten Ergebnisse noch nicht erzielt wurden, gehen die Autorenmeinungen auseinander.

Das folgende hier vorgestellte Phasenkonzept soll als Anregung zur Erstellung eines eigenen Projektplanes verstanden werden. Da kein Projekt dem anderen gleicht, müssen sich auch die Pläne zur Durchführung dieser Projekte im Detail unterscheiden. Von der »unkritischen« Übertragung eines standardisierten Phasenkonzeptes auf das eigene Projekt wird daher abgeraten. Vielmehr soll das folgende Phasenkonzept als »Gerüst« dienen, das jederzeit an die eigenen Bedürfnisse angepasst werden kann.

Die Entwicklung eines Anwendungssystems kann man definieren als das Entwerfen und Umsetzen von logischen und physikalischen Abläufen und Strukturen, um den Anforderungen und den Informationsbedarf von Benutzern in einer Organisation gerecht zu werden. Das Phasenkonzept stellt dabei eine Vorlage für diesen Entwicklungsprozess dar.

Da ein Anwendungssystem im Wesentlichen aus Funktionen und Daten besteht, werden in jeder Phase sowohl Funktionen als auch Daten entworfen. Jede einzelne Phase hat also zwei Ergebnisse: ein Funktionsmodell und ein Datenmodell. Das Datenmodell legt fest, welche Daten benötigt und in einer Datenbank gespeichert werden sollen. Das Funktionsmodell bestimmt dagegen, wie die darin gespeicherten Daten verarbeitet werden.

Obwohl sich dieses Buch vorwiegend mit der Datenmodellierung und der Umsetzung in eine Datenbankstruktur beschäftigt, soll im Folgenden auch kurz auf die Funktionsmodellierung eingegangen werden, da größere Datenbanksysteme inzwischen verbreitet funktionale Elemente enthalten können.

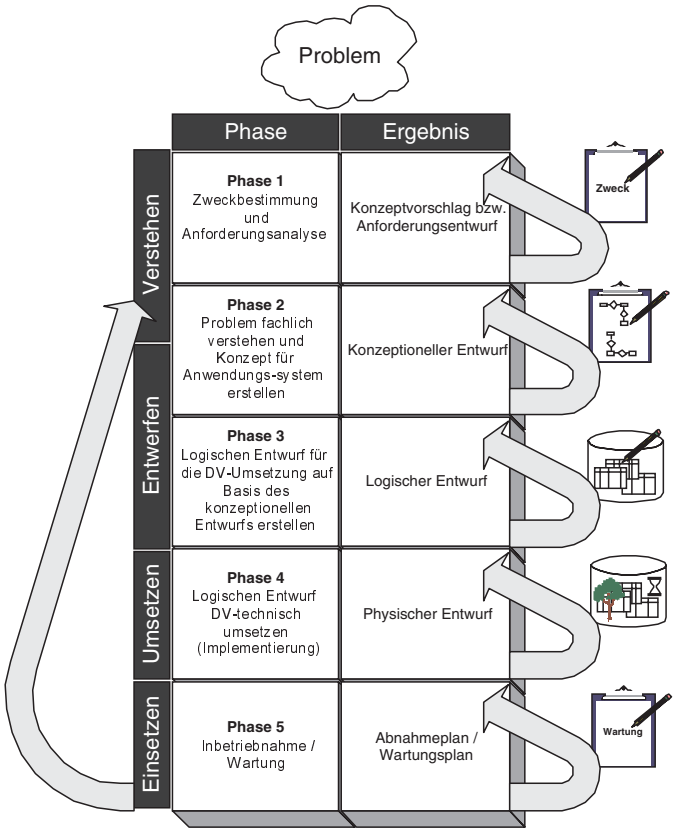


Abbildung 3.3: Phasenkonzept

Die **erste Phase** beschäftigt sich mit dem groben Kennenlernen des eigentlichen Problems. In unserem Beispiel klärt Herr Fleissig in einem ersten Gespräch mit dem Geschäftsführer von »KartoFinale«, warum ein Abrechnungssystem entwickelt werden soll, wo die Probleme bestehen und welche wesentlichen Eigenschaften das zu entwickelnde Abrechnungssystem enthalten soll. Dabei hat Herr Fleissig nur eine ungenaue Vorstellung davon, wie das spätere Abrechnungssystem einmal aussehen wird. Es geht hierbei ausschließlich um eine kurze Abgrenzung des Themas. Als Ergebnis dieser ersten Phase entsteht ein Konzeptvorschlag, der beschreibt, was eigentlich als Endergebnis des Projektes herauskommen und warum dieses Projekt in Angriff genommen werden soll.

In der **zweiten Phase** beschäftigt man sich mit dem geschäftlichen Umfeld. Man ermittelt,

- wie Geschäftsprozesse bisher abgewickelt werden (z.B. Kontoüberweisung bei einer Bank, Bestellung von Eintrittskarten),
- welche Geschäftsobjekte mit welchen Informationen es gibt (z.B. Kunde mit den Informationen Name, Vorname, Adresse usw.),

- welche Beziehungen zwischen den Geschäftsobjekten existieren (z.B. Kunde erhält Rechnung, Abteilung besteht aus mehreren Mitarbeitern),
- welche Einschränkungen zwischen Geschäftsobjekten bzw. Informationen existieren (z.B. darf eine Eintrittskarte für einen bestimmten Sitzplatz nur einmal verkauft werden).

Hier arbeitet man mit verschiedenen Informationsquellen. Man befragt zukünftige Benutzer des Anwendungssystems, sichtet Dokumente oder analysiert ein vielleicht bereits bestehendes Anwendungssystem und lernt so die Organisation und das Geschäftsumfeld kennen. Ebenso ist es wichtig, in Gesprächen mit den zukünftigen Benutzern des Anwendungssystems Begriffe zu erlernen, die im täglichen Geschäftsumfeld verwendet werden. Beide Seiten, der Analytiker und der zukünftige Benutzer, müssen die »gleiche Sprache« sprechen. Benutzer betrachten einen Geschäftsprozess in der Regel nur aus ihrer Sicht, je nachdem, welche Funktion sie im Unternehmen innehaben. Es ist deshalb notwendig, von möglichst vielen Benutzern Informationen über einen Geschäftsprozess zu erhalten, um so zu einer allgemeinen Sichtweise zu gelangen.

In dieser zweiten Phase sammelt man eine Menge an Informationen, die teilweise Lücken enthalten oder aber auch widersprüchlich sind. Um diese »Flut« an Informationen zu bewältigen, erstellt man ein konzeptionelles Funktions- und Datenmodell. Im Funktionsmodell beschreibt man, wie Geschäftsprozesse ablaufen, im Datenmodell, welche Daten dazu benötigt werden.

Neben der strukturierten textuellen Beschreibung werden die konzeptionellen Daten- und Funktionsmodelle zur Veranschaulichung grafisch dargestellt. Für die grafische Notation des Datenmodells wird in der Praxis vorwiegend das »Entity-Relationship-Modell« (ERM) verwendet (siehe Abschnitt 3.4), für die grafische Notation des Funktionsmodells die »Unified Modeling Language« (UML) (siehe Abschnitt 3.6). Dabei wird der UML zukünftig auch in der Datenmodellierung eine höhere Bedeutung zukommen. Zum einen, um eine mehr oder weniger einheitliche Notation zu verwenden, zum anderen aber auch, weil immer mehr funktionale Elemente durch das Datenbanksystem selbst abgebildet werden.

Das entstandene konzeptionelle Modell wird in der zweiten Phase zur Diskussion mit dem Kunden eingesetzt. Die zukünftigen Endbenutzer überprüfen das Modell auf noch gewünschte Funktionen oder eventuell vorhandene Fehler oder Missverständnisse. Das konzeptionelle Modell dient also als Diskussionsplattform mit dem Kunden, genauso wie ein Architekt den »Bauplan« mit dem zukünftigen Hausbesitzer durchspricht.

Der fertige konzeptionelle Entwurf dient in der **dritten Phase** als Grundlage zur Erstellung eines logischen Entwurfes. Der logische Entwurf setzt das Daten- und Funktionsmodell DV-technisch um. Es stellt sozusagen eine »Brücke« zwischen dem fachlichen Konzept und der eigentlichen Software dar. Die Notation des »Entity Relationship Modells« (ERM) kann nicht direkt in eine Datenbankstruktur transformiert werden. Schließlich dient das ERM ja vorwiegend dazu, die fachlichen Aspekte des Geschäftsumfeldes mit dem Kunden zu klären. Entsprechend ist es von der Notation sehr einfach aber auch eingängig gehalten. Es muss also in ein logisches Modell überführt werden. In der Praxis für relationale Datenbanken hat sich hier das Relationenmodell etabliert. Eine Überführung vom ERM in das Relationenmodell ist relativ einfach und

kann nach bestimmten Transformationsregeln vorgenommen werden. Das Relationenmodell wird detailliert im Kapitel 4 beschrieben.

Im Gegensatz zum Datenmodell wird beim Funktionsmodell in der Praxis nach dem konzeptionellen auch das logische Modell mit der UML entworfen.

Phase 4 stellt die eigentliche Implementierung des Anwendungssystems dar. Nachdem ein Modell in der dritten Phase erstellt wurde, das formal DV-technisch abbildbar ist, beginnt man in der vierten Phase mit der Programmierung und der DV-technischen Umsetzung der Datenbankstruktur. Dafür wird bei relationalen Datenbanken die »Structured Query Language« (SQL) verwendet, die wir genauer ab Kapitel 6 kennen lernen werden. Die Umsetzung des Funktionsmodells erfolgt mit gängigen Programmiersprachen wie C++, Java, Visual Basic oder C#, aber auch mit SQL. Seit der Verabschiedung des zweiten SQL-Standards im Jahr 1992 enthält SQL Sprachelemente prozeduraler Programmiersprachen. Im dritten Standard 1999 wurde SQL in dieser Hinsicht noch einmal erweitert. Es ist deshalb heutzutage üblich, mit dieser Sprache nicht nur die Datenbankstrukturen und Datenbankabfragen zu realisieren, sondern auch einen Großteil von Funktionen abzubilden. Hierauf werden wir genauer in den Kapiteln 12 bis 14 eingehen.

Als Ergebnis der vierten Phase entsteht die eigentliche Software, die zu diesem Zeitpunkt immer wieder getestet und mit dem Kunden auch auf fachliche Korrektheit überprüft werden sollte.

In der **fünften Phase** wird das getestete Anwendungssystem beim Kunden eingeführt und entsprechend einem vorher definierten Plan vom Kunden abgenommen. Ein Abnahmeplan sollte vorher festgelegte Szenarien enthalten, die die DV-technisch abgebildeten Geschäftsprozesse auf korrekte Funktionalität überprüfen. Wurde das Anwendungssystem vom Kunden abgenommen, so muss es gewartet werden, d.h. es muss Mitarbeiter geben, die die Computer warten, die Endbenutzer betreuen und eventuell schulen sowie die gesamte DV-Umgebung überwachen.

Das vorgestellte Phasenkonzept sollte generell niemals als ein durchgehender Prozess betrachtet werden. Gerade zwischen den Phasen 2 und 4 wird es oft wieder Rücksprünge geben, da fachliche oder technische Aspekte nicht in einer vorherigen Phase berücksichtigt wurden. Nehmen wir unser Beispiel der Firma »KartoFinale«. Dort wurde der Bezahlvorgang nach Rechnungsversand bisher noch gar nicht berücksichtigt. Würde das Projekt nun beginnen und erst bei der Umsetzung des Anwendungssystems in Phase 4 die Nichtberücksichtigung des Bezahlvorgangs bemerkt werden, eine DV-technische Umsetzung aber vom Kunden gewünscht ist, so muss entsprechend zur Phase 2 zurückgesprungen und das konzeptionelle und logische Modell geändert werden. Natürlich sollten wichtige Punkte eines Geschäftsprozesses wie der Bezahlvorgang von einem Analytiker bereits vorher berücksichtigt werden. Häufig sind es in der Praxis aber auch Kleinigkeiten oder technische Restriktionen, die zu einem Rücksprung führen. Generell gilt es jedoch, in den Phasen 1 bis 3 das Anwendungssystem so weit zu entwerfen, dass solche unliebsamen »Überraschungen« erst gar nicht auftreten. Dementsprechend sollte man diese Phasen nicht unterschätzen und hier ausreichend Zeit und Sorgfalt investieren. Gerade die Vernachlässigung dieser Phasen führt immer wieder zum Scheitern von Softwareprojekten.

3.3 Konzeptioneller Entwurf (Datenmodellierung)

3.3.1 Grundlagen

Im letzten Abschnitt haben wir kennen gelernt, wie ein komplettes Softwareprojekt vom Projektanstoß bis zum eigentlichen Einsatz umgesetzt wird. Wir haben gesehen, dass die Datenmodellierung selbst nur ein Teil des gesamten Entwicklungsprozesses ausmacht und von der Funktionsmodellierung getrennt wird. Diese Trennung ist sinnvoll, da ein Datenmodell nicht so sehr an ein bestimmtes Anwendungssystem angelehnt und abstrakter betrachtet werden sollte. Stellen Sie sich vor, Sie müssten für Ihre Firma ein Personalabrechnungssystem entwickeln. Würden Sie das Datenmodell nur für ein Personalabrechnungssystem erstellen, würden Beziehungen zu anderen wichtigen Geschäftsobjekten der Firma verloren gehen. Sie hätten eine Insellösung entwickelt, die ausschließlich für die Personalabrechnung funktioniert. Anwendungssysteme für weitere Geschäftsprozesse z. B. zur Kundenbetreuung müssten entweder eine eigene Datenbank erhalten oder die Datenbankstruktur der Personalabrechnung müsste in größerem Maße geändert werden. Die erste Lösung hätte wieder den uns bereits bekannten Nachteil der Datenredundanz: Mitarbeiterdaten würden in der Firma mehrfach gespeichert werden. Die zweite Lösung dagegen würde für jedes neu zu entwickelnde Anwendungssystem einen »Rattenschwanz« von Änderungen in bereits bestehenden Anwendungssystemen nach sich ziehen.

Man geht heute deshalb dazu über, so genannte unternehmensweite Datenmodelle (UDM) zu entwerfen, die ein Datenmodell und damit eine einzige Datenbasis für ein gesamtes Unternehmen abbilden. Auf Grundlage dieser Datenbasis sollten dann beliebige Anwendungssysteme für das Unternehmen entwickelt werden können, ohne die grundlegende Struktur der Datenbank in größerem Ausmaß zu ändern.

Daneben existieren Referenzmodelle bestimmter Branchen. Es ist also möglich, so ein Daten-Referenzmodell zu nehmen und es an die eigenen Bedürfnisse des Unternehmens anzupassen.

Doch zurück zum eigentlichen konzeptionellen Datenbankentwurf. Dieser wird für zwei Personengruppen erstellt. Zum einen für den Analytiker, der ihn als »Bauplan« zur Absprache mit dem Kunden verwendet. Zum anderen für den Datenbankdesigner, der den konzeptionellen Entwurf als Basis für die Umsetzung in einen logischen und schließlich in einen physischen Entwurf verwendet.

Beide Personengruppen haben entsprechend unterschiedliche Wünsche an ein Datenmodell. Für den Analytiker und den Kunden soll das Datenmodell möglichst leicht verständlich sein. Dies heißt jedoch auch, dass es weniger detailliert ist. Demgegenüber wünscht sich der Datenbankdesigner ein Datenmodell, das möglichst detailliert und komplett ist. Dieses geht jedoch in der Regel zu Lasten der Übersichtlichkeit und Verständlichkeit.

Generell lässt sich zusammenfassen, dass es in der Datenmodellierung um das Identifizieren und Klassifizieren von Geschäftsobjekten des Geschäftsumfeldes und deren kennzeichnenden Merkmale sowie der Beziehungen zwischen ihnen geht.

Ein Datenmodell besteht aus:

- strukturiertem Text,
- grafischer Notation.

Der strukturierte Text beschreibt detailliert die identifizierten und klassifizierten Geschäftsobjekte und deren Beziehungen, sowie vorhandene Regeln bzw. Einschränkungen. Der strukturierte Text dient vorwiegend dem Datenbankdesigner zum Entwurf des logischen und physischen Datenmodells.

Die grafische Notation dient der Übersicht und vor allem dem Verständnis der Geschäftsobjekte und deren Beziehungen, da grafische Modelle vom Menschen ganzheitlich besser aufgenommen werden können. Andererseits verzichten grafische Modelle auf Details.

Da es um Geschäftsobjekte und deren Beziehungen geht, gibt es verschiedene grafische Elemente für:

1) Geschäftsobjekte,
2) Attribute,
3) Eindeutige Identifizierer (Schlüssel),
4) Beziehungen,
5) Beziehungstypen,
6) Sub- bzw. Supertypen.

Im Folgenden wollen wir uns mit drei grafischen Notationen bzw. Modellen, die in der Praxis relevant sind, auseinandersetzen, dem »Entity-Relationship-Modell« (ERM) nach Chen, dem »Entity-Relationship-Modell« (ERM) nach Barker und der »Unified Modeling Language« (UML). Zum Schluss jedes vorgestellten Modells, wird anhand des praxisorientierten Fallbeispiels ein Datenmodell mit der jeweiligen Notation erstellt, um die Unterschiede zwischen diesen grafischen Notationen kennen zu lernen.

Bevor wir uns jedoch mit den drei grafischen Notationen auseinandersetzen, wollen wir erst einmal unser Fallbeispiel beschreiben, die oben aufgeführten sechs Elemente definieren und schließlich betrachten, wie man aus der Beschreibung eines Geschäftsumfeldes diese Elemente identifizieren kann.

3.3.2 Fallbeispiel

Herr Dr. Fleissig von der Unternehmensberatung »Software Consult AG« hat inzwischen verschiedene Dokumente und Formulare gesichtet und sich mit den meisten relevanten Mitarbeitern von »KartoFinale« unterhalten. Zunächst berichtet er seinem Geschäftsführer von den Kunden und den Artikeln, die »KartoFinale« verkauft:

»Ein Artikel kann entweder eine Eintrittskarte oder ein Werbeartikel zu einer Veranstaltung sein. Zu einer Veranstaltung kann es mehrere Werbeartikel geben, ein Werbe-

artikel ist immer genau einer Veranstaltung zugeordnet (z.B. »Cats T-Shirt« ist Veranstaltung »Cats« zugeordnet). Es gibt also keinen Werbeartikel, der für verschiedene Veranstaltungen verkauft wird. Eine Veranstaltung besteht aus mehreren Vorstellungsterminen, zu denen die Veranstaltung stattfindet. Eine Vorstellung findet in einer bestimmten Spielstätte statt. Nehmen wir einmal ein Beispiel: Zu der Veranstaltung »Don Giovanni von Mozart« gibt es drei Vorstellungen, und zwar am 1., 8. und 15. Oktober jeweils um 20 Uhr. Die Vorstellungen am 1. und 8. Oktober finden in der Spielstätte »Deutsche Staatsoper« in Hamburg statt, die Vorstellung am 15. Oktober im »Hamburger Operettenhaus«.

Ein Kunde kann mehrere Artikel kaufen und ein bestimmter Artikel kann von mehreren Kunden gekauft werden. Der Werbeartikel »Cats T-Shirt« kann von mehreren Kunden gekauft werden, da er mehr als einmal bei »KartoFinale« vorrätig ist. Entsprechend kann ein Kunde mehrere Werbeartikel kaufen, z.B. das »Cats T-Shirt« und auch das Plakat zu dieser Veranstaltung.

Bestellt der Kunde Artikel, so nimmt ein bestimmter Mitarbeiter die Bestellung telefonisch entgegen. Ein Mitarbeiter bearbeitet mehrere Bestellungen, andererseits wird eine bestimmte Bestellung genau von einem Mitarbeiter bearbeitet.

Jede Abteilung von »KartoFinale« besteht aus mehreren Mitarbeitern. Ein Mitarbeiter ist immer genau einer Abteilung zugeordnet.

3.3.3 Geschäftsobjekte

Ein Geschäftsobjekt, oder allgemeiner ein Objekt, ist ein Element, das eindeutig identifiziert und von anderen Objekten anhand seiner Eigenschaften unterschieden werden kann. Es stellt sozusagen eine »Schablone« dar, um gleichartige Dinge und deren Informationen zusammenzufassen. Ein Objekt kann entweder sein

- etwas real Existierendes (z.B. Person, Flugzeug, Buch),
- ein Ereignis (z.B. »Artikel bestellen«, »Tee trinken«, »Gespräch führen«, »Rechnung zahlen«),
- eine Rolle oder Person (z.B. Arzt, Patient, Richter, Kunde, Staatspräsident),
- eine Organisation (z.B. Firma, Abteilung, Behörden),
- ein Konzept (z.B. Projekt, Modellbeschreibung) oder
- eine Transaktion (z.B. Kaufvertrag).

Entsprechend fassen wir unter einem Objekttyp oder einer Objektklasse also Dinge zusammen, die durch gleichartige Merkmale beschrieben werden können und deren Merkmale jedes Element eindeutig identifizieren.

Betrachten wir hierzu einmal unser Fallbeispiel: Die Firma »KartoFinale« hat mehrere Mitarbeiter und diese Mitarbeiter haben Merkmale, die sie voneinander unterscheiden. In den Merkmalstypen stimmen die Mitarbeiter zwar überein (Name, Abteilung, Personalnummer, Adresse), aber nicht in den jeweiligen Ausprägungen.

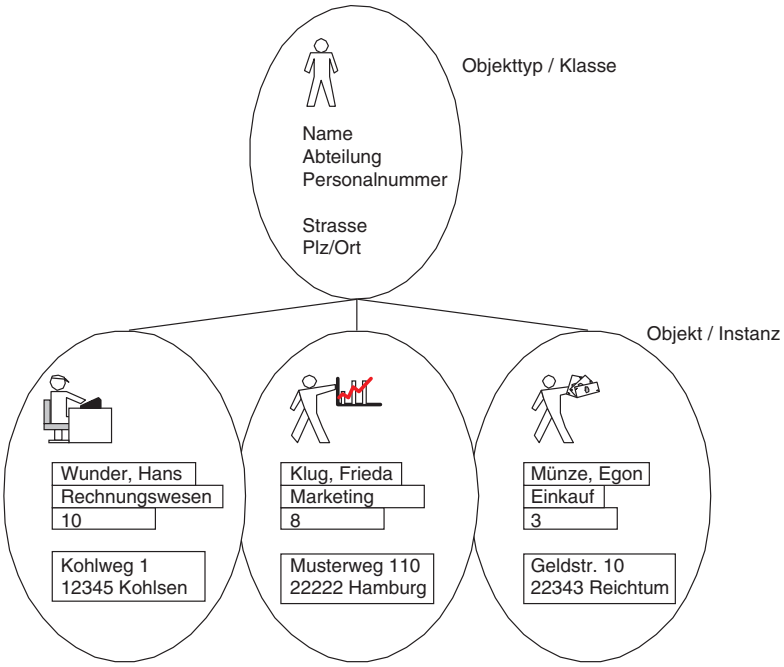


Abbildung 3.4: Objekte und deren kennzeichnende Merkmale

Objekttypen bzw. Objektklassen kann man in einer Textbeschreibung grammatikalisch häufig als Substantive ausmachen. Jedoch sind nicht alle im Text vorkommenden Substantive gleichzeitig auch Objekttypen. Jedes Objekt dieses Objekttyps muss über seine Merkmale eindeutig identifizierbar sein. Entsprechend sollte es mehr als ein beschreibendes Merkmal besitzen und natürlich einen definierten Geschäftszweck besitzen. So scheint es auf den ersten Blick so, als sei »Eintrittskarte« prädestiniert als Objekttyp, da es scheinbar einen wichtigen Geschäftszweck erfüllt. Überlegen wir uns jedoch, welche Merkmale eine Eintrittskarte besitzt. Dazu ist es sinnvoll, zunächst anhand eines Beispiels vorzugehen. Betrachten wir eine Eintrittskarte für »Don Giovanni« am 1. Oktober um 20 Uhr. Neben diesen Informationen ist auf der Eintrittskarte noch der Sitzplatz vermerkt. Überlegen wir, welchen Objekten diese Merkmale zugeordnet werden. Die Bezeichnung »Don Giovanni« ist primär eine Eigenschaft der Veranstaltung, der Termin eine Eigenschaft der Vorstellung und der Sitzplatz eine Eigenschaft der Spielstätte und damit indirekt der Vorstellung. Eintrittskarte ist also kein Objekt, da es keine Informationen beinhaltet. Indirekt stellt die Eintrittskarte ein Synonym für einen Sitzplatz dar. Ein Kunde kauft also abstrakt betrachtet, einen Sitzplatz für eine Vorstellung und keine Eintrittskarte. Damit ist die Eintrittskarte nur ein Beleg, um einen Sitzplatz zu einem bestimmten Termin in einer Spielstätte zu »mieten«.

Solche Erkenntnisse sind manchmal schwer zu durchschauen, weshalb gerade das Interview unterschiedlicher Personen zu dem jeweiligen Geschäftsumfeld wichtig ist.

In unserem Fallbeispiel ergeben sich folgende Objekttypen bzw. Objektklassen:

- Abteilung,
- Mitarbeiter,
- Kunde,
- Artikel,
- Werbeartikel,
- Vorstellung,
- Veranstaltung,
- Spielstätte,
- Sitzplatz.

3.3.4 Sub- bzw. Supertypen

Ein Subtyp ist eine Untermenge von übergeordneten Objekten eines Supertyps. Sub- bzw. Supertypen stellen deshalb eine Besonderheit von Objekttypen dar, um Informationen hierarchisch abzubilden.

Betrachten wir hierzu wiederum unser Fallbeispiel. Ein Kunde kauft einen Artikel. Ein Artikel stellt im obigen Sinne einen Objekttyp dar, da er durch verschiedene Merkmale wie Preis, Bezeichnung usw. charakterisiert wird. Andererseits verkauft »KartoFinale« zwei unterschiedliche Arten von Artikeln: Sitzplätze und Werbeartikel. Gemein ist Sitzplätzen und Werbeartikeln eine Artikelbezeichnung und eine Artikelnummer. Dagegen hat ein bestimmter Werbeartikel immer den gleichen Preis, ein Sitzplatz jedoch unterschiedliche Preise.

Es ist deshalb sinnvoll, einen übergeordneten Objekttyp (Supertyp) zu definieren, in diesem Fall Artikel, der die gemeinsamen Merkmale beinhaltet. Daneben entwirft man zwei weitere untergeordnete Objekttypen (Subtypen) für Werbeartikel und Sitzplatz, die vom Supertyp die Merkmale sozusagen erben. Aufgrund der Über- bzw. Unterordnung von Sub- und Supertypen spricht man auch von Hierarchisierung, Generalisierung und Spezialisierung. Spezialisierung bezieht sich in diesem Fall auf den Subtypen, der aufgrund zusätzlicher Attribute genauer spezifiziert wird. Generalisierung bezieht sich auf den Supertypen, der die gemeinsamen allgemeinen Attribute der untergeordneten Objekttypen enthält und Hierarchisierung auf die entstehende Struktur, die eine Hierarchie abbildet. Die Abbildung 3.5 verdeutlicht diesen Zusammenhang noch einmal am Beispiel von Gebäuden.

Der Objekttyp »Gebäude« ist dabei Supertyp von »Hochhaus«, »Einfamilienhaus« und auch von »Reihenhaus« und »Doppelhaushälfte«. Der Objekttyp »Einfamilienhaus« ist Supertyp von »Reihenhaus« und »Doppelhaushälfte«. »Einfamilienhaus« erbt also z. B. die Attribute »Baujahr« und »Architekt« von seinem Supertyp »Gebäude«, »Doppelhaushälfte« erbt zusätzlich zu den Attributen »Baujahr« und »Architekt« die Attribute »Dachform«, »Anzahl Zimmer« und »Größe«.

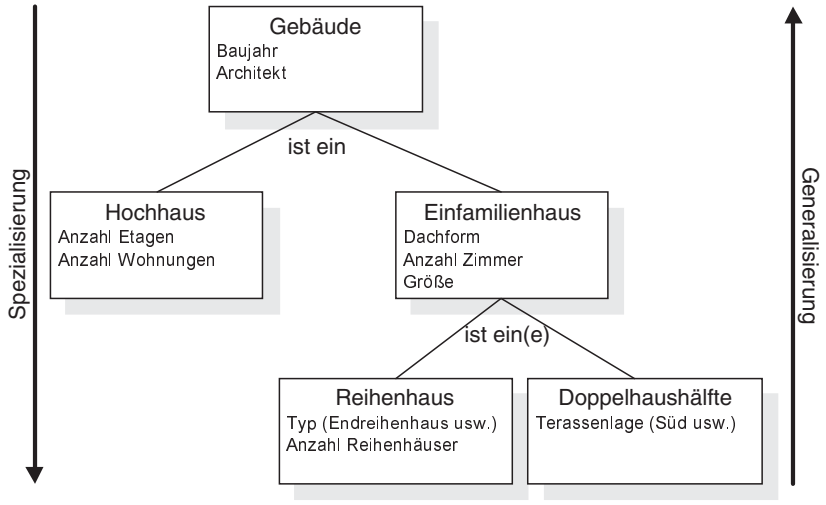


Abbildung 3.5: Beispiel für Super- und Subtypen

3.3.5 Attribute und Schlüssel

Attribute sind Informationsobjekte, die Objekttypen beschreiben oder identifizieren. Attribute, die einen Objekttyp eindeutig identifizieren bezeichnet man als Schlüssel bzw. Schlüsselattribut.

Attribute sollten vollständig für einen Objekttyp aufgeführt werden, soweit sie für den Geschäftszweck notwendig sind. Sie sollten atomar und unabhängig voneinander sein. So setzt sich z.B. der Name einer Person aus zwei Attributen zusammen, dem Vornamen und dem Nachnamen. Würde man nur ein Attribut Name für Vor- und Nachname definieren, so wäre dieses Attribut nicht mehr atomar, da es ja aus zwei Attributen besteht.

Die Werte von Attributen müssen unabhängig voneinander sein. Würde man für einen Objekttyp »Rechnung« zwei Attribute »Rabattprozentsatz« und »Rabattbetrag« definieren, so müsste bei einer Änderung des Rabattprozentsatzes auch der Rabattbetrag geändert werden. Der Rabattbetrag ist jedoch ein abgeleiteter Attributwert, da es ein prozentual berechneter Anteil vom Rechnungsbetrag ist, so dass auf das Attribut »Rabattbetrag« verzichtet werden kann. Hierbei kommt es aber auch auf die Sichtweise an. Angenommen, einem Objekttyp »Mitarbeiter« wurde das Attribut »Anzahl Kinder« zugewiesen. Das Attribut »Anzahl Kinder« ist nur dann sinnvoll, wenn es nicht gleichzeitig einen Objekttyp »Kinder« gibt, in dem die Namen der Kinder der Mitarbeiter gespeichert werden. Wäre dies der Fall, so könnte die Anzahl der Kinder ja berechnet werden, wodurch dann die Information über die Anzahl der Kinder doppelt gespeichert werden würde, einmal im Objekttyp »Mitarbeiter« und ein zweites Mal indirekt im Objekttyp »Kinder«.

Schließlich wird ein Attribut durch einen bestimmten Wertebereich gekennzeichnet. Eine Personalnummer ist in der Regel eine Zahl. Ein Attribut »Geschlecht« dürfte nur zwei mögliche Werte beinhalten, weiblich und männlich. Das Attribut »Zustand« des Objekttyps »Sitzplatz« kann z.B. nur die Werte für »belegt«, »frei« oder »reserviert« annehmen. Man spricht hier von Wertebereich oder Domäne (Domain).

Attribute von Objekttypen findet man am besten, indem man überlegt, welche Eigenschaften Objekte eines Objekttyps gemeinsam haben bzw. welche Objekteigenschaften man benötigt, um sagen zu können, um welchen Objekttyp es sich handelt. So reichen die beiden Attribute Name und Vorname nicht aus, um z.B. sagen zu können, ob »Egon Münze« ein Objekt des Objekttyps »Mitarbeiter« oder »Kunde« ist.

In unserem Fallbeispiel ergeben sich folgende Attribute für die jeweiligen Objekttypen:

Objekttyp	**Attribute**
Abteilung	Abteilungsbezeichnung, Abteilungsnummer, Abteilungsleiter
Mitarbeiter	Name, Vorname, Strasse, Plz, Ort
Ehepartner	Name, Vorname, Alter
Kunde	Name, Vorname, Strasse, Plz, Ort
Artikel	Artikelnummer, Artikelbezeichnung
Werbeartikel	Beschreibung, Preis, Lagerbestand
Vorstellung	Datum, Uhrzeit
Veranstaltung	Bezeichnung, Autor
Spielstätte	Haus, Strasse, Plz, Ort
Sitzplatz	Reihe, Sitz, Bereich, Preis, Zustand

Nachdem wir die Attribute für unsere Objekttypen definiert haben, müssen wir überlegen, welche Attribute bzw. Attributkombinationen notwendig sind, um ein bestimmtes Objekt eines Objekttyps eindeutig zu identifizieren. Dieses identifizierende Attribut bezeichnet man als Schlüssel. Gibt es mehrere Schlüssel, so spricht man von Schlüsselkandidaten. In diesem Falle muss ein Schlüsselkandidat ausgewählt werden, der als so genannter Primärschlüssel fungieren soll und über den ein Objekt eines Objekttyps immer eindeutig identifiziert wird. Ein Schlüssel kann auch aus mehreren Attributen zusammengesetzt sein, generell gilt jedoch, dass ein Primärschlüssel eine minimale Anzahl von Attributen enthalten sollte.

Betrachten wir unseren Objekttyp »Abteilung«. Sowohl die Abteilungsnummer als auch die Abteilungsbezeichnung identifizieren eindeutig eine Abteilung. Wir haben also zwei Schlüsselkandidaten und entscheiden uns für die Abteilungsbezeichnung als Primärschlüssel.

Betrachten wir nun, welches Attribut bzw. welche Attributkombination für Kunde als Schlüssel in Frage kommt. Da Namen wie »Hans Meier« mehr als einmal vorkommen können, ist die Kombination aus Vorname und Name nicht als Schlüssel geeignet.

Auch die Wahrscheinlichkeit, dass in einem Hochhaus in der gleichen Strasse und im gleichen Ort zwei Kunden mit gleichem Namen vorkommen können, ist möglich. Zwar können wir die Etage bei der Postadresse mit aufführen, müssten dann jedoch als Primärschlüssel alle vorhandenen Attribute nehmen. Das ist zwar möglich, aber nicht besonders praktikabel. In einem solchen Fall wird in der Regel ein künstlicher Schlüssel definiert. Wir wollen dieses Attribut als Kundennummer bezeichnen und definieren diese als Primärschlüssel. Wir können zwar auch einen Schlüssel aus Nachname und Kundennummer festlegen, dieses würde jedoch dem oben erwähnten Prinzip der Minimalität eines Primärschlüssels widersprechen.

In Abschnitt 3.3.3 haben wir kennen gelernt, dass die Objekttypen »Werbeartikel« und »Vorstellung« Subtypen von Artikel sind. Dadurch werden diese beiden Objekttypen indirekt durch den Primärschlüssel des Artikels eindeutig identifiziert.

Betrachten wir exemplarisch noch unseren Objekttypen »Sitzplatz«. Um einen Sitzplatz innerhalb einer Spielstätte eindeutig zu identifizieren, müssen in diesem Fall drei Attribute miteinander kombiniert werden. Ein Sitz wird eindeutig in einer Reihe bestimmt, eine Reihe wiederum eindeutig innerhalb eines Bereiches (Parkett, Rang usw.). Damit ergibt sich als Primärschlüssel für Sitzplatz die Attributkombination aus Bereich, Reihe und Sitz oder umgekehrt. Wenn für eine Vorstellung der Bereich, die Reihe und der Sitz bekannt sind, so kann das Objekt und damit auch seine Eigenschaften eindeutig gefunden werden. In diesem Fall auf den Preis und den Zustand für diesen Sitzplatz.

Für unser Beispiel ergeben sich folgende Primärschlüssel (jeweils unterstrichen dargestellt):

Objekttyp	**Attribute**
Abteilung	Abteilungsbezeichnung, Abteilungsnummer, Abteilungsleiter
Mitarbeiter	Personalnummer, Name, Vorname, Strasse, Plz, Ort
Ehepartner	Name, Vorname, Alter
Kunde	Kundennummer, Name, Vorname, Strasse, Plz, Ort
Artikel	Artikelnummer, Artikelbezeichnung
Werbeartikel	Artikelnummer, Beschreibung, Preis, Lagerbestand
Vorstellung	Vorstellungsnummer, Datum, Uhrzeit
Veranstaltung	Veranstaltungsnummer, Bezeichnung, Autor
Spielstätte	Haus, Strasse, Plz, Ort
Sitzplatz	Reihe, Sitz, Bereich, Preis, Zustand

Ein Problem haben wir hierbei allerdings noch: Betrachten wir einmal den Primärschlüssel vom Objekttyp »Sitzplatz«. Ist Bereich, Reihe und Sitz bekannt, so muss man zusätzlich noch wissen, in welcher Spielstätte und für welche Vorstellung diese Informationen gedacht sind. Entsprechendes gilt für Mitarbeiter und Ehepartner. Ohne die Personalnummer des Mitarbeiters kann der Ehepartner nicht ermittelt werden. Um

dieses Problem in einem Datenmodell zu berücksichtigen, müssen wir uns mit den Beziehungen zwischen Objekttypen auseinandersetzen.

3.3.6 Beziehungen und Beziehungstypen

Beziehungen sind Assoziationen zwischen Objekttypen und können grammatikalisch in der Regel durch Verben in einer Textbeschreibung ausgemacht werden. Die wohl bekannteste Beziehung ist die zwischen Mann und Frau. Da Beziehungen gegenseitiger Natur sind, ist die Beziehungsrichtung bei der Betrachtung von Bedeutung. Aus der Sicht des Mitarbeiters besteht die Beziehung »Mitarbeiter gehört zu Abteilung«, aus der Sicht der Abteilung lautet die Beziehung dagegen »Abteilung besteht aus Mitarbeitern«.

Bezogen auf unser Fallbeispiel ergeben sich z. B. folgende Beziehungen:

Beziehungsrichtung 1	Beziehungsrichtung 2
Mitarbeiter gehört zu Abteilung	Abteilung besteht aus Mitarbeitern
Mitarbeiter ist verheiratet mit Ehepartner	Ehepartner ist verheiratet mit Mitarbeiter
Mitarbeiter hat Vorgesetzten	Vorgesetzter hat Mitarbeiter
Kunde bestellt Artikel	Artikel wird von Kunde bestellt
Werbeartikel ist für Veranstaltung	Veranstaltung bietet Werbeartikel
Veranstaltung besteht aus Vorstellungen	Vorstellung gehört zu Veranstaltung
Vorstellung hat Sitzplätze	Sitzplatz gehört zeitlich zu Vorstellung
Spielstätte führt Vorstellungen auf	Vorstellung findet in Spielstätte statt
Artikel ist ein Werbeartikel oder ein Sitzplatz	

Die letzte Beziehung stellt keine typische Beziehung dar, sondern besteht, wie bereits oben erwähnt, aus einem Supertyp und zwei Subtypen. Die Beziehung »ist ein« gibt in der Regel einen Hinweis, dass es sich bei einer Beziehung um Super- und Subtypen handelt. Diese Beziehung wird entsprechend der englischen Übersetzung von »ist ein« häufig auch als »is a«-Beziehung bezeichnet.

Beziehungen zwischen Objekttypen enthalten unterschiedliche Merkmale wie Kardinalität und Optionalität. Kardinalität legt fest, wie viele Objekte des einen Objekttyps in Beziehung zu einem Objekt des anderen Objekttyps stehen und umgekehrt. Optionalität legt fest, ob ein Objekt eines Objekttyps immer in Beziehung zu einem anderen Objekt stehen muss, oder ob es optional ist. Bei der Beziehung zwischen Mitarbeiter und Ehepartner sollte genau ein Mitarbeiter eine Beziehung zu einem Ehepartner haben (Kardinalität von 1:1). Dennoch gibt es auch Mitarbeiter ohne Ehepartner, d. h. die Beziehung zwischen Mitarbeiter und Ehepartner ist nicht zwingend, nicht jeder Mitarbeiter hat eine Beziehung zu einem Ehepartner, die Beziehung ist also optional (Optionalität). Die Kardinalität kann also bei Ehepartner auch Null sein.

Dazu ein weiteres Beispiel: Eine Abteilung besteht aus mehreren Mitarbeitern und ein Mitarbeiter gehört zu einer Abteilung. Betrachten wir das Ganze einmal grafisch für unser Fallbeispiel:

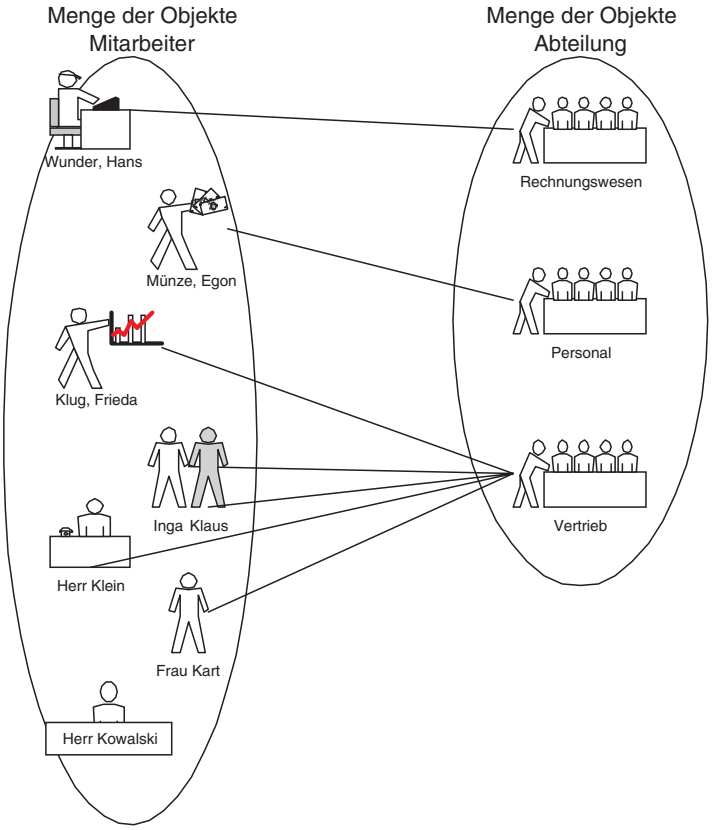


Abbildung 3.6: 1:N-Beziehung zwischen »Mitarbeiter« und »Abteilung«

Die Abteilung »Personal« besteht aus einem Mitarbeiter, Herrn Münze, und Herr Münze ist auch nur dieser Abteilung zugeordnet. Auf dieses Objekt bezogen haben wir also eine Kardinalität von 1:1. Betrachten wir nun jedoch den Vertrieb, so erkennen wir, dass der Vertrieb aus fünf Mitarbeitern besteht. Umgekehrt ist jeder dieser fünf Mitarbeiter auch nur dieser Abteilung zugeordnet. Es gibt also keinen Vertriebsmitarbeiter, der eventuell auch noch der Abteilung Rechnungswesen zugeordnet ist. Wir erkennen also eine Kardinalität von 1:5 bzw. allgemeiner von 1:N, wobei N eine beliebige Ganzzahl ist. Betrachten wir nun Herrn Kowalski. Als Geschäftsführer gehört er zwar zu dem Objekttyp der Mitarbeiter, ist aber keiner Abteilung zugeordnet, d.h. eine Zuordnung von einem Mitarbeiter zu einer Abteilung ist optional. Ein Mitarbeiter kann also einer oder auch keiner Abteilung angehören (max. 1 Abteilung) und eine Abteilung kann einen oder mehrere Mitarbeiter haben (max. N Mitarbeiter).

Wir sprechen deshalb bei der Beziehung von Mitarbeiter zu Abteilung von einem 1:N-Beziehungstyp. Neben dem 1:N-Beziehungstyp gibt es noch zwei weitere Beziehungstypen, den 1:1-Beziehungstyp und den N:M-Beziehungstyp. Bei einem 1:1-Beziehungstyp steht immer genau ein Objekt des einen Objekttyps in Beziehung zu genau einem Objekt des anderen Objekttyps. So ist z. B. Mitarbeiter »Hans Wunder« mit genau einem Ehepartner, nämlich »Karla Wunder«, verheiratet.

Um die Kardinalität von Objekttypen zueinander festzulegen, geht man am besten zunächst von einem Objekt aus und überlegt, ob es zu einem oder mehreren Objekten des anderen Objekttyps eine Beziehung hat. Diesen Vorgang kehrt man danach entsprechend um und überlegt dieses aus der Sicht des anderen Objekttyps.

In unserem Beispiel hat jeder Mitarbeiter genau einen Ehepartner oder keinen bzw. umgekehrt ist jeder Ehepartner mit genau einem Mitarbeiter verheiratet (Kardinalität 1:1).

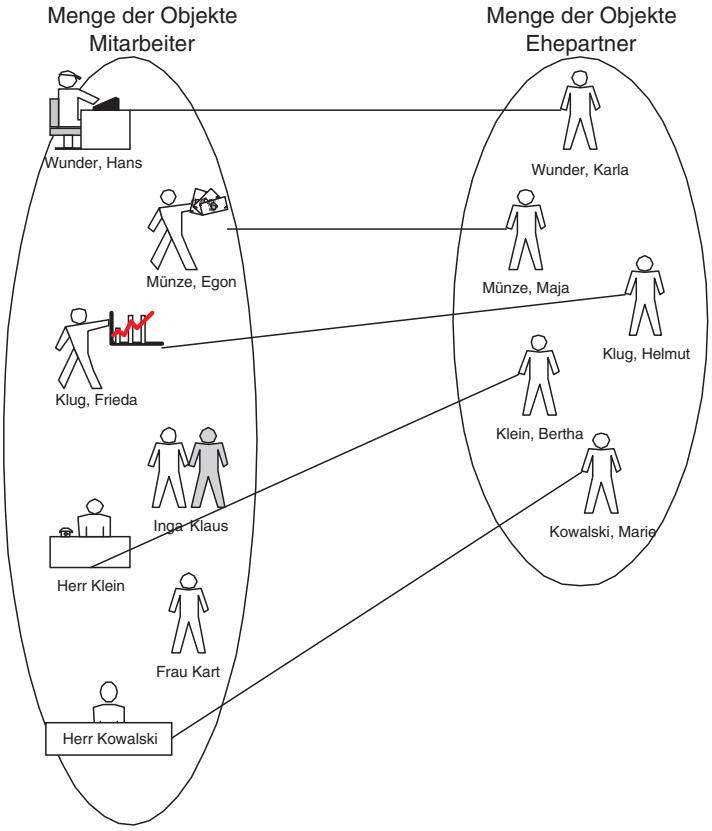


Abbildung 3.7: 1:1-Beziehung zwischen »Mitarbeiter« und »Ehepartner«

Vier Mitarbeiter sind verheiratet, die anderen Mitarbeiter nicht, entsprechend ist die Beziehung von Seiten des Mitarbeiters optional, von Seiten des Ehepartners jedoch nicht.

Bei einem N:M-Beziehungstyp stehen beliebig viele Objekte des einen Objekttyps in Beziehung zu beliebig vielen Objekten des anderen Beziehungstyps. In unserem Fallbeispiel finden wir so eine Beziehung zwischen Kunde und Artikel. Jeder Kunde kann beliebig viele Artikel kaufen bzw. jeder Artikel kann von unterschiedlichen Kunden gekauft werden.

Die Kundin »Frieda Wiegerich« kauft drei unterschiedliche Artikel. Der Artikel »Don Giovanni« wird z.B. von zwei unterschiedlichen Kunden gekauft.

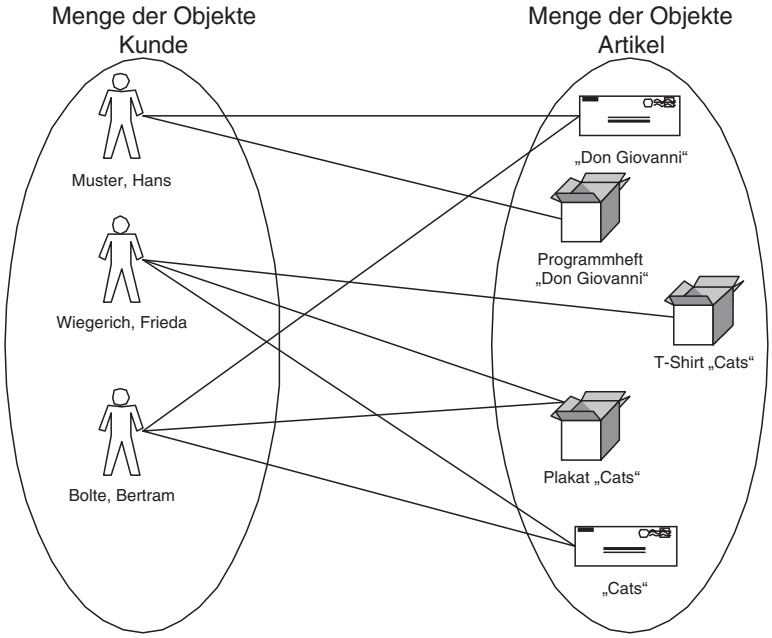


Abbildung 3.8: N:M-Beziehung zwischen »Kunde« und »Artikel«

Zum Schluss dieses Abschnitts wollen wir noch eine Besonderheit bei Beziehungen betrachten: Die Beziehung von Objekten eines Objekttyps zu Objekten des gleichen Objekttyps. Man spricht hier auch von rekursiven Beziehungen.

Betrachten wir wieder unser Fallbeispiel: Ein Mitarbeiter hat genau einen Vorgesetzten und ein Vorgesetzter hat mehrere Mitarbeiter. Ein Vorgesetzter ist jedoch auch ein Objekt des Objekttyps Mitarbeiter. Um nun diese Beziehung abzubilden, stellt man einen Verweis auf den Objekttyp selber her. So ist »Herr Kowalski« direkter Vorgesetzter von den Abteilungsleitern »Hans Wunder«, »Egon Münze« und »Frau Kart«. Frau Kart wiederum ist direkte Vorgesetzte von ihren vier Mitarbeitern, d.h. sie ist in der einen Beziehung Mitarbeiter und in einer anderen Beziehung Vorgesetzte.

Rekursive Beziehungen treten selten auf und wenn, dann meistens als Beziehungstyp 1:N, so wie es auch in unserem Beispiel der Fall ist. Dabei sind sie bei der Darstellung bzw. Umsetzung nicht schwieriger oder anders zu handhaben als Beziehungstypen zwischen zwei unterschiedlichen Objekttypen, da jede Beziehung für sich getrennt betrachtet wird.

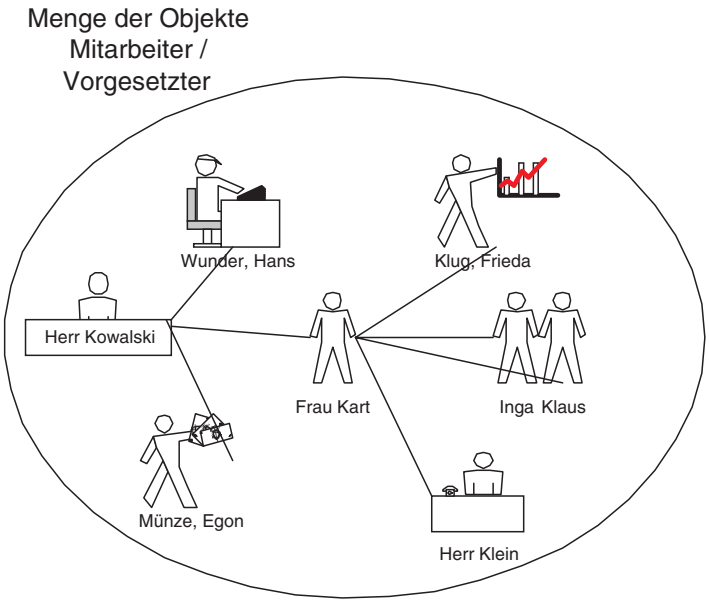


Abbildung 3.9: rekursive 1:N-Beziehung zwischen »Mitarbeitern« und »Vorgesetzten«

Für unser Fallbeispiel ergeben sich folgende Beziehungen und Beziehungstypen:

Ein Mitarbeiter gehört zu einer Abteilung	1:1
Eine Abteilung besteht aus mehreren Mitarbeitern	1:N
Abteilung – Mitarbeiter	**1:N-Beziehung**
Ein Mitarbeiter ist verheiratet mit einem Ehepartner	1:1
Ein Ehepartner ist verheiratet mit einem Mitarbeiter	1:1
Mitarbeiter – Ehepartner	**1:1-Beziehung**
Ein Mitarbeiter hat genau einen Vorgesetzten	1:1
Ein Vorgesetzter hat mehrere Mitarbeiter	1:N
Vorgesetzter- Mitarbeiter	**1:N-Beziehung**

Ein Kunde bestellt mehrere Artikel	1:N
Mehrere Artikel werden von einem Kunden bestellt	N:1
Kunde – Artikel	**N:M-Beziehung**
Ein Werbeartikel ist für eine bestimmte Veranstaltung	1:1
Eine Veranstaltung bietet mehrere Werbeartikel	1:N
Werbeartikel – Veranstaltung	**1:N-Beziehung**
Eine Vorstellung gehört zu einer Veranstaltung	1:1
Eine Veranstaltung enthält mehrere Vorstellungen	1:N
Veranstaltung – Vorstellung	**1:N-Beziehung**
Ein Sitzplatz gehört zeitlich zu einer Vorstellung	1:1
Eine Vorstellung hat mehrere Sitzplätze	1:N
Vorstellung – Sitzplatz	**1:N-Beziehung**
Eine Vorstellung findet in einer Spielstätte statt	1:1
Eine Spielstätte führt mehrere Vorstellungen auf	1:N
Spielstätte – Vorstellung	**1:N-Beziehung**
Ein Artikel ist ein Werbeartikel	1:1
Artikel – Werbeartikel	**1:1-Beziehung**
Ein Artikel ist ein Sitzplatz	1:1
Artikel – Sitzplatz	**1:1-Beziehung**

Nachdem wir uns mit den grafischen Elementen einer Notation für ein Datenmodell auseinandergesetzt haben, wollen wir uns im Folgenden mit den drei praxisrelevanten grafischen Notationen auseinandersetzen.

3.4 »Entity-Relationship-Modell« nach Chen

3.4.1 Grundlagen

Das heute am weitesten verbreitete Datenmodell ist das »Entity-Relationship-Modell« (ERM) bzw. »Entity-Relationship-Diagramm« (ERD). Es basiert auf einem Artikel von Peter Pin-Shan Chen aus dem Jahre 1976 im »ACM – Transactions on Database Systems«-Journal.

Das Modell von Chen ist leicht zu verstehen, begnügt sich mit den wichtigsten grafischen Elementen und hat sich gerade deshalb in der Kommunikation mit dem Kunden durchgesetzt. Außerdem ist es auch leicht in ein logisches Modell für relationale Datenbanken, dem Relationenmodell von E.F. Codd, umzusetzen. Die Umsetzung eines ERM in das Relationenmodell von Codd betrachten wir detailliert im Kapitel 4.

Mit den Jahren hat es von verschiedenen Personen Änderungen und Erweiterungen an dem »Entity-Relationship-Modell« gegeben, die sich u.a. auch auf die grafischen Symbole bezogen. Dies führte dazu, dass es heute keinen einheitlichen Standard gibt, was die grafischen Symbole und die Modellelemente betrifft. Die ursprüngliche Idee ist jedoch in den veränderten Modellen erhalten geblieben. Wir wollen zunächst das ERM nach Chen betrachten, das allerdings heute nicht mehr so oft in der Praxis eingesetzt wird. Danach schauen wir uns das ERM nach Barker an, das vor allem aufgrund seiner leichteren Lesbarkeit und Übersichtlichkeit heute in der Praxis am weitesten verbreitet ist.

3.4.2 Geschäftsobjekte

Geschäftsobjekte werden im ERM als Entity bezeichnet, daher spricht man hier auch nicht von Objekttypen, sondern von Entitytypen.

Entitytypen werden nach Chen in einem Rechteck dargestellt.

Abbildung 3.10: Darstellung von Entitytypen

3.4.3 Sub- bzw. Supertypen

In der ursprünglichen Notation von Chen gab es keine grafische Notation für Sub- und Supertypen. Sie wurden jedoch in der Folgezeit von Robert Brown und Mat Flavin (vgl. hierzu [Hay99]) hinzugefügt. Dabei wird für jeden Subtypen und Supertypen ein eigenes Rechteck verwendet. Die »ist ein«-Beziehung oder auf englisch »is a«-Beziehung wird über das grafische Symbol einer Raute dargestellt. Der kleine Querstrich vor dem Subtypen soll kennzeichnen, dass der Entitytyp »Sitzplatz« ohne seinen Supertypen »Artikel« nicht existieren kann.

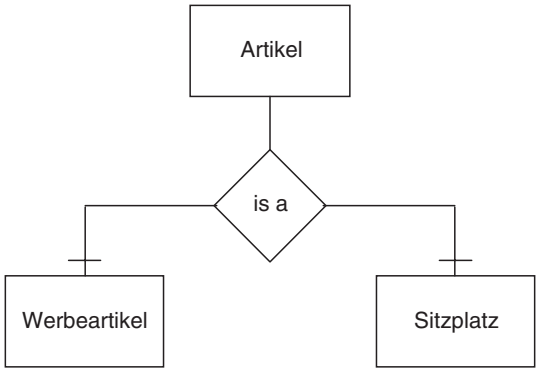


Abbildung 3.11: Darstellung von Sub- bzw. Supertypen

3.4.4 Attribute und Schlüssel

Attribute werden in Form eines Kreises an den jeweiligen Entitytyp angehängt. Primärschlüssel werden dabei nicht gesondert dargestellt, so dass man aus der Grafik nicht unmittelbar erkennen kann, ob es sich bei einem Attribut um einen Schlüssel handelt. Allerdings ist es heutzutage üblich, den Primärschlüssel unterstrichen darzustellen, weshalb wir uns dieser Konvention auch anschließen wollen.

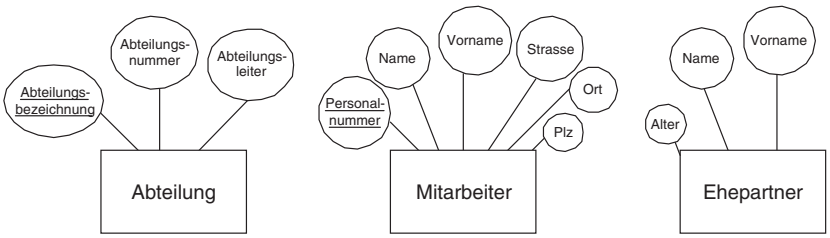


Abbildung 3.12: Darstellung von Attributen

3.4.5 Beziehungen und Beziehungstypen

Beziehungen werden, wie bereits oben erwähnt, über das grafische Symbol einer Raute dargestellt, wobei auch Beziehungen Attribute zugeordnet werden können. Z.B. kann man der Beziehung »Mitarbeiter heiratet Ehepartner« das zusätzliche Attribut »Datum« mitgeben (siehe Abb. 3.13). Eine Raute wird für die Darstellung der Beziehung über Linien mit den in Beziehung stehenden Entitytypen verbunden. Die Beziehung wird bei Chen durch ein Substantiv anstelle eines Verbs bezeichnet, z.B. »Heirat« anstelle des Verbs »heiratet«, da ein Substantiv wie »Heirat« ein Vorgang ist, der eigene Attribute wie ein Datum, einen Ort usw. besitzen kann. Die Kardinalitäten einer Beziehung werden durch die Zeichen 1, N und M gekennzeichnet.

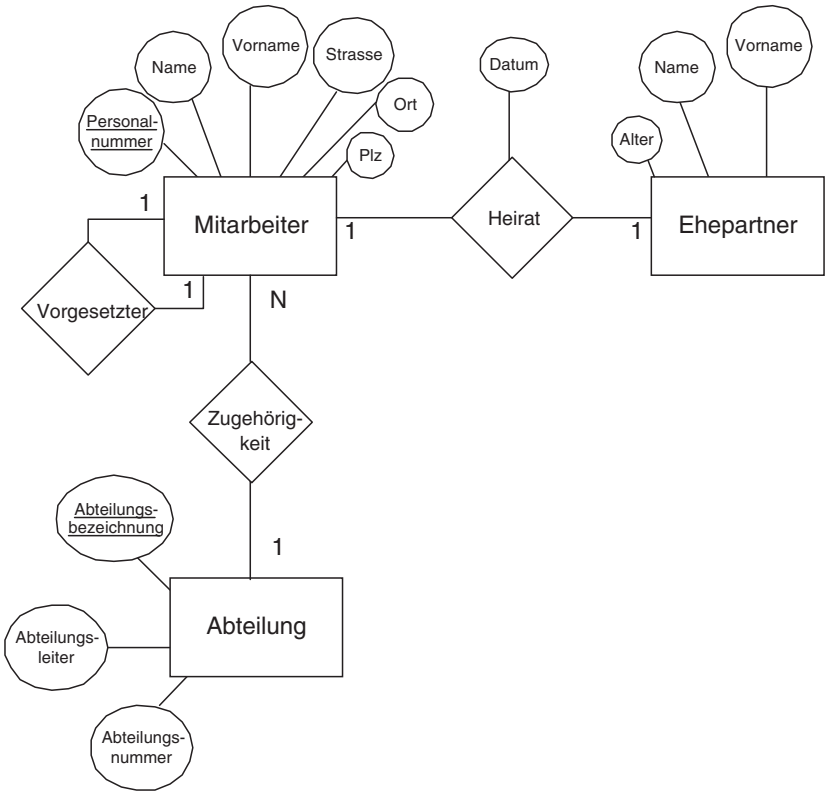


Abbildung 3.13: Darstellung von Beziehungen und Beziehungstypen

Einen Sonderfall bei Beziehungen stellen N:M-Beziehungen dar, da diese nicht ohne weiteres später in ein logisches Relationenmodell transformiert werden können. Ein N:M-Beziehungstyp stellt in der Regel einen weiteren Entitytypen dar. Meistens wird mit so einer Beziehung ein Konzept (z.B. ein Projekt, ein Konto) oder eine Transaktion (z.B. ein Kaufvertrag, eine Bestellung) dargestellt.

Dementsprechend müssen N:M-Beziehungen in zwei 1:N-Beziehungen umdefiniert werden. Dies geschieht grafisch, indem man einfach um die Beziehungsraute ein Rechteck zeichnet. Dadurch wird erkennbar, dass es sich nun um einen Entitytypen handelt, der aus einer N:M-Beziehung hervorgegangen ist.

Betrachten wir als Beispiel hierzu die Beziehung zwischen Kunde und Artikel, bei der es sich um eine N:M-Beziehung handelt. Im ersten Schritt wird diese Beziehung grafisch als N:M-Beziehung dargestellt. In einem zweiten Schritt erfolgt die Umdefinition der Beziehung in einen Entitytyp.

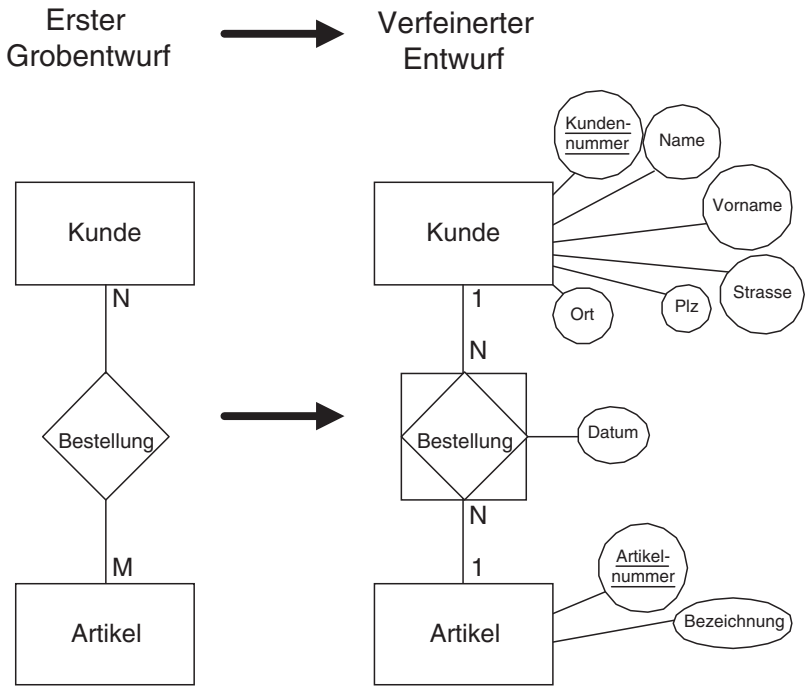


Abbildung 3.14: Umdefinition von N:M-Beziehungen

3.4.6 Fallbeispiel

Wir haben alle grafischen Elemente des ERM nach Chen kennengelernt und wollen uns nun ansehen, wie man von der Analyse des Textbeispiels zu einem grafischen ERM kommt. In der Regel geht man in sechs Schritten vor:

1) Identifizieren der Entitytypen und Beziehungen;
2) erster Grobentwurf mit Entitytypen und Beziehungen;
3) Super-, Subtypen modellieren;
4) Entwurf durch Umdefinition von N:M-Beziehung verfeinern und auf eventuell zusätzliche Beziehungen überprüfen (siehe Beziehung Bestellung zwischen Kunde und Artikel, Abb. 3.14);
5) Attribute hinzufügen;
6) Schlüsselattribute festlegen bzw. ggf. neue Schlüsselattribute hinzufügen.

Im ersten und zweiten Schritt werden Entitytypen und Beziehungen nur grob skizziert, um vor allem die Beziehungen zwischen Entitytypen zu erkennen und mit dem Kunden durchzusprechen. Nachdem der erste Entwurf als korrekt bewertet wurde, wird das Modell zunächst auf mögliche Super- bzw. Subtypen analysiert. Danach werden alle N:M-Beziehungen in jeweils zwei 1:N-Beziehungen umdefiniert und für die Beziehung ein Entitytyp gebildet. Im fünften Schritt überlegt man, welche Attribute den

jeweiligen Entitytypen beschreiben und im sechsten Schritt, welche dieser Attribute als Schlüssel in Frage kommen oder ob eventuell ein künstlicher Schlüssel verwendet werden soll.

Die folgende Grafik zeigt noch einmal die einzelnen Schritte auf. Zur besseren Übersicht werden im Folgenden nicht alle Entitytypen und Beziehungen unseres Beispiels dargestellt.

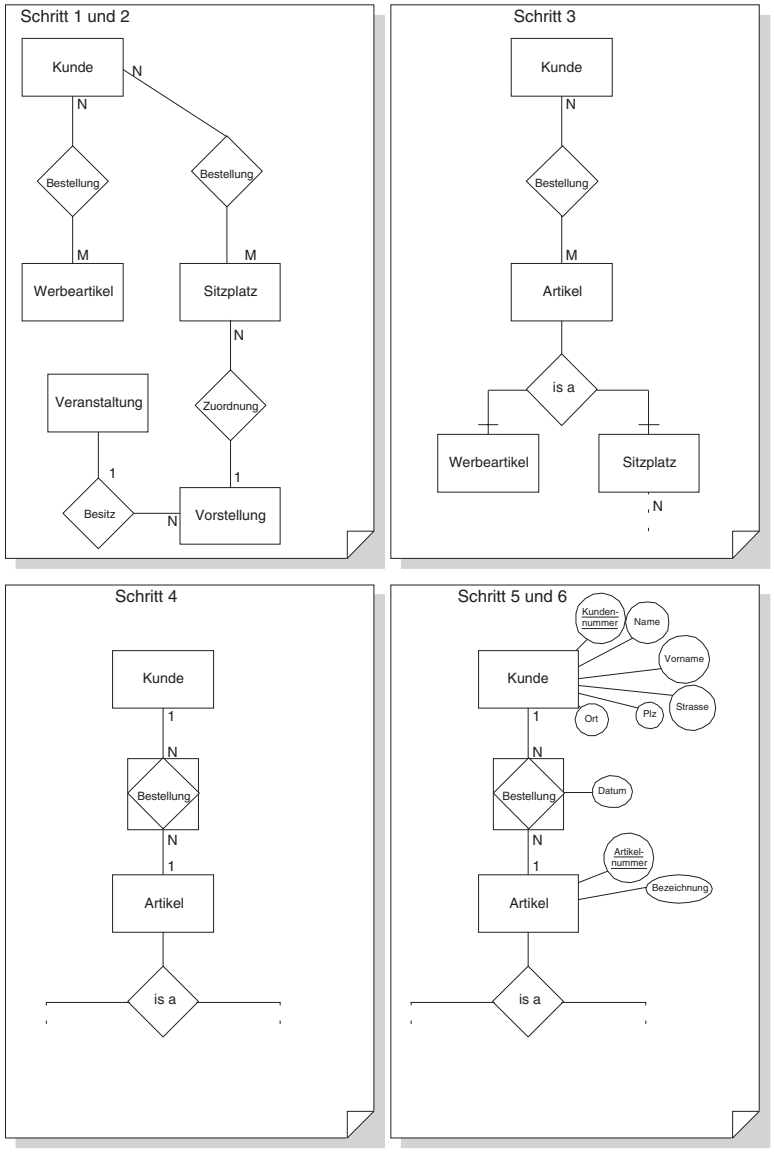


Abbildung 3.15: Sechs Schritte bis zum ERM nach Chen

Damit sieht das komplette ERM nach Chen für unser Fallbeispiel wie folgt aus:

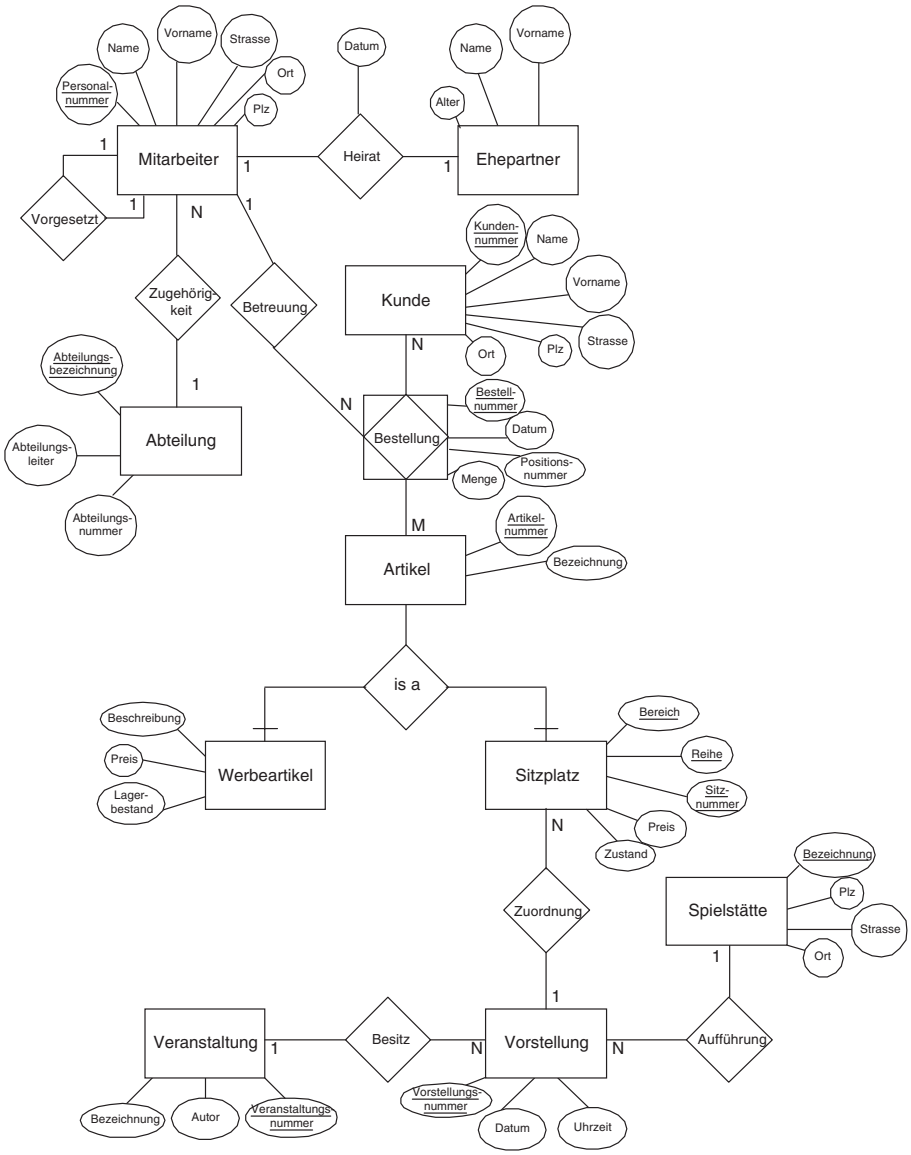


Abbildung 3.16: ERM nach Chen für unser Fallbeispiel

3.5 »Entity-Relationship-Modell« nach Barker

3.5.1 Grundlagen

Das ERM nach Barker stammt ursprünglich von der britischen Unternehmensberatung CACI und wurde von Richard Barker (vgl. hierzu [Barker90]) weiterentwickelt. Gegenüber dem ERM von Chen ist es übersichtlicher und teilweise aussagekräftiger. Dass es heute in der Praxis so weit verbreitet ist, liegt u.a. an der DV-technischen Unterstützung durch die Firma Oracle mit deren CASE-Tool.

3.5.2 Geschäftsobjekte

Geschäftsobjekte bzw. Entitytypen werden im ERM nach Barker durch abgerundete Rechtecke dargestellt.

Abbildung 3.17: Darstellung von Entitytypen

3.5.3 Sub- bzw. Supertypen

Subtypen werden innerhalb eines Supertyps als abgerundetes Recheck dargestellt.

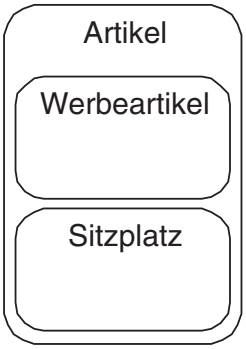


Abbildung 3.18: Darstellung von Sub- bzw. Supertypen

3.5.4 Attribute und Schlüssel

Attribute werden innerhalb des Rechtecks für den Entitytypen dargestellt. Ist der Wert eines Attributes für ein Entity optional, so wird dies durch einen offenen Kreis vor dem Attributnamen angezeigt. Ist das Attribut dagegen obligatorisch (engl. mandatory), so ist der Kreis geschlossen. Um Attribute als Teil des Primärschlüssels zu kennzeichnen,

erscheint vor dem Attributnamen das Nummernzeichen (#). Da der Primärschlüssel niemals optional sein kann, sondern immer angegeben werden muss, wird hier auf einen geschlossenen Kreis verzichtet.

Anstelle eines offenes Kreises wird häufig auch der Buchstabe 'o', anstatt eines geschlossenen Kreises das Zeichen '*' verwendet. Im Folgenden schließen wir uns dieser Konvention an.

Abbildung 3.19: Darstellung von Attributen

3.5.5 Beziehungen und Beziehungstypen

Beziehungen werden ausschließlich durch Verbindungslinien dargestellt. Dabei wird die Linie als zweigeteilt betrachtet. Je nachdem, in welchem Entitytyp die Linie endet, wird sie gestrichelt oder als geschlossene Linie gezeichnet. Eine gestrichelte Linie zwischen zwei Entitytypen zeigt an, dass die Kardinalität dort optional ist, wo die gestrichelte Linie endet. So genannte »Krähenfüsse« (»Crows foot«) an einem Entitytyp kennzeichnen die Kardinalität für »viele«. Eine einfache Linie gibt die Kardinalität für »genau eins« an. Im Gegensatz zum Modell von Chen hat Barker sich ausführlich mit der Vergabe von Namenskonventionen auseinandergesetzt. Die Bezeichnung einer Beziehung wird aus der Sichtweise jedes Entitytyps zweimal dargestellt, entsprechend werden hier Verben statt Substantive verwendet.

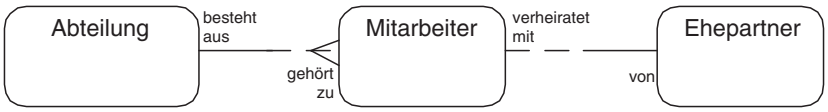


Abbildung 3.20: Darstellung von Beziehungen und Beziehungstypen

Für die Vergabe der Beziehungen hat Barker sich eine bestimmte Systematik überlegt, die zur sprachlichen Überprüfung der Beziehung herangezogen werden kann. Die allgemeine Satzstruktur für eine Beziehung wird ausgedrückt durch:

Jeder A (**muss** | **kann**) (**genau einem** B | **einen oder mehreren** B) R,

wobei A und B die Bezeichnung der Entitytypen darstellen und R die Beschreibung der Beziehung. Das Wort »muss« wird bei einer durchgezogenen Linie, also bei einer obligatorischen Beziehung, das Wort »kann« bei einer gestrichelten Linie, also einer optionalen Beziehung verwendet. Die Phrase »genau einem B« wird bei einer einzigen Linie, die Phrase »einen oder mehreren B« bei den Krähenfüssen verwendet.

Betrachten wir hierzu das Beispiel der Beziehung zwischen Abteilung zu Mitarbeiter, so ergibt sich:

Jede Abteilung **muss** aus **einem oder mehreren** Mitarbeitern bestehen.
Jeder Mitarbeiter **kann** zu **genau einer** Abteilung gehören.

Dabei muss der Satz natürlich noch einmal grammatikalisch angepasst werden. Leider ist aufgrund der Grammatik in der deutschen Sprache die Verwendung dieser Satzstruktur nicht systematisch anwendbar, dennoch kann sie zur Validierung von Beziehungen dienen.

Daneben kennt das ERM nach Barker das Konzept der Aggregation. Aggregation beschreibt Beziehungen der Form »besteht aus«. Um solche Zusammenhänge grafisch darzustellen, erscheint ein kleiner senkrechter Strich vor dem Entitytypen, der den Primärschlüssel übernehmen soll. In unserem Beispiel haben wir eine Aggregation zwischen Veranstaltung und Vorstellung vorliegen. Eine Vorstellung existiert nicht ohne eine Veranstaltung, entsprechend haben wir hier die Beziehung »besteht aus« vorliegen. Ein Entitytyp stellt eine Aggregation des anderen Entitytypen dar, wenn dieser ohne den anderen Entitytypen nicht existieren kann (Vorstellung kann nicht ohne eine Veranstaltung existieren).

Bezogen auf unser Beispiel von Abb. 3.19 sieht das Ergebnis damit wie folgt aus:

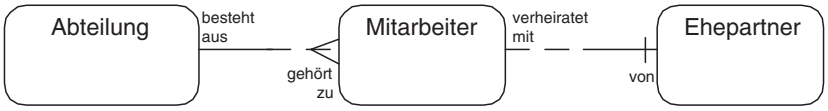


Abbildung 3.21: Berücksichtigung der Primärschlüssel bei 1:1- und 1:N-Beziehungen

3.5.6 Fallbeispiel

Wir haben alle grafischen Elemente des ERM nach Barker kennen gelernt und wollen uns nun ansehen, wie man von der Analyse des Textbeispiels zu einem grafischen ERM kommt. In der Regel geht man in sechs Schritten vor:

1) Identifizieren der Entitytypen und Beziehungen;
2) ersten Grobentwurf mit Entitytypen und Beziehungen erstellen;
3) Super-, Subtypen modellieren;
4) Entwurf durch Umdefinition von N:M-Beziehung verfeinern und auf eventuell zusätzliche Beziehungen überprüfen (siehe Beziehung Kunde – Artikel);
5) Aggregationen modellieren;
6) Attribute hinzufügen und festlegen, ob Attribut optional oder obligatorisch;
7) Schlüsselattribute festlegen oder hinzufügen.

Gegenüber der Vorgehensweise des ERM nach Chen muss bei dem ERM nach Barker zusätzlich festgelegt werden, wo der Primärschlüssel eines Entitytyps dem Primärschlüssel des anderen Entitytyps zugeordnet wird. Daneben ist festzulegen, inwiefern der Wert eines Attributes für ein Entitytyp zwingend vorhanden sein muss oder nicht (Schritt 6).

Die folgende Grafik zeigt noch einmal die einzelnen Schritte auf. Zur besseren Übersicht werden im Folgenden nicht alle Entitytypen und Beziehungen unseres Beispiels dargestellt.

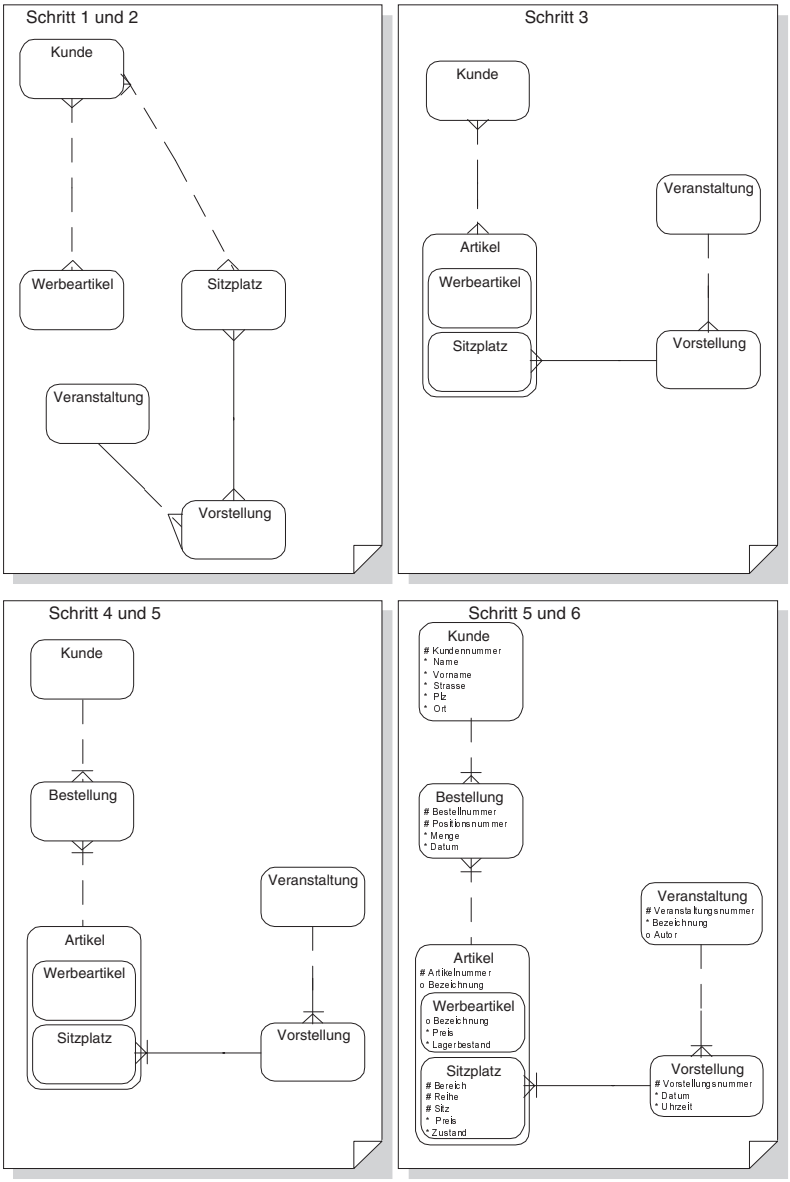


Abbildung 3.22: Sieben Schritte bis zum ERM nach Barker

Damit sieht das komplette ERM nach Barker für unser Fallbeispiel wie folgt aus:

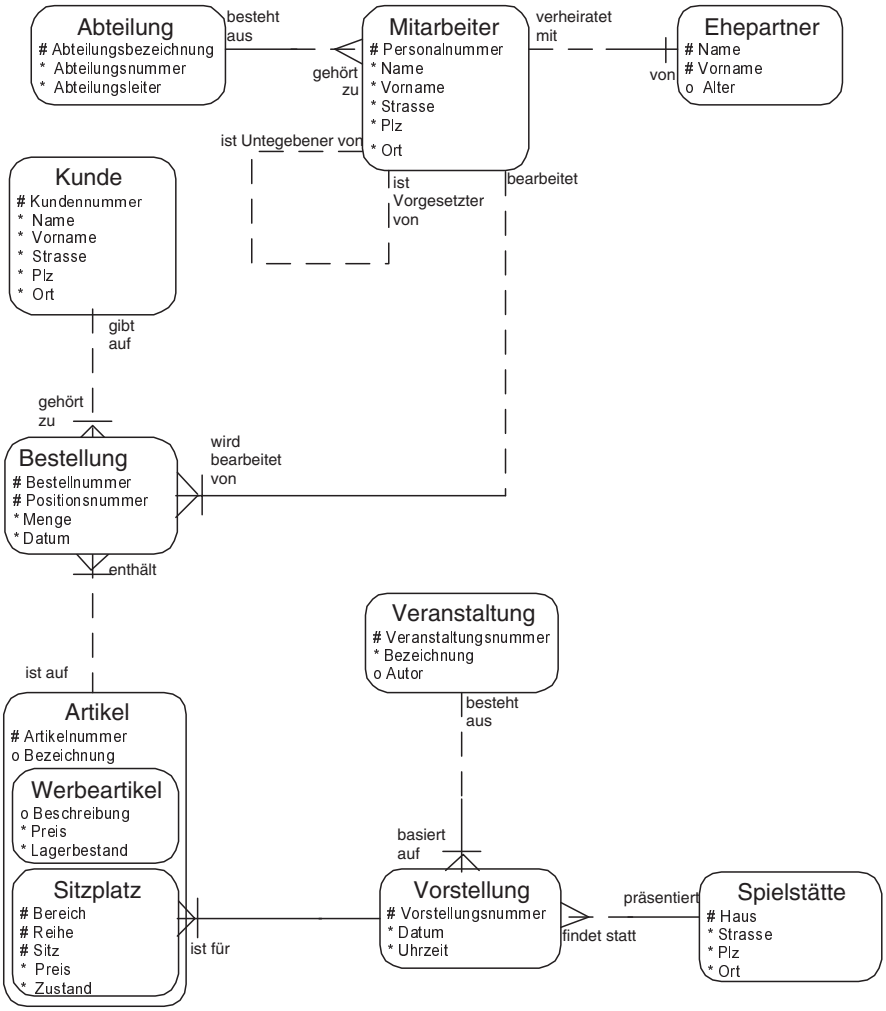


Abbildung 3.23: ERM nach Barker für unser Fallbeispiel

3.6 »Unified Modeling Language«

3.6.1 Grundlagen

Die »Unfied Modeling Language« (UML) ist primär keine grafische Notation zur Datenmodellierung. Eigentlich dient sie zur »Objektmodellierung« und damit hauptsächlich zur Modellierung von Anwendungen bzw. Funktionsmodellen. Ein Objekt

beinhaltet neben seinen Attributen, und damit den Daten, die es beschreibt, auch ein Verhalten in Form von Methoden. So könnte ein Objekt des Objekttyps »Sitzplatz« neben seinen Attributen auch Methoden zum Bearbeiten eines Sitzplatzes beinhalten, z.B. ReservierePlatz, StornierePlatz usw.

Die UML besteht aus unterschiedlichen Diagrammtypen. So beinhaltet eine grafische Notation der UML so genannte »Anwendungsfälle« (»Use Cases«). Diese dienen dazu, einen funktionalen Überblick über ein Geschäftsumfeld zu erhalten und sind innerhalb des Phasenkonzeptes der 1. und teilweise der 2. Phase zuzuordnen.

Der wichtigste Diagrammtyp der UML ist jedoch das Klassenmodell (»Class Modell«), um die logische Struktur eines Anwendungssystems zu modellieren. Zur Erstellung eines Datenmodells auf Basis der UML beschäftigen wir uns hier deshalb nur mit einem Teil der grafischen Notation eines Klassenmodells.

Die UML hat sich heutzutage als Standard zur objektorientierten Modellierung von Anwendungssystemen durchgesetzt. Bevor die UML als Standard von der »Object Management Group« (OMG) 1997 verabschiedet wurde, gab es im Bereich der objektorientierten Modellierung drei sich ähnelnde Notationen zur Modellierung von James Rumbaugh, Grady Booch und Ivar Jacobson. Um ihre Notationen zu vereinheitlichen und einen Standard zu schaffen, entwickelten die drei »Amigos« (wie sie von der objektorientierten »Gemeinde« bezeichnet werden) die UML. Heute arbeiten Rumbaugh, Booch und Jacobsen bei der Firma Rational Rose, die Software-Tools zur Modellierung von Anwendungen mit der UML erstellt. Inzwischen gibt es einen weiteren Vorschlag der drei »Amigos« zur Erweiterung der UML um grafische Notationen an die OMG. Diese Erweiterung betrifft vor allem die Umsetzung eines Klassenmodells in ein logisches Modell als Voraussetzung zur Erstellung einer relationalen Datenbank.

Wir wollen uns hier nicht mit dieser Erweiterung beschäftigen, sondern betrachten Klassenmodelle so, wie sie zur Modellierung von Anwendungssystemen verwendet werden. Dabei geht es nicht so sehr um die Vollständigkeit der Notation, sondern vielmehr um ein Kennenlernen der UML, bezogen auf die Datenmodellierung. Literaturhinweise zur UML sind im Anhang aufgeführt.

3.6.2 Geschäftsobjekte

Die UML dient zur Objektmodellierung. Entsprechend werden Geschäftsobjekte als Objekte bezeichnet und Objekttypen als Klassen. Eine Klasse ist also eine »Schablone« für ein Objekt, genau wie ein Entitytyp. Im Gegensatz zum Entitytypen definiert eine Klasse jedoch nicht nur die Attribute eines Objektes, sondern auch das Verhalten des Objektes in Form von Methoden. Klassen werden deshalb in der UML durch ein Rechteck repräsentiert, das dreigeteilt ist. Im oberen Bereich steht der Name der Klasse, im mittleren Teil die Attribute und im unteren Bereich die Methoden der Klasse. Eine Klasse hat private Attribute und Methoden, auf die nur die Klasse selbst zugreifen darf und öffentliche Attribute, auf die auch andere Klassen zugreifen können. Die Bezeichner von Klassen und Attributen dürfen keine Leerzeichen enthalten. So ist »Leiter der Abteilung« kein korrekter Bezeichner, statt dessen müsste man diese Klasse »LeiterDerAbteilung« oder besser »Abteilungsleiter« nennen.

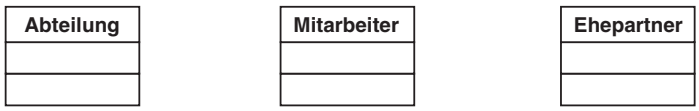


Abbildung 3.24: Darstellung von Klassen

3.6.3 Sub- bzw. Supertypen

Supertypen und Subtypen werden jeweils als Klasse dargestellt. Die Beziehung eines Supertyps zu seinen Subtypen durch eine »ist ein«-Beziehung wird in Form eines Dreiecks dargestellt.

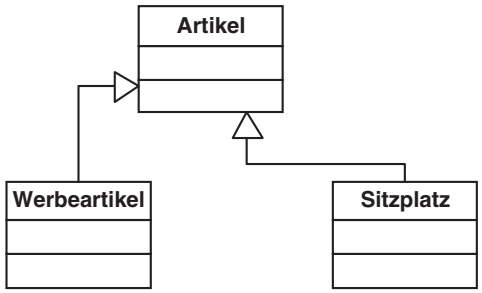


Abbildung 3.25: Darstellung von Sub- bzw. Supertypen

3.6.4 Attribute und Schlüssel

Attribute werden innerhalb des Rechtecks einer Klasse unterhalb des Klassenbezeichners aufgeführt. Das relationale Datenmodell sieht es nicht vor, Attribute als privat oder öffentlich zu kennzeichnen. Da die UML jedoch zur objektorientierten Modellierung dient, können hier die Attribute in der UML als nach außen sichtbar (public), nur sichtbar für Subtypen (protected) oder versteckt (private) gekennzeichnet werden. »public«-Attribute oder Methoden erhalten ein »+«-Zeichen vor ihrer Bezeichnung, »private«-Elemente ein »-«-Zeichen und »protected«-Elemente das »#«-Zeichen. Schlüssel existieren in der UML nicht, da in der UML ein Objekt bzw. eine Instanz eindeutig durch ihre Beziehungen zu anderen Objekten identifiziert werden kann. Dennoch ist es möglich, durch so genannte Sterotype anzugeben, welche Attribute als Schlüssel innerhalb einer relationalen Datenbank gelten sollen.

Sterotype dienen dazu, die Sprachelemente der UML zu erweitern, indem man einfach bestimmten Begriffen innerhalb eines Diagramms eine Semantik zuweist. Diese Begriffe werden durch Guillemets (französische Anführungszeichen), »<<« und »>>«, begrenzt. Für einen Primärschlüssel definieren wir deshalb den Sterotyp »<<PK>>«, der dazu dienen soll, Attribute als Primärschlüssel (»Primary Key«) zu kennzeichnen.

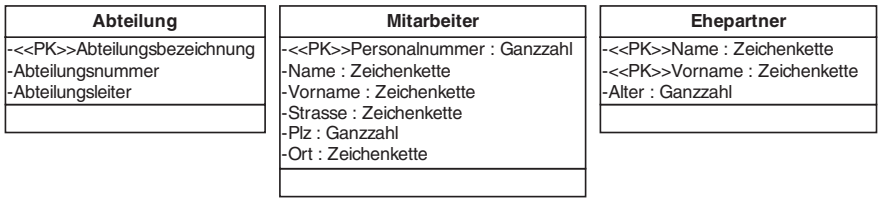


Abbildung 3.26: Darstellung von Sub- bzw. Supertypen

Auf Methoden wurde in dieser Darstellung verzichtet. Wir werden jedoch ab Kapitel 6 sehen, dass in Datenbanken auch Methoden über so genannte »User Defined Types«, »Trigger« und »Stored Procedures« gespeichert und ausgeführt werden können.

3.6.5 Beziehungen und Beziehungstypen

Beziehungen, in der UML Assoziationen genannt, werden durch Linien und Symbole an den Enden der Linien beschrieben.

1:1- und 1:N-Beziehungen werden durch eine einfache Linie gekennzeichnet, N:M-Beziehungen durch so genannte Assoziationsklassen, indem der Beziehung automatisch eine weitere Klasse zugeordnet wird. Ein gesonderter Schritt zur Umdefinition von N:M-Beziehungen ist deshalb nicht notwendig.

Daneben wird das Symbol einer Raute am Ende einer Linie verwendet, um Assoziationen der Form »ist Teil von« und »besteht aus« (Aggregation) darzustellen. In unserem Beispiel haben wir diesen Fall zwischen Veranstaltung und Vorstellung vorliegen. Eine Vorstellung existiert nicht ohne eine Veranstaltung, entsprechend haben wir hier die Beziehung »besteht aus«.

Eine Beziehung der Art »ist Teil von« existiert zwischen Mitarbeiter und Ehepartner. Ein Ehepartner ist Teil eines Mitarbeiters, gehört also zum Objekttyp »Mitarbeiter«. Ähnlich wäre es, wenn wir z. B. für Kunden mehrere Adressen zulassen würden. In diesem Fall müssten wir zusätzlich eine Klasse »Adresse« definieren. Die Klasse »Adresse« wäre dann »Teil von« der Klasse »Kunde«. Diese Art der Beziehung wird durch eine Raute dargestellt.

Ist die Raute gefüllt, so handelt es sich um eine zusammengesetzte Aggregation. In diesem Fall können die Objekte, aus denen sich das aggregierte Objekt zusammensetzt, keine weiteren Aggregationen mit anderen Objekten bilden. Hierzu ein Beispiel: Ein Gebäude besteht aus mehreren Räumen, ein Raum kann ohne das Gebäude nicht existieren. Ein ganz bestimmter Raum kann nur einem Gebäude zugeordnet sein und kann daher nicht für ein weiteres Gebäude verwendet werden.

Kardinalitäten werden mit Zeichen der Form »Minimum .. Maximum« dargestellt. Minimum und Maximum stehen dabei für die Anzahl der in Beziehung stehenden Objekte. Ein »*« steht für beliebig viele Objekte, eine 0 für optional. Eine Kardinalität von »1..1« bedeutet dementsprechend »genau einer«. Zur Verdeutlichung ein Beispiel: Eine Abteilung besteht aus einem oder beliebig vielen Mitarbeitern (1..*). Die 1 stellt

dabei das Minimum dar und das Zeichen »*« das Maximum. Würde man dagegen festlegen, dass eine Abteilung aus maximal 20 Mitarbeitern bestehen darf, so würde man 1..20 schreiben.

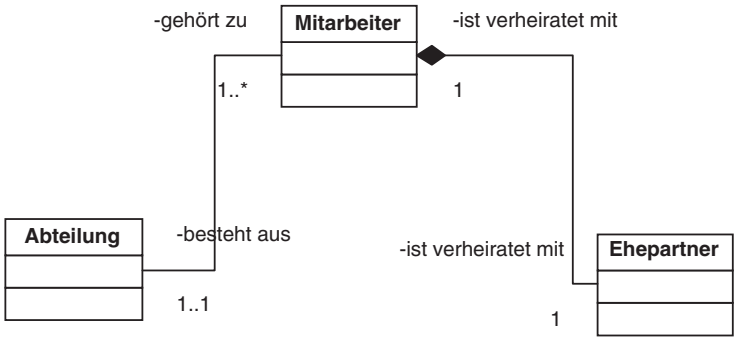


Abbildung 3.27: Darstellung von 1:1- und 1:N-Beziehungstypen

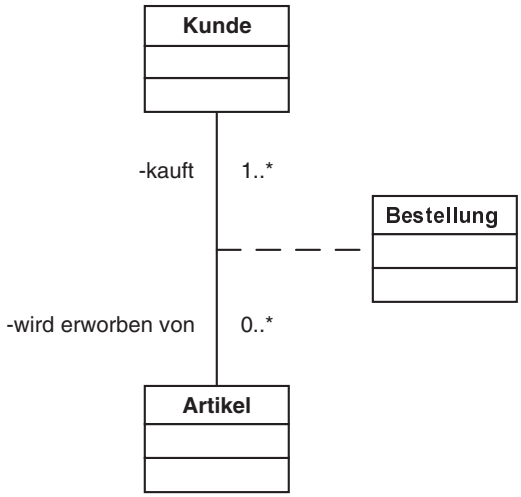


Abbildung 3.28: Darstellung von N:M-Beziehungstypen

3.6.6 Fallbeispiel

Wir haben die wichtigsten grafischen Elemente eines Klassendiagramms der UML kennen gelernt und wollen uns nun ansehen, wie man von der Analyse des Textbeispiels zu einem Klassendiagramm kommt. In der Regel geht man in sieben Schritten vor:

1) Identifizieren der Klassen und Assoziationen;
2) ersten Grobentwurf mit Klassen und Assoziationen erstellen;
3) Super-, Subtypen modellieren;

4) Aggregationen modellieren;
5) Attribute hinzufügen und festlegen, ob Attribut optional oder obligatorisch;
6) Schlüsselattribute festlegen oder hinzufügen;
7) Festlegen des Verhaltens der Klasse in Form von Methoden.

Gegenüber der Vorgehensweise des ERM wird bei der UML bei N:M-Beziehungen keine Umdefinition der Beziehung vorgenommen, da hier bereits bei der Modellierung von N:M-Beziehung eine eigene Klasse hinzugefügt wird (siehe Abb. 3.28). Neben dem eigentlichen Identifizieren und Modellieren der Daten sind jedoch gerade die Methoden von Klassen wichtig, die in einem sechsten Schritt mit der UML spezifiziert werden können.

Damit sieht das Klassendiagramm mit der UML für unser Fallbeispiel wie folgt aus:

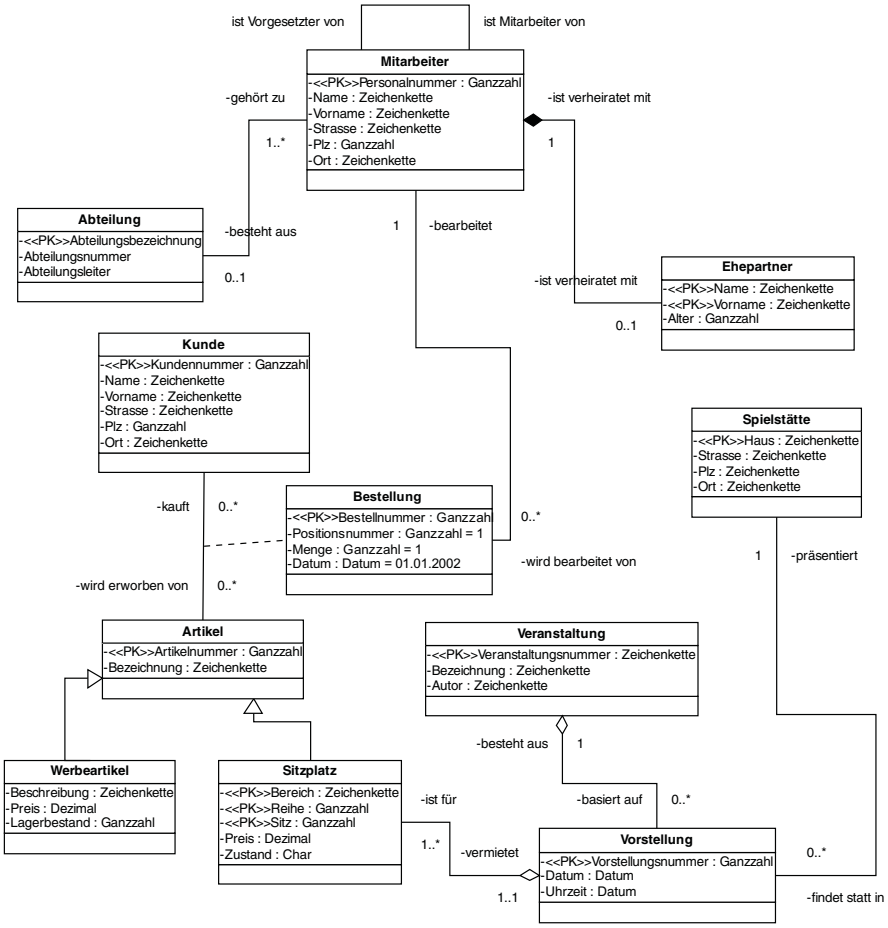


Abbildung 3.29: UML-Klassendiagramm für unser Fallbeispiel

3.7 Zusammenfassung

In diesem Kapitel haben wir kennen gelernt, wie man von einer Problembeschreibung zu einem Datenmodell kommt. Bevor mit der eigentlichen Datenmodellierung begonnen wird, stellt man einen Projektplan auf, der beinhaltet, in welchen Schritten man vorgeht. Als Grundlage verwendet man ein Phasenkonzept, das sozusagen als Vorlage zur Erstellung eines Projektplanes dient. Dabei haben wir auch gesehen, dass die Datenmodellierung selbst nur einen Teil des Entwicklungsprozesses von der Problembeschreibung zur Problemlösung darstellt.

Im ersten Schritt lernt man das Geschäftsumfeld, in dem Probleme auftreten, kennen. Dazu spricht man mit Personen, sichtet Dokumente und sieht sich eventuell vorhandene Anwendungssysteme an.

Daraufhin identifiziert und klassifiziert man die gefundenen Informationen und abstrahiert sie so weit, dass man so genannte Geschäftsobjekte und deren Beziehungen herausfindet. Geschäftsobjekte und deren Beziehungen werden in Datenmodellen dargestellt, die zum einen zur Kommunikation mit dem Kunden und andererseits als Grundlage zur Umsetzung in eine Datenbankstruktur dienen.

Zur Erstellung von Datenmodellen haben sich zwei Modellierungstechniken in der Praxis durchgesetzt, das »Entity-Relationship-Modell« (ERM) und die »Unified Modeling Language« (UML). Anhand eines Fallbeispiels haben wir ein Datenmodell in Form eines ERM nach Chen, eines ERM nach Barker und mit der UML erstellt und dabei die Unterschiede zwischen den Modellierungstechniken kennen gelernt.

Dieses Kapitel soll lediglich als Einführung in die Datenmodellierung dienen. Zur detaillierten Auseinandersetzung mit einer der Modellierungstechniken, empfehle ich die Literaturliste am Ende des Buches.

Zum Schluss sei noch erwähnt, dass das ERM und die UML nicht immer gewährleisten, dass ein in sich logisches Datenmodell entsteht. So enthält das Datenmodell unseres Fallbeispiels einen Teil, der noch nicht »sauber« modelliert ist. Wir werden hierauf in Kapitel 6 zurückkommen und dort eine Methode, nämlich die Normalisierung kennen lernen, die das erstellte Modell noch einmal validiert. Die Normalisierung wird allerdings auf das logische Relationenmodell angewendet. Bevor wir uns deshalb mit der Normalisierung beschäftigen, gehen wir im nächsten Kapitel auf das am weitesten verbreitete logische Modell für relationale Datenbanken ein, das Relationenmodell.

3.8 Aufgaben

Wiederholungsfragen

1) Aus welchen Phasen besteht das in diesem Buch vorgestellte Phasenkonzept?
2) In welchem Bezug steht das Phasenkonzept zum Projektplan?
3) Wozu dient der konzeptionelle Entwurf?
4) Worin besteht der Unterschied zwischen konzeptionellem und logischem Entwurf?

5) Welcher Phase sind die Modellierungstechniken des ERM und der UML zuzuordnen?
6) Wie wird eine N:M-Beziehung im ERM nach Chen, ERM nach Barker und mit der UML grafisch dargestellt?

Übungen

1) Ein Mitarbeiter bestellt bei einem Bürolieferanten Büromaterial. Welche der folgenden Attribute sind dem Geschäftsobjekt »Bürolieferant« zuzuordnen und welches Attribut würden Sie als Primärschlüssel für das Geschäftsobjekt festlegen?
   Name des Mitarbeiters, Lieferantennummer, Bestellnummer, bestellte Artikel, Bezeichnung der Firma des Mitarbeiters, Bezeichnung des Lieferanten, Adresse des Lieferanten.
2) Ein Online-Spielzeugwarengeschäft verkauft über das Internet Spielzeug. Ein Kunde gibt mehrere Bestellungen auf. Eine Bestellung enthält mehrere Artikel bzw. jeder Artikel kann auf verschiedenen Bestellungen erscheinen. Die Bestellungen werden von Mitarbeitern bearbeitet.
   Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.
3) Für ein Geo-Informationssystem soll ein Datenmodell erstellt werden. Auf der Erde gibt es mehrere Staaten. Jeder Staat besteht wiederum aus mehreren Regionen und jeder Staat hat genau einen Staatsherrscher. Ein Staatsherrscher kann entweder ein demokratisch gewählter Präsident oder ein Diktator sein. Merkmale eines Staatsherrschers sind sein Name und sein Geburtsdatum. Merkmale eines demokratisch gewählten Präsidenten sind seine Parteizugehörigkeit und die Anzahl der Wählerstimmen, die er bei der letzten Wahl erhalten hat. Merkmale eines Diktators sind sein geschätztes Vermögen.
   Ein Staat hat eine Hauptstadt, die, genau wie ein Staat, gekennzeichnet ist, durch die Anzahl der Bewohner, eine Gesamtfläche in Quadratmetern und durch eine Bevölkerungsdichte. Entsprechend existieren in jeder Region weitere Orte mit gleichen Kennzahlen.
   Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.
4) Eine Hotelkette hat mehrere Hotels in verschiedenen Orten. Jedes Hotel wird von einem Hotelmanager geleitet. Jedes Hotel hat mehrere Doppel- und Einzelzimmer und verschiedene Konferenzräume.
   In jedem dieser Räume befindet sich Inventar, wie z. B. Stühle, Betten, Fernseher. Ein Doppelzimmer kostet pro Übernachtung 220 Euro, ein Einzelzimmer 150 Euro und jeder Konferenzraum pro Quadratmeter 10 Euro. Die Räume werden sowohl an Privatpersonen vermietet, als auch an Firmen, sofern es sich um Konferenzräume handelt.
   Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.
5) In einer Arztpraxis arbeiten die Ärzte Hohlbein, Dürer und Grunewald. Hohlbein ist für Hals-/Nasen-/Ohrenkrankheiten zuständig, Dürer für innere Krankheiten und Grunewald für Hautkrankheiten.
   In die Arztpraxis kommen Patienten, die von den Ärzten, abhängig von ihrer Krankheit, behandelt werden. Jeder Patient wird mit Name, Vorname, Adresse und Krankenkasse registriert. Abhängig von der Krankheit, behandeln auch mehrere Ärzte einen Patienten.

Zu jeder Krankheit wird neben der Diagnose festgehalten, welcher Arzt diese behandelt und welche Kosten durch die Behandlung entstanden sind.
Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.

6) Im Finanzamt gibt es die Abteilungen »Betriebsprüfung«, »Gewerbesteuer« und »Einkommenssteuererklärungen«.
In der Abteilung »Einkommenssteuererklärungen« arbeiten die Finanzbeamten Muster, Mahlzahn, Hohlstein und Wichtig. Herr Wichtig ist der Abteilungsleiter. Ein Abteilungsleiter betreut mehrere Mitarbeiter, wobei ein Mitarbeiter immer genau einen Abteilungsleiter hat.
Muster bearbeitet die Anträge der Steuerpflichtigen, deren Nachname mit »A-H« beginnt, Mahlzahn von »I-M« und Hohlstein von »N-Z«.
Pro Einkommenssteuererklärung gibt es genau einen Bearbeiter. Bei der Einkommensteuererklärung wird neben dem Namen, der Adresse und der Steuernummer des Steuerpflichtigen auch der Bearbeitungsstatus vermerkt.
Bei schwierigen Fällen übergeben die Bearbeiter die Einkommenssteuererklärung ihrem Vorgesetzten Herrn Wichtig und benachrichtigen ihn über den Teil, bei dem sie Probleme hatten. Herr Wichtig führt die Bearbeitung dann zu Ende.
Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.

7) Zur Verwaltung der Fussballspiele möchte der »Hamburger Fussballverband« (HFV) eine Datenbank erstellen.
Der HFV besteht aus mehreren Mannschaften. Diese Mannschaften haben eine Bezeichnung und bis zu vier verschiedene Postadressen. Jede Mannschaft besteht aus mehreren Spielern. Jeder Spieler hat einen Vor- und Nachnamen, eine Adresse und eine bevorzugte Spielernummer. Jeder Spieler darf nur für eine einzige Mannschaft spielen.
Die Mannschaften spielen gegeneinander in einem Hin- und Rückspiel. Hin- und Rückspiele finden zu bestimmten Terminen und in einem bestimmten Ort statt. Ein Fussballspiel wird von einem Schiedsrichter und zwei Linienrichtern geleitet.
Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.

8) Die Unternehmensberatung »Software Consult AG« möchte ihre Projekte mit den Kunden in einer Datenbank verwalten. Ein Kunde beauftragt ein oder mehrere Projekte. Projekte werden von Mitarbeitern der Unternehmensberatung durchgeführt. Ein Mitarbeiter kann mehreren Projekten zugeordnet sein, und ein Projekt besteht in der Regel aus mehreren Mitarbeitern. Zusätzlich ist ein Mitarbeiter einer Abteilung zugeordnet. Für jedes Projekt gibt es einen Mitarbeiter, der Projektleiter für das Projekt ist.
Merkmale eines Projektes sind die Bezeichnung, Anfangs- und geplanter Fertigstellungstermin.
Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.

9) In einer Volkshochschule unterrichten die Dozenten Hansen, Meier und Oberdorf. Meier unterrichtet die Kurse »EDV Grundlagen« und »WinWord«, Hansen den Kurs »Kochen für Kinder« und Oberdorf die Kurse »Marketing« und »Controlling«. Zu jedem Kurs melden sich mehrere Teilnehmer an, die für diesen Kurs eine Kursgebühr zahlen müssen. Für jeden Kurs gibt es eine maximale Anzahl an Teilnehmern.
Entwerfen Sie ein Datenmodell nach dem ERM von Baker und nach der UML.

4 Datenbankentwurf – Vom »Bauplan« zur Datenbankstruktur

In Kapitel 4 sollen folgende Fragen geklärt werden:

- Wozu dient der logische Entwurf?
- Wie kommt man vom konzeptionellem Entwurf zum DV-technischen Entwurf (logisches Modell)?
- Welche Entwurfmethode gibt es für den logischen Entwurf von relationalen Datenbanken (Relationenmodell)?
- Wie kommt man vom Entity-Relationship-Modell (ERM) zum Relationenmodell?

4.1 Motivation

Betrachten wir zunächst wieder unser Fallbeispiel. Inzwischen sind sechs Wochen vergangen und Herr Dr. Fleissig von der Unternehmensberatung »Software Consult AG« hat das konzeptionelle Modell für »KartoFinale« erstellt. Er hat ein Entity-Relationship-Modell nach Baker erstellt und es so weit mit dem Kunden besprochen und immer wieder durchdiskutiert, dass er sich jetzt sicher ist, das Geschäftsumfeld von »KartoFinale«verstanden zu haben. Damit ist der erste Meilenstein des Projektplanes erfolgreich und »just-in-time«erledigt. Herr Fleissig, stolz auf seine Leistung, beginnt nun hochmotiviert mit der Umsetzung auf den Computer. Bevor er sich jedoch mit der Umsetzung des ERM auf den Computer beschäftigt, muss das ERM so umgewandelt werden, dass es überhaupt auf den Computer übertragen werden kann.

Zur Zeit hat Herr Fleissig ja nur eine Grafik und eine textuelle Beschreibung des Geschäftsumfeldes. Da er noch nicht weiß, welches relationale Datenbank-Management-System (RDBMS) später eingesetzt werden soll, muss er also ein Modell entwerfen, dass auf einen beliebigen Computer für ein beliebiges RDBMS übertragen werden kann.

Herr Fleissig kennt sich natürlich mit relationalen Datenbank-Management-Systemen aus und weiß, dass diese die Daten in Form von Tabellen speichern. Also entschließt er sich sein ERM so umzuwandeln, dass es in Form von Tabellen abgebildet werden kann. Zunächst überlegt er sich, jeden Entitytyp in genau eine Tabelle umzuwandeln. Damit hat er schon einmal die Daten der Entities abgebildet, indem er pro Zeile einer Tabelle einen Entity speichert.

Doch wie können Beziehungen in Tabellenform abgebildet werden? Um einen Bezug zwischen Tabellen herzustellen, erstellt er einfach in der einen Tabelle ein zusätzliches Attribut, das den Primärschlüssel der anderen Tabelle als Bezug speichert. Betrachten

wir hierzu die Beziehung zwischen Abteilung und Mitarbeiter. In Tabellenform aus dem ERM transformiert sieht das Ganze wie folgt aus:

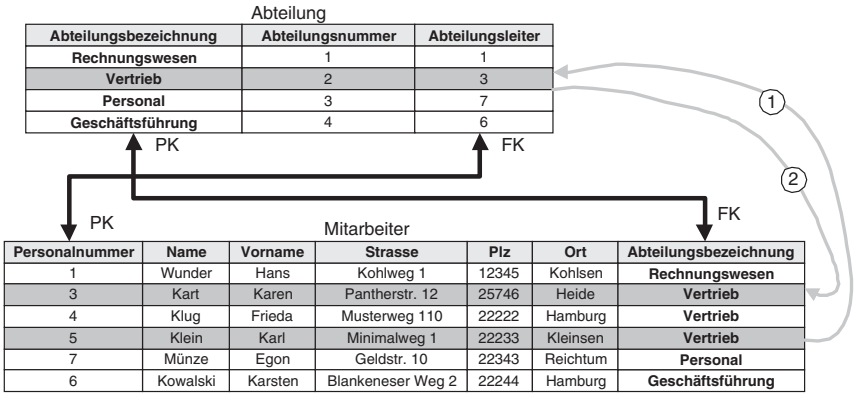


Abbildung 4.1: Tabellen und deren Beziehungen

Herr Fleissig hat den Primärschlüssel der Abteilung als Referenz in die Mitarbeitertabelle übernommen. Damit ist die 1:N-Beziehung zwischen Abteilung und Mitarbeiter abgebildet. Man kann damit sowohl herausbekommen, zu welcher Abteilung ein Mitarbeiter gehört und wer der entsprechende Abteilungsleiter ist. Umgekehrt kann man auch ermitteln, zu welcher Abteilung welche Mitarbeiter gehören.

Nehmen wir als Beispiel aus der Mitarbeitertabelle den Mitarbeiter Karl Klein. Herr Klein gehört zur Abteilung »Vertrieb«. Der Wert »Vertrieb« ist gleichzeitig Primärschlüssel in der Tabelle Abteilung. Sucht man in der Tabelle »Abteilung« nun den Eintrag für den Wert »Vertrieb« (1), so sieht man, dass der Abteilungsleiter dieser Abteilung die Personalnummer 3 hat. Entsprechend kann man in der Tabelle »Mitarbeiter« wieder nachsehen, um welchen Mitarbeiter es sich hier handelt, nämlich um Frau Kart (2).

Herr Fleissig erstellt die Tabellen in Textform. Dabei führt er jeweils die Bezeichnung der Tabelle und in Klammern die Attribute bzw. Spalten der Tabelle auf. Primärschlüssel stellt er unterstrichen dar und Referenzen jeweils kursiv. Für unsere Tabellen »Mitarbeiter« und »Abteilung« sieht das Ganze wie folgt aus:

Abteilung (Abteilungsbezeichnung, Abteilungsnummer, *Abteilungsleiter*)

Mitarbeiter (Personalnummer,Name,Vorname,Strasse, Plz, Ort, *Abteilungsbezeichnung*)

4.2 Relationenmodell

Im letzten Kapitel haben wir gesehen, dass man zunächst das Umfeld des zu lösenden Problemes analysieren und beschreiben muss. Als Ergebnis entsteht das konzeptionelle Modell, das häufig auch als Fachkonzept bezeichnet wird, da es ein Problem aus fachlicher und nicht technischer Sicht beschreiben soll. Bevor man sich nun an einen Computer setzt und dieses Fachkonzept auf diesen überträgt, muss ein Konzept zur Über-

tragung auf den Computer erstellt werden. Man spricht hier vom logischen Modell oder auch DV-Konzept, weil ein in sich logisches und mathematisch schlüssiges Konzept entstehen soll.

Bei der Datenmodellierung für relationale Datenbank-Management-Systeme hat sich hier das Relationenmodell durchgesetzt. Das Relationenmodell stammt von Dr. E. F. Codd und wurde erstmals 1970 in einem Artikel des Datenbankjournals des ACM (»Association for Computing Machinery«) beschrieben. Der Titel zu diesem Artikel lautet »A Relational Model of Data for Large Shared Data Banks«. Das Relationenmodell beschreibt die theoretischen mathematischen Grundlagen, die einem relationalen Datenbank-Management-System zugrunde liegen. Entsprechend ist ein logisches Modell auf Basis des Relationenmodells auch sehr einfach auf ein RDBMS zu übertragen. Da das logische Datenbankmodell auf dem Relationenmodell basiert, ist es unabhängig von einem ganz bestimmten RDBMS und einer ganz bestimmten Computerplattform.

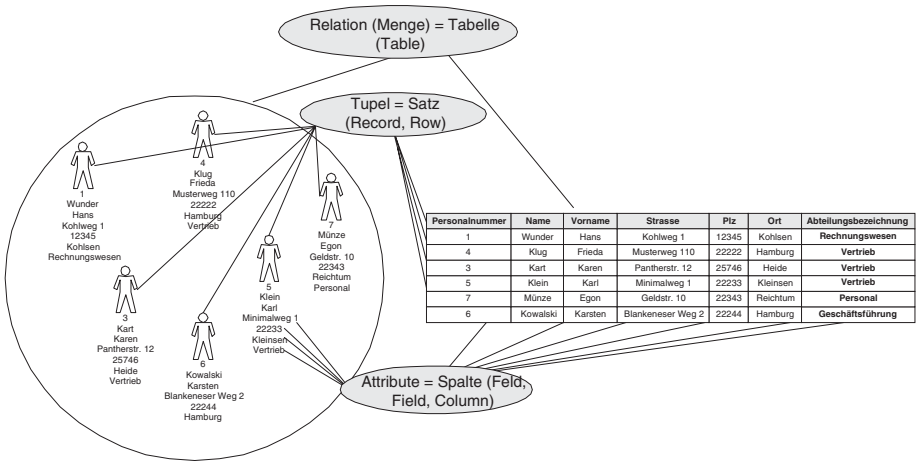
Im Folgenden werden wir uns pragmatisch mit dem Relationenmodell beschäftigen, ohne dabei zu theoretisch und mathematisch zu werden. Es werden nur die Aspekte des Relationenmodells beschrieben, die für die Umsetzung eines Datenmodells notwendig sind. Wir werden uns dabei vor allem darauf konzentrieren, wie man aus dem konzeptionellen Modell (ERM oder UML-Modell) ein Relationenmodell erstellt.

4.3 Grundlagen des Relationenmodells

Bevor wir uns mit der eigentlichen Umsetzung des konzeptionellen Modells in ein Relationenmodell beschäftigen, wollen wir kurz die wichtigsten Grundlagen des Relationenmodells kennen lernen. Das Relationenmodell basiert mathematisch auf der Mengenlehre und der Mengenalgebra (Relationentheorie).

Objekttypen werden dabei in Form von Mengen (Relationen) dargestellt. Jede Relation wiederum besteht aus mehreren Objekten bzw. Objektinstanzen, die gleichartig sind und durch gleiche Attribute beschrieben werden. So gehört in unserem Fallbeispiel Herr Egon Münze zu der Relation »Mitarbeiter« und besitzt die Attribute »Personalnummer«, »Name«, »Vorname«, »Strasse«, »Plz« und »Ort«. Eine Objektinstanz wird im Relationenmodell als Tupel bezeichnet. Da es sich um eine Menge handelt, existiert keine Reihenfolge bzw. Sortierung der Attribute oder auch der Tupel selber. Der Einfachheit halber werden Relationen nicht als Mengen gezeichnet, sondern in Form von 2-dimensionalen Tabellen dargestellt. Entsprechend werden im allgemeinen Sprachgebrauch die Begriffe Relation, Tupel und Attribut selten verwendet. Man spricht eher von Tabelle (Table), Satz (Record oder Row) und Spalte (Feld, Field, Column).

Betrachten wir hierzu die Menge der Mitarbeiter aus unserem Fallbeispiel, so haben wir sieben Tupel bzw. Sätze.



Personalnummer	Name	Vorname	Strasse	Plz	Ort	Abteilungsbezeichnung
1	Wunder	Hans	Kohlweg 1	12345	Kohlsen	**Rechnungswesen**
4	Klug	Frieda	Musterweg 110	22222	Hamburg	**Vertrieb**
3	Kart	Karen	Pantherstr. 12	25746	Heide	**Vertrieb**
5	Klein	Karl	Minimalweg 1	22233	Kleinsen	**Vertrieb**
7	Münze	Egon	Geldstr. 10	22343	Reichtum	**Personal**
6	Kowalski	Karsten	Blankeneser Weg 2	22244	Hamburg	**Geschäftsführung**

Abbildung 4.2: Relation »Mitarbeiter«

Jeder dieser Sätze ist eindeutig, d.h. kein Satz kommt mehrfach vor und jede Spalte hat einen eindeutigen Namen. Die Werte einer Spalte sind von der gleichen Art und haben einen Wertebereich (Domain). Der Wertebereich der Spalte Plz darf z.B. nur in Deutschland gültige Postleitzahlen enthalten. Der Wertebereich der Spalte Abteilungsbezeichnung darf dagegen nur Werte enthalten, die in der Tabelle Abteilung als Primärschlüssel auftreten.

Ein Satz innerhalb einer Tabelle wird eindeutig gekennzeichnet durch seinen Primärschlüssel (Primary Key). Bei einem Primärschlüssel handelt es sich, wie wir bereits kennen gelernt haben, um eine oder mehrere Spalten, die einen Satz eindeutig kennzeichnen. Existieren mehrere solcher Spalten, so wählt man einen dieser möglichen Schlüsselkandidaten als Primärschlüssel aus. So hat z.B. die Tabelle »Abteilung« zwei mögliche Schlüsselkandidaten, nämlich die Abteilungsnummer und die Abteilungsbezeichnung. In diesem Fall kann man sich für eine der beiden Spalten als Primärschlüssel entscheiden.

Der Wert eines Primärschlüssels muss immer einmalig und darf nicht optional sein. Außerdem darf er sich während der »Lebenszeit« eines Datensatzes nicht ändern. So darf z.B. der Wert »Vertrieb« des Primärschlüssels Abteilungsbezeichnung der Tabelle »Abteilung« nur genau einmal vorkommen. Ein Primärschlüssel sollte immer minimal sein, d.h. nur so viele Spalten zur eindeutigen Identifikation besitzen, wie unbedingt notwendig. So könnte man für die Tabelle »Abteilung« auch einen Primärschlüssel definieren, der aus den beiden Spalten Abteilungsbezeichnung und Abteilungsleiter besteht. Man hätte in diesem Fall also einen zusammengesetzten Schlüssel (»Compound Key«). Dies würde jedoch das eben erwähnte Prinzip der Minimalität eines Schlüssels verletzen, da bereits die Abteilungsbezeichnung zur eindeutigen Identifikation eines Satzes ausreicht.

Um nun Beziehungen (Assoziationen) zwischen Tabellen darzustellen, führt das Relationenmodell den Begriff des Fremdschlüssels (»Foreign Key«) ein. Hierbei handelt es

sich um eine Spalte, die einen Bezug zu einer anderen Tabelle herstellt. Um z.B. die Beziehung »Mitarbeiter gehört zu Abteilung« in Tabellen abzubilden, übernimmt die Tabelle »Mitarbeiter« den Primärschlüssel der »Abteilung« als so genannten Fremdschlüssel. Demnach gilt, dass die Werte eines Fremdschlüssel immer den Werten des dazugehörigen Primärschlüssels entsprechen müssen. So darf in der Tabelle »Mitarbeiter« die Spalte »Abteilungsbezeichnung« nur Werte enthalten, die auch in der Tabelle »Abteilung« in der Spalte »Abteilungsbezeichnung« vorkommen. In der Tabelle »Mitarbeiter« ist die Spalte »Abteilungsbezeichnung« Fremdschlüssel und in der Tabelle »Abteilung« ist sie Primärschlüssel. Ein Fremdschlüssel ist also immer in einer anderen Tabelle ein Primärschlüssel. Diese Einschränkung, dass der Wert eines Fremdschlüssel immer seinem dazugehörigem Primärschlüssel entsprechen muss, bezeichnet man als referentielle Integrität. Primär- und Fremdschlüssel dienen damit also zur Navigation zwischen den Tabellen.

Neben der referentiellen Integrität kennt das Relationenmodell noch zwei weitere allgemeine Einschränkungen:

- Entity-Integrität
- Domain-Integrität

Entity-Integrität haben wir bereits kennen gelernt. Hierunter versteht man die Tatsache, dass ein Primärschlüssel immer eindeutig einen Satz kennzeichnen muss und deshalb niemals leer sein darf. Domain-Integrität meint dagegen die Einschränkung von Spalten auf bestimmte Wertebereiche, die Spalte Plz darf nur gültige Postleitzahlen enthalten.

Tabellen werden nach dem Relationenmodell der Einfachheit halber nicht in Tabellenform geschrieben. Hierfür verwendet man eine einfache Notation, die die Spalten der Tabelle in Klammern aufführt. Primärschlüssel werden dabei unterstrichen und Fremdschlüssel kursiv dargestellt:

Tabellenname (Attribut 1, Attribut 2, *Attribut 3*, ..., Attribut n)

Fassen wir noch einmal zusammen. Eine Tabelle besteht aus Spalten mit eindeutigen Bezeichnern für die Spaltennamen. Die Spalten beschreiben Sätze der entsprechenden Tabelle. So macht es z.B. keinen Sinn, dass die Tabelle Mitarbeiter eine Spalte für die Bezeichnung der Spielstätte hat, da die Bezeichnung der Spielstätte nicht unmittelbar ein Merkmal eines Mitarbeiters ist. Jede Spalte enthält gleichartige Werte, die einem bestimmten Wertebereich (Domain) entsprechen. Beziehungen zwischen Tabellen werden über Primär- und Fremdschlüssel abgebildet, indem der Primärschlüssel der einen Tabelle als Fremdschlüssel in die andere Tabellen übernommen wird. Hierdurch kann zwischen den Tabellen navigiert werden.

4.4 Umsetzung des ERM in das Relationenmodell

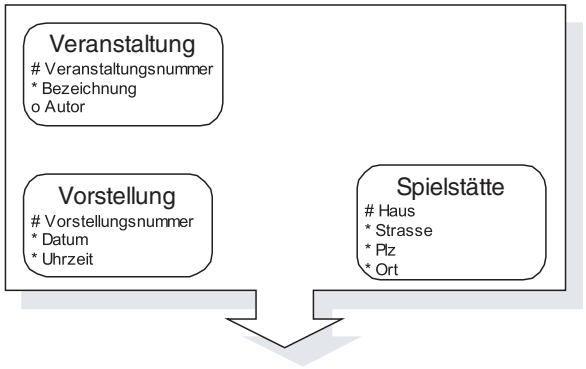
Im letzten Kapitel haben wir gesehen, dass ein konzeptionelles Datenbankmodell verschiedene Elemente zur Abbildung des Geschäftsumfeldes verwendet. Wir haben sechs Elemente unterschieden:

1) Geschäftsobjekte,
2) Attribute,
3) Eindeutige Identifizierer (Schlüssel),
4) Beziehungen,
5) Beziehungstypen,
6) Sub- bzw. Supertypen.

Im Folgenden gehen wir auf die Umsetzung der Elemente in das Relationenmodell ein.

Betrachten wir im ersten Schritt Geschäftsobjekte und deren Attribute. Diese werden im Relationenmodell jeweils als eigenständige Tabelle behandelt. Die Notation gibt vor, den Tabellennamen zu schreiben und in Klammern folgen dessen Attribute. Primärschlüssel werden hierbei unterstrichen dargestellt. Betrachten wir dazu exemplarisch das ERM nach Baker für die Geschäftsobjekte Spielstätte, Veranstaltung und Vorstellung. Die Umsetzung sieht dann wie folgt aus:

Schritt 1:



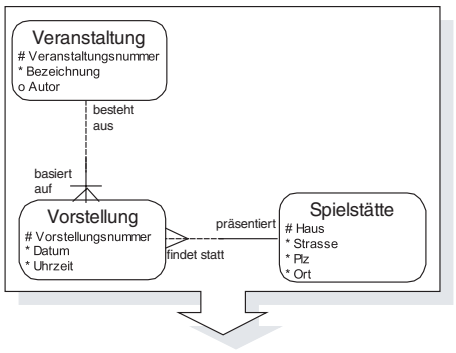
Veranstaltung (Veranstaltungsnummer, Bezeichnung, Autor)
Vorstellung (Vorstellungsnummer, Datum, Uhrzeit)
Spielstätte (Haus, Strasse, Plz, Ort)

Abbildung 4.3: Umwandlung ERM in Relationenmodell (Schritt 1)

Im zweiten Schritt werden die Beziehungen zwischen den Geschäftsobjekten abgebildet. Hierbei betrachten wir den jeweiligen Beziehungstyp, d.h. ob es sich um eine 1:1-, eine 1:N- oder eine N:M-Beziehung handelt. Da N:M-Beziehungen bereits im ERM bzw. UML-Modell in 1:N-Beziehungen umgewandelt wurden (siehe 3.4.4), kann dieser Beziehungstyp nicht mehr vorkommen.

Bei 1:N-Beziehungen wird der Primärschlüssel der 1er-Tabelle (Elterntabelle) als Fremdschlüssel in die N-er-Tabelle (Kindtabelle) übernommen. Für unser Beispiel mit der Spielstätte bedeutet das also, dass der Primärschlüssel der Tabelle Spielstätte, nämlich Haus, als Fremdschlüssel in die Tabelle Vorstellung übernommen wird. Damit ergibt sich im zweiten Schritt folgende Umsetzung, so dass zur Tabelle Vorstellung zwei Fremdschlüssel hinzukommen:

Schritt 2 (1:N-Beziehung):



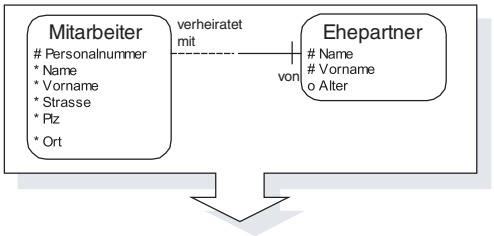
Veranstaltung (Veranstaltungsnummer, Bezeichnung, Autor)
Vorstellung (Vorstellungsnummer, Datum, Uhrzeit, *Veranstaltungsnummer*, *Haus*)
Spielstätte (Haus, Strasse, Plz, Ort)

Abbildung 4.4: Umwandlung ERM in Relationenmodell (Schritt 2)

Bei 1:1-Beziehungen gibt es verschiedene Möglichkeiten, diese Beziehung über Tabellen abzubilden.

Die einfachste Möglichkeit besteht darin, aus beiden Geschäftsobjekten eine einzige Tabelle zu erstellen. Die andere Alternative besteht darin, eine Tabelle pro Geschäftsobjekt zu erstellen und den Primärschlüssel der einen Tabelle als Fremdschlüssel in die andere Tabelle zu übernehmen. Im Gegensatz zur 1:N-Beziehung ist es dabei egal, welcher Primärschlüssel als Fremdschlüssel fungiert. Betrachten wir hierzu die Geschäftsobjekte »Mitarbeiter« und »Ehepartner«, die in einer 1:1-Beziehung zueinander stehen. Drei Möglichkeiten stehen zur Umsetzung zur Auswahl.

Schritt 2 (1:1-Beziehung):



1. Möglichkeit
Mitarbeiter (Personalnummer, Name, ..., Ort, NameEhepartner, VornameEhepartner, Alter)
2. Möglichkeit
Mitarbeiter (Personalnummer, Name, ..., Ort)
Ehepartner (Name, Vorname, Alter, *Personalnummer*)
3. Möglichkeit
Mitarbeiter (Personalnummer, Name, ..., Ort, *NameEhepartner*, *VornameEhepartner*)
Ehepartner (Name, Vorname, *Alter*)

Abbildung 4.5: Umwandlung ERM in Relationenmodell (Schritt 2)

Für welche der drei Möglichkeiten man sich entscheidet, hängt u.a. davon ab, wie oft z.B. diese Beziehung vorkommt. Würden wir annehmen, dass Mitarbeiter so gut wie nie verheiratet sind, so würde man bei Anwendung der 1. Alternative viele nicht vorhandene Werte für NameEhepartner, VornameEhepartner und Alter haben. In diesem Fall wäre es sinnvoll, die 2. oder 3. Möglichkeit anzuwenden.

Im letzten Schritt der Umsetzung des ERM in das Relationenmodell werden die Super- bzw. Subtypen in Tabellen abgebildet. Bei der Beziehung eines Supertyps zu seinen Subtypen handelt es sich um eine 1:1-Beziehung. Im Gegensatz zu einer normalen 1:1-Beziehung wird bei einer Super-Subtypen-Beziehung jedoch der Primärschlüssel des Supertyps als Fremdschlüssel in den Subtypen übernommen. Betrachten wir hierzu die Geschäftsobjekte Artikel (Supertyp) mit seinen beiden Subtypen Werbeartikel und Sitzplatz. Eine Umwandlung in Tabellen würde demnach wie folgt aussehen:

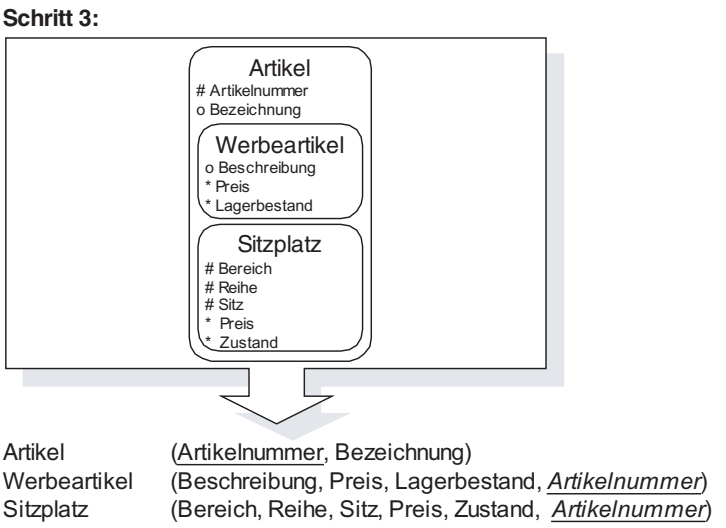


Abbildung 4.6: Umwandlung ERM in Relationenmodell (Schritt 3)

Der Sitzplatz erhält in diesem Fall also einen neuen Primärschlüssel, da ein Sitzplatz eindeutig nun auch über Artikelnummer identifiziert wird. Da »Artikelnummer« als Primärschlüssel aus nur einer Spalte besteht, muss diese Spalte als Primärschlüssel verwendet werden (Minimalität von Schlüsseln). Wir sehen hier auch, dass ein Primärschlüssel innerhalb einer Tabelle zugleich Primär- als auch Fremdschlüssel sein kann.

Rekursive Beziehungen, also Beziehungen, die sich auf das gleiche Geschäftsobjekt beziehen, werden genauso in Tabellen umgewandelt, wie es oben beschrieben wurde. Der einzige Unterschied besteht darin, dass die Beziehung sich auf die gleiche Tabelle bezieht. Nehmen wir als Beispiel die rekursive Beziehung zwischen Mitarbeiter und Vorgesetztem. Da ein Vorgesetzter auch zur Tabelle »Mitarbeiter« gehört und es sich bei der Beziehung zwischen Vorgesetztem und Mitarbeiter um eine 1:1-Beziehung handelt, wird der Primärschlüssel der Mitarbeitertabelle als Fremdschlüssel für den Vorgesetzten in die gleiche Tabelle übernommen. Das Ganze sieht dann wie folgt aus:

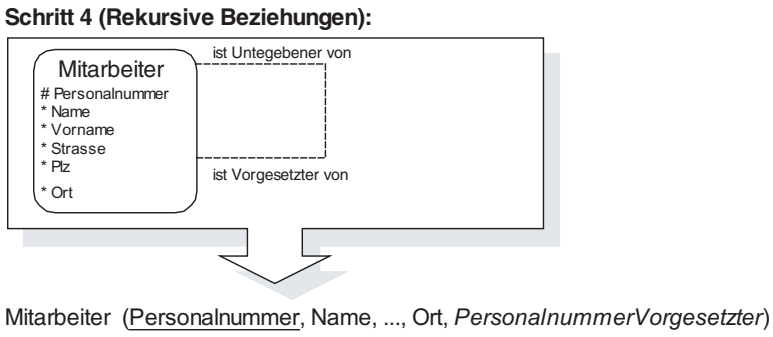


Abbildung 4.7: Umwandlung ERM in Relationenmodell (Schritt 4)

4.5 Fallbeispiel

Für unser Fallbeispiel ergibt sich folgendes Relationenmodell:

Relation	
Abteilung	(Abteilungsbezeichnung, Abteilungsnummer, *Abteilungsleiter*)
Mitarbeiter	(Personalnummer, Name, Vorname, Strasse, Plz, Ort, *Abteilungsbezeichnung*, NameEhepartner, VornameEhepartner, Alter, *PersonalnummerVorgesetzter*)
Kunde	(Kundennummer, Name, Vorname, Strasse, Plz, Ort)
Bestellung	(Bestellnummer, Positionsnummer, Menge, Datum, *Kundennummer, Personalnummer, Artikelnummer*)
Artikel	(Artikelnummer, Bezeichnung)
Werbeartikel	(Beschreibung, Preis, Lagerbestand, *Artikelnummer*)
Sitzplatz	(Bereich, Reihe, Sitz, Preis, Zustand, *Artikelnummer, Vorstellungsnummer*)
Veranstaltung	(Veranstaltungsnummer, Bezeichnung, Autor)
Vorstellung	(Vorstellungsnummer, Datum, Uhrzeit, *Veranstaltungsnummer, Haus*)
Spielstätte	(Haus, Strasse, Plz, Ort)

4.6 Zusammenfassung

In diesem Kapitel haben wir kennen gelernt, wie man aus dem Fachkonzept (konzeptionelles Modell) ein logisches Modell erstellt, das auf ein beliebiges relationales Datenbank-Management-System (RDBMS) übertragen werden kann. Wir haben uns dabei mit den Grundlagen des Relationenmodells von E. F. Codd beschäftigt. Das Relationenmodell besticht durch seine mathematische Einfachheit und bildet die Grundlage heutiger relationaler Datenbank-Management-Systeme.

Nach der Betrachtung der theoretischen Grundlagen des Relationenmodells haben wir gesehen, wie man aus einem ERM bzw. einem UML-Modell ein Relationenmodell erstellt. Zur Darstellung der Tabellen schreibt man den Tabellennamen und in Klammern die dazugehörigen Attribute. Primärschlüssel werden unterstrichen, Fremdschlüssel kursiv dargestellt. Folgende Vorgehensweise ergibt sich daraus:

1) Jedes Geschäftsobjekt wird durch eine Tabelle abgebildet.
2) Die Attribute des Geschäftsobjektes stellen die Spalten der Tabelle dar.
3) Bei einer 1:1-Beziehung werden entweder alle Attribute der einen Tabelle in die andere übertragen, so dass nur noch eine Tabelle existiert. Oder der Primärschlüssel einer der beiden Tabellen wird als Fremdschlüssel in die andere Tabelle übernommen.
4) Bei einer 1:N-Beziehung wird der Primärschlüssel der 1er-Tabelle als Fremdschlüssel in die N-er-Tabelle übernommen.
5) Bei einer Super-Subtypen-Beziehung wird der Primärschlüssel der Supertypen-Tabelle als Fremdschlüssel in die Subtypen-Tabellen übertragen. In der Regel stellt der Fremdschlüssel des Subtypen dann auch den Primärschlüssel des Subtypen dar.
6) Rekursive Beziehungen werden genau so aufgelöst, wie in den Punkten 3 und 4 beschrieben, mit dem einzigen Unterschied, dass Fremdschlüssel und Primärschlüssel in der gleichen Tabelle vorkommen.

Die Umwandlung eines ERM oder eines UML-Modells in ein Relationenmodell ist ohne weiteres systematisch möglich, indem man genau die beschriebenen sechs Schritte durchläuft.

Zur Erstellung von ERM oder auch UML-Modellen gibt es heutzutage Softwarewerkzeuge, die einem zwar nicht unbedingt die Arbeit des Fachkonzeptes abnehmen, aber weitgehend die Transformation in das logische Modell und auch in das physische Modell. Zu den bekanntesten gehören der »Oracle Designer«, »Rational Rose Professional Data Modeler« (UML), »Sybase PowerDesigner«, »Embarcadero ER/Studio« oder »Computer Associates ERwin«. Sofern man eines dieser Werkzeuge einsetzt, sollte man immer bedenken, dass die Software einem nur systematische Arbeiten abnehmen kann. Ein Fehler im konzeptionellen Modell, also auf fachlicher Ebene, kann auch das beste Softwarewerkzeug nur sehr bedingt erkennen.

Wir haben bisher gesehen, wie man von den fachlichen Anforderungen des konzeptionellen Modells zu einem logischen Modell, dem Relationenmodell, gelangt, das auf eine beliebige Computerplattform mit einem beliebigen RDBMS übertragen werden kann. Im nächsten Kapitel lernen wir, wie man das erstellte Relationenmodell noch einmal auf Korrektheit, Konsistenz und Vermeidung redundanter Daten überprüft.

4.7 Aufgaben

Wiederholungsfragen

1) Welche Eigenschaften besitzt ein Primärschlüssel?
2) Welche Eigenschaften besitzt ein Fremdschlüssel?
3) Was versteht man unter einem Tupel, einer Relation und einem Attribut?
4) Was versteht man unter referentieller Integrität?
5) Was versteht man unter Entity-Integrität?
6) Was versteht man unter Domain- bzw. Domänen-Integrität?
7) Wie wird ein Geschäftsobjekt in das Relationenmodell umgesetzt?
8) Wie wird eine 1:1-, eine 1:N- und eine N:M-Beziehung in das Relationenmodell umgesetzt?

Übungen

1) Die Umwandlung einer 1:N-Beziehung erfolgt, indem man den Primärschlüssel der 1er-Tabelle als Fremdschlüssel in die N-er-Tabelle übernimmt. Würde auch der umgekehrte Fall funktionieren, also den Primärschlüssel der N-er-Tabelle in die 1er-Tabelle übernehmen? Begründen Sie Ihre Antwort anhand eines Beispiels aus zwei Tabellen!
2) Angenommen bei der Beziehung zwischen Mitarbeiter und Vorgesetztem würde es sich um eine 1:N-Beziehung handeln, also jeder Mitarbeiter kann nicht nur einen, sondern mehrere Vorgesetzte haben. Wie würde dann die Tabelle Mitarbeiter aussehen?
3) Wandeln Sie die Entity-Relationship-Modelle bzw. die UML-Modelle der Übungen 2-9 aus Kapitel 3 in das Relationenmodell um!

5 Normalisierung – Überprüfung der Datenbankstruktur

In Kapitel 5 sollen folgende Fragen geklärt werden:

- Wie kann man das Relationenmodell auf Vermeidung mehrfach gespeicherter Daten überprüfen?
- Welche Abhängigkeiten existieren zwischen Attributen von Tabellen?
- Was versteht man unter dem Begriff »Normalisierung«?

5.1 Motivation

Herr Dr. Fleissig hat inzwischen vollständig das ERM für die Firma »KartoFinale« in das Relationenmodell übertragen. Bei der Tabelle »Bestellung«ist er sich noch unsicher. Um sich über eventuelle Probleme klar zu werden, überlegt er sich Testdaten und zeichnet diese in Tabellenform auf. Zugunsten der Übersichtlichkeit lässt er die Spalten Personalnummer und Artikelnummer weg.

Bestell-nummer	Positions-nummer	Menge	Datum	Kunden-nummer	Personal-nummer	Artikel-nummer
4711	1	4	29.09.2001	11112	4	1002
4711	2	2	29.09.2001	11112	4	1021
4711	3	1	29.09.2001	11112	4	1022
4711	4	1	29.09.2001	11112	4	2008
3001	1	8	06.01.2002	12343	5	1021
3001	2	7	06.01.2002	12343	5	1232

Abbildung 5.1: Redundante Daten in der Tabelle »Bestellung«

Dabei stellt er fest, dass für die Bestellung mit der Bestellnummer 4711 das Datum der Bestellung mehrfach, also redundant, vorhanden ist. Entsprechendes gilt für die Bestellnummer selbst, die Kundennummer des Kunden, der die Bestellung aufgegeben hat und die Personalnummer des Mitarbeiters, der die Bestellung entgegengenommen hat. Scheinbar ist Herrn Fleissig hier ein Fehler unterlaufen. Verunsichert holt er seinen neuen Kollegen Herrn Klever und bittet diesen um Rat. Herr Klever, der sich während seines Studiums auf Modellierung von Geschäftsprozessen und deren Daten spezialisiert hat, erkennt sofort, dass hier zwei Geschäftsobjekte miteinander vermischt wurden.

Er gibt Herrn Dr. Fleissig folgenden Rat: »Entweder modellierst du die Bestellung im ERM noch einmal neu, da der Entitytyp »Bestellung« z.Z. aus zwei verschiedenen

Entitytypen besteht. Du hast die Entitytypen »Bestellung« und »Bestellposten« miteinander vermischt. Entsprechend müsstest du dann auch das Relationenmodell noch einmal ändern. Die andere Möglichkeit besteht in der Anwendung der Normalformen. Hierbei untersucht man die Abhängigkeiten zwischen den Nichtschlüssel-Attributen zum Primärschlüssel. Nehmen wir mal deine Tabelle. Als Primärschlüssel hast du Bestellnummer und Positionsnummer gewählt. Alle Nichtschlüssel-Attribute müssen nach der zweiten Normalform vom gesamten Primärschlüssel abhängig sein. Datum und Kundennummer sind jedoch ausschließlich von der Bestellnummer direkt abhängig.«

Im weiteren Gesprächsverlauf erklärt Herr Klever seinem Kollegen die Einschränkungen der ersten bis dritten Normalform. Herr Fleissig, erfreut darüber etwas neues gelernt zu haben, wendet die Normalformen an und löst damit sein Problem.

5.2 Grundlagen der Normalisierung

Nachdem E. F. Codd die Grundlagen relationaler Datenbanken geschaffen hatte, hat er sich mit der Vermeidung redundanter Daten im Relationenmodell beschäftigt, der Normalisierung. Aus diesen Überlegungen heraus entstanden Vorgaben für bestimmte Einschränkungen auf Tabellen, die in den so genannten Normalformen beschrieben sind.

Die Normalisierung ist also eine Entwurfstechnik, die dazu dient, Tabellen auf mehrfach gespeicherte Daten (Redundanz), Konsistenz und Korrektheit zu überprüfen. Durch die Normalisierung teilt man Tabellen in weitere Tabellen auf, um redundante Daten zu vermeiden. Die Aufteilung muss dabei verlustfrei sein, d.h. ein späteres Zusammensetzen der geteilten Tabelle über Fremd- und Primärschlüssel muss wieder zur ursprünglichen Tabelle führen, ohne dass dabei Informationen verloren gehen.

Die Normalisierung wird in der Regel nach dem Erstellen des Relationenmodells verwendet, um dieses Modell noch einmal zu überprüfen.

Die Normalisierung geht generell in zwei Schritten vor: Zunächst werden Wiederholgruppen entfernt und danach redundant gespeicherte Daten. Bevor wir uns im Einzelnen mit den Normalformen auseinandersetzen, müssen wir die Begriffe der funktionalen und der vollen funktionalen Abhängigkeit verstehen.

Zwischen Attributen eines Objekttyps existieren Abhängigkeiten. Diese Abhängigkeiten basieren aus Beobachtungen der realen Welt. Betrachten wir zum Beispiel einmal den Objekttyp »Getränk«.

Getränk

Bezeichnung	Farbe
Kaffee	schwarz
Coca-Cola	schwarz
Mineralwasser	-
Milch	weiss

Abbildung 5.2: Funktionale Abhängigkeit von Attributen

Zwischen den beiden Attributen »Bezeichnung« und »Farbe« des Objekttyps »Getränk« existiert eine Abhängigkeit in der Form, dass man auf Grund der Bezeichnung des Getränks eindeutig sagen kann, wie der Wert des Attributes »Farbe« ist. So weiß man, dass ein Getränk mit der Bezeichnung »Kaffee« für das Attribut Farbe den Wert »schwarz« haben muss. Umgekehrt ist diese Schlussfolgerung jedoch nicht möglich. Wenn man weiß, welche Farbe ein Getränks hat, kann man daraus nicht bestimmen, wie die Bezeichnung des Getränks ist. Das Ganze ist für Sie natürlich schon ein »alter Hut«, da wir dies bereits in anderer Form bei der Bestimmung von Schlüsseln kennen gelernt haben. Beim Objekttyp »Getränk« ist das Attribut »Bezeichnung« der Primärschlüssel. Entsprechend bestimmt der Wert des Primärschlüssels eindeutig den Wert der anderen Attribute, in diesem Fall den Wert des Attributes Farbe. E. F. Codd schreibt in diesem Zusammenhang in Anlehnung an die Mathematik von funktionaler Abhängigkeit, da eine mathematische Funktion immer den gleichen Ausgabewert bei gleichen Eingabewerten liefert.

Die Funktion y = 2x liefert z. B. für den Wert x = 3 immer den Ausgabewert 6 für die Variable y. Für jede beliebige Zahl, die man für die Variable x einsetzt, ergibt sich also ein ganz bestimmter Wert für y. Man kann sagen, dass der Wert y abhängig ist vom Wert x bzw. x bestimmt y. Mathematisch spricht man auch von Determinante und schreibt: $x \rightarrow y$.

Doch kommen wir nun zu unserem Getränke-Beispiel zurück. Zwischen den Attributen »Bezeichnung« und »Farbe« besteht folgende funktionale Abhängigkeit:

Bezeichnung → Farbe

Das Attribut Farbe ist also funktional abhängig vom Attribut »Bezeichnung«. Erweitern wir nun unser Beispiel um ein weiteres Attribut »Geschmack«. Als Primärschlüssel wollen wir diesmal die beiden Attribute »Bezeichnung« und »Geschmack« festlegen. Betrachten wir nun wieder die funktionalen Abhängigkeiten, so stellen wir fest, dass das Attribut Farbe weiterhin funktional abhängig ist vom Primärschlüssel. Wenn die Bezeichnung und der Geschmack eines Getränkes bekannt sind, dann ergibt sich daraus eindeutig die Farbe des Getränks.

Wir können also schreiben: **Bezeichnung, Geschmack → Farbe**

Schauen wir uns nun die Abhängigkeiten zwischen Geschmack und Farbe an, so stellen wir fest, dass keine funktionale Abhängigkeit zwischen diesen beiden Attributen existiert. Ist der Geschmack eines Getränkes bekannt, so kann nicht eindeutig auf die Farbe geschlossen werden. Andererseits gilt nach wie vor für die Attribute Bezeichnung und Farbe, dass aufgrund der Bezeichnung eindeutig auf die Farbe geschlossen werden kann. In diesem Fall ist Farbe zwar funktional abhängig von dem zusammengesetzten Primärschlüssel Bezeichnung und Geschmack, auf das Attribut »Geschmack« kann jedoch verzichtet werden, um das Attribut »Farbe« eindeutig zu bestimmen.

Wir sprechen von voller funktionaler Abhängigkeit, wenn jedes Nichtschlüssel-Attribut durch den gesamten Primärschlüssel eindeutig bestimmt werden kann. In unserem Beispiel ist das nicht der Fall, da die Farbe auch dann eindeutig bestimmt werden kann, wenn das Attribut »Geschmack« nicht bekannt ist. Sicherlich werden Sie bemerkt

haben, dass volle funktionale Abhängigkeit unmittelbar in Zusammenhang mit der Forderung nach einem minimalen Primärschlüssel steht, wie wir es in Kapitel 4 kennen gelernt haben.

Getränk

Bezeichnung	Geschmack	Farbe
Kaffee	bitter	schwarz
Coca-Cola	süß	schwarz
Mineralwasser	neutral	-
Milch	neutral	weiss

Abbildung 5.3: Volle funktionale Abhängigkeit von Attributen

Der letzte Begriff, der noch zu klären ist, ist die so genannte transitive Abhängigkeit. Betrachten wir hierzu wieder ein Beispiel: Für einen Objekttyp »Fahrzeug« wurde als Primärschlüssel das Attribut »Bezeichnung« festgelegt.

Fahrzeug

Bezeichnung	Typ	Anzahl Räder	PS
VW Käfer	PKW	4	45
Hanomag II	LKW	4	220
Vespa Light	Motorroller	2	4
Suzuki GX	Motorrad	2	8
Ford Focus GL	PKW	4	65

Abbildung 5.4: Transitive funktionale Abhängigkeit von Attributen

Zunächst können wir erkennen, dass abhängig vom Primärschlüssel bestimmt werden kann, welchen Wert die anderen Attribute haben müssen. So bestimmt z.B. der Wert Vespa, dass es sich beim Fahrzeugtyp um einen Motorroller handelt, der 2 Räder und 4 PS haben muss. Da der Primärschlüssel aus nur einem Attribut besteht, müssen demnach auch alle Nichtschlüssel-Attribute voll funktional abhängig vom Primärschlüssel sein. Rein funktionale Abhängigkeit kann ja nur dann auftreten, wenn der Primärschlüssel aus zusammengesetzten Attributen besteht.

Untersuchen wir die Tabelle auf eventuell andere funktionale Abhängigkeiten, so erkennen wir, dass, wenn der Fahrzeugtyp bekannt ist, man daraus die Anzahl der Räder ableiten kann. Das Attribut »Anzahl Räder« scheint also voll funktional abhängig zu sein vom Fahrzeugtyp. Wir sprechen hier von einer transitiven Abhängigkeit, da wir folgende Schlussfolgerung ziehen können: Wenn eine Vespa ein Motorroller ist und jeder Motorroller zwei Räder hat, dann hat auch die Vespa zwei Räder. Oder allgemeiner: Wenn A den Wert von B bestimmt und B den Wert von C bestimmt, dann bestimmt auch A den Wert von C.

Fassen wir noch einmal zusammen:

1) Ein Attribut A ist von einem Attribut B funktional abhängig, wenn zu jedem Wert von B eindeutig der Wert von A bestimmt werden kann.
2) Von voller funktionaler Abhängigkeit spricht man, wenn ein Attribut A von einer Attributkombination B komplett funktional abhängig ist, und nicht nur von einem Teil dieser Attributkombination.
3) Transitive Abhängigkeit zwischen zwei Attributen A und C liegt vor, wenn ein Attribut A von einem Attribut B eindeutig bestimmt wird, das Attribut B aber wiederum von einem Attribut C eindeutig bestimmt wird.

Da wir soweit die Grundlagen der Normalisierung kennen gelernt haben, ist es jetzt relativ einfach, die Normalformen zu verstehen und anzuwenden. Bis 1978 wurden von E. F. Codd drei Normalformen aufgestellt, die entsprechend als erste, zweite und dritte Normalform bezeichnet werden. Daneben gibt es noch vier weitere Normalformen, die jedoch in der Praxis nicht so relevant sind, da Probleme, die durch diese Normalformen beseitigt werden, nur selten vorkommen. Wir wollen deshalb im Folgenden auf die ersten drei Normalformen eingehen. Anschließend wird anhand eines Beispiels deutlich, dass unter Umständen auch weitergehende Normalformen bedeutsam sein können.

5.3 1. Normalform

Die erste Normalform legt fest, dass eine Tabelle nur aus Attributen mit atomaren Werten bestehen und es keine Wiederholgruppen geben darf. Als Beispiel schauen wir uns die Tabelle Kunde aus unserem Fallbeispiel an. Die Tabelle wurde um die Namen und das Alter der Kinder erweitert. In der unnormalisierten Form der Tabelle stellt man fest, dass die Strasse aus zwei unterschiedlichen Werten besteht, nämlich dem Strassennamen und der Hausnummer. Das Attribut »Strasse« ist also nicht atomar und muss dementsprechend in zwei Attribute aufgeteilt werden.

Bei dem Attribut »Kind« erkennt man, dass im Kreuzungspunkt von Zeile und Spalte nicht ein Wert, sondern je nach Anzahl der Kinder, mehrere Werte gespeichert werden. Diese Wiederholgruppen eliminieren wir, indem wir pro Kind eine Zeile erstellen. Das Ganze sieht dann wie in Abbildung 5.5 aus.

Wie Sie sicher bemerkt haben, führt die Anwendung der ersten Normalform zunächst zu mehrfach gespeicherten Daten bei Wiederholgruppen. So existieren z.B. für den Kunden »Bolte, Bertram« jetzt drei Zeilen, da er drei Kinder hat. Zudem muss der Primärschlüssel neu definiert werden, da das Attribut »Kundennummer« alleine nicht mehr ausreicht. Wir haben zur Zeit also durch Anwendung der ersten Normalform unser Resultat eher verschlechtert.

Durch Anwendung der ersten Normalform erzeugt man also zunächst redundante Daten. Erst durch Anwendung der zweiten und dritten Normalform werden diese redundanten Daten wieder beseitigt. Das wollen wir uns jetzt ansehen.

Kunde (unnormalisiert)

Kunden-nummer	Name	Vorname	Strasse	Plz	Ort	Kind	Alter
1	Bolte	Bertram	Busweg 12	44444	Kohlscheidt	Katja, Ursula, Karl	5, 8, 12
2	Muster	Hans	Musterweg 12	22222	Karlstadt		
3	Wiegerich	Frieda	Wanderstr. 89	33333	Rettrich	Ursula, Enzo	16, 9

Überführung in 1. Normalform

Kunde (1. NF)

Kunden-nummer	Name	Vorname	Strasse	Haus-nummer	Plz	Ort	Kind	Alter
1	Bolte	Bertram	Busweg	12	44444	Kohlscheidt	Katja	5
1	Bolte	Bertram	Busweg	12	44444	Kohlscheidt	Ursula	8
1	Bolte	Bertram	Busweg	12	44444	Kohlscheidt	Karl	12
2	Muster	Hans	Musterweg	12	22222	Karlstadt		
3	Wiegerich	Frieda	Wanderstr.	89	33333	Rettrich	Ursula	16
3	Wiegerich	Frieda	Wanderstr.	89	33333	Rettrich	Enzo	9

Kunde (Kundennummer, Name, Vorname, Strasse, Hausnummer, Plz, Ort, Kind, Alter)

Abbildung 5.5: Anwendung der 1. Normalform

5.4 2. Normalform

Die zweite Normalform bezieht sich ausschließlich auf Tabellen, deren Primärschlüssel aus mehreren Attributen zusammengesetzt ist. Eine Tabelle befindet sich dann in der zweiten Normalform, wenn alle Nichtschlüssel-Attribute nicht nur von einem Teil, sondern vom gesamten Primärschlüssel voll funktional abhängig sind.

Nehmen wir dazu wieder unser Beispiel aus Abschnitt 5.3. Durch das Hinzufügen der Attribute für die Kinder der Kunden musste zunächst der Primärschlüssel neu definiert werden, da die Kundennummer alleine nicht mehr eindeutig alle anderen Attribute bestimmt. Als Primärschlüssel haben wir die Kombination aus den Attributen Kundennummer und Name des Kindes gewählt.

Wenden wir jetzt die zweite Normalform auf diese Tabelle an, so erkennen wir, dass z. B. das Attribut Strasse zwar durch den Primärschlüssel eindeutig bestimmt werden kann, aber zur Bestimmung der Strasse nur ein Teil des Primärschlüssels, das Attribut Kundennummer, ausreichen würde. Das gleiche gilt für die Attribute »Name«, »Vorname«, »Plz« und »Ort«.

Betrachten wir das letzte Attribut »Alter«, so stellen wir fest, dass dieses eindeutig durch den gesamten Primärschlüssel aus Kundennummer und Name des Kindes bestimmt werden kann. Nur ein Teil des Primärschlüssels, Name des Kindes oder die Kundennummer, reicht in diesem Fall also nicht aus. Kennen wir die Kundennummer, so kommen z. B. für den Kunden 1 die Werte 5, 8, 12 in Betracht. Die Kundennummer alleine ist also nicht eindeutig. Kennen wir den Namen eines Kindes, so gilt das gleiche. Für das Attribut »Kind« mit dem Wert »Katja« können wir zwar eindeutig bestimmen, dass das Attribut »Alter« 5 ist. Für den Wert »Ursula« ist dies jedoch nicht mög-

lich, da der Name dieses Kindes zweimal vorkommt, einmal für den Kunden mit der Kundennummer 1 und für den Kunden mit der Kundennummer 3. Das Attribut »Alter« ist also voll funktional abhängig vom gesamten Primärschlüssel aus Kundennummer und Name des Kindes.

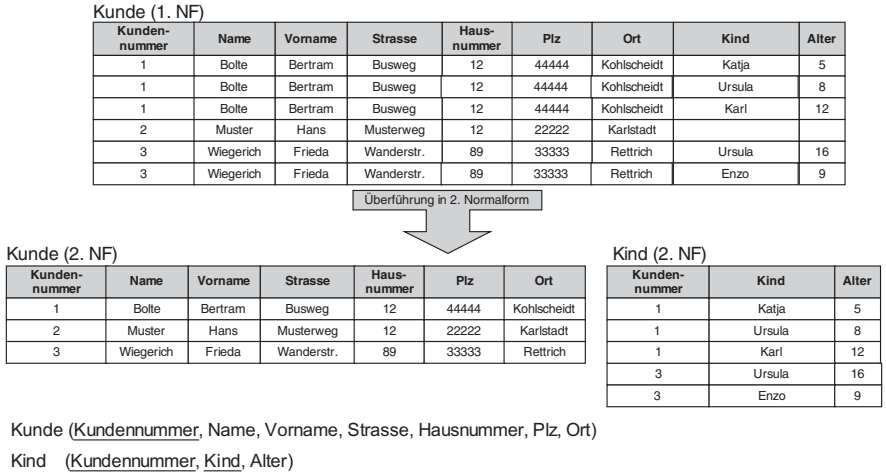


Abbildung 5.6: Anwendung der 2. Normalform

Um den Einschränkungen der zweiten Normalform gerecht zu werden, teilt man die Tabelle in zwei Tabellen auf. Die erste Tabelle erhält alle Attribute, die voll funktional abhängig sind von Kundennummer, die zweite Tabelle erhält alle Attribute, die voll funktional abhängig sind von Kundennummer und Name des Kindes.

5.5 3. Normalform

Die dritte Normalform bezieht sich auf funktionale Abhängigkeiten zwischen Nichtschlüssel-Attributen. Eine Tabelle befindet sich dann in der dritten Normalform, wenn alle Nichtschlüssel-Attribute ausschließlich vom Primärschlüssel funktional abhängig sind, und nicht transitiv über ein Nichtschlüssel-Attribut. Allerdings gibt es hierbei eine Ausnahme, und zwar darf ein Nichtschlüssel-Attribut nach wie vor von einem Schlüsselkandidaten abhängig sein.

Betrachten wir dazu wieder unser Beispiel. Der Primärschlüssel der Kundentabelle ist die Kundennummer. Über die Kundennummer kann eindeutig die Postleitzahl und der Ort eines Kunden bestimmt werden. Andererseits erkennen wir hier aber auch eine funktionale Abhängigkeit zwischen der Postleitzahl und dem Ort. Ist die Postleitzahl eines Kunden bekannt, so kann eindeutig auf den Ortsnamen geschlossen werden. Der Ort ist also voll funktional abhängig von dem Attribut Plz. Wir haben es hier mit einer transitiven Abhängigkeit zu tun, über die Kundennummer kann die Postleitzahl bestimmt werden und über die Postleitzahl der Ortsname. Das Attribut Plz ist also ein Schlüssel für Ortsname.

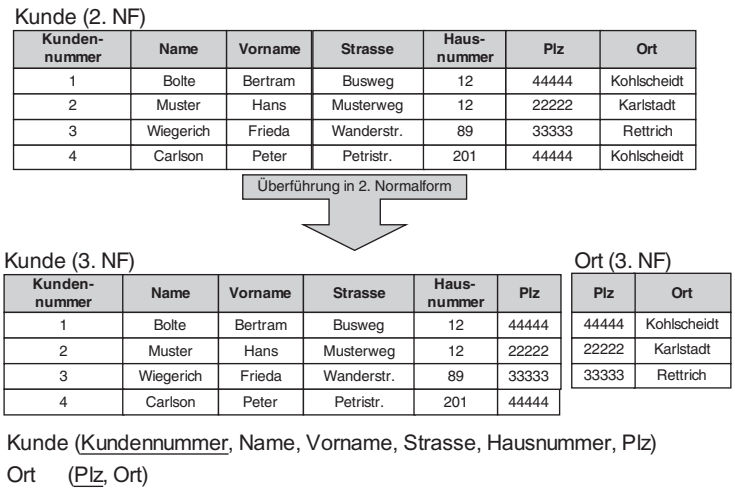


Abbildung 5.7: Anwendung der 3. Normalform

Um diese Abhängigkeit aufzulösen, teilt man die Tabelle wiederum in zwei Tabellen auf. Die erste Tabelle behält als Primärschlüssel die Kundennummer und entsprechend alle Attribute, die von diesem voll funktional, aber nicht transitiv abhängig sind. Dies sind also alle Attribute außer Ort. Die zweite Tabelle erhält als Primärschlüssel das Attribut Plz und als Nichtschlüssel-Attribut den Ortsnamen.

5.6 Weitere Normalformen

Die ersten drei Normalformen decken in der Regel die häufigsten Probleme ab, die bei einem Datenmodell auftreten können. Daher soll es an dieser Stelle genügen, nur die ersten drei Normalformen ausführlich zu erläutern und die weiteren Normalformen im Folgenden kurz zu erwähnen.

Als Ersatz für die dritte Normalform wird häufig die Boyce-Codd-Normalform angewendet. Sie sieht vor, alle Determinanten einer Tabelle zu bestimmen und für jede Determinante eine Tabellen zu erstellen.

Die vierte und die fünfte Normalform beschäftigen sich mit so genannten mehrwertigen Abhängigkeiten. Mehrwertige Abhängigkeiten treten nur auf, wenn der Primärschlüssel aus zwei oder mehr Attributen zusammengesetzt ist. Betrachten wir hierzu wieder unser Fallbeispiel. Angenommen, wir möchten zu jedem Kind eines Kunden dessen Hobbies speichern, so könnten wir ein zusätzliches Attribut »Hobby« der Tabelle »Kind« hinzufügen. Der Übersichtlichkeit halber wird das Attribut »Alter« weggelassen. Zur Erläuterung sehen wir uns diesen ersten Entwurf mit einigen Beispieldaten an:

Kunden-nummer	Kind	Hobby
1	Katja	Tennis
1	Ursula	Zeichnen
1	Ursula	Tennis
1	Ursula	Tanzen
1	Karl	Fussball
3	Ursula	Tennis
3	Enzo	Handball

Abbildung 5.8: Mehrwertige Abhängigkeiten

Bisher bestand der Primärschlüssel aus Kundennummer und Kind. Durch das Hinzufügen der Spalte »Hobby« muss der Primärschlüssel jedoch um dieses Attribut erweitert werden, um eindeutig zu sein, da z.B. der Wert »Ursula« des Kunden mit der Kundennummer 1 dreimal auftritt. Obwohl die Tabelle in der dritten Normalform ist, erkennen wir redundante Daten, so wird z.B. Ursula mit der Kundennummer 1 dreimal gespeichert.

Da Probleme dieser Art nur bei zusammengesetzten Primärschlüsseln auftreten, sind sie sehr selten. In der Regel versucht man einen Primärschlüssel auf ein, höchstens zwei Attribute zu beschränken. Auf Lösungen solcher Probleme wollen wir hier deshalb nicht weiter eingehen.

Schließlich gibt es noch die »Domain-Key-Normalform« (DKNF), die einen etwas anderen Weg beschreitet, indem sie Tabellen auf die Anwendung eines bestimmten Themas (»Single-Theme«) untersucht.

5.7 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie man ein vorher erstelltes Relationenmodell noch einmal auf logische Korrektheit überprüfen kann. Ziel der Normalisierung ist dabei die Vermeidung redundanter Daten. Dabei haben wir drei Normalformen kennen gelernt, die man nacheinander auf Tabellen anwenden sollte, um diese zu überprüfen.

Folgende Schritte sind zur Erfüllung der drei Normalformen durchzuführen:

1) Normalform
   - Attribute mit mehreren Werten, z.B. Adresse, in atomare Attribute aufspalten;
   - Wiederholgruppen, z.B. Kinder der Kunden, in mehrere Zeilen aufteilen.
2) Normalform
   - Überprüfen der vollen funktionalen Abhängigkeit der Nichtschlüssel-Attribute zum Primärschlüssel;

- Gegebenenfalls Aufteilen der Tabelle nach deren voller funktionaler Abhängigkeit (nur bei zusammengesetzten Primärschlüsseln notwendig).

3) Normalform
   - Überprüfen von transitiven Abhängigkeiten des Primärschlüssels zu den Nichtschlüssel-Attributen (Ausnahme: Funktionale Abhängigkeit von einem Schlüsselkandidaten);
   - Gegebenfalls Aufteilen der Tabelle, so dass transitiv abhängige Attribute mit ihrem funktional abhängigen Attribut eine eigene Tabelle bilden und transitiv abhängiges Attribut in Originaltabelle erhalten bleibt.

Generell gilt, dass ein »sauber« modelliertes ERM bzw. UML-Modell bereits in ein redundanzfreies Relationenmodell überführt wird.

Dennoch sollte die Normalisierung in drei Fällen angewendet werden. Zum einen zur nachträglichen Überprüfung von Tabellen, bei denen man Unstimmigkeiten vermutet, wie in unserem Fallbeispiel. Zum anderen zur Überprüfung, wenn nachträglich Attribute zur Datenbank hinzugefügt werden sollen. Und schließlich um eine bereits vorhandene relationale Datenbankstruktur nachträglich zu normalisieren. Häufig werden in Firmen relationale Datenbanken erstellt, ohne vorab ein konzeptionelles und logisches Modell zu entwerfen. In diesem Fall eignet sich die Normalisierung als nachträgliche Designtechnik, um das Datenmodell redundanzfrei zu entwerfen.

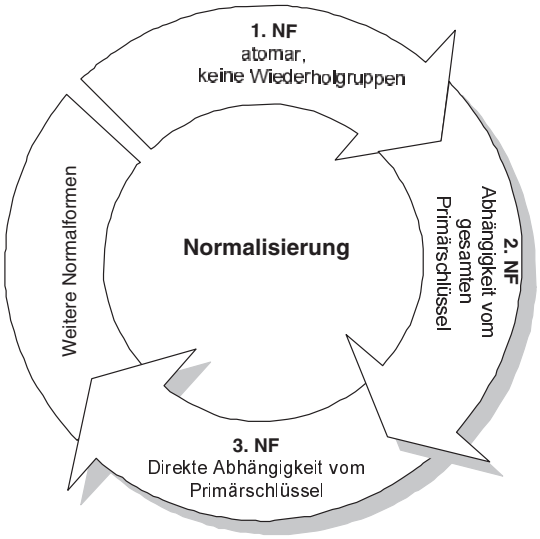


Abbildung 5.9: Zusammenfassung »Normalformen«

Wir sind jetzt soweit mit dem Entwurf unserer Datenbankstruktur fortgeschritten, dass diese im nächsten Kapitel physisch umgesetzt werden kann. Da wir in den folgenden

Kapiteln immer wieder auf dieses Datenmodell zurückkommen, soll deshalb hier noch einmal das endgültige ERM unseres Fallbeispiels und das dazugehörige Relationenmodell dargestellt werden.

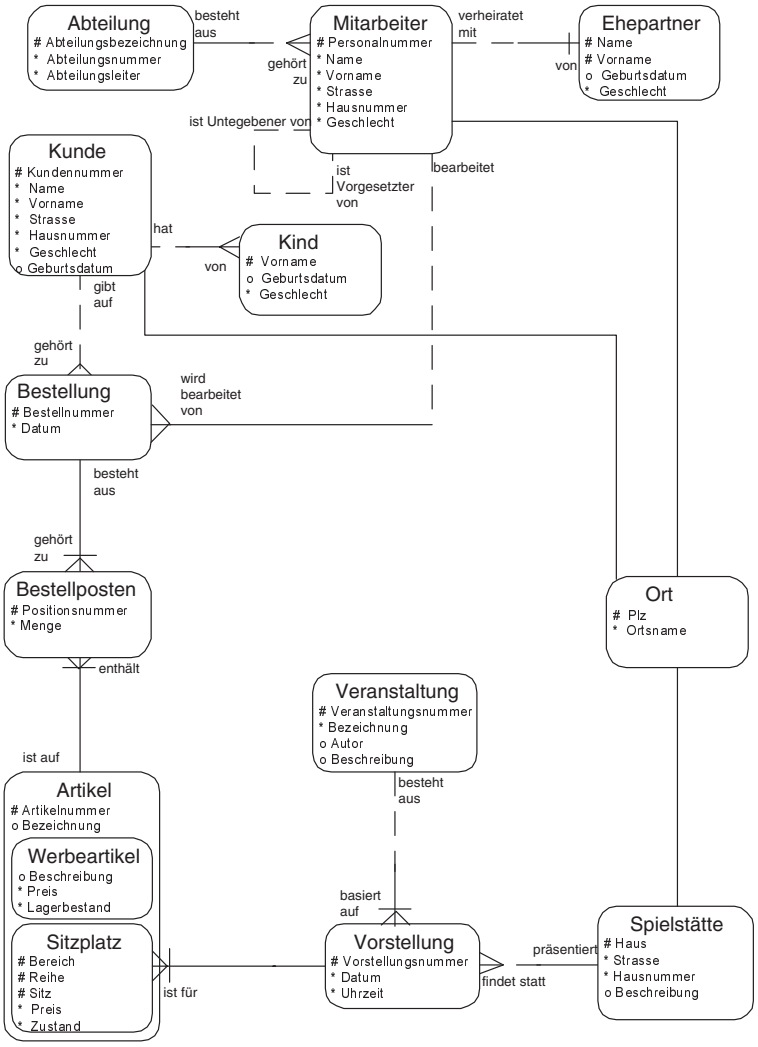


Abbildung 5.10: Das endgültige ERM des Fallbeispiels

Entsprechend sieht das Relationenmodell wie folgt aus:

Relation	
Abteilung	(Abteilungsbezeichnung, Abteilungsnummer, *PersonalnummerAbteilungsleiter*)
Mitarbeiter	(Personalnummer, Name, Vorname, Geschlecht, Strasse, Hausnummer, *Plz, Abteilungsbezeichnung*, NameEhepartner, VornameEhepartner, Geburtsdatum Ehepartner, GeschlechtEhepartner, Gehalt, *PersonalnummerVorgesetzter*)
Kunde	(Kundennummer, Name, Vorname, Geschlecht, Strasse, Hausnummer, *Plz*, Geburtsdatum)
Bestellung	(Bestellnummer, Datum, *Kundennummer, Personalnummer*)
Bestellposten	(Positionsnummer, *Bestellnummer*, Menge, *Artikelnummer*)
Artikel	(Artikelnummer, Bezeichnung)
Werbeartikel	(Beschreibung, Preis, Lagerbestand, *Artikelnummer*)
Sitzplatz	(Bereich, Reihe, Sitz, Preis, Zustand, *Artikelnummer*, *Vorstellungsnummer*)
Veranstaltung	(Veranstaltungsnummer, Bezeichnung, Autor, Beschreibung)
Vorstellung	(Vorstellungsnummer, Datum, Uhrzeit, *Veranstaltungsnummer, Haus*)
Spielstätte	(Haus, Strasse, Hausnummer, Beschreibung, *Plz*)
Ort	(Plz, Ort)
Kind	(Vorname, Geburtsdatum, Geschlecht, Kundennummer)

Tabelle 5.1: Relationenmodell

5.8 Aufgaben

Wiederholungsfragen

1) Wozu dient die Normalisierung?
2) Definieren Sie die erste, zweite und dritte Normalform!
3) Worin besteht der Unterschied zwischen funktionaler und voll funktionaler Abhängigkeit?
4) In Abschnitt 5.3 wurde das Attribut »Alter« für die Kinder eines Kunden eingeführt. Dies ist als problematisch zu betrachten. Warum?

Übungen

1) Wenden Sie auf folgende Tabelle die ersten drei Normalformen an!

Fahrzeug

Hersteller	Modell	Typ	Anzahl Räder	PS	Adresse	Land	Klima-anlage
VW	Käfer	PKW	4	45	Volkswagenstr. 10, 34444Wolfsburg	Deutschland	Ja
Hanomag	II	LKW	4	220	Musterstr. 1, 22222 Hamburg	Deutschland	Nein
Vespa	Vespa Light	Motorroller	2	4	La Strada 2, 122 Rom	Italien	Nein
Suzuki	GX	Motorrad	2	8	Koniciva 3, Yokohama	Japan	Nein
Ford	Focus	PKW	4	65	Fritzweg 2, 54455 Köln	Deutschland	Ja
VW	Golf	PKW	4	72	Volkswagenstr. 10, 34444Wolfsburg	Deutschland	Nein
VW	Golf	PKW	4	72	Volkswagenstr. 10, 34444Wolfsburg	Deutschland	Ja

Abbildung 5.11: Tabelle Fahrzeug

2) Im ERM-Entwurf unseres Fallbeispiels wurde die Tabelle »Bestellung« im Relationenmodell folgendermaßen entworfen:
Bestellung (Bestellnummer, Positionsnummer, Menge, Datum, *Kundennummer, Personalnummer, Artikelnummer*)
Wenden Sie die ersten drei Normalformen auf die Tabelle Bestellung an!

3) Wenden Sie auf folgende Tabelle die ersten drei Normalformen an!

Büromaterial

Material-nummer	Material-bezeichnung	Lieferanten-nummer	Lieferanten-bezeichnung	Lager-bestand	Preis
101	CD-Rohling	2	InnoPaper	61	1.99
221	Kugelschreiber	2	InnoPaper	156	0.99
101	CD-Rohling	31	Computer-Mendel	61	1.89
134	Papier A4	2	InnoPaper	702	8.99
002	Toner schwarz	31	Computer-Mendel	8	39.50
007	Heftzwecken	26	Muster-Büroartikel	81	0.79
026	Schnellhefter	2	InnoPaper	81	0.89

Abbildung 5.12: Tabelle Büromaterial

4) Wenden Sie auf folgende Tabelle die ersten drei Normalformen an!

Fußballverein

Vereins-bezeichnung	Gründungs-jahr	Spieler-nummer	Spieler-Name	Geburts-datum
1. FC Muster	1921	2	Karl Bein	1980-09-02
Eintracht Fritz	1949	11	Fritz Wurf	1979-02-23
1. FC Muster	1921	7	Hans Mehnel	1986-12-31
1. FC Muster	1921	11	Kurt Bebel	1975-06-16
Hehner SV	1971	2	Franz Bein	1971-11-30
Hehner SV	1971	3	Hubertus Klein	1983-08-09
Hehner SV	1971	1	Franz Bein	1982-06-26

Abbildung 5.13: Tabelle Fußballverein

5) Die folgende Tabelle enthält Daten zu Projekten. Aufgeführt sind Projekte und die erforderlichen Sprachkenntnisse, sowie die Mitarbeiter, die über die entsprechenden Sprachkenntnisse verfügen. Als Primärschlüssel ist ausschließlich eine Kombination aller drei Attribute möglich. Damit erfüllt die Tabelle die 3. Normalform. Dennoch sind redundante Daten vorhanden, da mehrwertige Abhängigkeiten vorkommen. Wie kann dieses Problem behoben werden?

Projektzuordnung

Personal-nummer	Projekt-nummer	Sprach-kenntnisse
1	1	Deutsch
1	1	Englisch
1	6	Englisch
2	1	Englisch
2	6	Franzosisch
3	1	Englisch
3	2	Spanisch

Abbildung 5.14: Tabelle Projektzuordnung

6 SQL – Anlegen der Datenbankstruktur

In Kapitel 6 sollen folgende Fragen geklärt werden:

- Wie kann man die Struktur des erstellten Relationenmodells auf ein relationales Datenbank-Management-System (RDBMS) übertragen?
- Was ist SQL?
- Was sind Datentypen?
- Wie speichert man Werte von Attributen, die man nicht kennt?
- Mit welchen Anweisungen der Sprache SQL erstellt man die Datenbankstruktur?
- Mit welchen Anweisungen der Sprache SQL kann man eine bestehende Datenbankstruktur wieder ändern?

6.1 Motivation

Herr Dr. Fleissig hat sein Relationenmodell nun soweit fertig, dass er es auf den Computer übertragen kann. Für einen ersten Test hat er sich die Datenbank-Management-Systemsoftware DB2 von der Firma IBM aus dem Internet heruntergeladen und installiert.

Nachdem er sich mit dem Produkt vertraut gemacht hat, startet er das dazugehörige Programm »Command Center« Mit Hilfe dieses Programms kann er Befehle in der Sprache SQL eingeben, um zunächst physikalisch eine leere Datenbank zu erzeugen. Danach gibt er entsprechende SQL-Anweisungen ein, um die Strukturen der Tabellen und deren Beziehungen in der Datenbank abzubilden.

6.2 Grundlagen

In den letzten beiden Kapiteln haben wir einen Teil der theoretischen Grundlagen des Relationenmodells kennen gelernt, auf dem relationale Datenbank-Management-Systeme (RDBMS) basieren. Ein RDBMS speichert die Datenbankstruktur in Form von Tabellen. Jede Tabelle besteht aus Zeilen und Spalten, in denen die Werte einzelner Objekte gespeichert werden.

Da durch ein RDBMS in der Regel mehrere tausend dieser Tabellen gespeichert werden, ist es sinnvoll, die Tabellen wiederum in eine bestimmte Hierarchie zu gliedern. Zur Übersicht verwendet eine RDBMS als oberste Hierarchiestufe die Kategorie. Eine Kategorie wiederum kann verschiedene Datenbank-Schemata speichern und ein Datenbank-Schema beinhaltet letztendlich die Tabellen mit den Spalten selbst.

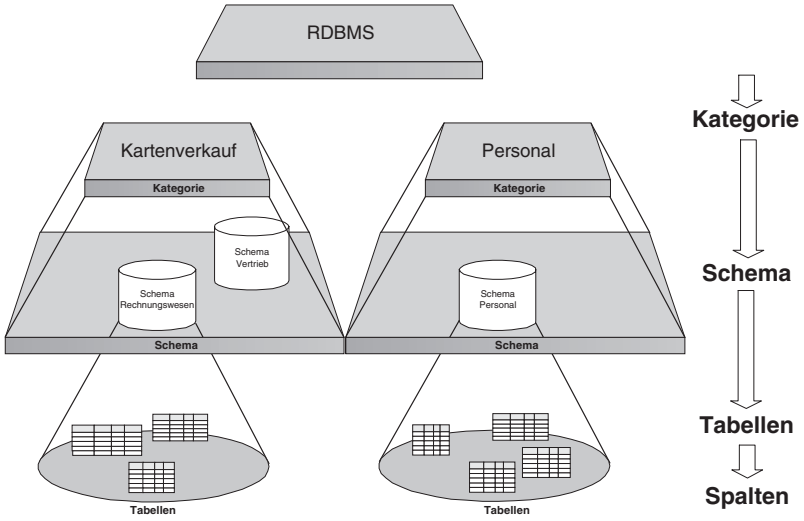


Abbildung 6.1: Hierarchische Struktur einer relationalen Datenbank

Diese Struktur dient ausschließlich der Übersicht, um nicht alle Tabellen eines RDBMS auf einer Hierarchiestufe speichern zu müssen.

Um nun Tabellenstrukturen in einer Datenbank anzulegen, verwendet man eine Datenbanksprache. Über diese Datenbanksprache kann man mit dem RDBMS kommunizieren, um diesem Anweisungen zu geben.

Für relationale Datenbanken hat sich die Sprache SQL durchgesetzt. SQL steht für »Structured Query Language« und wurde ursprünglich von der Firma IBM entwickelt. Nachdem E. F. Codd die Grundlagen über das Relationenmodell veröffentlicht hatte, begann IBM mit der Entwicklung eines ersten Prototypen für ein relationales Datenbank-Management-System mit dem Namen System/R. Ein Teil dieser Entwicklung bezog sich natürlich auch auf Abfragesprachen für System/R. Daraus entstand der Vorläufer des heutigen SQL, die Sprache SEQUEL (»Structured English Query Language«).

Mit der kommerziellen Verbreitung relationaler Datenbank-Management-Systeme seit Anfang der 80er Jahre, setzte sich auch SQL durch. Dies war der Anstoß dafür, dass sich das ANSI (»American National Standard Institute«) mit der Standardisierung der Sprache beschäftigte. 1986 wurde der erste SQL-Standard mit der Bezeichnung ANSI SQL-86 veröffentlicht. 1992 folgte die Publikation des ANSI SQL-92 Standards und 1999 des ANSI SQL:1999 Standards. Allgemein spricht man auch von SQL-1, SQL-2 und SQL-3.

Da die Standards keine zwingende Vorgabe sind für Firmen, die relationale Datenbank-Management-Systeme entwickeln, wurden sie bisher niemals komplett in einer der bekannten RDBMS implementiert. ANSI SQL ist zwar nicht das Maß aller Dinge, dennoch stimmen die grundlegenden Eigenschaften von SQL bei den jeweiligen RDBMS-Produkten weitgehend überein. Kennt man also den SQL Standard, ist es relativ einfach, Datenbankstrukturen für beliebige RDBMS-Produkte anzulegen, um dort

die gewünschten Daten zu verwalten. Allerdings ist es utopisch zu glauben, man beherrsche alle RDBMS-Produkte, wenn man den SQL-Standard kennt. Es ist eher vergleichbar mit dem Erlernen normaler Anwendungsprogramme. Wenn man sich z.B. mit dem Umgang eines ganz bestimmten Textverarbeitungsprogramms beschäftigt hat, fällt es einem in der Regel leichter, auch andere Textverarbeitungsprogramme zu bedienen, da die grundlegenden Prinzipien immer gleich sind.

Wir werden uns im Folgenden mit den wichtigsten Sprach-Eigenschaften des SQL:1999 Standards beschäftigen. Allerdings kann hier nicht der gesamte Standard behandelt werden, da dieser aus über 2000 Seiten besteht. Vielmehr handelt es sich um eine Einführung in aktuelle und neue Konzepte von SQL, die sich in den nächsten Jahren in den meisten RDBMS-Produkten durchsetzen werden bzw. schon vor der Verabschiedung des SQL:1999 Standards in den RDBMS-Produkten zu finden waren.

Bevor wir die erste SQL-Anweisung zum Anlegen von Tabellenstrukturen kennen lernen, müssen wir uns zunächst jedoch mit den Datentypen auseinandersetzen.

6.3 Datentypen

Wir haben in den letzen Kapiteln schon den Begriff des Wertebereichs oder der Domain kennen gelernt. Jede Eigenschaft eines bestimmten Objekts kann bestimmte Werte annehmen. Zum Beispiel kann das Alter einer Person in Jahren in der Regel nur Werte zwischen 0 und 150 annehmen, also nur ganzzahlige Werte (vorausgesetzt eine Person wird nicht älter als 150). Andererseits enthält die Eigenschaft Gehalt eines Mitarbeiters auch Zahlen, aber in diesem Fall mit Dezimalstellen. Der Name eines Mitarbeiters kann auch Buchstaben oder Sonderzeichen enthalten, das Geburtsdatum dagegen nur gültige Datumswerte. Es gibt also verschiedene Typen von Daten, je nachdem welche Werte ein Attribut annehmen darf.

SQL unterscheidet fünf Arten von vordefinierten atomaren Datentypen:

- Zahlen zum Speichern von numerischen Werten, z.B. Alter, Gehalt;
- Zeichenketten zum Speichern von beliebigen Zeichenfolgen, z.B. Beschreibung, Vorname, Strasse;
- Datum/Zeit zum Speichern von Zeitwerten, z.B. Geburtsdatum, Termin;
- Intervall zum Speichern von Zeitabständen, z.B. Dauer;
- Boolean zum Speichern von Wahr-/Falsch-Werten, z.B. »Führerschein vorhanden«.

Für jeden Datentyp kennt SQL ein bestimmtes Schlüsselwort, über das der Datentyp eines Attributes festgelegt wird. Will man einem Attribut einen bestimmten Wert zuweisen, so muss der Datentyp beachtet werden. Damit z.B. einer Zahl keine Zeichenkette zugewiesen werden kann, ist eine bestimmte Schreibweise für Werte von Datentypen vorgegeben. Man spricht hierbei von Literalen. Literale sind unveränderbare konstante Werte, die den Einschränkungen des zugehörigen Datentyps entsprechen.

Schlüsselwort	Beschreibung	Beispiel: Literal	Beispiel: Deklaration
SMALLINT	Ganzzahl mit geringerem Wertebereich als INTEGER	25	Alter SMALLINT
INTEGER	Ganzzahl	200399	Einwohnerzahl INTEGER
DECIMAL (p, s)	Zahl mit Dezimalstellen, p (precision) = Gesamtstellen, s (scale) = Dezimalstellen	12.99 +22 .6221	Gehalt DECIMAL (8,2)
NUMERIC (p, s)	Zahl mit Dezimalstellen, p = Gesamtstellen, s = Dezimalstellen	232.11 -1.22	Preis NUMERIC (5,2)
REAL	Angenäherte Zahl mit Dezimalstellen	4E3 -0.14E-8	Pi REAL Volumen REAL
FLOAT (p)	Angenäherte Zahl mit Dezimalstellen	4E3 -0.14E-8	Entfernung FLOAT (10) Breite FLOAT
DOUBLE	Angenäherte Zahl mit Dezimalstellen	4E3 -0.14E-8	Messwert DOUBLE Radius DOUBLE

Tabelle 6.1: Numerische Datentypen

Für alle numerischen Datentypen gilt, dass diese immer nur einen bestimmten Wertebereich abdecken können, da Computer nicht beliebig hohe bzw. beliebig niedrige Zahlen speichern können. So gilt z. B. bei den meisten RDBMS-Produkten für den Datentyp SMALLINT ein Wertebereich von -32.768 bis +32.767, dieser kann jedoch höher oder auch niedriger sein. Er ist aber immer auf einen bestimmten Bereich begrenzt. Eine Spalte, die also mit dem Datentyp SMALLINT angelegt wurde, kann als höchste Zahl den Wert +32.767 enthalten und als kleinste Zahl den Wert -32.768.

Generell unterscheidet man bei numerischen Datentypen zwischen exakten und angenäherten Datentypen. Die ersten vier Datentypen bilden einen numerischen Wert exakt ab. Dagegen gibt es angenäherte Datentypen, die berechnete Zahlen wie z. B. die Zahl Pi bis zu einer ganz bestimmten Anzahl Nachkommastellen speichern können.

Literale von angenäherten (approximierten) numerischen Datentypen werden in Form der E Notation ausgedrückt, wie es in Programmiersprachen wie C oder Pascal üblich ist. Der Buchstabe E steht dabei für »10 hoch irgendetwas«.
4E3 entspricht damit also der Zahl: $4 * 10^3 = 4 * 1000 = 4000$,
-321E-8 dagegen der Zahl $-321 * 10^{-8} = -321 * 0.00000001 = -0.00000321$.

Im kaufmännischen Bereich werden vorwiegend die numerischen Datentypen DECIMAL, NUMERIC und INTEGER benutzt. Numerische Datentypen mit angenähertem Wert sind hauptsächlich im wissenschaftlichen und mathematischen Bereich sinnvoll.

Zeichenketten sind sicherlich die am häufigsten verwendeten Datentypen innerhalb einer Datenbank, da Attribute dieses Typs allgemeine textuelle Beschreibungen enthal-

ten (schließlich kommunizieren wir ja mehr über Buchstaben und Wörter als über Zahlen). Attribute des Datentyps CHARACTER haben immer genau die in Klammern angegebene Länge und werden deshalb immer mit Leerzeichen aufgefüllt, wenn die eingetragenen Wörter weniger Buchstaben aufweisen. Attribute des Datentyps VARCHAR speichern ebenso wie CHARACTER auch Texte, allerdings in variabler Länge, also ohne Hinzufügen von Leerzeichen bei kürzeren Wörtern. Der Datentyp BIT wird selten eingesetzt, da er für das Speichern einzelner Bits (z.B. Ja/Nein-Zustände) gedacht ist. Mit SQL:1999 ist der Datentyp BOOLEAN hinzugekommen, der zwar auch zur Speicherung von Ja/Nein-Zuständen gedacht ist, aber benutzerfreundlicher ist, da hier mit Wahrheitswerten gearbeitet wird.

Schlüsselwort	Beschreibung	Beispiel: Literal	Beispiel: Deklaration
CHARACTER(n)	Zeichenkette mit fester Länge	'Hans'	Vorname CHAR(20)
CHARACTER VARYING (n) VARCHAR (n)	Zeichenkette variabler Länge	'Muster'	Name VARCHAR(50)
BIT (n)	Bitkette fester Länge	B'01101' X'12DEF'	Zustaende BIT (6)
BIT VARYING(n)	Bitkette variabler Länge	B'01101' X'12DEF'	Zustaende BIT (6)
CHARACTER LARGE OBJECT(n) CLOB (n)	Zeichenkette für Fließtext	'sehr langer Text'	Beschreibung CLOB(1M) Kapitel CLOB(20K) Lebenslauf CLOB(8000)
BINARY LARGE OBJECT(n) BLOB (n)	Bitkette für Binärdaten	B'01101' X'12DEF'	Foto BLOB (80M)

Tabelle 6.2: Zeichenketten

Schlüsselwort	Beschreibung	Beispiel: Literal	Beispiel: Deklaration
BOOLEAN	Boolescher Wert	TRUE FALSE	Fuehrerschein BOOLEAN

Tabelle 6.3: Boolean

Sind große Textdaten in einem Attribut zu speichern, so verwendet man den Datentyp CLOB. Als Parameter wird die Anzahl der Zeichen angegeben. Zur Vereinfachung darf man hier die beiden Buchstaben K für Kilobyte (=1024 Zeichen) und M für Megabyte (=1024*1024 Zeichen) verwenden.

Für große binäre Dateien, wie z.B. Audios, Videos oder Bilder, verwendet man den Datentyp BLOB.

Schließlich gibt es noch die folgenden Datentypen zur Speicherung von Datums- und Zeitwerten.

Schlüsselwort	Beschreibung	Beispiel: Literal	Beispiel: Deklaration
DATE	Datum	'2002-09-29'	Geburtsdatum DATE
TIME (p)	Uhrzeit	'20:15:00.000000' '18:00'	Spielbeg TIME
TIMESTAMP (p)	Datum und Uhrzeit	'2002-09-29 18:00'	Termin TIMESTAMP
INTERVAL YEAR(p) INTERVAL MONTH(p) INTERVAL YEAR(p) TO MONTH INTERVAL DAY(p) TO HOUR(p) INTERVAL DAY(p) TO MINUTE(p) INTERVAL DAY(p) TO SECOND(p) INTERVAL MINUTE(p) TO SECOND(p) INTERVAL SECOND(p,s)	Zeitintervalle	'3' '12' '62-10' '26 19:30' '-61 22:30' '21 31.0001' '15:31:0001' '59.000008'	Alter YEAR(3) Fallgeschw SECOND(2,6)

Tabelle 6.4: Datum / Zeit / Zeitintervall

Wir haben nun die wichtigsten Datentypen von SQL:1999 kennen gelernt und auch gesehen, wie man entsprechende Werte als Literale schreibt. Was in diesem Zusammenhang noch gar nicht angesprochen wurde, ist das Speichern von Werten, die entweder nicht bekannt sind oder nicht existieren. Angenommen, Sie sollen die Daten eines bestimmten Kunden in der Kundentabelle speichern, wissen jedoch nicht dessen Geburtsdatum. SQL bietet hierfür ein vordefiniertes Schlüsselwort: NULL.

Der Wort NULL steht nicht für die Zahl 0, NULL ist auch kein Wert, sondern soll einfach ausdrücken, dass ein Wert nicht bekannt ist. Dies hat zur Konsequenz, dass SQL mit einer dreiwertigen Logik arbeitet.

Betrachten wir hierzu ein Beispiel: Angenommen die Spalte Lagerbestand ist für einen bestimmten Werbeartikel nicht bekannt, also NULL. Würden wir nun vergleichen wollen, ob der Lagerbestand z.B. größer als 8 ist, so haben wir ein Problem. Der Vergleich »Lagerbestand > 8« ergibt weder wahr noch falsch, da wir den Lagerbestand ja nicht kennen. Das Ergebnis dieses Vergleiches ist dementsprechend also unbekannt.

Dieses Problem muss auch bei der Verknüpfung von Bedingungen mit UND/ODER berücksichtigt werden. Angenommen, wir wollen wissen, ob sowohl der Lagerbestand für einen bestimmten Werbeartikel größer 8 und gleichzeitig der Preis kleiner als 4,99 Euro ist. Die erste Bedingung ergibt, wie oben gesehen, unbekannt. Ist der Preis z.B. gleich 3 Euro, so ist die zweite Bedingung wahr. Eine UND-Verküpfung von unbekannt und wahr muss als Ergebnis wieder ein unbekannt ergeben, da wir ja nicht mit Sicher-

heit sagen können, ob die Bedingung erfüllt ist. Würden wir umgekehrt die beiden Bedingungen mit ODER verknüpfen, so ist das Ergebnis wahr. Betrachten wir abschließend die Wahrheitstabellen für wahr, falsch und unbekannt.

falsch	UND	falsch	=	falsch
falsch	UND	wahr	=	falsch
falsch	UND	unbekannt	=	falsch
wahr	UND	unbekannt	=	unbekannt
wahr	UND	wahr	=	wahr
unbekannt	UND	unbekannt	=	unbekannt

Tabelle 6.5: Wahrheitstabelle für UND-Verknüpfungen

falsch	ODER	falsch	=	falsch
falsch	ODER	wahr	=	wahr
falsch	ODER	unbekannt	=	unbekannt
wahr	ODER	unbekannt	=	wahr
wahr	ODER	wahr	=	wahr
unbekannt	ODER	unbekannt	=	unbekannt

Tabelle 6.6: Wahrheitstabelle für ODER-Verknüpfungen

6.4 Erzeugen und Bearbeiten einer Tabelle

6.4.1 Erzeugen einer Tabelle

Wie wird nun eine Tabelle angelegt? Hierzu kennt SQL die Anweisung CREATE TABLE. Betrachten wir hierzu die Tabelle Ort aus unserem Fallbeispiel. Diese besteht aus den beiden Attributen Ort und Postleitzahl. Die Anweisung, um diese Tabelle in einer Datenbank anzulegen, lautet:

```
CREATE TABLE Ort
(
       Plz   INTEGER,
       Ort   VARCHAR (200)
)
```

Mit dieser einfachen Anweisung haben wir die Tabelle »Ort« in unserer Datenbank angelegt. Da zu jeder Postleitzahl immer ein Ort bekannt ist, wollen wir allerdings verhindern, dass nicht aus Versehen eine Postleitzahl ohne Ort gespeichert wird. Dazu kann man in der CREATE TABLE-Anweisung angeben, ob ein Attribut NULL-Werte

enthalten darf. Daneben wollen wir unserem RDBMS auch bekannt geben, dass es sich bei dem Attribut Plz um den Primärschlüssel handelt. Die Anweisung dazu sieht wie folgt aus:

```
CREATE TABLE Ort
(
     Plz  INTEGER    NOT NULL ,
     Ort  VARCHAR (200)    NOT NULL ,
     -- Primärschlüssel festlegen
     CONSTRAINT primschluessel_Ort PRIMARY KEY (Plz)
)
```

Da die Attribute Plz und Ort kein NULL enthalten dürfen erscheinen nun hinter der Deklaration der Attribute und deren Datentypen die beiden Schlüsselwörter NOT NULL.

Integritätsbedingungen einer Tabelle gibt man in SQL über das Schlüsselwort CONSTRAINT bekannt. Hinter diesem Schlüsselwort folgt eine beliebige Bezeichnung für die Integritätsbedingung, und dann die Integritätsbedingung selbst. Über die beiden Schlüsselwörter PRIMARY KEY legt man den Primärschlüssel fest. In Klammern erscheint das Attribut oder die Attributkombination, die den Primärschlüssel darstellt, in unserem Beispiel das Attribut Plz.

Wie im letzten Beispiel gezeigt, kann man in SQL:1999 auch Kommentare einfügen, indem eine Zeile mit zwei Bindestrichen beginnt und dahinter der Kommentar folgt (z.B. -- Primärschlüssel festlegen). In dieser Form kann man einzeilige Kommentare schreiben. Mehrzeilige Kommentare werden durch die beiden Zeichen /* eingeleitet und durch die beiden Zeichen */ wieder beendet, z.B.

```
/* Dies ist ein Kommentar
   über zwei Zeilen */
```

Wir wollen jetzt eine weitere Tabelle aus unserm Fallbeispiel anlegen, die Tabelle »Spielstaette«. Um die Tabelle anzulegen, verwenden wir wieder die Anweisung CREATE TABLE. Über CONSTRAINT legen wir schließlich fest, welches Attribut als Primärschlüssel bestimmt wurde:

```
CREATE TABLE Spielstaette
(
     Haus          VARCHAR (100)   NOT NULL  ,
     Strasse       VARCHAR (50)    NOT NULL  ,
     Hausnummer    CHAR (6)                  ,
     Plz           INTEGER                   ,
     Beschreibung  CLOB(1M)                  ,

     CONSTRAINT primschluessel_Spielstaette PRIMARY KEY (Haus)
)
```

In den letzten beiden Kapiteln haben wir gesehen, dass Beziehungen zwischen Tabellen über Fremdschlüssel abgebildet werden. Unsere Tabelle Spielstaette besitzt einen solchen Fremdschlüssel, das Attribut Plz. Über das Attribut kann in der Tabelle Ort nachgesehen werden, welcher Ortsname zu welcher Postleitzahl gehört. In unserer obigen SQL-Anweisung haben wir dem RDBMS diese Beziehung jedoch noch nicht mitgeteilt. In Abschnitt 4.3 haben wir die verschiedenen Integritätsbedingungen kennen gelernt. Eine dieser Integritätsbedingungen, die referentielle Integrität, bezog sich auf die eben beschriebene Beziehung zwischen Primär- und Fremdschlüssel.

Da es sich bei der referenziellen Integrität wiederum um eine Einschränkung handelt, verwenden wir das SQL-Schlüsselwort CONSTRAINT gefolgt von einer beliebigen Bezeichnung für die Integritätsbedingung. Danach folgen die Schlüsselwörter, um einen Fremdschlüssel zu deklarieren: FOREIGN KEY. In Klammern wird dahinter das Attribut bzw. die Attributkombination angegeben, die den Fremdschlüssel darstellt. Schließlich muss man festlegen, auf welches Attribut in welcher Tabelle sich der Fremdschlüssel bezieht. Dies erfolgt über das Wort REFERENCES gefolgt vom Tabellennamen und dem Attribut. Mit Angabe des Fremdschlüssel wird die Tabelle »Spielstaette« wie folgt erzeugt:

```
CREATE TABLE Spielstaette
(
     Haus          VARCHAR (100)   NOT NULL  ,
     Strasse       VARCHAR (50)    NOT NULL  ,
     Hausnummer    CHAR (6)                  ,
     Plz           INTEGER                   ,
     Beschreibung  CLOB(1M)                  ,

     CONSTRAINT primschluessel_Spielstaette PRIMARY KEY (Haus),
     CONSTRAINT fremdschluessel_PlzOrt      FOREIGN KEY (Plz)
              REFERENCES Ort (Plz)
)
```

Betrachten wir hierzu folgendes Beispiel. Angenommen, in der Tabelle »Ort« wurde fälschlicherweise für den Ort Hamburg nicht die Postleitzahl 22287, sondern 33387 eingetragen. Auf diese Postleitzahl wird jedoch in der Tabelle »Spielstaette« referenziert:

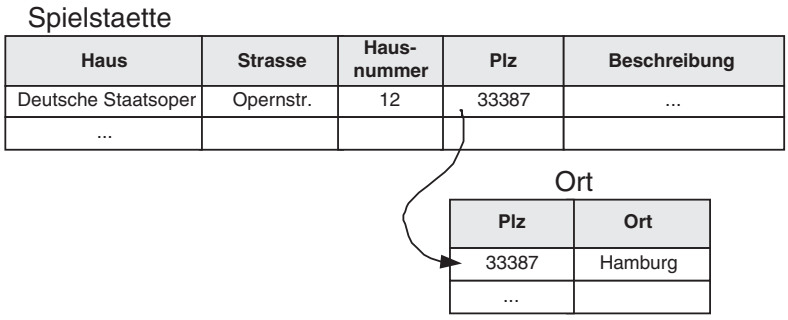


Abbildung 6.2: Problem der referenziellen Integrität

Würde man nun die Postleitzahl in der Tabelle Ort von 33387 in 22287 ändern, so würde die Spielstaette »Deutsche Staatsoper« auf einen Eintrag in der Tabelle Ort verweisen, den es nicht mehr gibt. Die Beziehung würde also sozusagen ins »Leere« laufen. Das Gleiche gilt, wenn der Satz mit der Plz 33387 aus der Tabelle »Ort« gelöscht würde.

Um solche Probleme zu vermeiden, erlaubt ein RDBMS das Ändern oder Löschen eines Satzes nicht, solange der Wert eines Fremdschlüssels noch auf die Tabelle verweist.

Da diese Einschränkung jedoch sehr restriktiv ist, kann man über SQL bei der Festlegung eines Fremdschlüssels angeben, wie mit dem Fremdschlüssel verfahren werden soll, wenn ein solches Problem auftritt. SQL:1999 kennt fünf mögliche Aktionen, die bei einer Fremdschlüsselverletzung ausgeführt werden können:

- RESTRICT oder NO ACTION
- SET NULL
- SET DEFAULT
- CASCADE

RESTRICT entspricht der Standardeinstellung und verhindert das Ändern oder Löschen eines Satzes, auf den sich ein Fremdschlüssel bezieht. In unserem Beispiel würde also das Ändern oder Löschen zurückgewiesen.

SET NULL legt fest, dass der Fremdschlüssel auf NULL gesetzt wird, somit kein Bezug mehr zur Elterntabelle vorhanden ist.

SET DEFAULT entspricht SET NULL, mit dem Unterschied, dass ein vorher definierter Standardwert für den Fremdschlüssel verwendet wird. Wir werden später in diesem Kapitel das Schlüsselwort DEFAULT kennen lernen. Hiermit kann ein Standardwert festgelegt werden, der bei Einfügen eines Satzes für ein Attribut verwendet wird, wenn für dieses Attribut nicht explizit ein Wert angegeben wurde. Angenommen, aus der Abteilungstabelle wird die Abteilung »Vertrieb« gelöscht. DEFAULT sorgt nun dafür, dass den Mitarbeitern, die der gelöschten Abteilung »Vertrieb« zugeordnet waren, der definierte Standardwert »Rechnungswesen« als Fremdschlüssel zugeordnet wird, sofern die Tabelle »Mitarbeiter« wie folgt angelegt wurde:

```
CREATE TABLE Mitarbeiter
(
     Personalnummer   INTEGER NOT NULL,
     ...
     Abteilungsbezeichnung VARCHAR (30) DEFAULT 'Rechnungswesen',
     ...
)
```

CASCADE bewirkt schließlich, dass der Wert des Fremdschlüssels automatisch auf den Wert des geänderten Primärschlüssels gesetzt wird bzw. der Satz sowohl aus der Elterntabelle als auch aus der Kindtabelle gelöscht wird.

Bei Anwendung der Aktionen SET NULL und CASCADE würden die Tabellen nach der Änderung wie folgt aussehen.

SET NULL

Spielstaette

Haus	Strasse	Haus-nummer	Plz	Beschreibung
Deutsche Staatsoper	Opernstr.	12	NULL	...
...				

Ort

Plz	Ort
22287	Hamburg
...	

Abbildung 6.3: Referenzielle Integrität bei SET NULL

CASCADE

Spielstaette

Haus	Strasse	Haus-nummer	Plz	Beschreibung
Deutsche Staatsoper	Opernstr.	12	22287	...
...				

Ort

Plz	Ort
22287	Hamburg
...	

Abbildung 6.4: Referenzielle Integrität bei CASCADE

Doch wie gibt man über SQL dem RDBMS bekannt, welche Aktion bei Verletzung der referentiellen Integrität verwendet werden soll? Die Aktion gehört mit zur Definition des Fremdschlüssels und folgt dieser direkt. Angenommen, wir möchten, dass das Attribut Plz auf NULL gesetzt wird, wenn der entsprechende Satz in der Tabelle »Ort« gelöscht wird. Außerdem soll die Plz in der Tabelle »Spielstaette« automatisch geändert werden, wenn diese in der Tabelle »Ort« geändert wird. Die SQL-Anweisung hierfür sieht folgendermaßen aus:

```
CREATE TABLE Spielstaette
(
     Haus               VARCHAR (100)    NOT NULL     ,
     Strasse            VARCHAR (50)     NOT NULL     ,
     Hausnummer         CHAR (6)                      ,
     Plz                INTEGER                       ,
     Beschreibung       CLOB(1M)                      ,

     CONSTRAINT primschluessel_Spielstaette PRIMARY KEY (Haus),
     CONSTRAINT fremdschluessel_PlzOrt      FOREIGN KEY (Plz)
                  REFERENCES Ort (Plz)
                  /* bei Löschen des zugehörigen Ortes die Plz
                     automatisch auf NULL setzen */
                                          ON DELETE SET NULL
                  /* bei Ändern der zugehörigen Plz in der
                  Orttabelle, die Plz von Spielstaette
                  automatisch anpassen */
                  ON UPDATE CASCADE
)
```

Zuerst erscheint das Ereignis (ON DELETE oder ON UPDATE) und dann die Aktion, die bei Auftreten des entsprechenden Ereignisses ausgeführt werden soll.

Von den drei Integritätsbedingungen, die wir in Abschnitt 4.3 behandelt haben, können wir bisher die referentielle Integrität (FOREIGN KEY) und die Entity-Integrität (PRIMARY KEY) über SQL abbilden. Fehlt also noch die Darstellung von Domain-Integritätsbedingungen in SQL.

Nehmen wir als Beispiel wieder unsere Tabelle Ort. Als Datentyp für die Postleitzahl haben wir eine Ganzzahl (INTEGER) verwendet. Der Wertebereich für einen INTEGER liegt bei den meisten RDBMS-Produkten zwischen -2.147.483.647 und +2.147.483.648. Allerdings wissen wir, dass eine Postleitzahl in Deutschland immer aus fünf Zahlen besteht, Werte größer als 99.999 also nicht möglich sind (es gibt bei Postleitzahlen zwar noch mehr Einschränkungen, der Übersichtlichkeit halber bleiben wir hier aber bei dieser vereinfachten Darstellung).

Um Einschränkungen solcher Art über SQL auszudrücken, kennt SQL verschiedene Sprachelemente. Ein Sprachelement, dass hierfür innerhalb der CREATE TABLE-Anweisung verwendet werden kann, ist das Schlüsselwort CHECK.

CHECK wird wie die anderen beiden Integritätsbedingungen auch durch das Schlüsselwort CONSTRAINT und eine Bezeichnung für diese Einschränkung eingeleitet. Darauf folgt das Wort CHECK und dann die Bedingung, die bei Einfügen eines Satzes in die Tabelle erfüllt sein muss.

Für unser Beispiel mit der Postleitzahl sieht das dann wie folgt aus:

```
CREATE TABLE Ort
(
     Plz  INTEGER    NOT NULL ,
```

```
    Ort  VARCHAR (200)    NOT NULL ,
    CONSTRAINT primschluessel_Ort PRIMARY KEY (Plz),
    CONSTRAINT pruef_Plz CHECK (Plz > 0 AND Plz <= 99999)
)
```

Bedingungen, wie für die Einschränkung der Postleitzahl, werden wir detaillierter in Kapitel 8 behandeln.

Zum Schluss wollen wir uns noch kurz ansehen, wie man Standardwerte definieren kann, die verwendet werden, wenn für eine Spalte nicht explizit angegeben wurde, welchen Wert diese beim Einfügen eines Satzes bekommen soll. Betrachten wir hierzu die Tabelle »Kunde«. Wird ein Satz in die Tabelle »Kunde« eingefügt, so soll für die Spalte »Geschlecht« automatisch der Wert ›W‹ für weiblich verwendet werden, sofern nicht explizit ein anderer Wert angegeben wurde. Die SQL-Anweisung zum Anlegen der Tabelle sieht dann wie folgt aus:

```
CREATE TABLE Kunde
(
    Kundennummer              INTEGER NOT NULL              ,
    Name                      VARCHAR (30) NOT NULL         ,
    Vorname                   VARCHAR (20) NOT NULL         ,
    Geschlecht                CHAR DEFAULT 'W'              ,
    Strasse                   VARCHAR (50) NOT NULL         ,
    Hausnummer                CHAR (6)                      ,
    Plz                       INTEGER                       ,
    CONSTRAINT pk_Kunde       PRIMARY KEY (Kundennummer)    ,
    CONSTRAINT fk_Plz         FOREIGN KEY (Plz)
              REFERENCES Ort (Plz)
               ON DELETE SET NULL
               ON UPDATE CASCADE,
    CONSTRAINT di_Geschlecht CHECK (Geschlecht IN ('M', 'W'))
)
```

Anstatt einen konstanten Wert als Standardwert anzugeben, besteht auch die Möglichkeit vom RDBMS vordefinierte Werte zu verwenden. So liefert z.B. das Schlüsselwort CURRENT_DATE das aktuelle Datum. Gibt man z.B. bei der Tabelle »Bestellung« als Standardwert CURRENT_DATE an, so wird bei Einfügen eines Satzes in diese Tabelle automatisch das aktuelle Datum als Bestelldatum gesetzt. Neben CURRENT_DATE gibt es noch Funktionen CURRENT_TIME für die aktuelle Zeit, CURRENT_TIMESTAMP für das aktuelle Datum und die aktuelle Zeit, sowie CURRENT_USER für den aktuell angemeldeten Benutzer.

Auf die Angabe der Länge einer Zeichenkette kann man verzichten, wenn diese genau 1 beträgt. Man könnte synonym also auch CHAR(1) schreiben. Dahinter folgt die Angabe des Vorgabewertes, der verwendet werden soll, wenn bei Einfügen eines Satzes der Wert für Geschlecht nicht explizit angegeben wurde. Damit Geschlecht nur die Werte ›W‹ für weiblich und ›M‹ für männlich annehmen kann, wurde eine entsprechende Einschränkung hinzugefügt. Über das Schlüsselwort IN kann überprüft wer-

den ob der Wert für ein Attribut einer bestimmten Wertemenge, in diesem Fall ›M‹ und ›W‹ entspricht. Das Schlüsselwort IN und weitere Bedingungsausdrücke werden wir detailliert in Kapitel 8 kennen lernen.

Im Folgenden wird beschrieben, wie SQL noch weitere Domain-Integritätsbedingungen unterstützt. Hierzu werden wir kennen lernen, wie man allgemeine Datentypen mit bestimmten Einschränkungen erstellen und einfache eigene Datentypen definieren kann.

6.4.2 Erstellen einfacher benutzerdefinierter Datentypen

Bisher haben wir ausschließlich vordefinierte Datentypen kennen gelernt, also Datentypen, die uns von SQL vorgegeben werden. SQL bietet jedoch die Möglichkeit, eigene Datentypen zu erstellen, die auf vordefinierten Datentypen basieren. Nehmen wir z. B. unser Attribut Plz. Dieses Attribut kommt in verschiedenen Tabellen vor, z. B. »Ort«, »Spielstaette« oder »Kunde«. In jeder Tabelle muss darauf geachtet werden, dass der gleiche vordefinierte Datentyp, nämlich INTEGER, verwendet wurde.

Es ist deshalb möglich, einen einfachen Datentyp selbst zu erstellen, den man dann z. B. mit DTyp_Plz bezeichnet. Das Anlegen einfacher Datentypen erfolgt über die SQL-Anweisung CREATE DISTINCT TYPE und sieht für das Attribut »Plz« dann folgendermaßen aus:

```
CREATE DISTINCT Dtyp_Plz AS INTEGER
```

Anstelle des vordefinierten Datentyps INTEGER kann nun für die Postleitzahl der benutzerdefinierte Datentyp DTyp_Plz verwendet werden. Für das Anlegen der Tabelle »Ort« bedeutet das:

```
CREATE TABLE Ort
(
      Plz             DTyp_Plz           NOT NULL  ,
      Ort             VARCHAR (200)      NOT NULL  ,
      CONSTRAINT primschluessel_Ort PRIMARY KEY (Plz)
      CONSTRAINT pruef_Plz CHECK (Plz > 0 AND Plz <= 99999)
)
```

Sicherlich werden Sie sich jetzt fragen, welchen Vorteil der benutzerdefinierte Typ DTyp_Plz gegenüber dem vordefinierten Datentyp INTEGER hat. Der Grund für die Verwendung benutzerdefinierter Datentypen liegt in der strengen Typüberprüfung. Sind z. B. sowohl das Attribut »Alter« als auch das Attribut »Plz« als INTEGER definiert, ist es ohne weiteres möglich, dem Attribut »Alter« den Wert des Attributes »Plz« zuzuweisen. Ebenso werden Vergleiche zwischen den Attributen gleichen Datentyps vermieden, wie z. B. »Alter« und »Plz«, die wenig sinnvoll sind.

Um solche Probleme zu vermeiden, verwendet man benutzerdefinierte Datentypen. Wird das Attribut »Plz« mit dem benutzerdefinierten Datentyp DTyp_Plz deklariert, so ist es nicht mehr möglich die beiden Attribute »Alter« und »Plz« miteinander zu vergleichen, da es sich ja nun sozusagen um unterschiedliche Datentypen handelt.

6.4.3 Überprüfungen von Wertebereichen

Seit dem ANSI SQL-92 Standard kennt SQL den Begriff der Domain. In SQL stellt dieses ein allgemeines Konzept dar, um Wertebereiche von Attributen zu überprüfen. Dazu erstellt man eine Domain und kann diese anstelle eines beliebigen Datentyps einsetzen. Diese Vorgehensweise erinnert stark an die gerade besprochenen benutzerdefinierten Datentypen. Im Gegensatz zu benutzerdefinierten Datentypen sind Domains aber eher eine Schreiberleichterung, da hiermit Datentypen und deren Einschränkungen an einer Stelle beschrieben und für unterschiedliche Attributdeklarationen verwendet werden können. Jedoch unterliegen durch Domain deklarierte Datentypen keiner strengen Typüberprüfung, was ja bei benutzerdefinierten Datentypen der Fall ist.

Eine Domain wird mit der SQL-Anweisung CREATE DOMAIN erstellt. Betrachten wir hierzu unsere Tabelle Kunde. Diese besitzt das Attribut »Geschlecht«, das auch in den Tabellen »Mitarbeiter« und »Kind« vorkommt. Das Erzeugen der Domain für dieses Attribut sieht wie folgt aus:

```
CREATE DOMAIN dom_Geschlecht AS CHAR(1)
             DEFAULT 'W'
             CHECK (VALUE IN ('W', 'M'))
```

Eine Domain wird ähnlich wie ein einfacher benutzerdefinierter Datentyp angelegt. Neben dem vordefinierten Datentyp, hier CHAR(1), kann der Wertebereich jedoch weitergehend überprüft werden, so wie wir es bei den Integritätsbedingungen zum Anlegen einer Tabelle bereits gesehen haben. Das Schlüsselwort VALUE kennzeichnet dabei den aktuellen Wert der Domain.

Für das Anlegen der Tabelle Kunde bedeutet das:

```
CREATE TABLE Kunde
(
      Kundennummer             INTEGER         NOT NULL     ,
      Name                     ARCHAR (30)     NOT NULL     ,
      Vorname                  VARCHAR (20)    NOT NULL     ,
      Geschlecht               dom_Geschlecht               ,
      Strasse                  VARCHAR ( 50)   NOT NULL     ,
      Hausnummer               CHAR (6)                     ,
      Plz                      DTyp_Plz                     ,
      CONSTRAINT pk_Kunde         PRIMARY KEY (Kundennummer),
      CONSTRAINT fk_Plz           FOREIGN KEY (Plz)
                  REFERENCES   Ort (Plz)
                  ON DELETE SET NULL
                  ON UPDATE CASCADE
)
```

Entgegen der ersten Lösung zum Anlegen dieser Tabelle, können wir nun auch auf die Einschränkung zum Überprüfen der Werte für Geschlecht verzichten, da wir dies ja bereits durch Anlegen der Domain beschrieben haben.

6.4.4 Reihen (Arrays)

ANSI SQL:1999 unterstützt auch Arrays. Hierbei handelt es sich um Wiederholgruppen eines bestimmten Attributs. Angenommen, Sie wollen in der Tabelle »Kunde« maximal drei unterschiedliche Telefonnummern speichern, so können Sie dies auf drei verschiedene Arten tun. Entweder definieren Sie drei verschiedene Attribute, »Telefon1«, »Telefon2« und »Telefon3«. Oder Sie erzeugen eine eigene Tabelle »Telefon«, die als Fremdschlüssel die Kundennummer enthält (also 1:N-Beziehung zwischen Kunde und Telefon). Die dritte Möglichkeit schließlich ist die Verwendung eines Arrays wie folgt:

```
CREATE TABLE Kunde
(
     ...
     Telefon    CHAR (10) ARRAY[3]    ,
     ...
```

Sicherlich werden Sie jetzt erwidern, dass dies aber doch der ersten Normalform und damit dem Relationenmodell widerspricht. Die erste Normalform besagt ja, dass Tabellen keine Wiederholgruppen enthalten dürfen und jeder Wert eines Attributs atomar sein soll. Das stimmt! Mit ANSI SQL:1999 weicht man das Relationenmodell in bestimmten Bereichen zugunsten pragmatischer Ansätze und der objektorientierten Softwareentwicklung auf.

6.4.5 Ändern und Löschen

Bis hierher haben wir gesehen, wie man Tabellen anlegt und dass man zum Beschreiben der Attribute vordefinierte Datentypen, benutzerdefinierte Datentypen und Domains verwenden kann. Allerdings haben wir diese Datenbankobjekte nur erzeugt, aber noch nicht besprochen, wie man diese löscht bzw. ändert.

SQL kennt die Schlüsselworter ALTER für das Ändern und DROP für das Löschen von Datenbankobjekten. Um z. B. die Tabelle »Ort« wieder zu löschen, schreibt man:

```
DROP TABLE Ort
```

Was ist jedoch, wenn auf Sätze der Tabelle »Ort« noch durch Fremdschlüssel verwiesen wird? Generell gilt, dass eine Tabelle nicht gelöscht werden kann, wenn ein Fremdschlüssel einer anderen Tabelle auf den Primärschlüssel dieser Tabelle verweist, da sonst die referenzielle Integrität verletzt würde. Dies gilt für alle Datenbankobjekte. Datenbankobjekte, auf die andere Datenbankobjekte verweisen, können also nicht gelöscht werden. D.h., bevor man ein Datenbankobjekt löschen kann, auf das verwiesen wird, müssen erst alle Datenbankobjekte gelöscht werden, die dieses verwenden.

Der SQL Standard bietet seit 1992 zwar die Möglichkeit, am Ende der DROP-Anweisung das Schlüsselwort CASCADE anzugeben, um automatisch alle Datenbankobjekte zu löschen, die auf dieses verweisen, doch selbst die bekannten RDBMS-Produkte unterstützen diese Spracheigenschaft nicht. Die Konsequenzen einer DROP-Anwei-

sung mit der Option CASCADE sind manchmal sicherlich auch nicht überschaubar. Denken Sie an ein Datenbankmodell von mehreren tausend Tabellen, die sich gegenseitig referenzieren. Würden Sie nun eine zentrale Tabelle löschen, so würde dies das automatische Löschen aller Tabellen nach sich ziehen, die diese verwenden. Die referenzierten Tabellen wiederum würden andere Tabellen nach sich ziehen usw. Also selbst wenn CASCADE in einem RDBMS als Spracheigenschaft für DROP unterstützt wird, sollte man es mit einer gewissen Vorsicht verwenden.

Exemplarisch löschen wir nun noch einmal unseren angelegten benutzerdefinierten Datentyp und unsere Domain:

```
DROP DISTINCT TYPE DTyp_Plz
DROP DOMAIN dom_Geschlecht
```

Das nachträgliche Ändern eines Datenbankobjektes ist natürlich etwas aufwändiger als das Löschen. Schauen wir uns hierzu das Ändern von Einschränkungen und Attributen einer Tabelle an. Um ein Attribut zu löschen, verwendet man zunächst die SQL-Anweisung ALTER TABLE gefolgt vom Tabellennamen. Danach folgt die Anweisung DROP zum Löschen eines Attributes und dann der Attributname.

```
ALTER TABLE Kunde DROP COLUMN Geschlecht
```

Um die Spalte »Geschlecht« nachträglich der Tabelle wieder hinzuzufügen, schreibt man:

```
ALTER TABLE Kunde ADD COLUMN Geschlecht CHAR(1)
```

Entsprechend legt man Integritätsbedingungen an bzw. löscht sie wieder. Um z. B. den Primärschlüssel der Tabelle »Ort« zu löschen und danach wieder anzulegen, schreibt man:

```
ALTER TABLE Ort
      DROP CONSTRAINT primschluessel_Ort
ALTER TABLE Ort
      ADD CONSTRAINT primschluessel_Ort PRIMARY KEY (Plz)
```

6.5 Zusammenfassung

In diesem Kapitel haben wir gelernt, wie man eine Datenbankstruktur mit Hilfe der SQL-Anweisung CREATE TABLE anlegt. Dazu haben wir gesehen, welche vordefinierten Datentypen SQL bietet, und dass einfache eigene Datentypen mit CREATE DISTINCT TYPE erstellt werden. Mit CREATE DOMAIN kann man eigene Wertebereiche erzeugen, die bestimmten Einschränkungen entsprechen. Benutzerdefinierte Datentypen und Domains können dabei überall dort eingesetzt werden, wo ein Datentyp für ein Attribut festgelegt werden muss.

Allgemein wird die Gruppe der SQL-Anweisungen zum Anlegen von Datenbankobjekten als »Data Definition Language« (DDL) bezeichnet.

Beim Erstellen der Tabelle haben wir auch gesehen, wie man Integritätsbedingungen über das Schlüsselwort CONSTRAINT definiert. Integritätsbedingungen dienen dazu, dem RDBMS Geschäftsregeln bekanntzugeben, damit dieses auf deren Einhaltung achten kann.

Am Ende haben wir betrachtet, wie man diese Datenbankobjekte wieder löschen bzw. nachträglich ändern kann.

Sicherlich werden Sie sich jetzt fragen, wie man denn nun eine Datenbank selbst erzeugt. ANSI SQL selbst bietet keine SQL-Anweisung, um leere Datenbanken anzulegen. Der Grund hierfür liegt darin begründet, dass das physikalische Erzeugen einer Datenbank zum großen Teil abhängig ist von der Computerplattform, weshalb man hierauf verzichtet hat. Dennoch bieten die meisten RDBMS-Produkte eigene SQL-Sprachelemente, um Datenbanken physikalisch zu erzeugen. In der Regel legt man eine leere Datenbank über die Schlüsselwörter CREATE DATABASE an und kann diese über DROP DATABASE wieder löschen. Unsere Beispieldatenbank können wir also wie folgt unter dem Namen »Vertrieb« anlegen:

```
CREATE DATABASE Vertrieb
```

Zum Schluss noch ein Hinweis zur Reihenfolge, in der Tabellen angelegt werden sollten. Wird beim Anlegen von Tabellen auf andere Tabellen über Fremdschlüssel referenziert, muss darauf geachtet werden, zuerst die Tabellen anzulegen, auf die in anderen Tabellen Bezug genommen wird. Betrachten wir hierzu noch einmal unser Beispiel mit den Tabellen »Ort« und »Spielstaette«. Die Tabelle »Spielstaette« referenziert die Tabelle »Ort« über den Fremdschlüssel »Plz«. Würde man nun versuchen, zuerst die Tabelle »Spielstaette« unter Angabe der Fremdschlüsselbeziehung anzulegen, kommt es zwangsläufig zu einer Fehlermeldung des RDBMS, da die Tabelle »Ort« noch nicht existiert. Häufig legt man deshalb zunächst alle Tabellen ohne Fremdschlüsselbezug an, also ohne FOREIGN KEY-Bedingung. Dies hat den Vorteil, dass man beim Anlegen der Tabellen auf keine bestimmte Reihenfolge achten muss. Danach definiert man über ALTER TABLE nachträglich alle referentiellen Integritätsbedingungen. Für »Ort« und »Spielstaette« würde das dann exemplarisch folgendermaßen aussehen:

```
CREATE TABLE Spielstaette
(
     Haus             VARCHAR (100)   NOT NULL     ,
     Strasse          VARCHAR (50) NOT NULL        ,
     Hausnummer       CHAR (6)                     ,
     Plz              INTEGER                      ,
     Beschreibung     CLOB(1M)                     ,
     CONSTRAINT primschluessel_Spielstaette PRIMARY KEY (Haus)
)
CREATE TABLE Ort
(
     Plz        INTEGER           NOT NULL ,
     Ort        VARCHAR (200)     NOT NULL ,
   CONSTRAINT primschluessel_Ort PRIMARY KEY (Plz)
```

```
)
ALTER TABLE Spielstaette ADD CONSTRAINT fremdschluessel_PlzOrt
                         FOREIGN KEY (Plz) REFERENCES Ort (Plz)
                         ON DELETE SET NULL
                         ON UPDATE CASCADE
```

Entsprechendes gilt für benutzerdefinierte Datentypen und Domains. Diese sollten natürlich vor dem Anlegen der Tabelle erstellt werden, in der sie verwendet werden.

Generell gilt also folgende Vorgehensweise:

1) Anlegen aller benutzerdefinierter Datentypen und Domains;
2) Erstellen der Tabellen ohne Berücksichtigung der referentiellen Integrität;
3) Referentielle Integritätsbedingungen nachträglich über ALTER TABLE den Tabellen hinzufügen.

6.6 Aufgaben

Wiederholungsfragen

1) Wie lautet der SQL-Befehl zum Anlegen einer Tabelle?
2) Wie lautet der SQL-Befehl zum Anlegen eines einfachen benutzerdefinierten Datentyps?
3) Welche numerischen Datentypen gibt es?
4) Welche Zeichenketten-Datentypen gibt es?
5) Geben Sie für folgende Attribute einen vordefinierten Datentyp an:
   »Vorname«, »Mitarbeiterfoto«, »Mitarbeiter_Lebenslauf«, »Plz«, »Gehalt«

Übungen

1) Die Firma »KartoFinale« will zukünftig international tätig sein. Dazu muss die Tabelle »Kunde« um ein Attribut für Land ergänzt werden. Wie lautet die SQL-Anweisung, um nachträglich diese Spalte hinzuzufügen?
2) Die in Übung 1 erstellte Spalte »Land« soll wieder gelöscht werden, da man den Wertebereich für »Land« einschränken möchte. Dazu soll eine Domain erstellt werden, die nur Buchstaben zulässt. Wie sieht die SQL-Anweisung zum Löschen der Spalte aus? Wie sieht die SQL-Anweisung zum Anlegen der Domain aus?
3) In der Tabelle »Veranstaltung« haben wir eine Spalte für den Autor vorgesehen. Eine Veranstaltung kann jedoch auch von mehreren Autoren sein. Wie lautet die SQL-Anweisung unter Verwendung eines ARRAY zum Anlegen dieser Tabelle, wenn man maximal 8 Autoren pro Veranstaltung speichern möchte?
4) Erstellen Sie alle Tabellen aus unserem Fallbeispiel!
   (Hinweis: SQL-Skripte für das Fallbeispiel sind im Internet unter »http://www.addison-wesley.de/DBMS« verfügbar)
5) Erzeugen Sie die Tabellen zu den Relationenmodellen von Übung 3 aus Kapitel 4!

7 Einfuegen, Aendern, Loeschen von Daten

In Kapitel 7 sollen folgende Fragen geklärt werden:

- Wie fügt man einen oder mehrere Datensätze in eine Tabelle ein?
- Wie löscht man einen oder mehrere Datensätze aus einer Tabelle?
- Wie ändert man nachträglich die Werte von Spalten eines oder mehrerer Datensätze?

7.1 Motivation

- Herr Dr. Fleissig hat die Datenbankstruktur für die Firma »KartoFinale«erstellt. Für den nächsten Tag hat er einen Termin mit Frau Kart vereinbart. Als vorbildliche Mitarbeiterin möchte Frau Kart so bald wie möglich anfangen, die Datenbank zu verwenden. Herr Fleissig erklärt ihr kurz, dass es zum Einfügen, Ändern und Löschen von Einträgen in die Tabellen Datenbankanweisungen gibt. Damit sie mit dem RDBMS »kommunizieren« kann, muss sie die Sprache des RDBMS, nämlich SQL, erlernen.
- Nach zwei Stunden ist Frau Kart schließlich so weit und beginnt die Kunden-, Mitarbeiter- und Abteilungsdaten zu erfassen.

7.2 Einfuegen von Datensaetzen

Um Daten in eine Tabelle einzufügen, verwendet man die SQL-Anweisung INSERT. Betrachten wir zunächst die einfachste Variante von INSERT am Beispiel unserer Tabelle »Ort« Das Einfügen des Ortes Kohlscheidt mit der Postleitzahl 44444 sieht wie folgt aus:

```
INSERT INTO Ort
VALUES ( 44444, 'Kohlscheidt' )
```

Um mehrere Sätze gleichzeitig einzufügen, kann man die Werte mehrerer Sätze jeweils durch Klammern begrenzt angeben. Um zwei weitere Sätze in die Tabelle »Ort«einzufügen, schreibt man:

```
INSERT INTO Ort
VALUES ( 22222, 'Karlstadt' ),
       ( 33333, 'Rettrich' )
```

Wir haben bisher Sätze eingefügt, bei denen wir die Werte aller Spalten und auch die Reihenfolgen der Spalten kennen mussten. Die Reihenfolge der Spalten ergibt sich aus der CREATE TABLE-Anweisung und entspricht der Reihenfolge, in der sie dort aufgeführt wurden. Um nicht immer die Reihenfolge beachten zu müssen, ist es möglich,

hinter dem Tabellennamen die Spaltennamen in Klammern aufzuführen, deren Werte hinter dem Schlüsselwort VALUES folgen. Betrachten wir hierzu die Tabelle Kunde. Wir wollen die Kundin Frieda Wiegerich in die Tabelle Kunde einfügen. Da beim Anlegen der Tabelle Kunde das Geschlecht ‚W' für weiblich als Vorgabewert definiert wurde, brauchen wir diese Spalte beim Einfügen nicht mit anzugeben. Die Hausnummer dieser Kundin ist uns nicht bekannt, also lassen wir diese vorerst weg. Die SQL-Anweisung dazu lautet:

```
INSERT INTO Kunde (Name, Vorname, Strasse, Plz, Kundennummer)
VALUES ( 'Wiegerich', 'Frieda', 'Wanderstr.', 33333, 3 )
```

Unsere Tabelle Kunde sieht danach wie folgt aus:

Kunden-nummer	Name	Vorname	Geschlecht	Strasse	Haus-nummer	Plz
3	Wiegerich	Frieda	W	Wanderstr.	NULL	33333

Abbildung 7.1: Tabelle Kunde

Die Kundennummer ist in dieser INSERT-Anweisung zum Schluss aufgeführt, also entspricht die Zahl 3 der Kundennummer. Ohne Angabe der Spaltennamen hätte der Wert für die Kundennummer in der SQL-Anweisung an erster Stelle stehen müssen.

Doch wie kann der Satz ohne Angabe der Spaltennamen eingefügt werden. Dazu verwendet man das Schlüsselwort DEFAULT, um explizit den Vorgabewert zu verwenden oder das Schlüsselwort NULL, um einen Wert als unbekannt zu kennzeichnen. Die folgende INSERT-Anweisung entspricht damit der oberen, ist nur etwas länger:

```
INSERT    INTO Kunde
VALUES    ( 3, 'Wiegerich', 'Frieda',
          DEFAULT, 'Wanderstr.', NULL, 33333 )
```

Was geschieht aber nun, wenn ein Kunde mit einer Postleitzahl eingefügt wird, die als Primärschlüssel in der Tabelle »Ort« noch nicht eingefügt ist? Die Antwort hierzu sollte nicht schwer fallen: Das Einfügen wird natürlich vom RDBMS zurückgewiesen, da die referentielle Integrität verletzt wurde. Sie sehen also, wie wichtig es ist, solche Integritätsregeln im RDBMS zu deklarieren. Je mehr das RDBMS an Geschäftsregeln kennt, um so besser kann es die Datenbank vor inkonsistenten Daten schützen. Referentielle Integrität ist dabei natürlich nur eine Möglichkeit, Geschäftsregeln dem RDBMS bekannt zu geben. Wir werden im weiteren Verlauf des Buches viele Möglichkeiten kennen lernen, solche Einschränkungen im RDBMS umzusetzen.

Zur Übung wollen wir nun noch drei weitere Kunden einfügen. Die Anweisung dazu sieht folgendermaßen aus:

```
INSERT INTO Kunde (Kundennummer, Name, Vorname, Geschlecht,
                   Strasse, Hausnummer, Plz)
VALUES ( 1, 'Bolte', 'Bertram', 'M', 'Busweg', '12', 44444 ),
       ( 2, 'Muster', 'Hans', 'M', 'Musterweg', '12', 22222 ),
       ( 4, 'Carlson', 'Peter', 'M', 'Petristr.', '201', 44444 )
```

Unsere Kundentabelle besteht zur Zeit aus vier Datensätzen und sieht folgendermaßen aus:

Kunden-nummer	Name	Vorname	Geschlecht	Strasse	Haus-nummer	Plz
1	Bolte	Bertram	M	Busweg	12	44444
2	Muster	Hans	M	Musterweg	12	22222
3	Wiegerich	Frieda	W	Wanderstr.	NULL	33333
4	Carlson	Peter	M	Petristr.	201	44444

Abbildung 7.2: Kundentabelle

7.3 Löschen von Datensätzen

Nachdem wir das Einfügen von Datensätzen gelernt haben, wollen wir uns nun anschauen, wie man Sätze wieder aus einer Tabelle löscht. Hierfür gibt es die SQL-Anweisung DELETE. In der einfachsten Form sieht die DELETE-Anweisung wie folgt aus:

```
DELETE FROM Kunde
```

Diese SQL-Anweisung sollte mit einer gewissen Vorsicht verwendet werden, da sie alle Sätze der Tabelle Kunde löscht. Würde man diese SQL-Anweisung ausführen, so wäre die Tabelle danach wieder leer und würde keine Kundendaten mehr enthalten.

Hinter der Angabe der Tabelle, aus der Datensätze gelöscht werden sollen, kann genauer spezifiziert werden, welche Datensätze gelöscht werden sollen. Dazu verwendet man das Schlüsselwort WHERE. Um z. B. alle Datensätze zu löschen, bei denen die Hausnummer der Zeichenfolge ‚12' entspricht, würde die SQL-Anweisung folgendermaßen aussehen:

```
DELETE FROM Kunde WHERE Hausnummer = '12'
```

Diese SQL-Anweisung bewirkt, dass die Sätze mit den Kundennummern 1 und 2 gelöscht werden, da bei diesen beiden Sätzen die Hausnummer 12 ist. Hinter der WHERE-Klausel können Bedingungen beliebig miteinander verknüpft werden. Hier gibt es sehr viele Möglichkeiten, genau zu spezifizieren, welche Sätze ausgewählt werden sollen. Da wir dies noch genauer beim Abfragen von Datensätzen ab Kapitel 8 kennen lernen werden, wollen wir uns vorerst einfach drei Beispiele ansehen und es damit erst einmal belassen:

```
DELETE FROM Kunde WHERE Hausnummer = '12' AND Name = 'Bolte'
DELETE FROM Kunde WHERE Hausnummer = '12' OR Plz = 44444
DELETE FROM Kunde WHERE Plz IN ( 44444, 22222 )
```

Die erste Anweisung löscht alle Sätze, bei denen die Hausnummer 12 ist und gleichzeitig der Kunde den Nachnamen Bolte hat. Die zweite Anweisung löscht alle Kunden, die in einem Ort mit der Postleitzahl 44444 wohnen und alle Kunden mit der Haus-

nummer 12. Die letzte Anweisung löscht schließlich alle Kunden, die in Orten mit den Postleitzahlen 44444 oder 22222 wohnen.

Abschließend noch der Hinweis, dass z.B. das Löschen eines Ortes mit einer Postleitzahl, die noch von einem Kundensatz verwendet wird, vom RDBMS abgelehnt wird. Auch hier greift natürlich wieder die Beziehung zwischen Primär- und Fremdschlüssel.

7.4 Ändern von Datensätzen

In Abschnitt 7.1 haben wir die Kundin Frieda Wiegerich in die Kundentabelle eingefügt, ohne deren Hausnummer zu kennen. Inzwischen haben wir erfahren, dass die Hausnummer der Kundin den Wert 89 hat. Dies wollen wir jetzt nachträglich eintragen. Hierzu verwenden wir die SQL-Anweisung UPDATE. Die einfachste Form der UPDATE-Anweisung wird, wie bei der DELETE-Anweisung auch, ohne die WHERE-Klausel verwendet, so dass eine Änderung sich auf alle Sätze bezieht. Die Anweisung

```
UPDATE Kunde
SET Geschlecht = 'M'
```

setzt demnach das Geschlecht aller Kunden auf ‚M' für männlich. Entsprechend sollte man genau überlegen, wann die Anweisung UPDATE in dieser Form angewendet werden soll.

Doch zurück zu unserer Kundin Frau Wiegerich. Zum Ändern der Hausnummer lautet die SQL-Anweisung:

```
UPDATE Kunde
SET Hausnummer = '89'
WHERE Name = 'Wiegerich'
```

Doch was würde passieren, gäbe es mehrere Kunden mit dem Nachnamen Wiegerich?

Alle Kunden, die diesen Nachnamen besitzen, bekämen als Adresse die Hausnummer 89 zugewiesen. Zur Vermeidung dieses Problems kommt der Primärschlüssel wieder zum Tragen. Um für einen ganz bestimmten Kunden Werte zu ändern, sollte man den Primärschlüssel in der Bedingung hinter der WHERE-Klausel verwenden. Die folgende Anweisung wäre deshalb besser geeignet, die Hausnummer von Frieda Wiegerich zu ändern:

```
UPDATE Kunde
SET Hausnummer = '89'
WHERE Kundennummer = 3
```

Da Kundennummer der Primärschlüssel der Tabelle Kunde ist, kann ein Kunde mit der Kundennummer 3 nur einmal vorkommen. Schließlich ist der Primärschlüssel, wie wir ja gelernt haben, immer eindeutig und darf niemals NULL sein. Entsprechend hat durch die obige SQL-Anweisung auch nur ein Satz die Hausnummer 89 zugewiesen bekommen.

Um mehrere Spalten eines Satzes zu ändern gibt es zwei Möglichkeiten. Entweder schreibt man durch Kommata getrennt, die Spalten und jeweiligen Werte oder man verwendet das Schlüsselwort ROW, mit dem man einen ganzen Satz komplett ändern kann. Frau Wiegerich ist umgezogen und ihre Adresse soll geändert werden. Außerdem sind Vor- und Nachname falsch geschrieben. Das Ergebnis folgender SQL-Anweisungen ist identisch:

```
UPDATE Kunde
SET Kundennummer = 3,
    Name = 'Wegerich',
    Vorname = 'Frida',
    Geschlecht = 'W',
    Strasse = 'Spatzenweg',
    Hausnummer = '81',
    Plz = 33333
WHERE Kundennummer = 3

UPDATE Kunde
SET ROW =
    ROW(3, 'Wegerich', 'Frida', 'W', 'Spatzenweg', '81', 33333)
WHERE Kundennummer = 3
```

Wie auch für die DELETE-Anweisung gilt, dass der WHERE-Klausel wesentlich kompliziertere Bedingungen zur Einschränkung der Ergebnismenge folgen können. Da wir dies jedoch ausführlich beim Abfragen von Tabellen behandeln, wollen wir exemplarisch nur drei weitere Beispiele betrachten:

```
UPDATE    Werbeartikel
          SET Lagerbestand = Lagerbestand - 6
WHERE Artikelnummer = 1001

UPDATE    Werbeartikel
          SET Preis = Preis * 1.10
WHERE Preis < 9.99

UPDATE    Bestellung
          SET Datum = CURRENT_DATE,
          Personalnummer = 2
WHERE Kundennummer = 3
```

Im ersten Beispiel können wir sehen, dass man in SQL-Anweisungen auch einfache Berechnungen vornehmen kann. In diesem Beispiel wird der Lagerbestand für den Artikel mit der Artikelnummer 1001 um 6 reduziert. Im zweiten Fall werden alle Datensätze geändert, bei denen der Artikel mehr als 9.99 Euro kostet und der Preis um 10 Prozent erhöht.

Das dritte Beispiel schließlich soll noch einmal die Verwendung von vordefinierten Werten verdeutlichen. Anstatt das aktuelle Datum direkt als Literal anzugeben, kann das aktuelle Datum über CURRENT_DATE in der UPDATE-Anweisung ermittelt werden.

7.5 Zusammenfassung

Wir haben in diesem Kapitel alle SQL-Anweisungen zum Einfügen, Ändern und Löschen von Datensätzen kennen gelernt. Allgemein bezeichnet man diese Gruppe von SQL-Anweisungen auch als »Data Manipulation Language« (DML).

Zum Einfügen von Datensätzen verwendet man die INSERT-Anweisung. Entweder kann man in dieser Anweisung explizit die Spalten aufführen, für die Werte angeben werden sollen oder man führt alle Werte für einen Datensatz auf, in der Reihenfolge, in der die Spalten in der CREATE TABLE-Anweisung angegeben wurden. Um den Vorgabewert einzufügen, verwendet man das Schlüsselwort DEFAULT, um einen Wert als nicht bekannt zu kennzeichnen, NULL.

Zum Löschen von Datensätzen ist die DELETE-Anweisung geeignet. Ohne Verwendung der WHERE-Klausel werden alle Datensätze einer Tabelle gelöscht. Durch die Angabe von Bedingungen hinter der WHERE-Klausel wird festgelegt, welche Datensätze gelöscht werden sollen.

Als letzte SQL-Anweisung haben wir zum Ändern von Datensätzen UPDATE kennen gelernt. Wie bei der DELETE-Anweisung auch, schränkt man die Anzahl der zu ändernden Sätze durch die WHERE-Klausel ein.

Für alle Anweisungen gilt, dass das Einfügen, Ändern oder Löschen von Datensätzen vom RDBMS zurückgewiesen wird, sofern festgelegte Integritätsbedingungen verletzt werden. Dadurch wird verhindert, dass die Datenbank inkonsistent oder inkorrekt wird. Bevor also z. B. ein Kunde eingefügt wird, muss sichergestellt sein, dass der Ort, in dem der Kunde wohnhaft ist, bereits in der Tabelle Ort existiert. Andernfalls muss der Ort neu eingefügt werden, damit die referentielle Integrität gewährleistet ist.

7.6 Aufgaben

Wiederholungsfragen

1) Wie lauten die SQL-Anweisungen zum Löschen, Ändern und Einfügen von Datensätzen in Tabellen?
2) Was ist mit dem Schlüsselwort DEFAULT gemeint und in welchen der drei kennen gelernten SQL-Anweisungen kann es verwendet werden?
3) Was ist mit NULL gemeint?

Übungen

1) Der Preis für Sitzplätze der Reihen 1-3 soll um 15% erhöht werden. Wie lautet die SQL-Anweisung?
2) Der Sitzplatz (Reihe 6, Sitz 2) im Bereich »Parkett« für die Vorstellung mit der Nummer 22 soll als »belegt« gekennzeichnet werden. Wie lautet die SQL-Anweisung?
3) Die Spielstätte »Vergissmeinnicht« hat Konkurs angemeldet. Alle Vorstellungen sind abgesagt worden. Wie lautet die SQL-Anweisung zum Löschen der Spielstätte? Was geschieht mit den Sätzen der Vorstellungen, die sich auf diese Spielstätte beziehen. Müssen diese explizit gelöscht werden?

1) Die Firma »KartoFinale« will mit ihrer neu entwickelten Datenbank in Produktion gehen. Dazu müssen zunächst die Kunden- und Vorstellungsdaten in die Datenbank eingefügt werden. Fügen Sie die folgenden Datensätze in die jeweiligen Tabellen ein. Beachten Sie die referenzielle Integrität, d.h. bevor Sie z.B. eine Vorstellung einfügen können, muss die entsprechende Veranstaltung eingefügt sein und die Spielstätte, in der die Vorstellung stattfindet.

Kunde

Kunden-nummer	Name	Vorname	Geschlecht	Strasse	Strassen-nummer	Plz	Geburtsdatum
1	Bolte	Bertram	M	Busweg	12	44444	1945-12-02
2	Muster	Hans	M	Musterweg	12	22222	1953-02-21
3	Wiegerich	Frieda	W	Wanderstr.	NULL	33333	1963-08-18
4	Carlson	Peter	M	Petristr.	201	44444	1971-09-26

Kind

Kunden-nummer	Vorname	Geburtsdatum	Geschlecht
1	Katja	1988-12-31	W
1	Ursula	1990-05-08	W
1	Karl	2001-05-21	M
3	Ursula	1992-08-19	W
3	Enzo	1990-05-12	M

Mitarbeiter

Personal-nummer	Name	Vorname	Geschlecht	Strasse	Haus-nummer	Plz	Abteilungs-bezeichnung	Personalnummer-vorgesetzter	Gehalt
3	Kart	Karen	W	Pantherstr.	12	22222	Vertrieb	6	3000
5	Klein	Karl	M	Minimalweg	1	22222	Vertrieb	3	2050
6	Kowalski	Karsten	M	Blankeneser Weg	2	22287	Geschäftsführung	NULL	4000

Abteilung

Abteilungs-bezeichnung	Abteilungs-nummer	Personalnummer-Abteilungsleiter
Vertrieb	1	3
Geschäftsführung	2	6

Abbildung 7.3: Datensätze

Ort

Plz	Ort
44444	Kohlscheidt
22222	Karlstadt
33333	Rettrich
22287	Hamburg
25746	Heide

Spielstaette

Haus	Strasse	Haus-nummer	Plz	Beschreibung
Deutsche Staatsoper	Opernstr.	1	22287	NULL
Operettenhaus	Bahndamm	88	22287	NULL
Sokratesbühne	Musterweg	81	22222	NULL
Nordseehalle	Nordfeldweg	200	25746	NULL
Kongresshalle	Hahndamm	21	33333	NULL

Veranstaltung

Veranstaltungs-nummer	Bezeichnung	Autor	Beschreibung
1	Don Giovanni	Amadeus Mozart	NULL
4	Phil Collins LIVE	Phil Collins	NULL
3	Zauberlehrling	Amadeus Mozart	NULL
8	Mutter Courage	Brecht	NULL
7	Cats	Webber	NULL

Vorstellung

Vorstellungs-nummer	Datum	Uhrzeit	Veranstaltungs-nummer	Haus
11	2002-07-21	20.00	1	Hamburg Opernhaus
12	2002-07-28	19.30	1	Hamburg Opernhaus
13	2002-08-04	19.30	1	Hamburg Opernhaus
31	2002-01-05	21.30	3	Operettenhaus
32	2002-01-06	21.30	3	Operettenhaus
41	2002-07-21	18.45	4	Nordseehalle
42	2002-08-01	19.00	4	Kongresshalle
44	2002-09-18	19.15	4	Hamburg Opernhaus
81	2002-01-31	19.30	8	Sokratesbühne
82	2002-02-28	19.30	8	Sokratesbühne
85	2002-03-31	20.00	8	Sokratesbühne
86	2002-04-30	20.00	8	Sokratesbühne
87	2002-05-31	20.00	8	Sokratesbühne

Abbildung 7.4: Weitere Datensätze

Artikel

Artikel-nummer	Bezeichnung
1001	Noten Don Giovanni
1003	Mozart T-Shirt
1012	CD: Phil Collins
1081	Zauberstock
1078	Opernführer
1077	Cats-Plakat
1101	Don Giovanni
1102	Don Giovanni
1103	Don Giovanni
1104	Don Giovanni
1105	Don Giovanni
1201	Don Giovanni
1202	Don Giovanni
1203	Don Giovanni
1204	Don Giovanni
1205	Don Giovanni
4101	Konzert von Phil Collins
4102	Konzert von Phil Collins
4103	Konzert von Phil Collins
4104	Konzert von Phil Collins
4105	Konzert von Phil Collins
4106	Konzert von Phil Collins

Werbeartikel

Artikel-nummer	Beschreibung	Preis	Lagerbestand
1001	Noten f. Klavier	89.00	26
1003	T-Shirt Farbe: rot	29.99	11
1012	Phil Collins in Concert	20.00	21
1081	Schwarzer Zauberstock	5.99	32
1077	Plakat mit den Musical-Katzen	33.50	89
1078	Opernführer	19.99	9999

Sitzplatz

Artikel-nummer	Bereich	Reihe	Sitz	Preis	Zustand	Vorstellungs-nummer
1101	Parkett	2	8	89.00	belegt	11
1102	Parkett	4	11	89.00	frei	11
1103	1. Rang	1	12	120.00	frei	11
1104	2. Rang	3	2	75.00	belegt	11
1105	3. Rang	7	1	60.00	reserviert	11
1201	Parkett	2	8	89.00	belegt	12
1202	Parkett	4	11	89.00	frei	12
1203	1. Rang	1	12	120.00	frei	12
1204	2. Rang	3	2	75.00	belegt	12
1205	3. Rang	7	1	60.00	reserviert	12
4101	Parkett	1	1	120.00	frei	41
4102	Parkett	1	2	120.00	frei	41
4103	Parkett	1	3	120.00	frei	41
4104	Parkett	11	14	60.00	frei	41
4105	Parkett	3	21	90.00	frei	41
4106	Parkett	8	26	85.99	frei	41

Bestellung

Bestell-nummer	Datum	Kunden-nummer	Personal-nummer
1	2002-01-26	1	NULL
3	2002-02-08	3	NULL
8	2001-12-21	2	NULL

Bestellposten

Bestell-nummer	Positions-nummer	Artikel-nummer	Menge
1	1	1101	1
1	1	1102	1
1	1	1103	1
1	2	1001	2
1	3	1003	3
8	1	4101	1
8	2	1012	2

Abbildung 7.5: Artikel und Bestellung

8 Eine Tabelle abfragen

In Kapitel 8 sollen folgende Fragen geklärt werden:

- Wie werden Tabellen einer Datenbank generell abgefragt?
- Wie sieht der Aufbau der SQL-Anweisung zur Abfrage aus?
- Wie gibt man den Inhalt einer Tabelle aus?
- Wie gibt man an, welche Spalten einer Tabelle man ausgegeben haben möchte?
- Welche Funktionen kann man auf Spalten anwenden?
- Wie sagt man dem RDBMS, welche Sätze es finden soll?
- Welche Bedingungsausdrücke gibt es?
- Wie sortiert man die Ergebnismenge?
- Wie gruppiert man Sätze der Ergebnismenge?

8.1 Motivation

Inzwischen ist eine Woche vergangen und Frau Kart hat alle Kunden, Mitarbeiter, Vorstellungen und Bestellungen der Firma »KartoFinale«in die Datenbank übernommen. Mit Herrn Dr. Fleissig hat sie deshalb für heute einen Termin vereinbart, damit er ihr erklärt, wie man nun Informationen aus der Datenbank abfragt.

Herr Fleissig erklärt Frau Kart, dass die Sprache SQL hierfür nur SELECT als einzige SQL-Anweisung vorsieht. Unter anderem erklärt er ihr, dass man nicht nur bestimmte Tabellen nach bestimmten Werten abfragen kann, sondern dass man auch Listen über Kunden nach Umsatz sortiert erstellen kann. Nachdem ihr Herr Fleissig die grundlegenden Eigenschaften der SELECT-Anweisung erklärt hat, beginnt Frau Kart, verschiedene Abfragen auszuprobieren. Dabei bemerkt sie, dass sie sehr viele Informationen erhalten kann, die vorher nur mit großem Aufwand zu beschaffen waren. Erfreut darüber teilt Frau Kart dies dem Geschäftsführer Herrn Kowalski mit, der von den Möglichkeiten und natürlich von der vorbildlichen Einstellung Frau Karts begeistert ist. Er beauftragt Frau Kart deshalb, mehrere Informationen aus der Datenbank für ihn abzufragen:

- Was kostet ein Werbeartikel im Durchschnitt?
- Welcher Werbeartikel wurde am häufigsten verkauft?
- Welche Vorstellungen finden im Mai statt?
- Wie viele Vorstellungen finden im Jahr 2002 jeweils in den einzelnen Monaten statt?
- Welcher Kunde hat die höchste Anzahl an Artikeln gekauft?
- Welcher Artikel wurde am wenigsten bestellt?

Frau Kart, noch etwas unsicher, ob sie alle diese Fragen beantworten kann, macht sich an die Arbeit.

8.2 Allgemeiner Aufbau einer Abfrage

Bevor wir uns mit dem allgemeinen Aufbau der SELECT-Anweisung beschäftigen, wollen wir uns noch einmal kurz die Grundlagen des Relationenmodells vor Augen führen. SQL basiert ja auf der Mengenlehre bzw. den Grundlagen des Relationenmodells nach E.F.Codd. Neben den Grundlagen des Relationenmodells, die wir in Kapitel 4 kennen gelernt haben, enthält das Relationenmodell auch mathematische Grundlagen, um Tabellen und in Beziehung stehende Tabellen abzufragen. Diese mathematischen Grundlagen zur Abfrage werden als Relationenalgebra und als Relationenkalkül bezeichnet. Auf die mathematischen Grundlagen brauchen wir hier nicht weiter einzugehen, da sie zur Verwendung von SQL nicht unbedingt notwendig sind.

Doch kommen wir zu SQL zurück. SQL wurde gemäß der Relationenalgebra als deklarative (beschreibende) Programmiersprache konzipiert. Im Gegensatz zu deklarativen Programmiersprachen gibt es prozedurale Programmiersprachen. Um den Unterschied zwischen beiden Arten von Programmiersprachen zu verstehen, wollen wir uns ein Beispiel aus der realen Welt ansehen:

Der Geschäftsführer einer Firma geht zum Leiter des Rechnungswesens, um von diesem eine Liste anzufordern. Er hat zwei Möglichkeiten, dem Leiter des Rechnungswesens sein Anliegen zu erklären.

Entweder sagt er ihm: »Erstellen Sie eine Liste aller Kunden, die in den letzten drei Monaten für mehr als 700 Euro Eintrittskarten gekauft haben und aus Regionen kommen, die mit der Postleitzahl 2 beginnen.«

Oder er sagt ihm: »Holen Sie den Ordner mit den Bestellungen. Suchen Sie dann alle Bestellungen der letzten 3 Monate heraus. Aus diesen Bestellungen suchen Sie wiederum die Bestellungen heraus, die von Kunden aus der Postleitzahlregion mit der Zahl 2 stammen. Danach sortieren Sie die Bestellungen nach Kunden und addieren dann die Bestellungen pro Kunde zusammen. Den Betrag und den Kundennamen schreiben Sie auf eine Liste. Schließlich streichen Sie die Kunden aus der Liste, deren Bestellsumme kleiner als 700 Euro ist.«

In beiden Fällen erhält der Geschäftsführer nach einer gewissen Zeit seine Ergebnisliste, die, je nachdem, wie er die Aufgabe formuliert hat, identisch sein sollte. Im ersten Fall hat er dem Leiter des Rechnungswesens beschrieben, was er als Ergebnisliste haben möchte. Im zweiten Fall hat er ihm beschrieben, in welchen Schritten der Rechnungsleiter vorgehen soll, um die Ergebnisliste zu erstellen.

Der erste Fall entspricht einer deklarativen Sprache, da beschrieben wird, **was** man als Ergebnis haben möchte. Der zweite Fall entspricht einer prozeduralen Sprache, da beschrieben wird, **wie** man zu einem Ergebnis kommt. Deklarative Programmiersprachen beschreiben also das Problem, prozedurale Programmiersprachen dagegen geben ein schrittweises Verfahren zur Problemlösung vor.

Wie Sie sicherlich bemerkt haben, ist die erste Vorgehensweise für den Geschäftsführer eindeutig einfacher, da er nicht jeden Schritt einzeln aufführen muss. Beide Arten von Sprachen haben jedoch ihre Berechtigung, je nachdem welche Probleme man lösen möchte.

Bei SQL nun handelte es sich ursprünglich um eine deklarative Programmiersprache. In SQL formuliert man also, was man vom RDBMS als Ergebnismenge geliefert bekommen möchte. Ab Anfang der 90er Jahre begannen die Hersteller von RDBMS-Produkten jedoch, auch prozedurale Sprachelemente in ihre Produkte einzubauen. Diese Sprachelemente wurden 1996 als neuer Bestandteil dem damaligen SQL-92 Standard vom ANSI unter der Bezeichnung SQL/PSM hinzugefügt. Dadurch ist SQL heute sowohl eine prozedurale als auch eine deklarative Programmiersprache. Wir wollen uns in diesem und in den nächsten Kapiteln zunächst mit den deklarativen Sprachelementen von SQL beschäftigen.

Wir können also festhalten, dass man durch SQL dem RDBMS in textueller Form mitteilt, was man von ihm wissen möchte, genauso wie der Geschäftsführer dies dem Leiter des Rechnungswesens mitteilt. Das RDBMS entscheidet dann selbständig, ob und wie es diese Informationen findet.

Da SQL-Abfragen auf der Mengenlehre basieren, ist das Ergebnis einer Abfrage auch wiederum eine Menge bzw. eine Tabelle. Mengen kennen keine bestimmten Reihenfolgen. Das Ergebnis einer SQL-Abfrage in Form einer Tabelle ist also nicht sortiert, es sei denn man gibt dies explizit an.

In den folgenden drei Kapiteln werden wir uns mit der SQL-Anweisung zum Abfragen von Tabellen, der SELECT-Anweisung, beschäftigen. Der allgemeine Aufbau der SELECT-Anweisung sieht folgendermaßen aus:

```
  SELECT   spaltenliste
[ FROM     tabellenliste ]
[ WHERE    bedingungsausdruck ]
[ GROUP BY spaltenliste ]
[ HAVING   bedingungsausdruck ]
[ ORDER BY spaltenliste ]
```

Die Angaben in eckigen Klammern sind optional, also nicht zwingend erforderlich.

8.3 Spalten auswählen

Ein einfaches Beispiel der SELECT-Anweisung sieht wie folgt aus:

```
SELECT Plz
FROM   Ort
```

Die SELECT-Klausel führt alle Spalten auf, die in der Ergebnisliste erscheinen sollen. Danach folgt das Schlüsselwort FROM, hinter dem man die Tabellen angibt, in denen sich die zu suchenden Daten befinden. Unser Beispiel gibt also alle Postleitzahlen aus, die in der Tabelle »Ort« gespeichert sind.

```
Plz
-----
44444
22222
33333
22287
25746
```

Alle Spalten, die wir von einer Tabelle als Ergebnis erhalten möchten, werden hinter SELECT aufgeführt. Sind es mehr als eine Spalte, so werden diese durch Kommata voneinander getrennt. Um Ort und Plz auszugeben, sieht die SELECT-Anweisung folgendermaßen aus:

```
SELECT Ort, Plz
FROM   Ort
```

Als Ergebnis erhält man eine Tabelle, die die Spalten in der Reihenfolge aufführt, in der sie hinter der SELECT-Anweisung angegeben sind.

```
Ort          Plz
-----------  -----
Karlstadt    22222
Hamburg      22287
Heide        25746
Rettrich     33333
Kohlscheidt  44444
```

SQL kennt so genannte Platzhalter-Zeichen. Eines dieser Platzhalter-Zeichen ist der »*«. Möchte man den Inhalt einer Tabelle ausgegeben haben, so ist es bei Tabellen mit vielen Spalten häufig mühselig, alle Spalten einer Tabelle hintereinander aufzuführen. Deshalb kann anstelle der Spaltenliste das »*«-Zeichen als Platzhalter für alle Spalten hinter der SELECT-Anweisung erscheinen. Dabei werden die Spalten in der Reihenfolge ausgegeben, in der sie in der CREATE TABLE-Anweisung einmal angelegt wurden.

```
SELECT *
FROM   Ort
```

Das Ergebnis sieht wie bei der vorherigen Abfrage aus, nur diesmal erscheint zuerst die Spalte »Plz« und dann die Spalte »Ort«, da die Spalten in der CREATE TABLE-Anweisung in dieser Reihenfolge angelegt wurden.

Um die Lesbarkeit von Spalten zu verbessern, kann man Spalten in der Ergebnistabelle auch andere Namen geben. Angenommen, Sie möchten statt Plz lieber Postleitzahl als Spaltenbezeichnung, so sieht die SELECT-Anweisung wie folgt aus:

```
SELECT Plz AS 'Postleitzahl', Ort
FROM   Ort
```

Das Umbenennen von Spalten über das Schlüsselwort AS ist besonders dann wichtig, wenn man mit Spalten rechnet. Denn auch das ist möglich: Sie können Berechnungen

mit beliebigen Spalten durchführen. Angenommen, Sie möchten eine Liste aller Werbeartikel ausgeben und die Liste soll mit Preisen angezeigt werden, die um 10% erhöht sind, dann sieht die SELECT-Anweisung dazu folgendermaßen aus:

```
SELECT Beschreibung, Preis, Preis * 1.10 AS 'Preis + 10%'
FROM   Werbeartikel
```

Das Ergebnis der Tabelle erhält den erhöhten berechneten Preis, der hier mit vier Nachkommastellen angezeigt wird. Allerdings haben wir das Attribut Preis der Tabelle Werbartikel mit dem Datentyp DECIMAL(8, 2) deklariert, also nur mit zwei Nachkommastellen. SQL konvertiert bei Berechnungen Zahlen automatisch in andere Datentypen, um korrekte Berechnungen darzustellen, in diesem Fall mit vier Nachkommastellen. Wir werden im Abschnitt über Funktionen sehen, dass man bei SQL eine solche Datenkonvertierung aber auch explizit angeben kann, d.h. in diesem Fall könnten wir die Berechnung dann wieder auf zwei Nachkommastellen begrenzen. Generell wählt SQL allerdings automatisch einen Datentyp mit einem größeren Wertebereich, sofern bei Berechnungen ein Wertebereich überschritten wird. Entsprechend sieht die Ergebnistabelle der Abfrage dann wie folgt aus:

```
Beschreibung              Preis  Preis + 10%
------------------------  -----  -----------
Noten f. Klavier          89.00  97.9000
T-Shirt Farbe: rot        29.99  32.9890
Phil Collins in Concert   20.00  22.0000
Plakat mit Musical-Katzen 33.50  36.8500
Schwarzer Zauberstock      5.99   6.5890
```

Dieses Beispiel zeigt Berechnungen mit konstanten Werten. Man kann aber auch Berechnungen mit mehreren Spalten vornehmen. Möchte man z.B. wissen, welchen Gesamtpreis ein Werbeartikel hat, der auf Lager liegt, so wird der Lagerbestand mit dem Einzelpreis multipliziert. Die SELECT-Anweisung dazu lautet:

```
SELECT Beschreibung, Preis * Lagerbestand AS 'Artikel Gesamtwert'
FROM   Werbeartikel
```

Hier hat das Ergebnis der Berechnung nur zwei Nachkommastellen, da Lagerbestand eine Ganzzahl ist, und damit die Anzahl der Nachkommastellen durch die Berechnung auch nicht überschritten werden kann.

```
Beschreibung              Artikel Gesamtwert
------------------------  ------------------
Noten f. Klavier          2314.00
T-Shirt Farbe: rot        329.89
Phil Collins in Concert   420.00
Plakat mit Musical-Katzen 2981.50
Schwarzer Zauberstock     191.68
```

Da der Preis in Euro in unsere Datenbank eingegeben wurde, fehlt nun noch die Währungseinheit hinter dem Preis. Als Spaltenliste hinter dem Schlüsselwort SELECT können auch konstante Werte wie z.B. »EUR« aufgeführt werden. Diese erscheinen in jeder Ergebniszeile mit dem gleichen Wert. Das ist genau das, was wir brauchen.

```
SELECT Beschreibung, Preis, 'EUR' AS 'Währung'
FROM   Werbeartikel
```

Damit sieht unsere Ergebnistabelle folgendermaßen aus:

```
Beschreibung               Preis  Währung
-------------------------  -----  -------
Noten f. Klavier           89.00  EUR
T-Shirt Farbe: rot         29.99  EUR
Phil Collins in Concert    20.00  EUR
Plakat mit Musical-Katzen  33.50  EUR
Schwarzer Zauberstock      5.99   EUR
```

Gemäß dem Relationenmodell darf ein Element der Ergebnismenge nur einmal vorkommen. Mit anderen Worten: Jede Zeile einer Tabelle darf nur ein einziges Mal auftreten. SQL ist in diesem Punkt nicht so konsequent wie das Relationenmodell. Betrachten wir dazu folgende SELECT-Anweisung und deren Ergebnistabelle:

```
SELECT Haus
FROM   Vorstellung

Haus
-----------------
Deutsche Staatsoper
Deutsche Staatsoper
Deutsche Staatsoper
Operettenhaus
Operettenhaus
Nordseehalle
Kongresshalle
Deutsche Staatsoper
Sokratesbühne
Sokratesbühne
Sokratesbühne
Sokratesbühne
Sokratesbühne
```

Am Beispiel dieser einfachen Abfrage können wir sehen, dass Sätze mit gleichen Werten in einer Ergebnistabelle mehrfach vorkommen können. Generell gilt, dass die Ergebnistabelle nicht unbedingt den Normalformen des Relationenmodells entsprechen muss. Um nun dennoch zu erreichen, dass Sätze der Ergebnistabelle nicht mehrfach angezeigt werden, verwendet man das Schlüsselwort DISTINCT direkt hinter dem Schlüsselwort SELECT. DISTINCT filtert Sätze mit gleichen Werten aus der Ergebnistabelle aus.

```
SELECT DISTINCT Haus
FROM   Vorstellung

Haus
-----------------
Deutsche Staatsoper
Kongresshalle
Nordseehalle
Operettenhaus
Sokratesbühne
```

Unsere Ergebnistabelle ist jetzt korrekt, alle Spielstätten, in denen Vorstellungen stattfinden, werden nur noch einmal aufgeführt.

Wir haben uns bis hierher vorwiegend damit beschäftigt, wie man die gewünschten Spalten einer Tabelle ausgeben kann. Mathematisch wird die Auswahl der Spalten gemäß dem Relationenmodell als Projektion bezeichnet. Abschließend wollen wir uns die Projektion noch einmal grafisch am Beispiel der Tabelle Werbeartikel anschauen, bevor wir zu Funktionen übergehen, die man auf Spalten anwenden kann.

```
SELECT  Bezeichnung, Lagerbestand
FROM    Werbeartikel
```

Artikel-nummer	Beschreibung	Preis	Lagerbestand
1001	Noten f. Klavier	89.00	26
1003	T-Shirt Farbe: rot	29.99	11
1012	Phil Collins in Concert	20.00	21
1081	Schwarzer Zauberstock	5.99	32
1077	Plakat mit den Musical-Katzen	33.50	89

Beschreibung	Lagerbestand
Noten f. Klavier	26
T-Shirt Farbe: rot	11
Phil Collins in Concert	21
Schwarzer Zauberstock	32
Plakat mit den Musical-Katzen	89

Abbildung 8.1: Projektion (Auswahl von Spalten)

8.4 »Built-in«-Funktionen

8.4.1 Grundlagen

SQL:1999 legt in seinem Standard mehrere Gruppen von eingebauten Funktionen (»Built-in«-Funktionen) fest. Funktionen führen bestimmte immer wiederkehrende Aufgaben aus, wie z.B. das Umwandeln einer Zeichenkette in Großbuchstaben. Funktionen haben die Eigenschaft, dass ihnen Werte übergeben werden können, die Funktionen diese Werte verarbeiten und ein Ergebnis zurückliefern. Damit sind Funktionen nützliche, aber auch sehr wichtige »Helfer« für den SQL-Programmierer. Funktionen können in SQL-Ausdrücken auf Spalten oder auf Literale angewendet werden.

Betrachten wir hierzu exemplarisch die einfache Funktion UPPER. Der Funktion UPPER wird eine Zeichenkette übergeben, die diese in Großbuchstaben umwandelt und dann wieder zurückliefert. In einer einfachen SELECT-Anweisung könnten wir so z.B. alle Orte in Großbuchstaben ausgeben:

```
SELECT UPPER(Ort)AS 'Ortsname'
FROM   Ort

Ortsname
-----------
KARLSTADT
HAMBURG
HEIDE
RETTRICH
KOHLSCHEIDT
```

SQL:1999 kennt ingesamt folgende Gruppen von Funktionen, wobei gerade der Funktionsumfang bei den einzelnen RDBMS-Produkten sehr groß ist und vor allem Unterschiede in der Syntax bestehen:

- Wertfunktionen mit numerischem Rückgabewert (»Numeric Value Functions«);
- Wertfunktionen mit Zeichenkette als Rückgabewert (»String Value Functions«);
- Wertfunktionen mit Datums-/Zeit-Rückgabewert (»Datetime Value Functions«);
- Wertfunktionen mit Zeitintervall-Rückgabewert (»Interval Value Functions«);
- NULL-Funktionen und Datentypkonvertierung;
- Mengenfunktionen (»Set Functions«).

Wir wollen uns im Folgenden mit den wichtigsten Funktionen von SQL beschäftigen und deren Anwendung in Zusammenhang mit der SELECT-Anweisung kennen lernen.

8.4.2 »Numeric Value Functions«

Wertfunktionen mit numerischem Rückgabewert liefern als Ergebnis immer eine Zahl und haben daher ihren Namen. Die ersten beiden Funktionen, die wir uns ansehen

wollen, sind die Funktionen ABS und MOD. Beiden werden numerische Werte übergeben. ABS liefert als Ergebnis den absoluten Wert einer Zahl, MOD entspricht mathematisch der Modulofunktion und liefert den Rest einer Division zurück.

Betrachten wir hierzu ein Beispiel:

```
SELECT ABS(-26.00) AS 'Absoluter Wert'

Absoluter Wert
--------------
26.00
```

Dieses recht einfache Beispiel übergibt einen konstanten Wert -26.00 und bekommt als Ergebnis den absoluten positiven Wert 26.00 zurückgeliefert.

Genauso wird die Modulofunktion MOD verwendet. Diesmal wollen wir die Funktion allerdings nicht auf einen konstanten Wert anwenden, sondern auf eine Spalte. Wir wollen ermitteln, welche Werbeartikel in einer geraden Anzahl im Lager vorhanden sind. Ergibt also der Rest einer Division durch 2 die Zahl 1, so handelt es sich um eine ungerade Menge, ist der Rest 0 um eine gerade Menge. Die SELECT-Anweisung hierzu sieht wie folgt aus:

```
SELECT Lagerbestand, MOD( Lagerbestand, 2 ) AS 'Divisionsrest'
FROM   Werbeartikel

Lagerbestand  Divisionsrest
------------  -------------
26            0
11            1
21            1
89            1
32            0
```

Der Funktion CHAR_LENGTH oder CHARACTER_LENGTH wird, wie der Name schon sagt, eine Zeichenkette als Wert übergeben. Als Ergebnis liefert die Funktion die Länge der Zeichenkette. Das folgende Beispiel gibt die Namen aller Orte, sowie die Länge des Ortsnamens aus:

```
SELECT Ort, CHARACTER_LENGTH(Ort)
FROM   Ort

Ort
------------  -------------
Karlstadt      9
Hamburg        7
Heide          5
Rettrich       8
Kohlscheidt    11
```

Die Funktion POSITION sucht innerhalb einer Zeichenkette nach dem Auftreten einer anderen Zeichenkette. Angenommen, Sie möchten die Position der Zeichenkette 'stadt' innerhalb jedes Ortsnamens ermitteln, so sieht die entsprechende SELECT-Anweisung wie folgt aus:

```
SELECT Ort, POSITION('stadt' IN Ort)
FROM   Ort

Ort
------------  -------------
Karlstadt     5
Hamburg       0
Heide         0
Rettrich      0
Kohlscheidt   0
```

Die Zeichenfolge »stadt« wurde nur im ersten Ortsnamen Karlstadt gefunden und beginnt dort an der fünften Stelle. Für alle anderen Datensätze wurde im jeweiligen Ortsnamen die Zeichenfolge nicht gefunden und damit der Wert 0 von der Funktion POSITION zurückgeliefert.

Als letzte numerische Wertfunktionen wollen wir uns die Funktion EXTRACT ansehen. Sie extrahiert aus einem Datumswert den Tag, Monat oder das Jahr eines Datums. Die folgende SELECT-Anweisung gibt das Bestelldatum jeweils getrennt nach Tag, Monat und Jahr in einer einzelnen Spalte aus:

```
SELECT EXTRACT (DAY   FROM Datum) AS 'Tag',
       EXTRACT (MONTH FROM Datum) AS 'Monat',
       EXTRACT (YEAR  FROM Datum) AS 'Jahr'
FROM   Bestellung

Tag  Monat  Jahr
---  -----  ----
26   1      2002
8    2      2002
21   12     2001
```

8.4.3 »String Value Functions«

»String Value Functions« liefern als Ergebnis immer eine Zeichenkette. Ein einfaches Beispiel so einer Funktion haben wir bereits kennen gelernt, die Funktion UPPER, welche Zeichenketten in Großbuchstaben umwandelt. Analog zur UPPER-Funktion gibt es auch eine LOWER-Funktion, die Zeichenketten in Kleinbuchstaben umwandelt. Neben diesen beiden eher unwichtigen Funktionen ist vor allem die Funktion TRIM von Bedeutung. Mit TRIM können Leerzeichen oder auch andere Zeichen vor und hinter einer Zeichenkette entfernt werden. Im ersten Moment mag dies für Sie auch eine relativ unwichtige Funktion sein, doch hier kommt der Datentyp CHARACTER ins Spiel. Wir haben ja gelernt, dass Werte von Spalten des Datentyps CHARACTER immer

genau die angegebene Anzahl an Zeichen belegen, die hinter CHARACTER angegeben wird. Angenommen, Sie haben die Spalte Vorname mit CHARACTER(20) definiert, so wird ein Vorname wie »Hans« in dieser Spalte nicht 4 Zeichen, sondern immer 20 Zeichen belegen. Dabei füllt SQL den Rest des Wertes mit Leerzeichen auf. Hinter dem Wort »Hans« würden also noch 16 Leerzeichen folgen. Um diese Leerzeichen bei Abfragen wieder zu entfernen, eignet sich die Funktion TRIM. Neben der Zeichenkette, aus der die Leerzeichen entfernt werden sollen, gibt man der Funktion noch bekannt, ob führende (LEADING), nachfolgende (TRAILING) oder führende und nachfolgende (BOTH) Leerzeichen entfernt werden sollen. Um die Leerzeichen bei Vor- und Nachname eines Kunden zu entfernen, schreiben wir:

```
SELECT TRIM( TRAILING ' ' FROM Name ),
       TRIM( TRAILING ' ' FROM Vorname)
FROM   Kunde
```

Folgendes weiteres Beispiel soll noch einmal die verschiedenen Varianten von TRIM verdeutlichen. Es entfernt Semikolons vor oder hinter einem konstanten Text.

```
SELECT TRIM( TRAILING ';' FROM ';;;Test;;;' ),
       TRIM( LEADING  ';' FROM ';;;Test;;;' ),
       TRIM( BOTH     ';' FROM ';;;Test;;;' )

-------  -------  -------
;;;Test  Test;;;  Test
```

Mit der Funktion SUBSTRING kann ein Teil aus einer Zeichenkette herausgetrennt werden. Angenommen, wir haben beim Autor einer Veranstaltung Vor- und Nachnamen in einer Spalte gespeichert (was man ja nach der 1. Normalform eigentlich nicht machen sollte). Wir möchten nun alle Veranstaltungen ausgeben, sowie die Nachnamen der Autoren. Ein Autor ist Phil Collins, dessen Nachname beginnt ab der sechsten Stelle und endet an der zwölften Stelle, also nach sieben Zeichen. Mit Hilfe der Funktion SUBSTRING ist folgende SELECT-Anweisung möglich:

```
SELECT Autor, SUBSTRING( Autor FROM 6 FOR 7 )  AS 'Nachname'
FROM Veranstaltung

Autor            Nachname
---------------  --------
Amadeus Mozart   us Moza
Amadeus Mozart   us Moza
Phil Collins     Collins
Webber           r
Brecht           t
```

Das Ergebnis ist allerdings nicht besonders zufrieden stellend, da nur für Phil Collins gilt, dass der Nachname ab der sechsten Stelle beginnt. Für Amadeus Mozart wird zum Beispiel ab der sechsten bis zur zwölften Stelle nur die Zeichenfolge »us Moza« ausgegeben.

Man kann nun aber Funktionen ineinander schachteln und entsprechend zuerst die Position ermitteln, ab der das Leerzeichen erscheint und diese Position in der Funktion SUBSTRING verwenden:

```
SELECT Autor,
  SUBSTRING(Autor FROM POSITION( ' ' IN Autor)+1 ) AS 'Nachname'
FROM Veranstaltung

Autor            Nachname
---------------  --------
Amadeus Mozart   Mozart
Amadeus Mozart   Mozart
Phil Collins     Collins
Webber Webber
Brecht Brecht
```

Betrachten wir dazu die ineinander geschachtelten Funktionen SUBSTRING und POSITION. Zunächst wird die Position des Leerzeichens über POSITION(' ' IN Autor) ermittelt. Für den Namen Phil Collins würde diese Funktionen die Position 5 zurückliefern. Um das Leerzeichen selbst zu überspringen wird 1 dazu addiert. Dieser Wert wird dann an SUBSTRING übergeben und entsprechend für die Zeichenfolge Phil Collins die Funktion SUBSTRING(Autor FROM 6) ausgeführt.

Die letzte Zeichenkettenfunktion, die wir betrachten wollen, dient zum Ersetzen einer Zeichenfolge durch eine andere Zeichenfolge. Ein einfaches Beispiel um das Kürzel »z.B.« durch die Zeichenfolge »zum Beispiel« zu ersetzen, sieht wie folgt aus:

```
SELECT OVERLAY( 'z.B. Saft' PLACING 'zum Beispiel' FROM 1 FOR 4)

-----------------
zum Beispiel Saft
```

Die Zeichenkette »z.B. Saft« wurde durch die Funktion OVERLAY geändert in »zum Beispiel Saft«.

Betrachten wir noch ein Beispiel, in dem wir wieder Funktionen schachteln. Wir haben in der Spalte Beschreibung unserer Werbeartikel die Abkürzung »f.« für das Wort »für« verwendet. Bei der Ausgabe unserer Ergebnistabelle möchten wir aber anstelle der Abkürzung das Wort »für« ausgeben. Dazu verwenden wir die Funktion OVERLAY zusammen mit der Funktion POSITION wie folgt:

```
SELECT Beschreibung,
       OVERLAY( Beschreibung PLACING 'für'
                FROM POSITION( 'f.' IN Beschreibung) FOR 2 )
       AS 'Beschreibung ohne Kürzel'
FROM   Werbeartikel
```

```
Beschreibung               Beschreibung ohne Kürzel
-------------------------  -------------------------
Noten f. Klavier           Noten für Klavier
T-Shirt Farbe: rot         T-Shirt Farbe: rot
Phil Collins in Concert    Phil Collins in Concert
Plakat mit Musical-Katzen  Plakat mit Musical-Katzen
Schwarzer Zauberstock      Schwarzer Zauberstock
```

8.4.4 »Datetime Value Functions«

Funktionen aus der Gruppe der Wertfunktionen, die ein Datums-/Zeitwert zurückliefern, haben wir bereits in Kapitel 6 beim Anlegen von Tabellen über CREATE TABLE kennen gelernt, und zwar die Funktionen CURRENT_DATE, CURRENT_TIME und CURRENT_TIMESTAMP. Diese Funktionen liefern einfach das aktuelle Datum, die aktuelle Uhrzeit oder aber beides zurück. Folgendes Beispiel verdeutlicht noch einmal die Verwendung der drei Funktionen:

```
SELECT CURRENT_DATE() AS 'Aktuelles Datum',
       CURRENT_TIME(2) AS 'Aktuelle Zeit',
       CURRENT_TIMESTAMP(6) AS 'Aktuelles Datum/Zeit'

Aktuelles Datum  Aktuelle Zeit  Aktuelles Datum/Zeit
---------------  -------------  ----------------------
2002-10-15       09:25:12.92    2002-10-15 09:25:12.920
```

8.4.5 NULL-Funktionen und Datentypkonvertierung

Oft werden in Datenbanken anstelle von NULL andere Daten verwendet, um anzuzeigen, dass ein Wert nicht vorhanden oder unbekannt ist. Angenommen, in unserem Fallbeispiel hätte Frau Kart bei dem Lagerbestand für die Werbeartikel immer die Zahl 9999 anstelle von NULL verwendet, um anzuzeigen, dass der Lagerbestand zur Zeit unbekannt ist. Würden wir die Tabelle Werbartikel abfragen, so erhielten wir überall dort, wo der Wert von Lagerbestand 9999 ist, nicht NULL für unbekannt sondern 9999. Um bei einer Abfrage Werte, die für unbekannt stehen, als NULL auszugeben, gibt es die Funktion NULLIF. Der Funktion NULLIF werden zwei Werte übergeben, sind diese beiden Werte identisch, so wird einfach NULL zurückgegeben. Sehen wir uns hierzu unser Beispiel mit den Werbeartikeln an:

```
SELECT Beschreibung,
       Lagerbestand,
       NULLIF(Lagerbestand, 9999) AS 'Lagerbestand'
FROM Werbeartikel
```

```
Beschreibung               Lagerbestand  Lagerbestand
-------------------------  ------------  ------------
Noten f. Klavier           26            26
T-Shirt Farbe: rot         11            11
Phil Collins in Concert    21            21
Plakat mit Musical-Katzen  89            89
Opernführer                9999          NULL
Schwarzer Zauberstock      32            32
```

Als Ergebnis erhalten wir den Lagerbestand, wie er in der Tabelle »Werbeartikel« gespeichert ist und den Lagerbestand, wenn die Zahl 9999 als NULL interpretiert werden soll. Für den Werbeartikel mit der Beschreibung »Opernführer« wurde als Lagerbestand 9999 eingetragen Über die Funktion NULLIF wird er als NULL und damit unbekannt ausgegeben.

Eine weitere Funktion, die man auf Spalten mit NULL anwendet, ist COALESCE. Der Funktion COALESCE kann man eine beliebige Anzahl an Spalten oder Konstanten übergeben. Als Rückgabe wird der erste Wert der übergebenen Parameter zurückgegeben, der nicht NULL ist. *COALESCE(NULL, 'Zweiter')* würde also den Wert »Zweiter« zurückliefern. COALESCE wird dort eingesetzt, wo man bei einer Abfrage für NULL z.B. ein Fragezeichen oder das Wort »Unbekannt« ausgeben möchte. Sollen z.B. alle Kundennamen und deren Strasse und Hausnummer, in der sie wohnen, ausgegeben werden und für Strassennamen und Hausnummern, die nicht bekannt sind, das Wort »Unbekannt«, so sieht die Abfrage folgendermaßen aus:

```
SELECT Name,
       COALESCE( Strasse, '<Unbekannte Strasse>' ),
       COALESCE( Hausnummer, '<Unbekannte Hausnummer>' )
FROM   Kunde

Name
---------  ----------  ----------------------
Bolte      Busweg      12
Muster     Musterweg   12
Wiegerich  Wanderstr.  <Unbekannte Hausnummer>
Carlson    Petristr.   201
```

Die dritte Funktion, die wir uns in diesem Abschnitt ansehen wollen, dient zur Konvertierung von Werten eines bestimmten Datentyps in einen anderen Datentyp. In Abschnitt 8.3 haben wir gesehen, dass man mit Spalten auch rechnen kann. In einer Abfrage haben wir eine Ergebnistabelle ausgegeben, die den Preis um 10% erhöht anzeigt. Die Abfrage dazu lautete:

```
SELECT Beschreibung, Preis, Preis * 1.10 AS 'Preis + 10%'
FROM   Werbeartikel
```

Zwar wurde der Preis mit dem Datentyp DECIMAL(8, 2) deklariert, durch die Multiplikation ergab sich jedoch ein Ergebnis, das mehr als zwei Nachkommastellen benötigte. Diese Konvertierung in einen Datentyp mit einem größeren Wertebereich wurde

vom RDBMS selbständig durchgeführt. Da ein Preis mit vier Nachkommastellen in der Regel nicht gewollt ist, kann man diesen Datentyp explizit in einen ganz bestimmten Datentyp konvertieren. Dazu verwendet man die Funktion CAST. Betrachten wir zunächst ein einfaches Beispiel:

```
SELECT CAST( 9.98 AS INTEGER),
       CAST( 1 AS DECIMAL(6,3)),
       CAST( '12' AS INTEGER ) * 3

-----  -----  -----
9      1.000  36
```

Wir können erkennen, dass die erste Zahl 9.98 einem numerischem Datentyp mit Nachkommastellen entspricht. Eine Konvertierung in eine Ganzzahl, in diesem Fall dem Datentyp INTEGER, »schneidet« sozusagen die Nachkommastellen ab, um den Wert in eine Ganzzahl zu konvertieren. Die zweite Spalte zeigt die Ganzzahl 1 als 1.000, die in den Datentyp DECIMAL(6, 3) konvertiert wurde, und deshalb drei Nachkommastellen erhalten hat. Das letzte Beispiel schließlich wandelt eine Zeichenkette in eine Ganzzahl um, so dass mit dieser gerechnet werden kann (die Konvertierung von '12' ergibt die Zahl 12, die dann mit 3 multipliziert 36 ergibt).

Betrachten wir nun noch einmal unser obiges Beispiel mit dem erhöhten Preis. Um das Ergebnis der Berechnung auf zwei Nachkommastellen zu begrenzen, müssen wir einfach das Ergebnis in den Datentyp DECIMAL(8, 2) konvertieren:

```
SELECT Beschreibung, Preis,
       CAST( Preis * 1.10 AS DECIMAL(8, 2) ) AS 'Preis + 10%'
FROM   Werbeartikel

Beschreibung               Preis  Preis + 10%
-------------------------  -----  -----------
Noten f. Klavier           89.00  97.90
T-Shirt Farbe: rot         29.99  32.99
Phil Collins in Concert    20.00  22.00
Plakat mit Musical-Katzen  33.50  36.85
Opernführer                19.99  21.99
Schwarzer Zauberstock      5.99   6.59
```

8.4.6 »Set Functions«

Mengenfunktionen oder »Set Functions« beziehen sich, wie der Name schon sagt, immer auf eine Menge von Werten und berechnen aus dieser Menge je nach Funktion ein bestimmtes Ergebnis. So gibt es Mengenfunktionen, um die Werte einer Spalte aufzusummieren, den Durchschnittswert zu berechnen, den kleinsten oder größten Wert einer Spalte zu ermitteln oder einfach nur um die Anzahl der Werte festzustellen. Da Mengenfunktionen Werte aggregieren, werden sie häufig auch als Aggregatfunktionen bezeichnet.

Um z.B. den Durchschnittspreis eines Werbeartikels zu ermitteln, würde das RDBMS mit Hilfe der Mengenfunktion AVG folgendermaßen vorgehen:

```
SELECT  AVG( Preis ) AS 'Durchschnitt'
FROM    Werbeartikel
```

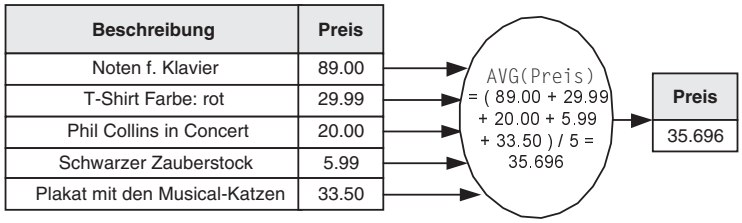


Abbildung 8.2: Verwendung von Mengenfunktionen (z.B. AVG)

Schauen wir uns als Erstes die Funktion COUNT an. Der Funktion COUNT wird in der Regel der Name einer Spalte übergeben. COUNT ermittelt dann, wie viele Sätze es gibt, die einen Wert in dieser Spalte haben.

Betrachten wir hierzu die Tabelle »Kunde«. Wir wissen, dass die Tabelle »Kunde« vier Datensätze enthält, da wir diese in Kapitel 7 über die INSERT-Anweisung eingefügt haben. Um nun die Anzahl der Zeilen über eine SELECT-Anweisung zu erhalten, geben wir ein:

```
SELECT COUNT(Kundennummer) AS 'Satzanzahl'
FROM   Kunde

Satzanzahl
----------
4
```

Mengenfunktionen berücksichtigen bei ihren Berechnungen keine NULL-Werte. Wäre also die Kundennummer eines Datensatzes in der Tabelle unbekannt, so würde die obige Anweisung als Ergebnis 3 zurückgeben. Wir wissen aber, dass »Kundennummer« als Primärschlüssel definiert wurde. Da Primärschlüssel immer eindeutig sein müssen, können hier niemals NULL-Werte auftreten. Würden wir jedoch anstatt der »Kundennummer« die Spalte »Hausnummer« an die Funktion übergeben, wäre das Ergebnis 3, da bei der Kundin Frieda Wiegerich die Hausnummer nicht eingetragen, also NULL ist.

Als einzige Mengenfunktion bietet COUNT die Möglichkeit, anstelle eines Spaltennamens auch das Platzhaltersymbol »*« zu verwenden. In diesem Fall wird bei COUNT immer die komplette Anzahl Sätze zurückgegeben, da COUNT dann alle Spaltenwerte beim Zählen berücksichtigt.

Die Mengenfunktionen MAX und MIN geben den höchsten und den niedrigsten Wert einer Spalte zurück. Um z.B. den Werbeartikel mit dem höchsten Preis und den Werbeartikel mit dem niedrigsten Preis zu ermitteln, lautet die SELECT-Anweisung:

```
SELECT MAX( Preis ), MIN( Preis )
FROM   Werbeartikel

-----  -----
89.00  5.99
```

Der teuerste Werbeartikel kostet also 89.00 Eur und der günstigste 5.99 Eur.

MIN und MAX funktionieren nicht nur auf Spalten mit numerischem Datentyp. Um den Namen eines Kunden zu erhalten, der alphabetisch an erster bzw. an letzter Stelle steht, lautet die SQL-Anweisung:

```
SELECT MAX( Name), MIN( Name)
FROM   Kunde

---------  ------
Wiegerich  Bolte
```

Um den Durchschnitt der Werte einer Spalte zu berechnen, gibt es die Funktion AVG (»Average«) und um die Werte einer Spalte aufzusummieren SUM.

Betrachten wir hierzu jeweils ein Beispiel. Der Durchschnittspreis aller Werbeartikel ergibt sich durch folgende SELECT-Anweisung:

```
SELECT AVG( Preis )
FROM   Werbeartikel

----------
33.078333
```

Durch Anwendung der Funktion CAST, die wir im vorherigen Abschnitt kennen gelernt haben, könnten wir zusätzlich noch die Nachkommastellen auf zwei begrenzen, indem wir das Ergebnis von AVG(Preis) an die Funktion CAST übergeben:

```
CAST( AVG( Preis ) AS DECIMAL(8, 2))
```

Ebenso kann eine Spalte über die Funktion SUM aufsummiert werden. Als Beispiel wollen wir einmal den Gesamtwert aller Werbeartikel ausgeben. Der Gesamtwert eines einzelnen Werbeartikels ergibt sich aus dem Preis multipliziert mit dem Lagerbestand. Auf diesen berechneten Wert muss dann die SUM-Funktion wie folgt angewendet werden:

```
SELECT SUM( Preis * Lagerbestand ) AS 'Gesamtwert'
FROM   Werbeartikel

Gesamtwert
----------
206117.08
```

Doch halt, diese Summe erscheint etwas hoch! Im letzten Abschnitt haben wir gesehen, dass Frau Kart für Werbeartikel, deren Lagerbestand nicht bekannt war, die Zahl 9999 eingetragen hat, d.h. 9999 steht für unbekannt, für NULL. Die Funktion SUM weiß natürlich nicht, dass mit 9999 eigentlich unbekannt gemeint ist und betrachtet dies als den Lagerbestand für den Opernführer. Um das korrekte Ergebnis zu erhalten, müssen wir die Funktion NULLIF auf Lagerbestand innerhalb der Berechnung wie folgt verwenden:

```
SELECT SUM( Preis * NULLIF(Lagerbestand, 9999) ) AS 'Gesamtwert'
FROM   Werbeartikel

Gesamtwert
----------
6237.07
```

Als Ergebnis erhalten wir die korrekte Gesamtsumme von 6237,07 Eur. Wir erkennen hier also ein Problem mit unbekannten Werten: Mengenfunktionen berücksichtigen bei ihren Berechnungen NULL nicht, was ja auch durchaus einleuchtend erscheint. Wurde nun aber innerhalb einer Spalte ein Wert verwendet, der semantisch für unbekannt steht, so kann dies zu falschen Ergebnissen führen. In unserem Fall wäre es also sinnvoll, anstelle der Zahl 9999 die Kennung NULL zu verwenden.

8.5 Ausdrücke

Wir haben Ausdrücke bereits indirekt in den bisherigen Anweisungen kennen gelernt. Der Vollständigkeit halber wollen wir sie hier noch einmal zusammenfassen, ohne im Detail darauf einzugehen, da sie sich in der Regel aus den Beispielen ergeben.

Mit numerischen Datentypen kann gerechnet werden. Dementsprechend gibt es Operatoren für Addition, Subtraktion, Division, Multiplikation sowie Klammern zur Festlegung der Berechnungs-Reihenfolge.

Ausdruck	Beschreibung	Beispiel
+	Addition	Preis + 1.00
-	Subtraktion	Lagerbestand – 6
*	Multiplikation	Preis * 1.10
/	Division	Gesamtwert / Produktanzahl
( )	Klammern	(Lagerbestand – 6) * 7

Tabelle 8.1: Numerische Ausdrücke

Zeichenketten kennen nur einen einzigen Ausdruck, nämlich den zum Verbinden von Zeichenketten. Um z.B. innerhalb einer Abfrage Vor- und Nachname in einer Spalte in der Ergebnistabelle auszugeben, verbindet man diese beiden Spalten über die beiden Zeichen | | miteinander:

```
SELECT Name || ', ' || Vorname
FROM   Kunde

-----------------
Bolte, Bertram
Muster, Hans
Wiegerich, Frieda
Carlson, Peter
```

Ausdruck	Beschreibung	Beispiel
||	Zeichenketten verbinden	Name || ', ' || Vorname

Tabelle 8.2: Zeichenketten Ausdrücke

Zu Datums- und Zeitwerten können Zeitintervalle addiert oder aber subtrahiert werden.

Ausdruck	Beschreibung	Beispiel
+	Addition	Datum + INTERVAL '7' DAY
-	Subtraktion	Termin – INTERVAL '2' HOUR

Tabelle 8.3: Datum/Zeit Ausdrücke

Neben den dargestellten Ausdrücken gibt es noch so genannte logische Ausdrücke. Da diese aber eher beim Auswählen von Zeilen von Bedeutung sind, werden wir sie auch dort behandeln.

Wir haben bisher gelernt, wie man über die SELECT-Anweisung angibt, welche Spalten einer Tabelle man als Ergebnis erhalten möchte und wie man die unterschiedlichsten Funktionen auf die angegebenen Spalten anwenden kann. Wie die einzelnen Zeilen ausgewählt werden, die man als Ergebnistabelle erhalten möchte, wollen wir uns im nächsten Abschnitt ansehen.

8.6 Sätze auswählen

8.6.1 Grundlagen

Der allgemeine Aufbau der SELECT-Anweisung sieht, wie wir bereits kennen gelernt haben, wie folgt aus:

```
  SELECT   spaltenliste
[ FROM     tabellenliste ]
[ WHERE    bedingungsausdruck ]
[ GROUP BY spaltenliste ]
[ HAVING   bedingungsausdruck ]
[ ORDER BY spaltenliste ]
```

Hinter dem Schlüsselwort SELECT wird angegeben, welche Spalten man als Ergebnistabelle erhalten möchte und hinter dem Schlüsselwort FROM, in welcher Tabelle sich die angegebenen Spalten befinden. Über das Schlüsselwort WHERE werden nun die Zeilen ausgewählt, die in der Ergebnistabelle ausgegeben werden sollen. Nehmen wir dazu ein einfaches Beispiel: Sie möchten die Tabelle »Werbeartikel« ausgeben, jedoch nur die Zeilen, bei denen der Artikel mehr als 20.00 Eur kostet. Die folgende Abbildung verdeutlicht das Vorgehen:

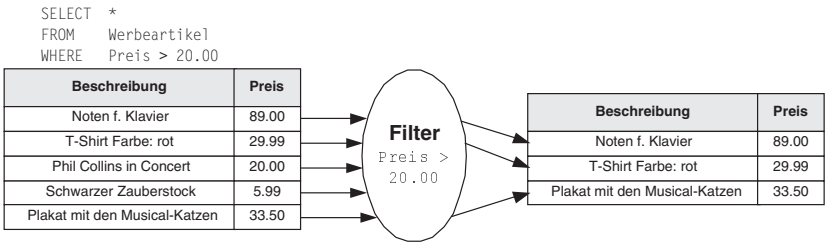


Abbildung 8.3: Selektion (Auswahl von Datensätzen)

Zunächst gibt man hinter der SELECT-Anweisung an, welche Spalten ausgegeben werden sollen. In diesem Fall alle, also wird das Platzhaltersymbol »*« verwendet. Danach muss hinter FROM bestimmt werden, in welcher Tabelle sich die gewünschten Spalten befinden, hier: »FROM Werbeartikel«. Nun soll die Ergebnistabelle noch auf die Zeilen beschränkt werden, deren Preis größer als 20.00 Eur ist. Dafür gibt man hinter WHERE den entsprechenden Bedingungsausdruck »Preis > 20.00« an. Jede Zeile der Tabelle Werbeartikel wird nun auf diese Bedingung hin überprüft. Der erste Artikel kostet 89.00, damit durchläuft er die Bedingung, die in diesem Fall »wahr« ergibt. Dasselbe gilt für den zweiten Artikel mit einem Preis von 29.99. Der dritte Artikel kostet genau 20.00 Eur. Als Bedingung haben wir angegeben, dass der Preis größer als 20.00 Eur sein soll, daher ergibt die Bedingung hier »falsch« und der Satz wird nicht in die Ergebnistabelle übernommen.

Ein Bedingungsausdruck kann aus einer oder mehreren Bedingungen bestehen, die über logische Verknüpfungsausdrücke wie OR oder AND miteinander verbunden sind. Der Bedingungsausdruck wird für jeden einzelnen Satz auf »wahr« oder »falsch« überprüft. Ergibt die Überprüfung »wahr«, so wird er Satz in die Ergebnistabelle übernommen, andernfalls nicht. Operatoren von Bedingungen, die als Ergebnis »wahr«, »falsch« oder »unbekannt« zurückliefern, werden als Prädikate bezeichnet.

Wir wollen uns zunächst mit Bedingungsausdrücken mit nur einer einzigen Bedingung auseinandersetzen. Danach werden wir sehen, wie man die einzelnen Bedingungen miteinander verknüpft.

8.6.2 Vergleichsprädikate und IS NULL

Ein Vergleichsprädikat haben wir bereits im einleitenden Beispiel kennen gelernt, das »>«-Zeichen. Insgesamt gibt es sechs Vergleichsprädikate. Für die Überprüfung auf unbekannte Werte, also auf NULL, gibt es ein gesondertes Prädikat.

Vergleichsprädikat	Beschreibung	Beispiel
=	gleich	Name = 'Muster'
<>	ungleich	Lagerbestand <> 0
>	größer als	Preis > 20.00
<	kleiner als	Preis < 20.00
>=	größer als oder gleich	Preis >= 20.00
<=	kleiner als oder gleich	Preis <= 20.00
IS NULL	gleich NULL	Hausnummer IS NULL
IS NOT NULL	ungleich NULL	Lagerbestand IS NOT NULL

Tabelle 8.4: Vergleichsprädikate

Da in SQL das Schlüsselwort NULL keinen Wert darstellt, erlaubt SQL auch nicht den Vergleich von NULL mit dem Vergleichsprädikat »=«. Der Bedingungsausdruck »Hausnummer = NULL« ist also nicht korrekt. SQL sieht ein gesondertes Prädikat, IS NULL, hierfür vor. Um z.B. die Namen aller Kunden auszugeben, deren Hausnummer nicht bekannt ist, gibt man an:

```
SELECT Name, Vorname
FROM   Kunde
WHERE  Hausnummer IS NULL

Name       Vorname
---------  -------
Wiegerich  Frieda
```

8.6.3 LIKE und SIMILAR Prädikat

Mit LIKE bzw. SIMILAR kann man Zeichenketten auf bestimmte Zeichenfolgen hin untersuchen. Betrachten wir hierzu ein Beispiel: Um die Namen aller Kunden auszugeben, deren Nachname »Muster« ist, lautet die Abfrage:

```
SELECT Name, Vorname
FROM   Kunde
WHERE  Name = 'Muster'
```

Über diese Abfrage erhalten wir alle Kunden, deren Nachname genau »Muster« lautet. Wollen wir dagegen die Namen aller Kunden, deren Namen mit einem »B« beginnt, so müssen wir das Prädikat LIKE verwenden. LIKE kann wie der Gleichheitsoperator verwendet werden, nur mit dem Unterschied, dass LIKE Platzhaltersymbole kennt. Um nun alle Kunden, die mit einem »B« beginnen auszugeben, verwendet man das Platzhaltersymbol »%«, dass für eine beliebige Folge von Zeichen steht. Die SELECT-Anweisung sieht wie folgt aus:

```
SELECT Name, Vorname
FROM   Kunde
WHERE  Name LIKE 'B%'
```

Möchte man dagegen alle Kunden ausgegeben haben, die an einer beliebigen Stelle in ihrem Nachnamen die Zeichen »er« stehen haben, so lautet die Anweisung:

```
SELECT Name, Vorname
FROM   Kunde
WHERE  Name LIKE '%er%'
```

Der erste Teil des Namens kann beliebige Zeichenfolgen enthalten, dann müssen die Zeichen »er« folgen und anschließend wieder eine beliebige Zeichenfolge, was durch das Platzhaltersymbol »%« angedeutet wird.

LIKE kennt neben »%« noch ein weiteres Platzhaltersymbol, den Unterstrich. Er steht im Gegensatz zum »%«-Symbol für genau ein einziges beliebiges Zeichen. Um also alle Kunden auszugeben, deren Nachname ab der dritten Stelle die beiden Buchstaben »lt« enthält, müssen hinter LIKE zwei Unterstriche für zwei beliebige Buchstaben angegeben werden:

```
SELECT Name, Vorname
FROM   Kunde
WHERE  Name LIKE '_ _lt%'
```

Allerdings haben wir nun noch ein Problem mit dem LIKE-Prädikat: Was ist, wenn wir nach einer Zeichenkette suchen, die das »%«-Zeichen selbst enthält. Dazu bietet das LIKE-Prädikat über das Schlüsselwort ESCAPE die Möglichkeit, ein Zeichen zu definieren, das man vor einem Platzhaltersymbol schreiben kann, wenn dieses in die Suche miteinbezogen werden soll. Um also in der Bezeichnung einer Veranstaltung nach der Zeichenfolge »hundert%ig gut« zu suchen, lautet der SELECT-Befehl:

```
SELECT Bezeichnung
FROM   Veranstaltung
WHERE  Bezeichnung LIKE '%hundert#%ig gut%' ESCAPE '#'
```

Das Prädikat LIKE ist zwar sehr nützlich, aber nicht immer ausreichend. Deshalb kennt SQL:1999 ein weitere Prädikat zum Vergleichen von Zeichenketten: SIMILAR. Wer sich mit UNIX auskennt, hat dort vielleicht schon einmal mit regulären Ausdrücken gearbeitet. Mit regulären Ausdrücken kann für jede Position innerhalb einer Zeichenkette genau angegeben werden, welches Zeichen man suchen möchte. Hier soll nicht detailliert auf das Prädikat SIMILAR eingegangen, sondern nur kurz ein Beispiel erläutert werden. Um z. B. nach einer Postleitzahl zu suchen, die als erste Zahl eine 2 oder 3, dann drei beliebige Zahlen und als letzte Stelle eine 6 oder 3 hat, lautet die SELECT-Anweisung:

```
SELECT Plz
FROM   Spielstaette
WHERE  Plz SIMILAR TO '[23][0-9][0-9][0-9][63]'
```

```
Plz
-----
33333
25746
```

Für jede Position innerhalb der Suchzeichenkette gibt man in eckigen Klammern an, welchen Wert diese Stelle annehmen darf. Dabei sind auch Bereichsangaben, wie z.B. 0-9, möglich. Hiermit soll beispielhaft gezeigt werden, wie leistungsfähig reguläre Ausdrücke sind. Über die Suchzeichenkette, also den regulären Ausdruck, können noch wesentlich genauere Suchangaben gemacht werden, als hier vorgestellt. Als kurze Einführung soll dies jedoch ausreichen.

8.6.4 BETWEEN-Prädikat

Um innerhalb einer Abfrage Spalten nach bestimmten Wertebereichen und nicht nur nach einem einzelnen Wert abzufragen, kennt SQL das BETWEEN-Prädikat. Hiermit können z.B. Fragen wie »Welche Werbeartikel gibt es, die zwischen 20.00 und 60.00 Eur kosten?« beantwortet werden. Betrachten wir die SELECT-Anweisung hierzu:

```
SELECT Beschreibung, Preis
FROM   Werbeartikel
WHERE  Preis BETWEEN 20.00 AND 60.00

Beschreibung               Preis
-------------------------  -----
T-Shirt Farbe: rot         29.99
Phil Collins in Concert    20.00
Plakat mit Musical-Katzen  33.50
```

Über BETWEEN…AND wird der Bereich angegeben, in dem die gesuchten Werte liegen sollen. Das Prädikat BETWEEN kann auch negiert werden, d.h. will man alle Werbeartikel ausgeben, die im Preis nicht zwischen 20.00 und 60.00 Eur liegen, so lautet die SQL-Anweisung:

```
SELECT Beschreibung, Preis
FROM   Werbeartikel
WHERE  Preis NOT BETWEEN 20.00 AND 60.00
```

8.6.5 IN-Prädikat

Mit dem Prädikat IN kann eine bestimmte Spalte auf mehrere Werte hin überprüft werden. Möchte man z.B. die Postleitzahl von den Orten Rettrich, Karlstadt oder Kohlscheidt ermitteln, so lautet die SELECT-Anweisung mit dem IN-Prädikat:

```
SELECT *
FROM   Ort
WHERE  Ort IN ( 'Rettrich', 'Karlstadt', 'Kohlscheidt')
```

```
Plz    Ort
-----  ---------
22222  Karlstadt
33333  Rettrich
44444  Kohlscheidt
```

Wie beim BETWEEN-Prädikat auch, kann das IN-Prädikat über das Schlüsselwort NOT negiert werden.

8.6.6 Logische Operatoren

Bisher haben wir uns ausschließlich Beispiele mit einer einzigen Bedingung angesehen. Bedingungen können jedoch über die logischen Operatoren AND, OR und NOT miteinander verbunden werden. Um z. B. alle Kunden auszugeben, die männlich sind und in einem Ort mit der Postleitzahl 44444 wohnen, lautet die SELECT-Anweisung:

```
SELECT Name, Geschlecht, Plz
FROM   Kunde
WHERE  Geschlecht = 'M' AND Plz = '44444'

Name     Geschlecht  Plz
-------  ----------  -----
Bolte    M           44444
Carlson  M           44444
```

Möchte man dagegen alle Kunden ausgegeben haben, die entweder männlich sind oder in einem Ort mit der Postleitzahl 44444 wohnen, so verknüpft man die beiden Bedingungen wie folgt mit OR:

```
SELECT Name, Geschlecht, Plz
FROM   Kunde
WHERE  Geschlecht = 'M' OR Plz = '44444'

Name     Geschlecht  Plz
-------  ----------  -----
Muster   M           22222
Bolte    M           44444
Carlson  M           44444
```

Bedingungen können beliebig mit AND, OR oder NOT verknüpft werden. Um die Reihenfolge der Auswertung der Ausdrücke zu bestimmen, verwendet man Klammern.

Betrachten wir hierzu ein weiteres Beispiel. Wir möchten alle Kunden ausgegeben haben, die männlich sind oder in einem Ort mit der Postleitzahl 44444 wohnen und deren Name an einer beliebigen Stelle ein »e« enthält.

```
SELECT Name, Geschlecht, Plz
FROM   Kunde
WHERE  ( Geschlecht = 'M' OR Plz = '44444' ) AND Name LIKE '%e%'
```

```
Name     Geschlecht  Plz
-------  ----------  -----
Muster   M           22222
Bolte    M           44444
```

Diesmal fällt der Kunde Carlson aus der Ergebnismenge heraus. Zunächst überprüft das RDBMS die Bedingung in der Klammer. Diesen Bedingungsausdruck erfüllt Herr Carlson. In der zweiten Bedingung wird geprüft, ob der Name des Kunden ein »e« enthält. Diese Bedingung erfüllt Herr Carlson nicht. Da nun beide Ergebnisse über den AND-Operator verknüpft werden, Herr Carlson aber nur den ersten Bedingungsausdruck erfüllt und nicht den zweiten, erscheint er letztendlich nicht in der Ergebnistabelle.

Sehen wir uns nun einmal an, wie das Ergebnis aussieht, wenn wir die Klammern weglassen:

```
SELECT Name, Geschlecht, Plz
FROM   Kunde
WHERE  Geschlecht = 'M' OR Plz = '44444' AND Name LIKE '%e%'

Name     Geschlecht  Plz
-------  ----------  -----
Muster   M           22222
Bolte    M           44444
Carlson  M           44444
```

Jetzt ist Herr Carlson wieder in der Ergebnistabelle enthalten. Der Grund hierfür liegt in der Vorgehensweise von SQL bei der Auswertung des Bedingungsausdrucks. Ohne Klammerung hat der AND-Operator Vorrang vor dem OR-Operator. Zunächst werden alle Kunden herausgesucht, deren Name ein »e« enthält und die gleichzeitig aus einem Ort mit der Postleitzahl 44444 stammen. Danach werden alle Kunden gesucht, die männlich sind. Diese Bedingung erfüllt Herr Carlson wieder und da die Verknüpfung durch OR erfolgt, erscheint er auch in der Ergebnistabelle.

Wir haben nun die wichtigsten Sprachelemente der SELECT-Anweisung kennen gelernt, die sich auf eine einzige Tabelle beziehen. In den letzten beiden Abschnitten wollen wir noch einmal sehen, wie man die Ergebnistabelle sortiert oder nach bestimmten Spalten gruppiert.

8.7 Sätze zusammenfassen

Betrachten wir zunächst wieder den allgemeinen Aufbau der SELECT-Anweisung:

```
  SELECT   spaltenliste
[ FROM     tabellenliste ]
[ WHERE    bedingungsausdruck ]
[ GROUP BY spaltenliste ]
[ HAVING   bedingungsausdruck ]
[ ORDER BY spaltenliste ]
```

Bisher haben wir gesehen, wie man Spalten hinter dem Schlüsselwort SELECT, Tabellen hinter dem Schlüsselwort FROM und Zeilen aufgrund von Bedingungsausdrücken hinter dem Schlüsselwort WHERE auswählt. Damit sind wir schon in der Lage, die unterschiedlichsten Informationen aus einer Datenbank, bezogen auf eine einzige Tabelle, herauszusuchen.

Neben dem reinen Suchen und Finden von Daten kann man mit der SELECT-Anweisung aber auch Daten kumulieren. Damit dient die SELECT-Anweisung u.a. auch der Analyse von Daten, indem sie Werte aggregiert. Hierzu kennt die SELECT-Anweisung die Schlüsselwörter GROUP BY. Über GROUP BY kann man die Daten nach bestimmten Spalten gruppieren, d.h. es werden Gruppen mit gleichen Werten gebildet. Für diese Gruppen wird in der Regel eine Berechnung durchgeführt. Berechnungen auf mehrere Werte einer Spalte haben wir bisher nur bei den Mengen- oder Aggregatfunktionen kennen gelernt. GROUP BY ist in der Regel auch nur in Zusammenhang mit einer Mengenfunktion sinnvoll.

Betrachten wir ein einfaches Beispiel: Wir wollen Gruppen von Kunden bilden, die aus dem gleichen Ort stammen, die also die gleiche Postleitzahl haben. Um alle Postleitzahlen auszugeben, sieht die SELECT-Anweisung folgendermaßen aus:

```
SELECT Plz
FROM   Kunde

Plz
-----
44444
22222
33333
44444
```

In der Ergebnistabelle werden Postleitzahlen auch doppelt aufgeführt, da zwei Kunden (Bolte und Carlson) aus dem gleichen Ort mit der Postleitzahl 44444 stammen. Um doppelte Werte zu entfernen, haben wir vorher bereits das Schüsselwort DISTINCT kennen gelernt. Fügen wir also direkt hinter SELECT das Schlüsselwort DISTINCT ein, so wird die Postleitzahl 44444 nur einmal aufgeführt. Doch es gibt noch eine andere Möglichkeit: Wir können gleiche Werte über GROUP BY zusammenfassen. Das Ganze sieht dann wie folgt aus:

```
SELECT Plz
FROM Kunde
GROUP BY Plz

Plz
-----
22222
33333
44444
```

Damit haben wir zunächst das gleiche Ergebnis erzielt wie mit dem Schlüsselwort DISTINCT. Hier werden jetzt aber die Mengenfunktionen interessant. Normalerweise kann man Mengenfunktionen in der Spaltenliste nicht mit Spaltennamen kombinieren. Folgende SELECT-Anweisung würde also eine Fehlermeldung erzeugen:

```
SELECT COUNT(Plz), Name     -- falsche SELECT-Anweisung
FROM   Kunde
```

Das ist auch einleuchtend, denn COUNT(Plz) würde nur eine Zeile, nämlich die Zahl 4 zurückliefern. Name dagegen würde vier Werte zurückliefern, nämlich Bolte, Muster, Wiegerich und Carlson.

Wir können nun aber beide Abfragen miteinander kombinieren, indem wir zum einen die Anzahl der Plz über COUNT zählen, gruppiert nach den unterschiedlichen Plz. Die folgende SELECT-Anweisung gibt alle unterschiedlichen Postleitzahlen, und die Anzahl der Kunden zurück, die in dem Ort mit der jeweiligen Postleitzahl wohnen:

```
SELECT Plz, COUNT(Plz) AS 'Anzahl Kunden/Plz'
FROM   Kunde
GROUP BY Plz

Plz    Anzahl Kunden/Plz
-----  -----------------
22222  1
33333  1
44444  2
```

Grafisch sieht das Ganze wie folgt aus:

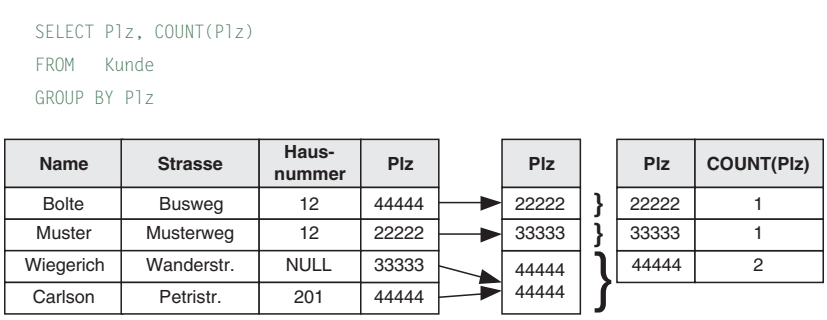


Abbildung 8.4: Zusammenfassen von Sätzen

Wir sehen also, dass unsere Kunden aus Orten mit den Postleitzahlen 22222, 33333 und 44444 kommen. Zwei Kunden wohnen in dem Ort mit der Postleitzahl 44444 und je ein Kunde in den Orten mit den anderen beiden Postleitzahlen.

Betrachten wir noch einmal den allgemeinen Aufbau der SELECT-Anweisung am Anfang diese Abschnitts 8.7, so sehen wir unmittelbar unter der GROUP BY-Klausel

das Schlüsselwort HAVING. Mit HAVING kann die Ergebnistabelle einer GROUP BY-Abfrage eingeschränkt werden. Angenommen es sollen alle Postleitzahlen und die Anzahl der in diesen Orten wohnenden Kunden ermittelt werden, aber nur wenn die Anzahl der Kunden pro Ort größer als 1 ist, so lautet die SELECT-Anweisung:

```
SELECT Plz, COUNT(Plz) AS 'Anzahl Kunden/Plz'
FROM   Kunde
GROUP  BY Plz
HAVING COUNT(Plz) > 1

Plz    Anzahl Kunden/Plz
-----  -----------------
44444  2
```

Jetzt werden nur die Postleitzahlen von Orten angezeigt, wo mindestens zwei Kunden wohnen. Wie würde nun die Abfrage lauten, wenn man die Anzahl der Kunden pro Ort ermitteln möchte, diesmal aber nur für Postleitzahlen größer 22222? Hierfür gibt es zwei Möglichkeiten: Entweder über die HAVING-Klausel, wie wir es gerade am Beispiel gesehen haben, oder aber über die WHERE-Klausel. Die beiden folgenden SELECT-Anweisungen sind identisch und liefern die gleiche Ergebnistabelle:

```
SELECT Plz, COUNT(Plz) AS 'Anzahl Kunden/Plz'
FROM   Kunde
GROUP  BY Plz
HAVING Plz > '22222'

SELECT Plz, COUNT(Plz) AS 'Anzahl Kunden/Plz'
FROM   Kunde
WHERE  Plz > '22222'
GROUP  BY Plz
```

Die GROUP BY-Klausel dient in der Regel analytischen Zwecken, um z.B. den Umsatz der letzten drei Monate gruppiert nach Kundengruppen o.ä. auszugeben. Man spricht deshalb in diesem Zusammenhang auch von OLAP (»OnLine Analytical Processing«): Mengen von Daten werden zusammengefasst und analysiert. OLAP enthält jedoch viel mehr Eigenschaften als das reine Zusammenfassen und Gruppieren von Werten. Der SQL:1999 Standard sieht deshalb einen eigenen Teil für OLAP vor, der allerdings nicht Bestandteil dieses Buches ist.

SQL:1999 kennt jedoch noch zwei weitere grundlegende Schlüsselwörter, die man in Zusammenhang mit der GROUP BY-Klausel verwendet: ROLLUP und CUBE.

Bei Verwendung der GROUP BY-Klausel haben wir bisher immer nur nach einem ganz bestimmten Wert gruppiert. Haben wir z.B. als Gruppierungsspalte die beiden Attribute Plz und Geschlecht angegeben, so wurden immer alle Datensätze zusammengefasst, für die beide Werte identisch sind. ROLLUP ist dann sinnvoll, wenn man nach mehr als einer einzigen Spalte gruppieren möchte, in diesem Fall also nach Plz und Geschlecht. Um eine Liste gruppiert nach Plz, eine weitere Liste gruppiert nach

Geschlecht und eine dritte Liste gruppiert nach beiden Spalten zu erhalten, müssten folgende drei SELECT-Anweisungen geschrieben werden:

```
SELECT Geschlecht, COUNT(*) AS 'Anzahl Kunden'
FROM   Kunde
GROUP BY Geschlecht

SELECT Plz, COUNT(*) AS 'Anzahl Kunden'
FROM   Kunde
GROUP BY Plz

SELECT Geschlecht, Plz, COUNT(*) AS 'Anzahl Kunden'
FROM   Kunde
GROUP BY Geschlecht, Plz
```

Die erste Abfrage gibt die Anzahl der Kunden bezogen auf ihr Geschlecht zurück, die zweite die Anzahl der Kunden bezogen auf den Wohnort und die letzte Abfrage die Anzahl der Kunden bezogen auf beide Attribute.

Die Ergebnistabellen sehen also wie folgt aus:

```
Geschlecht  Anzahl Kunden
----------  -------------
M           3
W           1

Plz    Anzahl Kunden
-----  -------------
22222  1
33333  1
44444  2

Geschlecht  Plz    Anzahl Kunden
----------  -----  -------------
M           22222  1
W           33333  1
M           44444  2
```

Um nun nicht drei verschiedene Abfragen zu erstellen verwendet man ROLLUP und CUBE, um aus diesen drei Ergebnistabellen eine einzige zu erzeugen. Mit ROLLUP kann man die erste und dritte Abfrage in einer Abfrage ausdrücken. Die SELECT-Anweisung und die dazugehörige Ergebnistabelle sehen dann wie folgt aus:

```
SELECT Geschlecht, Plz, COUNT(*) AS 'Anzahl Kunden'
FROM   Kunde
GROUP BY ROLLUP( Geschlecht, Plz )
```

```
Geschlecht  Plz    Anzahl Kunden
----------  -----  -------------
M           22222  1
M           44444  2
M           NULL   3
W           33333  1
W           NULL   1
NULL        NULL   4
```

Aus dieser Ergebnistabelle sind fast die gleichen Informationen herauszulesen, wie aus den oberen drei Tabellen. Insgesamt gibt es 4 Kunden, 1 weiblichen und 3 männliche Kunden. Aus dem Wohnort mit der Postleitzahl 44444 kommen 2 männliche Kunden, ansonsten gibt es keine Kunden, die in einem Ort mit der gleichen Postleitzahl wohnen.

Allerdings fehlt in diesem Ergebnis die Gruppierung nach der Postleitzahl. ROLLUP gruppiert nach beiden Attributen und zusätzlich nach dem ersten angegebenen Attribut hinter GROUP BY.

Um nun zusätzlich auch nach dem zweiten Attribut, also Plz gesondert zu gruppieren, verwendet man CUBE. Die SELECT-Anweisung und die dazugehörige Ergebnistabelle sehen dann folgendermaßen aus:

```
SELECT Geschlecht, Plz, COUNT(*) AS 'Anzahl Kunden'
FROM   Kunde
GROUP BY CUBE( Geschlecht, Plz )
```

```
Geschlecht  Plz    Anzahl Kunden
----------  -----  -------------
M           22222  1
M           44444  2
M           NULL   3
W           33333  1
W           NULL   1
NULL        NULL   4
NULL        22222  1
NULL        33333  1
NULL        44444  2
```

8.8 Sätze sortieren

Die Reihenfolge der Datensätze unserer bisherigen Ergebnistabellen waren bisher eher zufällig. Da SQL mathematisch auf der Mengenlehre basiert, kennt es generell keine Reihenfolgen von Spalten oder Sätzen. SQL bietet deshalb die Möglichkeit, die Ergebnistabelle zu sortieren. Betrachten wir hierzu noch einmal den allgemeinen Aufbau der SELECT-Anweisung:

```
  SELECT   spaltenliste
[ FROM     tabellenliste ]
[ WHERE    bedingungsausdruck ]
[ GROUP BY spaltenliste ]
[ HAVING   bedingungsausdruck ]
[ ORDER BY spaltenliste ]
```

Die ORDER BY-Klausel erscheint immer am Ende einer SELECT-Anweisung und dient zum Sortieren der Ergebnistabelle. Hinter ORDER BY folgt eine Spaltenliste, nach der die Ergebnistabelle sortiert werden soll. Hinter jeder Spalte wiederum kann über die Schlüsselwörter ASC (»ascending«) oder DESC (»descending«) festgelegt werden, ob nach dieser Spalte aufsteigend oder absteigend sortiert werden soll.

Betrachten wir hierzu ein einfaches Beispiel: Wir wollen Name, Vorname und Plz aller Kunden ausgeben, sortiert nach der Spalte »Plz« in aufsteigender Reihenfolge.

```
SELECT Plz, Name, Vorname
FROM   Kunde
ORDER  BY Plz

Plz    Name       Vorname
-----  ---------  -------
22222  Muster     Hans
33333  Wiegerich  Frieda
44444  Carlson    Peter
44444  Bolte      Bertram
```

Möchte man aufsteigend sortieren, kann man auf das Schlüsselwort ASC verzichten, wie in der SELECT-Anweisung zu sehen. Anstatt einen Spaltennamen anzugeben, kann auch eine Zahl als Spalte angeben werden. Diese Zahl entspricht der Rangfolge der Spalten, wie sie hinter der SELECT-Anweisung aufgeführt werden. Um z.B. absteigend nach Name zu sortieren, kann man schreiben:

```
SELECT Plz, Name, Vorname
FROM   Kunde
ORDER  BY 2 DESC

Plz    Name       Vorname
-----  ---------  -------
33333  Wiegerich  Frieda
22222  Muster     Hans
44444  Carlson    Peter
44444  Bolte      Bertram
```

Generell ist es allerdings vorteilhafter, den Namen einer Spalte zu verwenden. Ändert man nämlich die Reihenfolge der Spalten hinter SELECT, muss auch die Zahl zur Sortierung geändert werden. Dieser zweite Arbeitsschritt entfällt bei Verwendung des Spaltennamens.

Ergebnistabellen können auch nach berechneten Spalten sortiert werden. Da berechnete Spalten normalerweise keinen eigenen Spaltennamen haben, muss eine Zahl als Sortierspalte angeben werden. Besser ist jedoch, der berechneten Spalte über das Schlüsselwort AS einen Namen zu geben und diesen bei der Sortierung zu verwenden. Um z.B. den Gesamtwert eines jeden Werbeartikels auszugeben und nach diesem zu sortieren, schreibt man folgende SELECT-Anweisung:

```
SELECT Beschreibung, Preis * Lagerbestand AS 'Gesamtwert'
FROM   Werbeartikel
ORDER  BY Gesamtwert

Beschreibung               Gesamtwert
-------------------------  ----------
Schwarzer Zauberstock      191.68
T-Shirt Farbe: rot         329.89
Phil Collins in Concert    420.00
Noten f. Klavier           2314.00
Plakat mit Musical-Katzen  2981.50
```

Innerhalb der ORDER BY-Klausel können auch Berechnungen durchgeführt und nach diesen Berechnungen sortiert werden. Spalten, nach denen sortiert wurde, müssen nicht unbedingt hinter SELECT aufgeführt sein. Um also die Werbeartikel nach ihrem Gesamtwert zu sortieren, kann man auch schreiben:

```
SELECT Beschreibung
FROM   Werbeartikel
ORDER  BY Preis * Lagerbestand
```

8.9 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie man Daten aus einer Tabelle abfragen kann und dass die Ergebnisse auch als Tabellen ausgegeben werden. Die SQL-Anweisung zum Abfragen von Tabellen lautet SELECT. Ihr allgemeiner Aufbau sieht folgendermaßen aus:

```
  SELECT   spaltenliste          -- Spaltenauswahl
[ FROM     tabellenliste ]       -- Tabellenauswahl
[ WHERE    bedingungsausdruck ]  -- Zeilenauswahl
[ GROUP BY spaltenliste ]        -- Gruppieren/Aggregieren
[ HAVING   bedingungsausdruck ]  -- Gruppenauswahl
[ ORDER BY spaltenliste ]        -- Sortieren
```

Wir haben uns in diesem Kapitel mit den einzelnen Bestandteilen der SELECT-Anweisung auseinandergesetzt.

Über SELECT werden Spalten auswählt, die in der Ergebnistabelle angezeigt werden sollen. Auf Spalten kann man Berechnungen durchführen oder so genannte »Built-in«-

Funktionen anwenden. Hierbei gibt es Funktionen zum Ändern oder Heraustrennen von Zeichenketten, zum Konvertieren in einen anderen Datentyp, zum Herauslesen der einzelnen Bestandteile eines Datums oder einer Uhrzeit usw. Eine Gruppe dieser Funktionen sind die Mengen- oder Aggregatfunktionen. Im Gegensatz zu den anderen Funktionsgruppen, liefern sie ein Ergebnis zurück, dass sich auf mehrere Sätze beziehen kann. Mit Mengenfunktionen können Werte einer Spalte summiert werden, es kann der Durchschnitt ermittelt werden usw. Gerade bei den einzelnen Funktionen gibt es zur Zeit erhebliche Unterschiede zwischen den einzelnen RDBMS-Produkten. Die Funktionalität der hier vorgestellten Funktionen existiert in der Regel zwar bei allen RDBMS-Produkten. Häufig jedoch unterscheiden sich die Funktionen der RDBMS-Produkte und des SQL:1999 Standards in den Bezeichnungen und dem Aufruf der jeweiligen Funktionen. Die hier kennen gelernten Funktionen entsprechen dem SQL:1999 Standard.

Mit der FROM-Klausel wird spezifiziert, in welchen Tabellen sich die Spalten befinden, die ausgewählt werden sollen. Bestimmte Datensätze wiederum werden mit dem Schlüsselwort WHERE ausgewählt. Hinter WHERE schreibt man einen Bedingungsausdruck, der die Ergebnistabelle auf die gewünschten Sätze beschränkt. Der Bedingungsausdruck setzt sich aus verschiedenen Prädikaten zusammen. Die wichtigsten Prädikate sind die Vergleichsprädikate, daneben gibt es IS NULL, LIKE, SIMILAR, BETWEEN und IN. Mehrere Bedingungen werden über die logischen Operatoren AND, OR und NOT verknüpft.

Die GROUP BY-Klausel wird in Zusammenhang mit der Anwendung von Mengenfunktionen verwendet. Über GROUP BY kann man Datensätze zusammenfassen und aggregierte Werte für die zusammengefassten Gruppen errechnen. Um die gruppierten Datensätze wieder auf bestimmte Gruppen zu beschränken, verwendet man HAVING.

Der letzte SELECT-Bestandteil dient schließlich dem Sortieren der Ergebnistabelle. Über die Schlüsselwörter ORDER BY kann die Ergebnistabelle nach Spalten sortiert werden.

Bis hierher haben wir also gelernt, wie man aus einer einzelnen Tabelle Informationen heraussucht. Das nächste Kapitel beschäftigt sich daher mit dem Finden von Daten aus mehreren Tabellen.

8.10 Aufgaben

Wiederholungsfragen

1) Über welches Schlüsselwort benennt man eine Spalte um?
2) Wozu dienen die Funktionen TRIM, SUBSTRING, UPPER, POSITION und EXTRACT?
3) Was liefert der Ausdruck NULLIF(10.99, 10.99) zurück?
4) Was liefert der Ausdruck POSITION('ist' IN 'Dies ist ein Beispieltext') zurück?
5) Was liefern die Funktionen CURRENT_DATE und CURRENT_TIME zurück?
6) Welches Ergebnis liefert der Ausdruck »10 NOT BETWEEN 20 AND 30«?
7) Welches Ergebnis liefert der Ausdruck »NOT(10 >= 20 AND 10 >= 30)«?

8) Welches Ergebnis liefert der Ausdruck »'Hamburg' IN ('München', 'Frankfurt', 'Berlin')« ?
9) Welches Ergebnis liefert der Ausdruck »'Hamburg' = 'München' OR 'Hamburg' ='Frankfurt' OR 'Hamburg' = 'Berlin')« ?
10) Welches Ergebnis liefert der Ausdruck »10 <> 10«?
11) Wozu dienen die Schlüsselwörter ROLLUP und CUBE?
12) Wozu dienen die Funktionen AVG, SUM, COUNT, MAX und MIN?

Übungen

1) Geben Sie alle Vorstellungen aus, die von Mai bis Juli 2002 stattfinden!
2) Geben Sie die Namen aller Kunden aus, deren Nachname an der zweiten Stelle den Buchstaben »u« enthält!
3) Berechnen Sie die Anzahl aller auf Lager liegenden Werbeartikel und geben Sie der berechneten Spalte den Spaltennamen »Anzahl auf Lager liegende Werbeartikel«!
4) Für welche Bestellung wurde der Mitarbeiter nicht eingetragen, der diese entgegengenommen hat?
5) Welche Vorstellungen finden in der Nordseehalle oder in der Kongresshalle statt?
6) Wie viele verschiedene Vorstellungen gibt es?
7) Geben Sie die Anzahl der Vorstellungen aus, die pro Veranstaltung stattfinden!
8) Welcher Kunde hat wie viele Kinder (absteigend sortiert nach der Anzahl der Kinder)?
9) Herr Kowalski hatte mehrere Fragen, die Frau Kart ihm aus der Datenbank beantworten sollte:
   Was kostet ein Werbartikel im Durchschnitt?
   Welcher Werbeartikel wurde am häufigsten verkauft?
   Welche Vorstellungen finden im Mai statt?
   Wie viele Vorstellungen finden im Jahr 2002 jeweils in den einzelnen Monaten statt?
   Welcher Kunde hat die höchste Anzahl an Artikeln gekauft?
   Welcher Artikel wurde am wenigsten bestellt?
   Helfen Sie Frau Kart!
10) Wie viele Vorstellungen finden im jeweiligen Jahr jeweils in den einzelnen Monaten und insgesamt statt?
11) Wie viele Kinder der Kunden sind Mädchen und wie viele Kinder sind Jungen?
12) Geben Sie die Kundennummer der Kunden aus, die Kinder haben, deren Vorname mit dem Buchstaben »K« beginnt (Kundennummern sollen nur einmal ausgegeben werden)!
13) Wie lauten die Namen der Kinder, die im Mai oder Juni Geburtstag haben?
14) Hat der Kunde mit der Kundennummer 1 sowohl ein Mädchen als auch einen Jungen?
15) Was kostet ein Sitzplatz im Parkett im Durchschnitt, im Maximum und im Minimum?

16) Geben Sie die Gesamtmenge an Artikeln an, die durch die Bestellung mit der Nummer 1 bestellt wurden?

17) Geben Sie die Bestellnummern, sowie die Anzahl der Bestellposten pro Bestellung aus?

18) Welche Vorstellung hat noch keine reservierten oder belegten Sitzplätze?

19) Wie viele Sitzplätze sind für die Vorstellung mit der Nummer 12 noch frei?

20) Wie viele Sitzplätze sind zur jeweiligen Vorstellung belegt, frei oder reserviert (sortiert nach der Vorstellungsnummer)?

21) Wie viele Sitzplätze sind zur jeweiligen Vorstellung belegt, frei oder reserviert und wie viele Sitzplätze gibt es ingesamt zu jeder Vorstellung?

9 Abfragen auf mehrere Tabellen

In Kapitel 9 sollen folgende Fragen geklärt werden:

- Wie verbindet man Tabellen wieder, die vorher durch Modellierung in mehrere Tabellen aufgespalten wurden?
- Wie kann man in einer Abfrage auf Spalten mehrerer Tabellen zugreifen?
- Welche Möglichkeiten gibt es, Tabellen wieder miteinander zu verbinden?

9.1 Motivation

Frau Kart von der Firma »KartoFinale« hat inzwischen die Abfragen, um die der Geschäftsführer Herr Kowalski sie gebeten hatte, erstellt und die Ergebnisse ausgedruckt. Voller Stolz zeigt sie Herrn Kowalski die Ergebnisse, der sichtlich beeindruckt ist, welche Informationen und teilweise auch Analysen er über seine Geschäftsdaten erhalten kann.

Also beauftragt er Frau Kart gleich mit weiteren Fragen, die er beantwortet haben möchte:

- Erstellen Sie mir eine Liste aller Kundennamen und deren Kinder.
- Erstellen Sie mir eine Liste aller Kunden und deren Bestellungen. Es sollen alle Kunden aufgeführt werden, auch die, die bisher noch nichts bestellt haben.
- Erstellen Sie mir eine Liste aller Vorstellungen mit Bezeichnung und Adresse der Spielstätte, Bezeichnung der Veranstaltung und Vorstellungstermin.
- Erstellen Sie eine Liste aller Bestellungen und deren Bestellposten.

Frau Kart, optimistisch durch den ersten Abfrage-Erfolg, setzt sich sofort daran, die gewünschten Listen zu erstellen. Doch gleich bei der ersten Abfrage hat sie Probleme. Die Namen der Kunden und die Namen der Kinder befinden sich in unterschiedlichen Tabellen. Wie kann sie nun aber eine Abfrage erstellen, die sich auf beide Tabellen bezieht?

Also ruft sie Herrn Fleissig von der Unternehmensberatung an und schildert ihr Problem. Herr Fleissig erklärt ihr, dass man Tabellen, die über den Fremd- und Primärschlüssel miteinander in Beziehung stehen wieder zu einer Tabelle verbinden kann. Er erläutert ihr dies anhand der ersten Abfrage: »Der Primärschlüssel des Kunden befindet sich als Fremdschlüssel in der Kind-Tabelle. Um diese beiden Tabellen zu einer Tabelle zu verbinden, erstellt man eine Abfrage, die den Primär- und Fremdschlüssel, also die Kundennummer, auf Gleichheit prüft«

Nachdem Herr Fleissig mit ihr die entsprechende SELECT-Anweisung zum Abfragen mehrerer Tabellen durchgegangen ist, kann Frau Kart schließlich die gewünschten Listen selbst erstellen.

9.2 Grundlagen

Wir haben in den Kapiteln zur Datenmodellierung gesehen, dass zur Vermeidung mehrfach gespeicherter Daten, das Relationenmodell die Daten in mehrere Tabellen aufspaltet. Die Beziehungen zwischen den Tabellen werden dabei über Fremd- und Primärschlüssel abgebildet.

Um nun durch eine Abfrage Informationen aus mehreren Tabellen zu erhalten, müssen die in Beziehung stehenden Tabellen wieder miteinander verbunden werden. Das Verbinden (»Join«) von Tabellen zu einer großen Tabelle ist eine wichtige Aufgabe eines RDBMS, da in der Regel für die meisten Abfragen Informationen aus verschiedenen Tabellen benötigt werden. Das Verbinden von Tabellen ist dabei nicht nur beschränkt auf das Wiederzusammenführen von Tabellen über Primär- und Fremdschlüsselspalten, auch wenn dieses sicherlich die Hauptaufgabe von »Joins« darstellt. So sind durchaus Abfragen sinnvoll, die sich nicht auf Fremd- und Primärschlüssel beziehen. Um herauszufinden, welche Kunden im gleichen Ort wohnen, in der auch eine Spielstätte vorhanden ist, würde man die Postleitzahl des Kunden mit der Postleitzahl der Spielstätte vergleichen, auch wenn es sich hierbei um keine Fremd-/Primärschlüsselbeziehung handelt.

Einfach ausgedrückt verbindet man Tabellen also dadurch, dass Fremd- und Primärschlüssel auf Gleichheit bzw. Spalten auf Gleichheit überprüft werden, die den gleichen Datentyp haben und sich natürlich semantisch entsprechen. So macht es natürlich keinen Sinn, zwei Spalten wie »Ortsname« und »Artikelbezeichnung« auf Gleichheit zu überprüfen, da es hier keine inhaltlichen Übereinstimmungen gibt, auch wenn der Datentyp identisch sein mag.

Dadurch, dass das Relationenmodell Verbindungen von beliebigen Tabellen zulässt, sind beliebige Abfragen denkbar, an die beim Entwurf der Datenbankstruktur vielleicht noch gar nicht gedacht wurde. Diese Flexibilität ist ein wesentlicher Vorteil des Relationenmodells.

Doch kommen wir nun zu unserer SELECT-Anweisung, mit der wir Tabellen abfragen können. Wir wollen zunächst schrittweise betrachten, wie ein »Join« funktioniert. Nehmen wir dazu wieder unser Beispiel mit den Tabellen Kunde und Kind. Zunächst müssen wir die Spalten angeben, die unsere SELECT-Anweisung ausgeben soll. Wir möchten den Namen des Kunden und die Vornamen der Kinder ausgeben. Die Informationen zu diesen Spalten finden wir in den Tabellen Kunde und Kind. Der erste Teil der SELECT-Abfrage könnte mit unserem bisherigen Wissen folgendermaßen aussehen:

```
SELECT Name, Vorname, Vorname  -- mehrdeutige Spaltennamen
FROM   Kunde, Kind
```

Doch hier haben wir schon unser erstes Problem: Die Spalte »Vorname« existiert sowohl in der Tabelle »Kunde« als auch in der Tabelle »Kind«, der Spaltenname »Vorname« ist also mehrdeutig. Wie können wir dieses Problem nun lösen? Eine Möglichkeit wäre natürlich, den Spaltennamen »Vorname« in der Tabelle »Kind« über ALTER TABLE umzubenennen, so dass wir in allen Tabellen eindeutige Spaltennamen hätten. Dies ist jedoch nicht besonders praktikabel. SQL kennt deshalb so genannte qualifizierte Namen. Ein Spaltenname ist, wie wir gerade gesehen haben, zwischen zwei oder mehreren Tabellen nicht immer eindeutig. Wird jedoch der Tabellenname der Spalte vorangestellt, so ist eindeutig festgelegt, welche Spalte gemeint ist. Genau das ist mit qualifizierten Namen gemeint. Man stellt einfach den Tabellennamen vor den Spaltennamen und trennt die beiden Bezeichner durch einen Punkt. Um also eindeutig auf den Vornamen in der Tabelle Kind zu verweisen, lautet der qualifizierte Bezeichner

```
Kind.Vorname
```

Um dagegen eindeutig auf den Vornamen des Kunden zu verweisen, lautet der qualifizierte Bezeichner:

```
Kunde.Vorname
```

In Kapitel 6, Abschnitt 2, haben wir gesehen, dass ein RDBMS eine Datenbank weiter strukturiert, als in Tabellen und Spalten. Den Tabellen übergeordnet ist das Schema und über dem Schema steht die Kategorie. Eine Tabelle kann also innerhalb von zwei Schemata auch den gleichen Namen bekommen. Allgemein sieht die Vergabe qualifizierter Bezeichner deshalb folgendermaßen aus:

```
Kategorie.Schema.Tabelle.Attribut
```

In diesem Buch wird jedoch ausschließlich mit einem einzigen Schema gearbeitet, daher reicht die Qualifizierung des Spaltennamens über Tabelle und Spalte.

Doch kommen wir zu unserem Ausgangspunkt zurück. Die gewünschte Abfrage mit qualifizierten Spaltennamen sieht jetzt folgendermaßen aus:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde, Kind
```

Wie sieht nun das Ergebnis dieser Abfrage aus? Schließlich haben wir noch keinen Vergleich von Primär- und Fremdschlüssel (Kundennummer) angegeben. Die Mengenlehre, auf der das Relationenmodell basiert, arbeitet mit dem kartesischen Produkt, d.h. jedes Element der einen Tabelle wird mit jedem Element der anderen Tabelle kombiniert. Das kartesische Produkt selbst liefert vorwiegend unsinnige Daten, da jeder Kunde mit jedem Kind kombiniert wird. Danach wäre sozusagen, jeder Kunde Elternteil jedes Kindes. Das Ergebnis der Abfrage würde also folgendermaßen aussehen:

```
Kunde.Name  Kunde.Vorname  Kind.Vorname
----------  -------------  ------------
Bolte       Bertram        Karl
Bolte       Bertram        Katja
Bolte       Bertram        Ursula
Bolte       Bertram        Enzo
Bolte       Bertram        Ursula
Muster      Hans           Karl
Muster      Hans           Katja
Muster      Hans           Ursula
Muster      Hans           Enzo
Muster      Hans           Ursula
Wiegerich   Frieda         Karl
Wiegerich   Frieda         Katja
Wiegerich   Frieda         Ursula
Wiegerich   Frieda         Enzo
Wiegerich   Frieda         Ursula
Carlson     Peter          Karl
Carlson     Peter          Katja
Carlson     Peter          Ursula
Carlson     Peter          Enzo
Carlson     Peter          Ursula
```

Um nun jedem Kunden die dazugehörigen Kinder zuzuordnen, muss das kartesische Produkt eingeschränkt werden, indem nur die Sätze ausgegeben werden, bei denen die Kundennummer in beiden Tabellen identisch ist. Betrachten wir hierzu die Abbildung 9.1 auf der nächsten Seite. Zunächst wird das kartesische Produkt der Tabellen Kunde und Kind gebildet. Danach wird überprüft, wo die Kundennummer der Tabelle Kunde der Kundennummer der Tabelle Kind entspricht. Denn das sind ja genau die Kinder, die zu diesem Kunden gehören. Als Ergebnis erhält man alle Kundennamen und deren zugehörige Kinder, indem man alle Datensätze über eine WHERE-Klausel herausfiltert, bei denen die Kundennummer nicht übereinstimmt. Die endgültige SELECT-Anweisung sieht also wie folgt aus:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde, Kind
WHERE  Kunde.Kundennummer = Kind.Kundennummer

Kunde.Name  Kunde.Vorname  Kind.Vorname
----------  -------------  ------------
Bolte       Bertram        Karl
Bolte       Bertram        Katja
Bolte       Bertram        Ursula
Wiegerich   Frieda         Enzo
Wiegerich   Frieda         Ursula
```

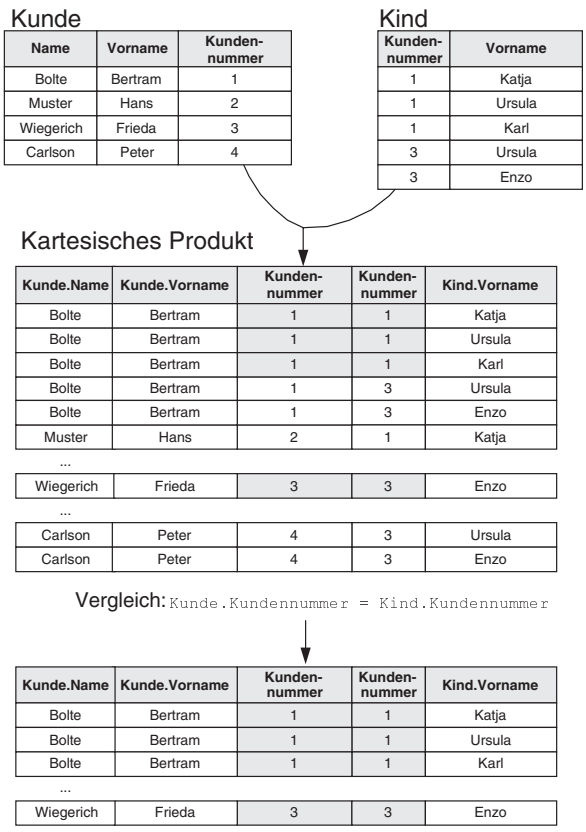


Abbildung 9.1: »Join«von »Kunde«und »Kind«

Der Kunde Bertram Bolte hat also drei Kinder und die Kundin Frieda Wiegerich zwei. Auch in der WHERE-Klausel werden qualifizierte Namen verwendet, da die Spalte Kundennummer in beiden Tabellen vorkommt.

Wir haben soeben den häufigsten Typ von »Joins« kennen gelernt, den »Natural Join«. SQL:1999 kennt insgesamt vier Typen von »Joins«:

- CROSS JOIN
- INNER JOIN
- NATURAL JOIN
- OUTER JOIN

Wir wollen uns im Folgenden die einzelnen »Join«-Typen ansehen.

9.3 CROSS JOIN

Der CROSS JOIN entspricht dem kartesischen Produkt. Wir haben ihn bereits kennen gelernt, als wir die Tabellen Kunde und Kind hinter der FROM-Klausel aufgeführt haben ohne die WHERE-Klausel zu verwenden. Wie bereits erwähnt, liefert er in der Regel unsinnige Daten, weshalb er in der Praxis kaum Verwendung findet. Die klassisch komma-getrennte Version des CROSS JOIN wurde oben bereits verwendet. Seit SQL-1992 gibt es jedoch auch die beiden Schlüsselwörter CROSS JOIN, mit denen für zwei Tabellen das kartesische Produkt gebildet werden kann. Die dazugehörige SELECT-Anweisung für unsere beiden Tabellen sieht dabei folgendermaßen aus:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde CROSS JOIN Kind
```

Die Anweisung entspricht der SELECT-Anweisung aus Abschnitt 9.2 ohne WHERE-Klausel. Um jedoch besser erkennen zu können, ob es sich bei einem Vergleich um einen »Join« oder ein Auswählen von Sätzen handelt, hat man 1992 diese Schlüsselwörter im Standard eingeführt.

9.4 INNER JOIN

Ein INNER JOIN berücksichtigt niemals NULL, er verbindet also nur Spalten, die Werte enthalten. Ein typisches Beispiel eines INNER JOINS ist unsere Kunde-Kind-Beziehung. Es werden alle Kunden aufgeführt und deren Kinder. Dabei werden die kinderlosen Kunden weggelassen. In Abschnitt 9.2 haben wir die klassische kommagetrennte Version des INNER JOIN kennen gelernt. Seit SQL-1992 gibt es wie beim CROSS JOIN auch zwei neue Schlüsselwörter, nämlich INNER JOIN. Um die obige Abfrage mit den Schlüsselwörtern INNER JOIN darzustellen, schreibt man:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde INNER JOIN Kind
       ON Kunde.Kundennummer = Kind.Kundennummer
```

Als Ergebnis erhält man die beiden Kunden, Bertram Bolte und Frieda Wiegerich, die Kinder haben. Wenn man zwei Spalten auf Gleichheit überprüft, erlaubt SQL eine einfachere Syntax über das Schlüsselwort USING. Folgende SQL-Anweisung liefert das gleiche Ergebnis:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde INNER JOIN Kind
       USING (Kundennummer, Kundennummer)
```

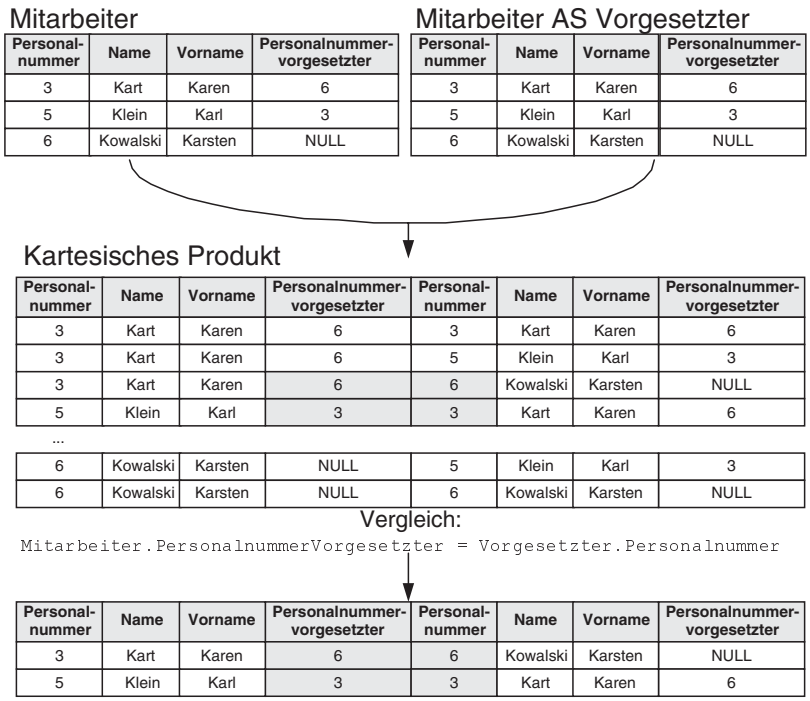
Ist der Name der Spalte in beiden Tabellen identisch, so braucht die Spalte nur einmal aufgeführt zu werden. Um also alle Kunden, die Kinder haben, mit den Namen ihrer Kinder auszugeben, gibt es verschiedene Möglichkeiten. Folgende SELECT-Anweisungen sind identisch und liefern das gleiche Ergebnis:

```
-- klassisch komma-getrennter Join
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde, Kind
WHERE  Kunde.Kundennummer = Kind.Kundennummer

-- Bedingungs-Join
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde INNER JOIN Kind
       ON Kunde.Kundennummer = Kind.Kundennummer

-- Spaltennamen-Join
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde INNER JOIN Kind
       USING (Kundennummer)
```

Wie werden »Joins« durchgeführt, die sich auf die gleiche Tabelle beziehen? Wenn wir uns noch einmal an die Tabelle Mitarbeiter zurückerinnern, so wissen wir, dass hier eine rekursive Beziehung zwischen Mitarbeiter und Vorgesetztem abgebildet wurde. Wir haben hier also den Sonderfall, dass der Fremdschlüssel »PersonalnummerVorgesetzter« auf die gleiche Tabelle verweist, nämlich den Primärschlüssel »Personalnummer«. Verknüpfungen einer Tabelle mit sich selbst bezeichnet man als »Self Join«. Betrachten wir folgendes Beispiel: Es sollen die Namen aller Mitarbeiter, sowie die Namen der dazugehörigen Vorgesetzten ausgegeben werden. Hier tritt das gleiche Problem wie mit mehrdeutigen Spaltennamen auf: Soll eine Tabelle mit sich selbst verknüpft werden, so muss sie zweimal in der FROM-Klausel aufgeführt werden. SQL bietet die Möglichkeit, den Namen von Tabellen über das Schlüsselwort AS umzubenennen. Das Ganze funktioniert also genauso wie das Umbenennen von Spalten. Damit sieht die Abfrage folgendermaßen aus:

```
SELECT Mitarbeiter.Name, Vorgesetzter.Name
FROM   Mitarbeiter, Mitarbeiter AS Vorgesetzter
WHERE  Mitarbeiter.PersonalnummerVorgesetzter =
       Vorgesetzter.Personalnummer

Mitarbeiter.Name  Vorgesetzter.Name
----------------  -----------------
Kart              Kowalski
Klein             Kart
```

Gibt man in der FROM-Klausel zweimal die gleiche Tabelle an, so betrachtet SQL die beiden Tabellen als zwei unterschiedliche Tabellen, die z. B. miteinander verbunden werden können.



Personal-nummer	Name	Vorname	Personalnummer-vorgesetzter
3	Kart	Karen	6
5	Klein	Karl	3
6	Kowalski	Karsten	NULL

Personal-nummer	Name	Vorname	Personalnummer-vorgesetzter	Personal-nummer	Name	Vorname	Personalnummer-vorgesetzter
3	Kart	Karen	6	3	Kart	Karen	6
3	Kart	Karen	6	5	Klein	Karl	3
3	Kart	Karen	6	6	Kowalski	Karsten	NULL
5	Klein	Karl	3	3	Kart	Karen	6
...							
6	Kowalski	Karsten	NULL	5	Klein	Karl	3
6	Kowalski	Karsten	NULL	6	Kowalski	Karsten	NULL

Personal-nummer	Name	Vorname	Personalnummer-vorgesetzter	Personal-nummer	Name	Vorname	Personalnummer-vorgesetzter
3	Kart	Karen	6	6	Kowalski	Karsten	NULL
5	Klein	Karl	3	3	Kart	Karen	6

Abbildung 9.2: »Self Join« der Tabelle »Mitarbeiter«

Entsprechend könnte man z.B. auch herausfinden, welche Mitarbeiter die gleiche Postleitzahl haben. In diesem Fall würde man eine Verknüpfung nicht über Fremd- und Primärschlüssel vornehmen, sondern über semantisch vergleichbare Spalten:

```
SELECT M1.Name, M2.Name
FROM   Mitarbeiter AS M1 INNER JOIN Mitarbeiter AS M2
       ON M1.Plz = M2.Plz

M1.Name   M2.Name
--------  -------
Kart      Kart
Klein     Kart
Kart      Klein
Klein     Klein
Kowalski  Kowalski
```

Doch unser Ergebnis ist noch nicht besonders befriedigend. Zwar werden Frau Kart und Herr Klein aufgeführt, die beide im Ort mit der Postleitzahl 22222 wohnen. Es werden allerdings sowohl einzelne Personen, als auch Personenkombinationen, nur in umgekehrter Reihenfolge, doppelt ausgegeben. Betrachten wir noch einmal die Vorgehensweise eines RDBMS bei der Auswertung einer Abfrage. Zunächst wird das kartesisches Produkt gebildet. In diesem Fall wird jeder Satz der Tabelle Mitarbeiter (M1) mit

jedem Satz der Tabelle Mitarbeiter (M2) kombiniert. Da z.B. Herr Kowalski natürlich in beiden Tabellen vorhanden ist, gibt es auch eine Kombination, in der er zweimal auftritt. Um gleiche Sätze aus M1 und M2 auszufiltern, fügen wir eine Bedingung hinzu, die überprüft, ob der Primärschlüssel zwischen M1 und M2 unterschiedlich ist. Doch wir haben noch eine Unstimmigkeit, die Kombination Klein und Kart wird zweimal aufgeführt. Um doppelte Kombinationen auszufiltern, vergleicht man einfach, ob der Primärschlüssel von M1 größer als der Primärschlüssel von M2 ist. Die SELECT-Anweisung sieht also folgendermaßen aus:

```
SELECT M1.Name, M2.Name
FROM   Mitarbeiter AS M1 INNER JOIN Mitarbeiter AS M2
       ON M1.Plz = M2.Plz
WHERE  M1.Personalnummer <> M2.Personalnummer
AND    M1.Personalnummer > M2.Personalnummer

M1.Name  M2.Name
-------  -------
Klein    Kart
```

Damit können wir auch die erste Bedingung auf Ungleichheit der Personalnummern weglassen, da die zweite Bedingung diese Einschränkung schon mitberücksichtigt.

9.5 NATURAL JOIN

Der NATURAL JOIN ist ein Spezialfall des INNER JOIN. Ein INNER JOIN muss nicht unbedingt zwei Spalten auf Gleichheit prüfen, als Bedingung sind hier alle Vergleichsoperatoren, also >, <, <>, <= und >= möglich. »Joins«, die einen anderen Vergleichsoperator als das Gleichheitszeichen verwenden, sind jedoch selten. Daher wird hier auch nicht näher darauf eingegangen.

Der NATURAL JOIN ist ein INNER JOIN, der zwei Spalten ausschließlich auf Gleichheit prüft. Damit waren alle bisherigen Beispiele in diesem Kapitel also NATURAL JOINs. Seit SQL-1992 gibt es aber auch für den NATURAL JOIN spezielle Schlüsselwörter. Um also alle Kunden und deren Kinder auszugeben, kann man auch schreiben:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde NATURAL JOIN Kind
```

In diesem Fall wird die erste Spalte verwendet, die in beiden Tabellen den gleichen Namen hat und auf Gleichheit überprüft.

9.6 OUTER JOIN

Im Gegensatz zum INNER JOIN berücksichtigt der OUTER JOIN Spalten, die NULL enthalten. Betrachten wir hierzu wieder unser Beispiel mit den Tabellen Kunde und Kind. Bei einem INNER JOIN werden nur die Kunden aufgeführt, die Kinder haben.

Die Namen der Kunden, die keine Kinder haben, werden nicht aufgeführt. Dies ist jedoch sinnvoll, wenn man z.B. eine komplette Kundenliste erstellen möchte, in der u.a. auch die Namen der Kinder aufgeführt sind. Seit SQL-1992 gibt es deshalb als Gegenstück zu INNER JOIN die beiden Schlüsselwörter OUTER JOIN.

Ein OUTER JOIN gibt es in drei verschiedenen Varianten, je nachdem von welcher Tabelle aus man die NULL-Werte betrachtet:

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

Sehen wir uns dazu die SELECT-Anweisung zur Ausgabe der Kunden mit den dazugehörigen Kindern an:

```
SELECT Kunde.Name, Kunde.Vorname, Kind.Vorname
FROM   Kunde LEFT OUTER JOIN Kind
       ON Kunde.Kundennummer = Kind.Kundennummer

Kunde.Name  Kunde.Vorname  Kind.Vorname
----------  -------------  ------------
Bolte       Bertram        Karl
Bolte       Bertram        Katja
Bolte       Bertram        Ursula
Muster      Hans           NULL
Wiegerich   Frieda         Enzo
Wiegerich   Frieda         Ursula
Carlson     Peter          NULL
```

Es werden die Namen aller Kunden ausgegeben, auch die der kinderlosen Kunden, also Hans Muster und Peter Carlson. Die Spalten der Tabelle, in der ihr Primärschlüssel nicht als Fremdschlüssel vorhanden ist, werden als NULL dargestellt.

Über LEFT und RIGHT kann festgelegt werden, ob die Spalten der linken oder der rechten Tabelle ausgegeben werden sollen, wenn keine Beziehung zu der entsprechenden Tabelle besteht. Da ein Kind immer einem Kunden zugeordnet ist, diese Beziehung aus der Sicht des Kindes also niemals optional ist, würde RIGHT OUTER JOIN in diesem Fall einem NATURAL JOIN entsprechen.

Betrachten wir hierzu ein weiteres Beispiel. Wir wollen eine Liste aller Mitarbeiter und deren Abteilung ausgeben. Es sollen auch Mitarbeiter ohne zugehörige Abteilung ausgegeben werden. Über einen LEFT OUTER JOIN würde das dann folgendermaßen aussehen:

```
SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter LEFT OUTER JOIN Abteilung
       ON Mitarbeiter.Abteilungsbezeichnung =
          Abteilung.Abteilungsbezeichnung
```

```
Name      Vorname  Abteilung.Abteilungsbezeichnung
--------  -------  -------------------------------
Kart      Karen    Vertrieb
Klein     Karl     Vertrieb
Kowalski  Karsten  NULL
```

Wir sehen, dass der Geschäftsführer Herr Kowalski keiner Abteilung zugeordnet ist. Wollen wir nun umgekehrt wissen, welche Abteilungen es gibt und welche Mitarbeiter zu diesen Abteilungen gehören, so verwenden wir anstelle des LEFT ein RIGHT. Dadurch werden auch Abteilungen ausgegeben, denen kein Mitarbeiter zugeordnet ist.

```
SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter RIGHT OUTER JOIN Abteilung
       ON Mitarbeiter.Abteilungsbezeichnung =
          Abteilung.Abteilungsbezeichnung
```

```
Name      Vorname  Abteilung.Abteilungsbezeichnung
--------  -------  -------------------------------
NULL      NULL     Geschäftsführung
NULL      NULL     Rechnungswesen
Kart      Karen    Vertrieb
Klein     Karl     Vertrieb
```

Da Frau Kart und Herr Klein zu einer Abteilung gehören, sind sie wieder in der Ergebnistabelle zu finden. Herr Kowalski ist nicht aufgeführt, da er keiner Abteilung zugeordnet ist. Dafür sind aber alle Abteilungen, die es gibt, in der Ergebnistabelle zu sehen, auch wenn kein Mitarbeiter zugeordnet ist.

Möchte man nun eine Liste, in der alle Mitarbeiter und auch alle Abteilungen aufgeführt sind, unabhängig davon ob ein Mitarbeiter zu einer Abteilung gehört oder ob eine Abteilung Mitarbeiter hat, so verwendet man schließlich einen FULL OUTER JOIN:

```
SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter FULL OUTER JOIN Abteilung
       ON Mitarbeiter.Abteilungsbezeichnung =
          Abteilung.Abteilungsbezeichnung
```

```
Name      Vorname  Abteilung.Abteilungsbezeichnung
--------  -------  -------------------------------
Kowalski  Karsten  NULL
NULL      NULL     Geschäftsführung
NULL      NULL     Rechnungswesen
Kart      Karen    Vertrieb
Klein     Karl     Vertrieb
```

Jetzt enthält die Ergebnistabelle die Namen aller Mitarbeiter und aller Abteilungen.

Wie bei einem INNER JOIN auch, kann anstelle der ON-Bedingung das Schlüsselwort USING verwendet werden, sofern man auf Gleichheit überprüft. Folgende SELECT-Anweisungen sind also identisch:

```
SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter OUTER JOIN Abteilung
       ON Mitarbeiter.Abteilungsbezeichnung =
          Abteilung.Abteilungsbezeichnung

SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter OUTER JOIN Abteilung
       USING (Abteilungsbezeichnung)
```

9.7 »Joins« auf mehrere Tabellen

Um mehr als zwei Tabellen miteinander zu verbinden, gibt es zwei Möglichkeiten. Die erste Alternative ist der klassisch komma-getrennte Stil. In diesem Fall gibt man hinter der WHERE-Klausel alle Primär- und Fremdschlüsselbeziehungen an, über die die einzelnen Tabellen miteinander verknüpft sind und verbindet diese Beziehungen mit AND. Um z.B. die Namen und den Wohnort aller Kunden, sowie die Namen der dazugehörigen Kinder auszugeben, sieht die SELECT-Anweisung dann wie folgt aus:

```
SELECT Kunde.Name, Kunde.Vorname, Ort.Ort, Kind.Vorname
FROM   Kunde, Kind, Ort
WHERE  Kunde.Kundennummer = Kind.Kundennummer
AND    Kunde.Plz = Ort.Plz
```

Die Anzahl der zu verknüpfenden Tabellen bestimmt dabei die Anzahl der Bedingungen hinter WHERE, die jeweils durch AND verbunden werden.

Die zweite Möglichkeit, mehrere Tabellen untereinander zu verbinden, ist die Verwendung der Schlüsselwörter JOIN und ON. Hierbei verknüpft man immer das Ergebnis des ersten Joins mit dem zweiten Join usw. Das Ganze sieht dann folgendermaßen aus:

```
SELECT Kunde.Name, Kunde.Vorname, Ort.Ort, Kind.Vorname
FROM   Kunde INNER JOIN Kind
       ON Kunde.Kundennummer = Kind.Kundennummer
       INNER JOIN Ort
       ON Kunde.Plz = Ort.Plz
```

In diesem Fall wird zunächst das Ergebnis aus dem ersten »Join« zwischen Kunde und Kind gebildet. Danach wird dieses Ergebnis verwendet und über einen »Join« mit der Tabelle »Ort« verknüpft.

9.8 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie man mehrere Tabellen zu einer Tabelle verbindet. Auf diese Tabelle kann man dann wieder alle Ausdrücke und Prädikate anwenden, die wir im letzten Kapitel kennen gelernt haben.

Eine Möglichkeit, um zwei Tabellen miteinander zu verbinden, ist die Anwendung des klassischen komma-getrennten Stils. Dabei verwendet man die normalen Bedingungsausdrücke, um Fremd- und Primärschlüssel miteinander zu vergleichen. »Joins« sind dabei nicht nur auf Fremd- und Primärschlüssel beschränkt. Letztendlich kann jede Tabelle mit einer anderen Tabelle verbunden werden, sofern diese zwei semantisch gleiche Spalten besitzen. Auch eine Verbindung über das Gleichheitszeichen ist nicht zwingend notwendig. Ebenso können alle anderen Vergleichsoperatoren verwendet werden, auch wenn diese Fälle in der Praxis selten auftreten.

Neben dem komma-getrennten Stil gibt es spezielle Schlüsselwörter, um »Joins« zu kennzeichnen. Dabei unterscheidet man generell zwischen

- OUTER JOIN und
- INNER JOIN

Ein INNER JOIN berücksichtigt im Gegensatz zum OUTER JOIN unbekannte Werte über NULL nicht. Ein Spezialfall des INNER JOIN ist der NATURAL JOIN, dessen Hauptaufgabe in der Verknüpfung von Tabellen über Primär- und Fremdschlüssel besteht.

9.9 Aufgaben

Wiederholungsfragen

1) Worin besteht der Unterschied zwischen einem INNER JOIN und einem CROSS JOIN?
2) Worin besteht der Unterschied zwischen einem RIGHT OUTER JOIN und einem LEFT OUTER JOIN?
3) Folgende Anweisung verbindet die Tabellen Mitarbeiter und Abteilung:

   ```
   Mitarbeiter NATURAL JOIN Abteilung
   ```

   Geben Sie SQL-Ausdrücke an, die diesem entsprechen!
4) Wie können Sie folgende SQL-Anweisung durch Hinzufügen einer WHERE-Klausel so ändern, dass sie einem LEFT OUTER JOIN entspricht?

```
SELECT Name, Vorname, Abteilung.Abteilungsbezeichnung
FROM   Mitarbeiter FULL OUTER JOIN Abteilung
       ON Mitarbeiter.Abteilungsbezeichnung =
          Abteilung.Abteilungsbezeichnung
WHERE ...
```

5) Geben Sie ein Beispiel für einen »Self Join«!
6) Was ist bei einem »Self Join« bei der Angabe der Tabellennamen zu beachten?

Übungen

1) Geben Sie die Namen aller Spielstätten aus und den Ortsnamen, in dem sich die Spielstätte befindet!
2) In welchen Spielstätten finden Veranstaltungen statt, die von Mozart sind?
3) Geben Sie die Name der Kunden und die Anzahl der Bestellungen aus, die von diesem stammen!
4) Erstellen Sie eine Liste aller Bestellungen und deren Bestellposten!
5) Erstellen Sie eine Liste aller Vorstellungen mit Bezeichnung und Adresse der Spielstätte, Bezeichnung der Veranstaltung und Angabe der Vorstellungstermine!
6) Geben Sie die Bezeichnung und den Ortsnamen der Spielstätten aus, die die gleiche Postleitzahl besitzen?
7) Geben Sie die Namen aller Kunden aus, die im gleichen Ort wohnen!
8) Welche Sitzplätze sind zur Veranstaltung »Phil Collins LIVE« am 21.07.2002 noch frei?
9) Zu welchen Terminen gibt es Vorstellungen zu der Veranstaltung »Don Giovanni«?
10) Welche Kunden haben die Veranstaltung »Don Giovanni« zu welchem Termin gebucht und welche Sitzplätze haben sie erhalten?
11) In welcher Spielstätte finden keine Vorstellungen statt?
12) Zu welcher Veranstaltung gibt es zur Zeit keine Vorstellungen?
13) Geben Sie die Namen der Kunden aus, die bei einem Bestellvorgang mehr als drei Artikel bestellt haben!
14) Geben Sie die Namen der Kunden aus, deren Kinder das gleiche Geschlecht haben. Geben Sie dabei auch die Namen der Kinder aus!

10 Abfragen mit Unterabfragen

In Kapitel 10 sollen folgende Fragen geklärt werden:

- Wie kann man das Ergebnis einer Abfrage als Teil einer neuen Abfrage verwenden?
- Wie kann man in einer einzigen Anweisung ermitteln, welche Mitarbeiter über dem Durchschnitt verdienen?

10.1 Motivation

Inzwischen hat Frau Kart die neuen Abfragen fertiggestellt und zeigt sie ihrem Geschäftsführer, Herrn Kowalski. Der ist begeistert und angesichts der Tatsache, dass das Jahr bald zu Ende geht, möchte Herr Kowalski einen Bonus an alle Mitarbeiter ausgeben. Dabei sollen alle Mitarbeiter die über dem Durchschnitt verdienen, einen Bonus von 10% und Mitarbeiter unter dem Durchschnitt einen Bonus von 15% erhalten. Also beauftragt er Frau Kart, zwei Listen zu erstellen. Eine Liste soll alle Mitarbeitern enthalten, die überdurchschnittlich verdienen und die zweite Liste die Mitarbeiter, die unter dem Durchschnitt verdienen.

Zunächst ermittelt Frau Kart das Durchschnittsgehalt mit der Mengenfunktion AVG und schreibt dieses auf. Mit zwei einfachen SELECT-Anweisungen kann sie nun die beiden Listen erstellen. Ihr Ergebnis ist zwar korrekt, dennoch ist sie unzufrieden, da Herr Fleissig von der Unternehmensberatung ihr erzählt hat, dass die meisten Abfragen durch eine einzige SELECT-Anweisung ausgeführt werden können. Sie ruft also wieder Herrn Fleissig an, der ihr erzählt, dass man mehrere SELECT-Anweisungen »schachteln«kann.

10.2 Grundlagen

Betrachten wir zunächst unser Fallbeispiel: Die SELECT-Anweisung zum Ermitteln des durchschnittlichen Gehaltes lautet:

```
SELECT AVG( Gehalt ) AS 'Durchschnittsgehalt'
FROM   Mitarbeiter

Durchschnittsgehalt
-------------------
3016.666666
```

Den ermittelten Wert für das Durchschnittsgehalt können wir nun verwenden, um alle Mitarbeiter abzufragen, die unter dem Durchschnitt liegen. Entsprechend einfach sieht die dazugehörige SELECT-Anweisung aus:

```
SELECT Name, Vorname
FROM   Mitarbeiter
WHERE  Gehalt < 3016.66

Name   Vorname  Gehalt
-----  -------  -------
Kart   Karen    3000.00
Klein  Karl     2050.00
```

Um zum gewünschten Ergebnis zu gelangen, mussten also zwei Abfragen ausgeführt werden. Das Resultat der ersten Abfrage wurde in der zweiten Abfrage weiter verwendet.

Man kann nun beide SELECT-Anweisungen miteinander schachteln und das Ergebnis der ersten Abfrage direkt in die zweite Abfrage einfliessen lassen. Dazu kombiniert man beide Abfragen einfach wie folgt:

```
SELECT Name, Vorname, Gehalt
FROM   Mitarbeiter
WHERE  Gehalt < ( SELECT AVG( Gehalt )
                  FROM   Mitarbeiter )
```

Wir erhalten das gleiche Ergebnis durch eine einzige Abfrage, indem die Anweisung zur Ermittlung des Durchschnittsgehalts in der WHERE-Bedingung verwendet wird. Aufgrund dieser Vorgehensweise bezeichnet man die Abfragen, die in die WHERE-Bedingung einfließen als Unterabfrage (Subquery) oder innere Abfrage. Entsprechend wird die zweite Abfrage als äußere Abfrage bezeichnet. Unterabfragen sind also Abfragen innerhalb einer Abfrage.

In unserem Beispiel liefert die innere Abfrage einen einzigen Wert zurück, nämlich das Durchschnittsgehalt. Wir wissen aber aus den letzen Kapiteln, dass Abfragen nicht nur einzelne Werte, sondern Zeilen oder ganze Ergebnistabellen zurückliefern können. Je nachdem, was von der inneren Abfrage zurückgeliefert wird, unterscheidet man zwischen:

- Unterabfragen mit einem Rückgabewert (»Scalar Subquery«),
- Unterabfragen mit einer zurückgegebenen Zeile (»Row Subquery«),
- Unterabfragen mit mehreren zurückgegebenen Zeilen (»Table Subquery«).

Wir wollen uns im Folgenden die unterschiedlichen Typen von Unterabfragen und die dazugehörigen Abfragemöglichkeiten ansehen.

10.3 Unterabfragen mit einem Rückgabewert

Diese Art von Unterabfragen haben wir bereits im einleitenden Beispiel kennen gelernt. Voraussetzung dafür, dass die Abfrage korrekt ausgeführt wurde, war die Tatsache, dass die innere Abfrage genau einen Wert zurückliefert. Wären mehrere Werte von der Unterabfrage zurückgeliefert worden, so hätte das RDBMS die Abfrage nicht ausführen können. Betrachten wir hierzu ein Beispiel: Es sollen alle Mitarbeiter ausgegeben werden, die weniger verdienen als Frau Kart. Eine gültige SELECT-Anweisung könnte lauten:

```
SELECT Name, Vorname, Gehalt
FROM   Mitarbeiter
WHERE  Gehalt < ( SELECT Gehalt
                  FROM   Mitarbeiter
                  WHERE  Name LIKE 'K%' )
```

Hier würde das RDBMS eine Fehlermeldung ausgeben, da die innere Abfrage mehr als einen Wert zurückliefert. Sowohl das Gehalt von Frau Kart als auch das von Herrn Kowalski werden von der Unterabfrage zurückgegeben.

Bei skalaren Unterabfragen muss also gewährleistet sein, dass die innere Abfrage zum einen nur einen einzigen Wert zurückliefert und dass die Spalten, die man miteinander vergleicht, zueinander passen. Die innere Abfrage muss also einen Wert zurückliefern, der zur der Spalte passt, mit der verglichen wird (in unserem Fall verwenden innere und äußere Abfrage die Spalte »Gehalt«).

Damit die innere Abfrage nur einen Wert zurückliefert, sollte man in diesem Fall besser den Primärschlüssel, also die Personalnummer von Frau Kart, als Bedingung angeben.

Betrachten wir hierzu ein weiteres Beispiel. Um alle Mitarbeiter auszugeben, die im gleichen Ort wohnen, wie Hans Muster, sollte die Abfrage also folgendermaßen aussehen:

```
SELECT Name, Vorname, Plz
FROM   Mitarbeiter
WHERE  Plz = ( SELECT Plz
               FROM   Kunde
               WHERE  Kundennummer = 2 )
```

Diesmal wird die Postleitzahl von Hans Muster über den Primärschlüssel ermittelt, so dass nur ein Wert von der inneren Abfrage zurückgeliefert werden kann.

10.4 Unterabfragen mit einer zurückgegebenen Zeile

Unterabfragen mit einer einzigen zurückgegebenen Zeile werden von den meisten RDBMS zur Zeit noch nicht unterstützt. Generell gleichen solche Abfragen den Abfragen mit nur einem Wert, nur dass hier mehrere Spalten beim Vergleich angegeben werden können. Um also z.B. alle Mitarbeiter auszugeben, die sowohl im gleichen Ort als auch in der gleichen Strasse wie Hans Muster wohnen, lautet die Anweisung:

```
SELECT Name, Vorname, Plz
FROM   Mitarbeiter
WHERE  (Plz, Strasse) = ( SELECT Plz, Strasse
                          FROM   Kunde
                          WHERE  Kundennummer = 2 )
```

10.5 Unterabfragen mit mehreren zurückgegebenen Zeilen

Um in Unterabfragen mehrere Zeilen abzufragen, reichen die bisherigen Vergleichsoperatoren und Prädikate nicht aus. SQL kennt deshalb für diese Art von Abfragen spezielle Prädikate, die wir uns im Folgenden ansehen wollen.

10.5.1 IN

Das Prädikat IN entspricht eigentlich einer OR-Verknüpfung von Werten. Wenn eine Unterabfrage mehrere Werte einer Spalte zurückliefert, so kann mit IN überprüft werden, ob einer dieser Werte einer gewünschten Bedingung entspricht. Um z.B. alle Mitarbeiter auszugeben, die im gleichen Ort wie die Kunden Hans Muster und Bertram Bolte wohnen, lautet die SELECT-Anweisung:

```
SELECT Name, Vorname, Plz
FROM   Mitarbeiter
WHERE  Plz IN ( SELECT Plz
                FROM   Kunde
                WHERE  Kundennummer = 1 OR Kundennummer = 2)
```

Um demgegenüber alle Mitarbeiter zu ermitteln, die nicht im gleichen Ort wie Hans Muster und Bertram Bolte wohnen, verwendet man einfach die Schlüsselwörter NOT IN.

10.5.2 EXISTS

Über das Schlüsselwort EXISTS kann getestet werden, ob eine Unterabfrage Zeilen zurückliefert. Ist dies der Fall, so liefert die Unterabfrage TRUE zurück, andernfalls

FALSE. Um z. B. alle Abteilungen auszugeben, denen mindestens ein Mitarbeiter zugeordnet ist, lautet die SELECT-Anweisung:

```
SELECT Abteilungsbezeichnung
FROM   Abteilung
WHERE EXISTS ( SELECT *
               FROM Mitarbeiter
               WHERE Mitarbeiter.Abteilungsbezeichnung =
                     Abteilung.Abteilungsbezeichnung)

Abteilungsbezeichnung
--
Vertrieb
```

Wie bei einem »Join« auch, müssen Fremd- und Primärschlüssel miteinander verbunden werden. In diesem Fall stehen innere und äußere Abfrage in einer Wechselbeziehung zueinander, da die innere auf die äußere Abfrage verweist. Man spricht dann von »korrelierenden Unterabfragen«. Im Gegensatz zu den bisherigen Unterabfragen wertet ein RDBMS korrelierende Unterabfragen anders aus. Bei einer nicht-korrelierenden Abfrage kann das RDBMS zunächst die innere Abfrage ausführen und erhält dann ein Ergebnis. Dieses Ergebnis kann es dann verwenden, um die äußere Abfrage auszuführen. Bei korrelierenden Unterabfragen ist dies nicht möglich, da die innere Abfrage immer von der gerade bearbeiteten Zeile der äußeren Abfrage abhängt. Für jede Zeile der äußeren Abfrage wird also auch die innere Abfrage ausgewertet. Betrachten wir dazu folgende Abbildung:

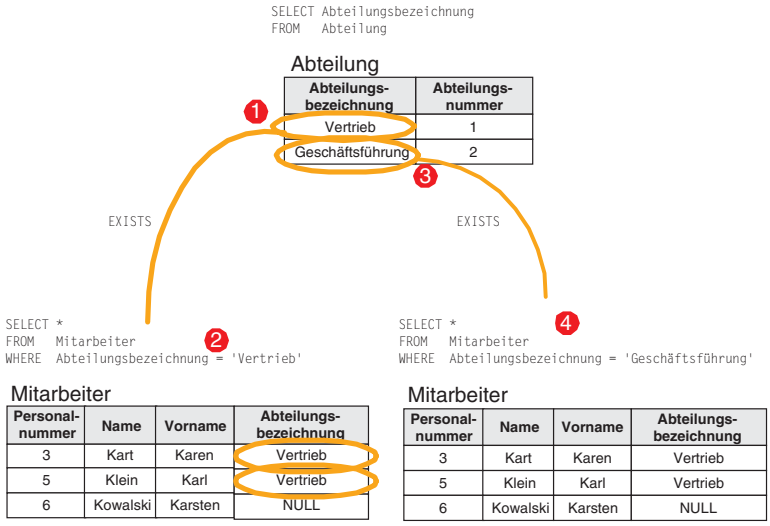


Abbildung 10.1: Korrelierende Unterabfrage mit EXISTS

Zunächst wird über die äußere Abfrage die erste Abteilungsbezeichnung ermittelt, nämlich Vertrieb (1). Dieser Wert wird nun in der inneren Abfrage verwendet, um herauszubekommen, welche Mitarbeiter im Vertrieb arbeiten (2). Da es in diesem Fall zwei Mitarbeiter sind, erfüllt die Abteilungsbezeichnung »Vertrieb« die Bedingung und gehört zur Ergebnistabelle. Nun ermittelt die äußere Abfrage die nächste Abteilungsbezeichnung, nämlich »Geschäftsführung« (3). Die innere Abfrage stellt fest, dass es keinen Mitarbeiter gibt, der zu dieser Abteilung gehört (4). Entsprechend gibt sie FALSE zurück.

Wie beim Mengenoperator IN kann auch EXISTS über das Schlüsselwort NOT negiert werden.

10.5.3 ANY / SOME / ALL

Mit ANY oder SOME kann man überprüfen, ob ein Wert größer als ein beliebiger Wert aus einer Menge ist. Hiermit sind Abfragen möglich, wie z.B. »Geben Sie das Gehalt aller Mitarbeiter an, die mehr als ein anderer Mitarbeiter verdienen«. Die dazugehörige SELECT-Anweisung lautet:

```
SELECT Name, Vorname, Gehalt
FROM   Mitarbeiter
WHERE  Gehalt > ANY (SELECT Gehalt FROM Mitarbeiter )

Name      Vorname  Gehalt
--------  -------  -------
Kart      Karen    3000.00
Kowalski  Karsten  4000.00
```

Frau Kart verdient mehr als Herr Klein und Herr Kowalski verdient sowieso mehr als jeder andere Mitarbeiter. Herr Klein ist nicht aufgeführt, da es keinen anderen Mitarbeiter gibt, der weniger als er verdient.

Entsprechend kann man über das Schlüsselwort ALL ermitteln, welcher Mitarbeiter mehr als jeder andere Mitarbeiter verdient. Die SELECT-Anweisung dazu sieht dann wie folgt aus:

```
SELECT Name, Vorname, Gehalt
FROM   Mitarbeiter
WHERE  Gehalt >= ALL (SELECT Gehalt FROM Mitarbeiter )

Name      Vorname  Gehalt
--------  -------  -------
Kowalski  Karsten  4000.00
```

In diesem Fall wird nur Herr Kowalski ausgegeben, da es keinen Mitarbeiter gibt, der mehr als er verdient.

Sie haben sicherlich bemerkt, dass man die beiden obigen Abfragen auch mit den Mengenfunktionen MIN und MAX als Unterabfrage hätte realisieren können. Letztendlich gibt es in SQL häufig mehrere Möglichkeiten, Abfragen zu formulieren.

Das gilt für viele Abfragen. Betrachten wir hierzu ein weiteres Beispiel: Um z. B. alle Mitarbeiter auszugeben, die in einem Ort wohnen, in dem auch ein beliebiger Kunde wohnt, führen folgende SELECT-Anweisungen zum gleichen Ergebnis:

```
SELECT Name, Vorname
FROM   Mitarbeiter
WHERE  Plz = ANY (SELECT Plz FROM Kunde)

SELECT Mitarbeiter.Name, Mitarbeiter.Vorname
FROM   Mitarbeiter, Kunde
WHERE  Mitarbeiter.Plz = Kunde.Plz
```

Generell gilt, dass die meisten Abfragen, die man als »Join« formulieren kann, auch als »Subquery« gestellt werden können. In diesem Fall bleibt es Geschmackssache, welche Formulierung man als »besser« empfindet. Eine Ausnahme, die man nicht als »Join« formulieren kann, ist das Verwenden von Mengenfunktionen in der inneren Abfrage, wie wir es gleich am Anfang kennen gelernt haben.

10.6 Zusammenfassung

In diesem Kapitel haben wir Unterabfragen kennen gelernt. Gerade in Zusammenhang mit der Verwendung von Mengenfunktionen sind Unterabfragen sehr nützlich. Unterabfragen sind Abfragen in einer Abfrage. Man kann die Ergebniswerte einer Abfrage als Vergleichswerte in einer anderen Abfrage verwenden. Man unterscheidet Unterabfragen danach, ob sie einen einzigen Wert, eine einzige Zeile oder mehrere Zeilen zurückliefert.

Liefert eine Unterabfrage mehrere Zeilen zurück, so können die Prädikate IN, EXISTS, ANY/SOME oder ALL verwendet werden. IN dient zur Überprüfung einer zurückgelieferten Menge von Werten, so wie wir es bereits bei einfachen Abfragen gesehen hatten. Das Prädikat EXISTS gibt TRUE oder FALSE aus, je nachdem ob die Unterabfrage Sätze zurückliefert oder nicht. ANY bzw. SOME testet, ob aus einer Menge von Werten wenigstens ein Wert einer Bedingung entspricht. ALL prüft, ob alle Werte einer Menge einer definierten Bedingung entsprechen.

10.7 Aufgaben

Wiederholungsfragen

1) Was ist mit innerer Abfrage und was mit äußerer Abfrage gemeint?
2) Was ist eine korrelierende Unterabfrage?

3) Wie wird eine nicht-korrelierende Unterabfrage vom RDBMS ausgewertet? Erklären Sie dies anhand eines Beispiels.
4) Wie wird eine korrelierende Unterabfrage vom RDBMS ausgewertet? Erklären Sie dies anhand eines Beispiels.
5) Worauf muss man achten, wenn man in der äußeren Abfrage ausschließlich Vergleichsoperatoren verwendet?
6) Wozu dienen die Prädikate NOT IN, NOT EXISTS, ALL, SOME?

Übungen

1) Welche Mitarbeiter verdienen mehr als der Mitarbeiter mit dem kleinsten Gehalt?
2) Welcher Abteilung sind keine Mitarbeiter zugeordnet?
3) Erhöhen Sie das Gehalt aller Mitarbeiter, die unter dem Durchschnitt verdienen, um 5%! (Hinweis: Auch bei der UPDATE-Anweisung kann eine Unterabfrage hinter der WHERE-Klausel folgen.)
4) Löschen Sie alle Werbeartikel, deren Preis unter dem Durchschnitt liegt! (Hinweis: Auch bei der DELETE-Anweisung kann eine Unterabfrage hinter der WHERE-Klausel folgen.)
5) Welche Sitzplätze der Vorstellung 11 kosten überdurchschnittlich viel?
6) Geben Sie die Namen der Mitarbeiter aus, die keinen Vorgesetzten haben! (Hinweis: Verwenden Sie die Schlüsselwörter NOT EXISTS.)
7) Geben Sie die Namen der Spielstätten aus, die die gleiche Postleitzahl besitzen!

11 Transaktionen

In Kapitel 11 sollen folgende Fragen geklärt werden:

- Wie kann man mehrere Änderungen an Datensätzen zu einer Einheit zusammenfassen?
- Warum sollte man mehrere Änderungen an Datensätzen zu einer Einheit zusammenfassen?
- Wie kann man Änderungen wieder rückgängig machen?
- Welche Probleme treten auf, wenn mehrere Benutzer gleichzeitig auf die gleichen Datensätze einer Tabelle zugreifen?
- Wie können Probleme, die im Mehrbenutzerbetrieb auftreten können, verhindert werden?

11.1 Motivation

Täglich gehen neue Bestellungen bei der Firma »KartoFinale«ein. Herr Klein, von Frau Kart inzwischen eingearbeitet, trägt die Bestellungen in die Tabellen Bestellung und Bestellposten ein. Gerade ist er dabei, eine Bestellung von der Kundin Frau Wiegerich zu erfassen. Sie möchte für den 27.8. zwei Karten für die Oper »Don Giovanni« Zunächst sieht Herr Klein mit Hilfe der SELECT-Anweisung in der Tabelle Sitzplatz nach, ob noch freie Plätze für diese Vorstellung mit der Nummer 12 vorhanden sind. Darauf ändert er den Zustand der beiden freien gefundenen Sitzplätze auf den Status »belegt« Hierzu verwendet er die UPDATE-Anweisung. Nun will er einen Datensatz in die Tabelle Bestellung einfügen. Er gibt die dazugehörige INSERT-Anweisung ein und gerade in dem Moment, wo er die Anweisung zum RDBMS senden will, fällt der Strom aus. Die Datenbank von »KartoFinale« befindet sich in einem inkonsistenten Zustand, zwei Sitzplätze wurden als »belegt«gekennzeichnet, obgleich es keine dazugehörige Bestellung gibt.

11.2 Transaktionen

In der realen Welt gibt es immer Folgen von Arbeiten, die zusammengenommen ein Ergebnis liefern. Diese Arbeiten müssen entweder vollständig oder gar nicht ausgeführt werden. Das bekannteste Beispiel hierfür ist die Überweisung eines Geldbetrages bei einer Bank. Der Geldbetrag eines Kontos wird einem anderen Konto gutgeschrieben. Dazu muss zunächst der Gelbetrag von einem Konto abgezogen und erst danach dem anderen Konto hinzuaddiert werden. Wird der zweite Schritt nicht korrekt ausgeführt, würde scheinbar Geld verschwinden.

Um so etwas zu vermeiden, fasst man mehrere Arbeitsanweisungen zu einer Transaktion zusammen. Eine Transaktion innerhalb einer Datenbank ist eine Folge von SQL-Anweisungen, die zusammengenommen eine Einheit bilden. Wichtigste Eigenschaft einer Transaktion ist ihre vollständige Ausführung. Wird eine Transaktion nicht vollständig ausgeführt, so muss der Zustand vor Beginn der Transaktion wiederhergestellt werden. Eine Transaktion wird also ganz oder gar nicht ausgeführt.

In unserem Fallbeispiel haben wir gesehen, dass das Erfassen einer Bestellung aus mehreren SQL-Anweisungen besteht und somit ein typisches Beispiel für eine Transaktion darstellt. In unserem Fall müssten also die Änderungen des Zustandes der beiden Sitzplätze wieder automatisch rückgängig gemacht werden.

Das Problem des Zurückführens einer Transaktion tritt jedoch nicht nur bei Hardwarefehlern auf. Programmfehler können auch dazu führen, dass eine Transaktion wieder zurückgeführt werden muss. Angenommen, Herr Klein würde die Bestellung in einer anderen Reihenfolge verarbeiten. Zunächst fügt er einen Datensatz in die Tabelle Bestellung ein, danach überprüft er erst, ob es freie Sitzplätze gibt. Ist dies nicht der Fall, müßte der erste Schritt dieser Transaktion selbständig wieder rückgängig gemacht werden.

SQL kennt vier Anweisungen zum Zurückführen und endgültigen Sichern von Änderungen.

11.3 ROLLBACK und COMMIT

Gemäß dem SQL:1999 Standard beginnt eine Transaktion immer automatisch mit der ersten SQL-Anweisung. Es gibt also im Standard keine Anweisung, um eine Transaktion explizit zu starten. Beendet wird eine Transaktion durch die SQL-Anweisung ROLLBACK oder COMMIT. Die SQL-Anweisung ROLLBACK stellt den Ursprungszustand vor Beginn der Transaktion wieder her. COMMIT dagegen beendet eine Transaktion erfolgreich und schreibt die Änderungen unwiderruflich in die Datenbank.

Betrachten wir hierzu wieder unser Fallbeispiel. Gehen wir davon aus, dass Herr Klein zunächst einen Datensatz in die Tabelle Bestellung einfügt und dann erst überprüft, ob freie Plätze vorhanden sind. Entsprechend gibt er folgende SQL-Anweisungen ein:

```
INSERT INTO Bestellung VALUES ( 9, '2002-02-18' , 3, NULL )

SELECT *
FROM   Sitzplatz
WHERE  Vorstellungsnummer = 12
AND    Zustand = 'frei'

-- wenn keine Plätze mehr frei sind, Transaktion zurückführen
ROLLBACK
```

Wenn Herr Klein feststellt, dass keine freien Plätze mehr vorhanden sind, muss er nicht überlegen, welche Änderungen die bisherige Transaktion durchgeführt hat. Das RDBMS »merkt« sich alle Änderungen einer Transaktion und macht diese bei Erhalt der Anweisung ROLLBACK selbständig wieder rückgängig.

Sind allerdings noch Plätze frei, so sieht das Ganze folgendermaßen aus:

```
INSERT INTO Bestellung VALUES ( 9, '2002-02-18' , 3, NULL )

SELECT *
FROM   Sitzplatz
WHERE  Vorstellungsnummer = 12
AND    Zustand = 'frei'

-- es sind zwei Plätze frei
UPDATE Sitzplatz
SET    Zustand = 'frei'
WHERE  Vorstellungsnummer = 12
AND    Artikelnummer IN (1202,1203)

INSERT INTO Bestellposten VALUES ( 9, 1, 1202, 1 )
INSERT INTO Bestellposten VALUES ( 9, 2, 1203, 1 )

COMMIT
```

Zunächst wird wieder die Bestellung eingefügt und danach überprüft, ob freie Sitzplätze vorhanden sind. Da dies der Fall ist, werden die zwei Sitzplätze mit der Artikelnummer 1202 und 1203 als »belegt« gekennzeichnet. Danach werden die beiden Artikel der Bestellung als Bestellposten hinzugefügt. Die Transaktion ist damit erfolgreich beendet und kann über die SQL-Anweisung COMMIT endgültig in die Datenbank geschrieben werden.

Was muss man aber nun tun, wenn innerhalb einer Transaktion z.B. der Strom ausfällt, muß man dann manuell Transaktionen zurückführen? Die Antwort darauf lautet eindeutig »Nein«, denn schließlich wissen sie ja gar nicht, welche Transaktionen zum Zeitpunkt des Stromausfalls gestartet waren, da ja mehrere Benutzer mit der Datenbank arbeiten. Ein RDBMS protokolliert, welche Transaktionen es gibt und welche Änderung jede Transaktion vornimmt. Erst wenn für eine Transaktion ein COMMIT oder ROLLBACK ausgeführt wird, wird die Transaktion als beendet angesehen. Bei einem Stromausfall nun erkennt das RDBMS beim erneuten Starten eigenständig, welche Transaktionen vor dem Stromausfall aktiv waren und führt diese automatisch über ein ROLLBACK zurück und überführt die Datenbank wieder in einen konsistenten Zustand.

11.4 SAVEPOINT

Da Transaktionen nicht explizit gestartet werden können, kann man sie natürlich auch nicht schachteln. Nun kommt es aber häufig vor, dass Transaktionen aus sehr vielen SQL-Anweisungen bestehen. Die Folge ist, dass das RDBMS sehr viel protokollieren muss und ein COMMIT bzw. ROLLBACK dazu führt, dass sehr viele Änderungen zu einem bestimmten Zeitpunkt geschrieben bzw. zurückgeführt werden müssen. Um dies zu vermeiden, führt SQL:1999 so genannte Savepoints ein. Savepoints dienen dazu, einen Teil einer Transaktion bereits vorzeitig wegzuschreiben bzw. zu einem ganz bestimmten Punkt einer Transaktion zurückzuspringen. Savepoints werden mit dem Schlüsselwort SAVEPOINT angelegt. Betrachten wir hierzu wieder ein Beispiel. Ein neuer Kunde ruft an und möchte den Werbeartikel »Opernführer« bestellen. Dazu muss zunächst der Kunde in die entsprechende Tabelle eingefügt werden. Dann wird ein Datensatz in die Tabelle Bestellung geschrieben und überprüft, ob der Werbeartikel vorhanden ist. Ist dies nicht der Fall, müsste die Transaktion normalerweise komplett zurückgeführt werden. Will der Kunde nun aber etwas anderes bestellen, so muss sein Datensatz wieder vollständig neu erfasst werden. Um dies zu vermeiden, fügt man nach dem Anlegen des Kunden einen Savepoint ein, zu dem man über ein ROLLBACK zurückspringen kann.

```
INSERT INTO Kunde
VALUES ( 5, 'Meyer', 'Michael', 'M', 'Menzelstr.', '108', 44444)

SAVEPOINT bestellung_schreiben
INSERT INTO Bestellung VALUES ( 9, '2002-02-18' , 3, NULL )

SELECT NULLIF(Lagerbestand, 9999)
FROM   Werbeartikel
WHERE  Beschreibung LIKE 'Opernführer'

-- Opernführer nicht auf Lager, also zurück zu
-- bestellung_schreiben
ROLLBACK TO SAVEPOINT bestellung_schreiben
```

Will der Kunde keinen anderen Artikel bestellen, so kann mit ROLLBACK die komplette Transaktion rückgängig gemacht werden:

```
-- Opernführer nicht auf Lager, Kunde will nichts bestellen,
-- also alles rückgängig machen, auch das Erfassen des Kunden
ROLLBACK
```

Abschließend sei darauf hingewiesen, dass die meisten RDBMS-Produkte in der Praxis nach jeder SQL-Anweisung in der Regel automatisch ein COMMIT ausführen (häufig als Autocommit bezeichnet). D.h., jede Änderung an der Datenbank wird sofort in die Datenbank geschrieben. Um in diesem Fall Transaktionen zu verwenden, muss das Autocommit ausgeschaltet werden.

11.5 Mehrbenutzerbetrieb (»Concurreny Control«)

11.5.1 Grundlagen

Bisher haben wir SQL-Anweisungen und Transaktionen immer nur so betrachtet, als würden wir alleine mit der Datenbank arbeiten. In der Realität greifen in der Regel jedoch sehr viele Benutzer auf eine Datenbank zu. Wenn zwei Benutzer zur gleichen Zeit mit den gleichen Daten arbeiten und diese verändern, können Probleme auftreten. Wir wollen uns im Folgenden vier solcher Probleme ansehen.

11.5.2 »Lost Update«

Betrachten wir zunächst wieder unser Fallbeispiel. Herr Klein nimmt gerade eine Bestellung für den Werbeartikel mit der Nummer 1001 entgegen. Zunächst stellt er fest, dass der Werbeartikel noch 26 mal auf Lager liegt. Der Kunde bestellt diesen Artikel dreimal, entsprechend korrigiert er den Lagerbestand.

Zur gleichen Zeit erhält der Einkauf diesen Artikel 10 mal neu, bringt ihn auf Lager und benachrichtigt Frau Kart. Diese trägt den Lagerbestand neu ein. Betrachten wir einmal den Ablauf:

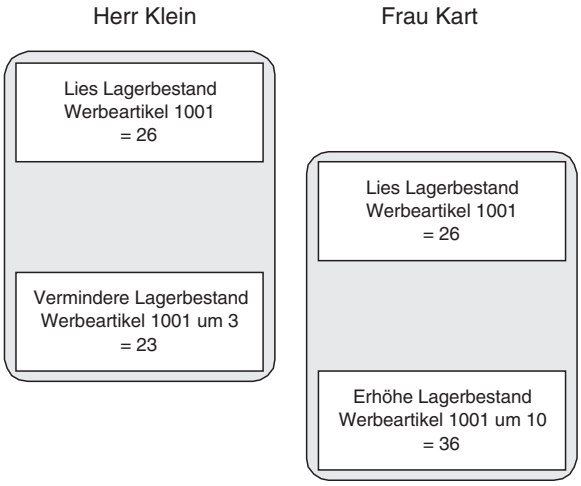


Abbildung 11.1: »Lost Update«Problem

Zuerst liest Herr Klein und danach Frau Kart den Lagerbestand. Herr Klein ändert den Datenbestand auf 23. Frau Kart bekommt davon nichts mit, da sie den Lagerbestand ja schon vorher gelesen hat und erhöht ihn auf 36.

Die Änderung von Herrn Klein geht verloren, da diese von der Änderung durch Frau Kart überschrieben wird. Dieses Problem wird deshalb als »Lost Update«-Problem bezeichnet.

11.5.3 »Dirty Read«

Das zweite Problem besteht darin, dass zunächst zwei Benutzer die gleichen Daten lesen und anschließend ein Benutzer die Daten ändert. Betrachten wir hierzu wieder unser Beispiel. Herr Klein hat den Werbeartikel mit der Nummer 1001 gelesen und danach den Lagerbestand entsprechend der Bestellung auf 23 angepasst. Zur gleichen Zeit erstellt Frau Kart eine Lagerliste mit den Beständen der Werbeartikel:

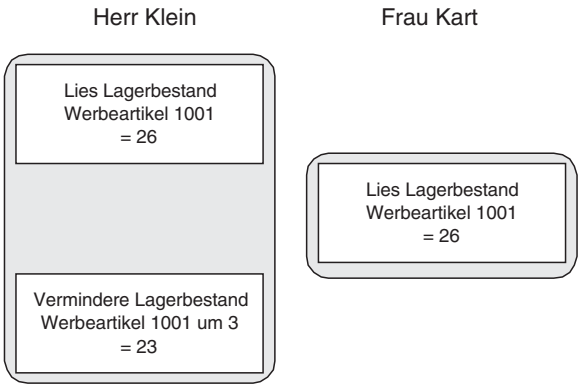


Abbildung 11.2: »Dirty Read«-Problem

Zunächst lesen beide Benutzer den Lagerbestand. Dann ändert Herr Klein den Lagerbestand und Frau Kart bekommt davon nichts mit. Die Lagerliste enthält fälschlicherweise einen Lagerbestand von 26 anstatt korrekterweise 23.

11.5.4 »Non-repeatable Read«

Das dritte Problem behandelt das erneute Lesen von Daten. Angenommen Herr Klein liest den Lagerbestand und ändert ihn danach auf 23. Zur gleichen Zeit liest Frau Kart den Lagerbestand zunächst, um nachzusehen, ob von diesem Artikel genug auf Lager ist. Da der Lagerbestand über 25 liegt, bestellt sie nicht neu. Danach liest sie den Lagerbestand erneut, um eine Lagerliste auszudrucken. Dieses mal erhält sie jedoch einen anderen Wert für den Lagerbestand.

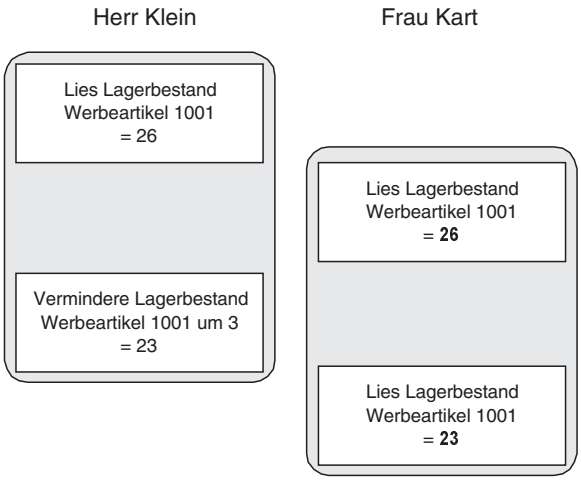


Abbildung 11.3: »Non-repeatable Read«-Problem

11.5.5 »Phantom Read«

Das vierte und letzte Problem beschäftigt sich mit unterschiedlichen Ergebnistabellen, obgleich jedesmal die gleiche Abfrage ausgeführt wird. Betrachten wir hierzu wieder ein Beispiel: Frau Kart erstellt eine Liste aller Werbeartikel. Gleichzeitig löscht der Geschäftsführer aus dieser Tabelle Werbeartikel, die nicht so oft verkauft wurden. Erstellt Frau Kart danach noch einmal die Liste, so fehlen Datensätze, die vorher noch in der Ergebnistabelle zurückgeliefert wurden:

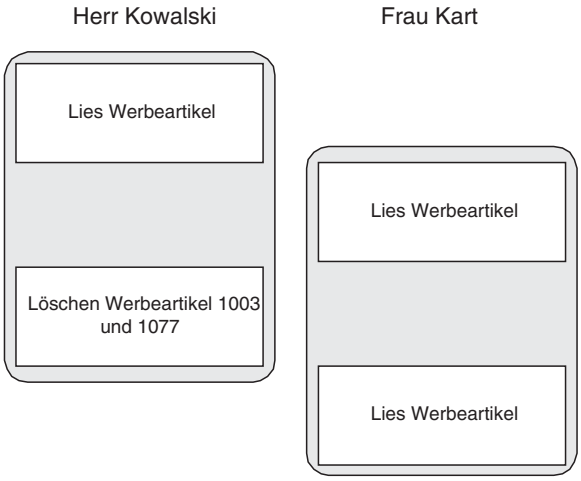


Abbildung 11.4: »Phantom Read«-Problem

11.5.6 Isolationslevel

Alle gerade beschriebenen Probleme müssen natürlich nicht als gegeben hingenommen werden, sondern werden automatisch vom RDBMS berücksichtigt. So muss man nicht dafür sorgen, dass eine bestimmte Tabelle exklusiv für einen gesperrt ist, das RDBMS führt dies automatisch für eine Transaktion durch.

Dennoch können mit SQL bestimmte Probleme zugelassen werden. Betrachten wir hierzu wieder unser Beispiel: Wenn Frau Kart eine Liste aller Werbartikel erstellen möchte, so ist es nicht in jedem Fall unbedingt notwendig, dass diese zeitgenau aktuell ist. Es ist also nicht schlimm, wenn »Dirty Read«-, »Non-repeatable Read« und »Phantom Read«-Probleme vorkommen.

Wenn mehrere Benutzer gleichzeitig auf Daten zugreifen und das RDBMS das Ganze so koordinieren muss, dass die aufgeführten Probleme nicht auftreten, so kostet das Ressourcen und Abfragen erfolgen für die Benutzer natürlich langsamer. Läßt man nun bei bestimmten Transaktionen bestimmte Probleme zu, so kann sich das RDBMS auf andere Dinge »konzentrieren« und Abfragen können schneller abgewickelt werden.

Um explizit Probleme zuzulassen, kennt SQL die SET TRANSACTION-Anweisung. Sie erlaubt, den Isolationslevel einer Transaktion einzustellen. Einer der folgenden Schlüsselwörter ist möglich:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Die folgende Tabelle gibt Auskunft darüber, welchen Isolationslevel man einstellen muß, um bestimmte Probleme zuzulassen. SERIALIZABLE entspricht dabei der Standardeinstellung und verhindert alle aufgeführten Probleme.

Isolationslevel	»Dirty Read«	»Non-Repeatable Read«	»Phantom Read«
READ UNCOMMITED	Ja	Ja	Ja
READ COMMITTED	Nein	Ja	Ja
REPEATABLE READ	Nein	Nein	Ja
SERIALIZABLE	Nein	Nein	Nein

Tabelle 11.1: Isolationslevel

Um alle Probleme für eine Transaktion zuzulassen, schreibt man in SQL vor Beginn der Transaktion:

```
SET TRANSACTION READ UNCOMMITED
```

11.6 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie man mehrere SQL-Anweisungen zu einer Einheit, einer so genannten Transaktion, zusammenfasst. Die SQL-Anweisungen einer Transaktion werden entweder vollständig oder gar nicht ausgeführt. Um eine Transaktion wieder zurückzuführen zu dem Zustand vor Beginn der Transaktion, verwendet man die SQL-Anweisung ROLLBACK. Um die Änderungen einer Transaktion endgültig in die Datenbank zu schreiben, verwendet man die SQL-Anweisung COMMIT. Um größere Transaktionen einfacher handhabbar zu machen, kann man so genannte »Savepoints« definieren, zu denen über ein ROLLBACK »zurückgesprungen« werden kann.

Durch den Mehrbenutzerbetrieb können verschiedene Probleme auftreten. Diese Probleme werden jedoch weitestgehend vom RDBMS selbständig behoben. Dafür benötigt das RDBMS allerdings Ressourcen, wodurch Abfragen anderer Benutzer dann durchaus langsamer ausgeführt werden. Um das RDBMS zu entlasten, können bestimmte Probleme für bestimmte Transaktionen wieder zugelassen werden.

11.7 Aufgaben

Wiederholungsfragen

1) Welche Eigenschaften muss eine Transaktion besitzen?
2) Wie werden die Änderungen einer Transaktion wieder rückgängig gemacht?
3) Wie werden die Änderungen einer Transaktion endgültig in die Datenbank geschrieben?
4) Was sind »Savepoints«?

Übungen

1) Gegeben sei eine leere Tabelle »Person« mit den Spalten »Name«, »Vorname«, »Alter«. Welchen Inhalt hat diese Tabelle nachdem folgende SQL-Anweisungen ausgeführt wurden?

```
INSERT INTO Person VALUES ( 'Muster', 'Hans', 63 )
INSERT INTO Person VALUES ( 'Carlson', 'Peter', 26 )
INSERT INTO Person VALUES ( 'Friedrichs', 'Frieda', 12 )
ROLLBACK
```

2) Welchen Inhalt hat diese Tabelle, nachdem folgende SQL-Anweisungen ausgeführt wurden?

```
INSERT INTO Person VALUES ( 'Muster', 'Hans', 63 )
INSERT INTO Person VALUES ( 'Carlson', 'Peter', 26 )
INSERT INTO Person VALUES ( 'Friedrichs', 'Frieda', 12 )
COMMIT
```

3) Welchen Inhalt hat diese Tabelle, nachdem folgende SQL-Anweisungen ausgeführt wurden?

```
INSERT INTO Person VALUES ( 'Muster', 'Hans', 63 )
SAVEPOINT sp1
INSERT INTO Person VALUES ( 'Carlson', 'Peter', 26 )
INSERT INTO Person VALUES ( 'Friedrichs', 'Frieda', 12 )
ROLLBACK TO sp1
```

4) Welchen Inhalt hat diese Tabelle, nachdem folgende SQL-Anweisungen ausgeführt wurden?

```
INSERT INTO Person VALUES ( 'Muster', 'Hans', 63 )
SAVEPOINT sp1
INSERT INTO Person VALUES ( 'Carlson', 'Peter', 26 )
INSERT INTO Person VALUES ( 'Friedrichs', 'Frieda', 12 )
ROLLBACK
```

12 »Stored Procedures« und »Prozedurale Sprachelemente«

In Kapitel 12 sollen folgende Fragen geklärt werden:

- Wie kann man mehrere Anweisungen unter einem Namen zusammenfassen und in einem Schritt ausführen lassen?
- Was sind prozedurale Sprachelemente?
- Was sind gespeicherte Prozeduren (»Stored Procedures«)?
- Was sind Funktionen und Prozeduren?
- Was sind Variablen?
- Welche prozeduralen Anweisungen kennt SQL zur Steuerung des Programmablaufs?

12.1 Motivation

Täglich erfassen Herr Klein und Frau Kart neue Bestellungen, indem sie einzelne SQL-Anweisungen eingeben, um Datensätze in die Tabellen »Bestellung«, »Bestellposten« und »Kunde«einzufügen. Das erscheint Frau Kart zu Recht sehr umständlich, also ruft sie Herrn Fleissig von der Unternehmensberatung an und schildert ihr Problem.

Herr Fleissig erzählt ihr, dass man mehrere SQL-Anweisungen auf dem Datenbankserver unter einem selbstdefinierten Namen in einer Routine speichern kann. Über den Namen der Routine kann man dann die Folge von SQL-Anweisungen in einem Schritt ausführen. Mit Routinen kann man aber noch viel mehr machen, da SQL auch prozedurale Sprachelemente enthält, die einer vollständigen Programmiersprache entsprechen.

Er empfiehlt Frau Kart, eine Routine mit dem Namen »ErfasseBestellung«zu erstellen. Dieser Routine kann sie dann die Werte einer Bestellung übergeben.

12.2 Routinen

Bisher haben wir SQL als rein deklarative (beschreibende) Programmiersprache kennen gelernt, d.h. man beschreibt, was man als Ergebnis vom RDBMS erhalten möchte. In Kapitel 8 wurde auch schon kurz erwähnt, dass SQL prozedurale Sprachelemente enthält. Prozedurale Sprachelemente dienen dazu, dem RDBMS zu beschreiben, wie ein gewünschtes Ergebnis schrittweise bearbeitet wird. Folgen von prozeduralen SQL-Anweisungen sind zwar schwieriger zu programmieren, da man dem RDBMS jeden einzelnen Arbeitsschritt vorgeben muss, dafür ist man mit prozeduralen SQL-Anwei-

sungen wesentlich flexibler als mit rein deklarativen Sprachelementen. Gerade die Kombination bei SQL von deklarativen und prozeduralen Sprachelementen macht dessen Leistungsfähigkeit aus. In diesem Kapitel werden wir uns mit den prozeduralen Sprachelementen beschäftigen. Das Kapitel soll lediglich als Einführung dienen, um einen Überblick zu erhalten, was mit prozeduralen Sprachelementen machbar ist.

Bevor wir uns jedoch mit den prozeduralen Sprachelementen beschäftigen, müssen zunächst die Begriffe Funktion, Prozedur und Variable geklärt werden. Folgen von SQL-Anweisungen kann man unter einem gemeinsamen Namen speichern und über diesen Namen wieder aufrufen. Diese Folgen von SQL-Anweisungen werden als Routine bezeichnet. Bei Routinen unterscheidet man in SQL zwischen Funktionen und Prozeduren. Betrachten wir zunächst ein Beispiel für eine gespeicherte Prozedur, die alle Werbeartikel ausgeben soll, die über 20,-- Eur kosten. Die SELECT-Anweisung dazu ist für uns kein Problem mehr, sie lautet:

```
SELECT *
FROM   Werbeartikel
WHERE  Preis > 20.00
```

Um nun nicht jedesmal die komplette SELECT-Anweisung zur Ermittlung einer Liste dieser Werbeartikel eingeben zu müssen, wird eine Prozedur mit dem Namen »Lies-Werbeartikel« erstellt. Dies erfolgt über die SQL-Anweisung CREATE PROCEDURE:

```
CREATE PROCEDURE LiesWerbeartikel ()
      SELECT *
      FROM   Werbeartikel
      WHERE  Preis > 20.00;
```

Nun kann über die SQL-Anweisung CALL die Prozedur wie folgt wieder aufgerufen werden und man erhält das gleiche Ergebnis, als wenn die dazugehörige SELECT-Anweisung eingegeben worden wäre:

```
CALL LiesWerbeartikel
```

Jetzt wäre es natürlich komfortabel, wenn wir einen Schwellwert für den Preis an die Prozedur übergeben könnten, damit nicht immer nur alle Artikel über 20,-- Eur ausgegeben werden. Man kann deshalb Werte, sogenannte Parameter, an eine Prozedur übergeben. Für den Parameter »PreisSchwellwert« muss die Prozedur wie folgt geändert werden:

```
CREATE PROCEDURE LiesWerbeartikel(PreisSchwellwert DECIMAL(8,2))
      SELECT *
      FROM   Werbeartikel
      WHERE  Preis > PreisSchwellwert;
```

Bevor die Prozedur jedoch erneut angelegt werden kann, muss diese vorher natürlich gelöscht werden. Dies erfolgt über die SQL-Anweisung DROP PROCEDURE. Um also die Prozedur zu löschen, schreibt man einfach:

```
DROP PROCEDURE LiesWerbeartikel
```

Unser bisheriges Beispiel war sehr einfach. Damit wir kompliziertere Abfragen schrittweise bearbeiten können, müssen wir den Begriff Variable verstehen. Eine Variable ist in einer Programmiersprache ein Platzhalter für einen Wert, auf den man über einen Namen zugreifen kann. Diesem Namen bzw. dieser Variablen kann man je nach Bedarf beliebige Werte zuweisen. Eine Variable haben wir eben bereits bei der Prozedur »LiesWerbeartikel« kennen gelernt: »PreisSchwellwert«. Dieser Parameter kann beliebige Zahlen speichern, die aus maximal 8 Stellen mit 2 Nachkommastellen bestehen. Eine Variable hat also einen bestimmten Datentyp und damit auch einen bestimmten Wertebereich. Um explizit eine Variable anzulegen, verwendet man die SQL-Anweisung DECLARE. Um z.B. in einer Prozedur den Durchschnittspreis eines Werbeartikels zu ermitteln und diesen in einer Variablen zu speichern, kann man in zwei Schritten vorgehen:

```
CREATE PROCEDURE ErmittleDurchschnitt ()
      DECLARE durchschnitt DECIMAL(8,2);

      SELECT AVG(Preis) INTO durchschnitt
      FROM   Werbeartikel;

      SELECT durchschnitt;
```

Zunächst wird die Variable »durchschnitt« deklariert, danach die SELECT-Anweisung zum Ermitteln des Durchschnittspreises aufgerufen und über das Schlüsselwort INTO in der Variablen »durchschnitt« gespeichert. Danach kann über die SQL-Anweisung SELECT die Variable ausgegeben werden.

Soll der Durchschnittswert nach Aufruf der Prozedur weiterverwendet werden, können Parameter als Outputparameter deklariert werden, in die der Wert dann zurückgeben wird. Das Ganze könnte folgendermaßen aussehen:

```
CREATE PROCEDURE ErmittleDurchschnitt(OUT rueckgabe DECIMAL(8,2))
      DECLARE durchschnitt DECIMAL(8,2);

      SELECT AVG(Preis) INTO durchschnitt
      FROM   Werbeartikel;

      SET rueckgabe = durchschnitt;
```

Vor der Angabe des Übergabeparameters »rueckgabe« gibt man über das Schlüsselwort OUT bekannt, dass dieser Parameter in der Prozedur einen Wert zugewiesen bekommt, den man nach der Ausführung weiterverwenden möchte. Standardmäßig sind Übergabeparameter mit dem Schlüsselwort IN festgelegt, übergeben also einen Wert an die Prozedur. In diesem Fall steht IN als Abkürzung für Input. Ein Übergabeparameter kann sowohl einen Wert an eine Prozedur übergeben als auch einen Wert als Ausgabeparameter speichern. Er muss dann mit dem Schlüsselwort INOUT deklariert werden. Wird bei einem Übergabeparameter weder IN, OUT noch INOUT angegeben, so handelt es sich standardmäßig um einen Eingabeparameter (IN). Doch wie ruft man nun die Prozedur auf? Zuerst deklariert man eine Variable, die nach Aufruf der Prozedur den Durchschnittspreis enthalten soll. Diese Variable übergibt man dann an die Prozedur »ErmittleDurchschnitt« und erhält danach in ihr den Durchschnittspreis zurück:

```
DECLARE zahl DECIMAL(8,2);
CALL ErmittleDurchschnitt (zahl);
SELECT 'Durchschnittspreis für Werbeartikel: ' || zahl;
```

Bisher haben wir gesehen, wie man Prozeduren anlegt, die auf dem Datenbankserver gespeichert werden und wie man Werte als Parameter an die Prozedur übergibt. Dabei können Parameter nur Werte an die Prozedur übergeben (IN), zurückliefern (OUT) oder beides (INOUT). Aufgerufen werden gespeicherte Prozeduren über die SQL-Anweisung CALL.

Am Anfang dieses Abschnitts haben wir erfahren, dass es neben Prozeduren auch Funktionen gibt. Funktionen sind weitgehend identisch mit Prozeduren, liefern jedoch immer einen skalaren Wert zurück. Wir haben in Kapitel 8 bereits vordefinierte Funktionen wie TRIM, SUBSTRING oder ABS kennen gelernt. Funktionen kann man auch selbst über die SQL-Anweisung CREATE FUNCTION erzeugen und über DROP FUNCTION wieder löschen. Im Gegensatz zur Prozedur wird bei einer Funktion zusätzlich angegeben, welchen Datentyp der zurückgegebene Wert hat. Betrachten wir dazu, wie eine Funktion aussieht, die den Durchschnittspreis für Werbeartikel zurückliefert:

```
CREATE FUNCTION ErmittleDurchschnittspreis() RETURNS
DECIMAL(8,2)
   RETURN (SELECT AVG(Preis) FROM Werbeartikel);
```

Hinter CREATE FUNCTION folgt der Name der Funktion und danach das Schlüsselwort RETURNS, das angibt, von welchem Datentyp der zurückgelieferte Wert ist. Anschließend wird die SELECT-Anweisung zum Ermitteln des Durchschnittspreises aufgerufen und über die SQL-Anweisung RETURN zurückgegeben. Jede Funktion in SQL muss mit der SQL-Anweisung RETURN einen Wert des angegebenen Datentyps zurückgeben.

Funktionen werden nicht wie Prozeduren mit dem Schlüsselwort CALL aufgerufen, sondern können innerhalb von Anweisungen direkt verwendet werden.

Um nun z.B. alle Werbeartikel und in jeder Zeile gleichzeitig den Durchschnittspreis mit auszugeben, kann man schreiben:

```
SELECT ErmittleDurchschnittspreis() AS 'Durchschnitt',
       Beschreibung, Lagerbestand, Preis
FROM   Werbeartikel

Durchschnitt  Beschreibung              Lagerbestand  Preis
------------  ------------------------  ------------  -----
33.08         Noten f. Klavier          26            89.00
33.08         T-Shirt Farbe: rot        11            29.99
33.08         Phil Collins in Concert   21            20.00
33.08         Plakat mit Musical-Katzen 89            33.50
33.08         Opernführer               9999          19.99
33.08         Schwarzer Zauberstock     32            5.99
```

Das Beispiel läßt erahnen, wie leistungsfähig Funktionen und Prozeduren sind. Das gleiche Ergebnis mit einer einzigen SELECT-Anweisung wäre so nicht ohne weiteres möglich. Wir wissen nun, was Funktionen und Prozeduren sind.

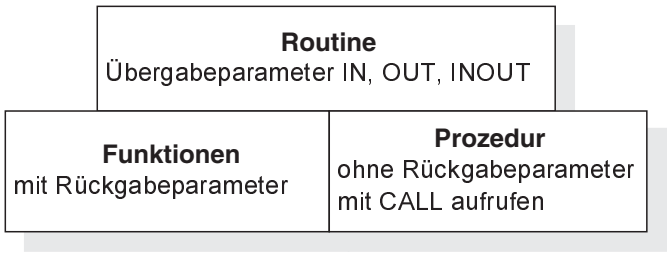


Abbildung 12.1: Funktionen und Prozeduren

Im Folgenden wollen wir sehen, wie man prozedurale Sprachelemente verwenden kann, um leistungsfähige und wiederverwendbare Routinen zu schreiben. Zunächst werden alle prozeduralen Sprachelemente vorgestellt. Am Schluss dieses Kapitels wird dann eine etwas komplexere Funktion programmiert, in der diese Sprachelemente angewendet werden.

12.3 Sprachelemente zur Kontrollsteuerung

Bisher haben wir nur SQL-Anweisungen kennen gelernt, die sequenziell, also immer nach der gleichen Reihenfolge abgearbeitet wurden. Über Sprachelemente zur Kontrollsteuerung kann die Abarbeitung der einzelnen Anweisungen gesteuert werden.

12.3.1 BEGIN...END

In unserem letzten Beispiel haben wir in der Funktion nur eine einzige SQL-Anweisung ausgeführt. Über die Schlüsselwörter BEGIN...END können mehrere SQL-Anweisungen zusammengefasst werden.

Die Funktion »ErmittleDurchschnittspreis« kann man dann auch wie folgt schreiben:

```
CREATE FUNCTION ErmittleDurchschnittspreis() RETURNS
DECIMAL(8,2)
BEGIN
      DECLARE durchschnitt DECIMAL(8,2);

      SELECT AVG(Preis) INTO durchschnitt FROM Werbeartikel;
      RETURN (durchschnitt);
END
```

Zunächst wird in der Funktion eine Variable deklariert, in der der Durchschnittspreis gespeichert werden soll. Dann wird der Durchschnittspreis ermittelt und über

RETURN zurückgeliefert. Im Gegensatz zu unserer vorherigen Version von »Ermittle-Durchschnittspreis« ist diese Variante etwas umfangreicher, dafür aber verständlicher. Über die Schlüsselwörter BEGIN...END werden die Anweisungen zu einem Anweisungsblock zusammengefasst.

12.3.2 IF..THEN...ELSE

Die IF...THEN...ELSE-Anweisung dient zum Abbilden von »Wenn...dann«-Entscheidungen. Abhängig von einer bestimmten Bedingung wird ein Anweisungsblock ausgeführt oder nicht. Betrachten wir hierzu ein Beispiel: Abhängig davon, wie viele Bestellungen ein Mitarbeiter bearbeitet hat, soll er einen Bonus ausgezahlt bekommen. Bei einer oder keiner bearbeiteten Bestellung gibt es keinen Bonus, bei zwei bearbeiteten Bestellungen 10,-- Eur und bei mehr als drei Bestellungen 20,-- Eur. Hierzu schreiben wir eine Prozedur, die als Übergabeparameter die Personalnummer erhält und dann ausgibt, wie hoch der Bonus für den Mitarbeiter ist:

```
CREATE PROCEDURE HoeheBonus(Personalnr INTEGER,
                            OUT Bonus DECIMAL(4,2))
BEGIN
   DECLARE anzahlBestellungen INTEGER;

   -- Ermitteln der bearbeiteten Bestellungen
   SELECT COUNT(Personalnummer) INTO anzahlBestellungen
   FROM   Bestellung
   WHERE  Personalnummer = Personalnr;

   -- Variable Bonus auf 0 setzen
   SET Bonus = 0.00;

   IF anzahlBestellungen = 2 THEN
       SET Bonus = 10.00;
   ELSEIF anzahlBestellungen >= 3 THEN
       SET Bonus = 20.00;
   END IF
END
```

Zunächst wird in der Funktion die Anzahl der Bestellungen ermittelt, die der Mitarbeiter mit der übergebenen Personalnummer bearbeitet hat. In der IF-Anweisung wird dann abhängig von der Anzahl der bearbeiteten Bestellungen die Übergabevariable »Bonus« auf den entsprechenden Betrag gesetzt.

12.3.3 CASE

Die CASE-Anweisung entspricht weitestgehend einer mehrfach geschachtelten IF-Anweisung. Betrachten wir dazu die IF-Anweisung aus dem letzten Beispiel. Mit einer CASE-Anweisung könnte man das Gleiche wie folgt erreichen:

```
SET Bonus =
    CASE
      WHEN anzahlBestellungen  = 2 THEN 10.00
      WHEN anzahlBestellungen >= 3 THEN 20.00
      ELSE 0.00
    END
```

Eingeleitet durch das CASE-Schlüsselwort folgt hinter dem Schlüsselwort WHEN jeweils die Bedingung die erfüllt sein muss, damit der Wert hinter THEN verwendet wird. Der ELSE-Zweig wird ausgeführt, wenn keine der aufgeführen WHEN-Bedingungen eintritt.

Soll immer die gleiche Variable ausschließlich auf Gleichheit mit bestimmten Werten getestet werden, ist die Verwendung der CASE-Anweisung noch einfacher. Angenommen, wir möchten über eine SELECT-Anweisung alle Kunden ausgeben. Anstelle der Kürzel »M« und »W« für Geschlecht soll allerdings in der Ergebnistabelle »männlich« bzw. »weiblich« ausgegeben werden. Da man in diesem Fall immer die gleiche Spalte auf Gleichheit mit »W« oder »M« testet, kann diese direkt hinter der CASE-Anweisung aufgeführt werden. Dann muss hinter der WHEN-Klausel der Spaltenname nicht explizit aufgeführt werden:

```
SELECT Name,
       Vorname,
       CASE Geschlecht
           WHEN 'M' THEN 'Männlich'
           WHEN 'W' THEN 'Weiblich'
           ELSE 'unbekannt'
       END AS 'Geschlecht'
FROM   Kunde

Name       Vorname  Geschlecht
---------  -------  ----------
Bolte      Bertram  Männlich
Muster     Hans     Männlich
Wiegerich  Frieda   Weiblich
Carlson    Peter    Männlich
```

12.3.4 REPEAT...UNTIL

REPEAT wiederholt einen Anweisungsblock so lange, bis eine definierte Bedingung eintritt. Der Anweisungblock wird bei REPEAT mindestens einmal durchlaufen, da die Bedingung erst am Ende des Anweisungsblocks überprüft wird. Nehmen wir als Beispiel eine Funktion, die die n-te Potenz einer Zahl berechnen soll. Wir wollen diese Funktion Potenz nennen und es werden zwei Übergabeparameter übergeben: Zum einen die Zahl, von der die Potenz berechnet werden soll und zum anderen die Zahl für die n-te Potenz. Mit Hilfe der REPEAT-Anweisung wird die Multiplikation dann entsprechend oft durchlaufen:

```
CREATE FUNCTION Potenz (basis FLOAT, exponent INTEGER)
RETURNS FLOAT
BEGIN
   DECLARE ergebnis FLOAT;

   SET ergebnis = basis;

   -- multipliziere ergebnis so oft mit basis,
   -- bis exponent kleiner gleich 1
   REPEAT
       SET ergebnis = ergebnis * basis;
       SET exponent = exponent - 1;
       UNTIL exponent <= 1
   END REPEAT

   RETURN ergebnis;
END
```

Der Anweisungblock zwischen REPEAT und END REPEAT, wird solange wiederholt, bis die Bedingung hinter UNTIL eintritt. Beachtet werden muss, dass das Beispiel nicht mit negativen Exponenten und dem Exponenten 0 rechnen kann. Um das zu ermöglichen, muss das Beispiel einfach mit Hilfe der IF-Anweisung erweitert werden, was hier nicht gezeigt werden, sondern für Sie eine Übungsmöglichkeit darstellen soll.

12.3.5 WHILE...DO

WHILE dient wie REPEAT dazu, einen Anweisungsblock mehrfach zu durchlaufen. Im Gegensatz zu REPEAT überprüft die WHILE-Anweisung die Abbruchsbedingung jedoch vor dem Anweisungsblock. Die Funktion Potenz kann dementsprechend wie folgt umgeschrieben werden, so dass WHILE anstelle der REPEAT-Anweisung verwendet wird:

```
…
WHILE exponent > 1 DO
    SET ergebnis = ergebnis * basis;
    SET exponent = exponent - 1;
END WHILE
…
```

In diesem Fall wird über die WHILE-Anweisung der Anweisungsblock solange durchlaufen, wie die Variable exponent einen Wert größer 1 hat.

12.3.6 LEAVE

Die SQL-Anweisung LEAVE ist nur in Zusammenhang mit Wiederholungen von Anweisungsblöcken über WHILE oder REPEAT sinnvoll und dient dazu, eine Wiederholungsschleife vorzeitig zu beenden. Betrachten wir dazu wieder unser letztes Bei-

spiel mit der Funktion Potenz. Angenommen, es sollen keine Ergebnisse größer als 10000 zurückgegeben werden, so kann die Schleife vorzeitig über das Schlüsselwort LEAVE verlassen werden. Demnach würde die WHILE-Schleife folgendermaßen aussehen:

```
…
WHILE exponent > 1 DO
    SET ergebnis = ergebnis * basis;
    IF ergebnis > 10000 THEN
        LEAVE; -- Verlassen der Wiederholungsschleife
    END IF
    SET exponent = exponent - 1;
END WHILE
…
```

12.4 Beispiel: Bestellung erfassen

Zum Abschluss dieses Kapitels sollen alle Erkenntnisse noch einmal an zwei Beispielen vertieft werden. Im ersten Beispiel soll eine Prozedur geschrieben werden, die für einen Kunden den Namen des Kindes in die dazugehörige Tabelle einträgt. Da das Geburtsdatum des Kindes i.d.R. kleiner als das des Elternteils ist, soll die Prozedur dies überprüfen und wenn nötig, eine Fehlermeldung zurückliefern. Außerdem soll überprüft werden, ob das Geburtsdatum niedriger als das aktuelle Tagesdatum ist.

Der Prozedur »KindErfassen« werden vier Eingabe- und ein Ausgabeparameter übergeben. Die Eingabeparameter »Kundennr«, »VornameKind«, »GeburtsdatumKind«, »GeschlechtKind« enthalten die Kundennummer des Elternteils sowie die Informationen über das Kind. Der Ausgabeparameter »Rueckgabe« liefert eine Zeichenkette zurück, in der nach Ausführen der Prozedur beschrieben ist, ob Probleme beim Einfügen aufgetreten sind oder ob das Einfügen erfolgreich war.

```
CREATE PROCEDURE KindErfassen( Kundennr INTEGER,
                               VornameKind VARCHAR(20),
                               GeburtsdatumKind DATE,
                               GeschlechtKind CHAR(1),
                               OUT Rueckgabe VARCHAR(100) )
...
```

Danach werden über eine IF-Anweisung mehrere Werte auf Plausibilität überprüft. Die erste Bedingung testet, ob das Geburtsdatum des Kindes in der Zukunft liegt. Ist dies der Fall, wird eine Fehlermeldung in der Variablen »Rueckgabe« zurückgeliefert. Andernfalls wird das Geburtsdatum des Elternteils über die SELECT-Anweisung ermittelt und überprüft, ob das Kind dem Geburtsdatum nach älter als das Elternteil ist. Auch in diesem Fall erfolgt eine Fehlermeldung. Ist das Kind jünger als das Elternteil, werden die Daten in die Tabelle »Kind« über die INSERT-Anweisung eingefügt.

```
IF GeburtsdatumKind > CURRENT_DATE THEN
   SET Rueckgabe = 'Problem: Geburtsdatum in der Zukunft!';
ELSEIF ( SELECT Geburtsdatum
         FROM Kunde
         WHERE Kundennummer = Kundennr ) >= GeburtsdatumKind
   SET Rueckgabe = 'Problem: Kind Geburtsdatum aktueller!';
ELSE
   INSERT INTO Kind VALUES( Kundennr, VornameKind,
                            GeburtsdatumKind, GeschlechtKind);
   SET Rueckgabe = 'Datensatz erfolgreich eingefügt!'
END IF
```

Zum Aufrufen unserer Prozedur müssen wir eine Variable für den Übergabeparameter »Rueckgabe« deklarieren, in der die Zeichenkette mit der Fehlermeldung zurückgegeben wird. Danach wird die Prozedur über die SQL-Anweisung CALL aufgerufen:

```
DECLARE ergebnis VARCHAR(100);
CALL KindErfassen( 1, 'Hans', '2003-01-01', DEFAULT, ergebnis );
SELECT ergebnis;

------------------------------------
Problem: Geburtsdatum in der Zukunft!
```

12.5 Beispiel: Phonetischer Vergleich

Im zweiten Beispiel geht es um das Erstellen einer Funktion zur phonetischen Suche. Einige RDBMS-Produkte wie z.B. IBM DB2 oder Microsoft SQL Server bieten die Funktion SOUNDEX an, mit der man Wörter auf Ähnlichkeit im Klang überprüfen kann. Dieser phonetische Vergleich basiert auf einem Algorithmus, der bestimmten ähnlich klingenden Buchstaben eine Zahl zuordnet und Vokale beim Vergleich nicht mit berücksichtigt. Dadurch kann man Wörter, die ähnlich klingen, aber unterschiedlich buchstabiert werden, finden. Personen, die den Namen Meyer, Meier, Maier oder Mayer haben, ergeben dabei die gleiche Zahlenfolge.

Betrachten wir zunächst den Algorithmus: Der erste Buchstabe eines Wortes bleibt erhalten und wird in einen Großbuchstaben umgewandelt. Allen folgenden Buchstaben werden Zahlen zugeordnet, außer es handelt sich um Vokale oder die Buchstaben H, W und Y.

Buchstaben	Zugeordnete Zahl
B, P, F, V	1
C, S, K, G, J, Q, X, Z	2
D, T	3
L	4
M, N	5
R	6

Tabelle 12.1: »Soundex-Algorithmus«

Von dem ermittelten Soundex-Code werden dann die ersten 4 Zeichen berücksichtigt. Besteht der Code aus weniger als 4 Zeichen, werden die restlichen Zeichen mit der Zahl 0 aufgefüllt.

Betrachten wir als Beispiel den Nachnamen »Muster« und berechnen den Code einmal manuell, so ergibt sich »M236«. Der erste Buchstabe des Wortes wird übernommen. Der zweite Buchstabe, das »u« wird ignoriert (Vokal), dem Buchstabe »s« wird die Zahl 2 zugeordnet, »t« die Zahl 3, »e« wird wieder ignoriert (Vokal) und »r« entspricht der Zahl 6. Somit ergibt sich als Soundex-Code der Wert »M236«. Dieser Algorithmus wird übrigens vom »Amerikanischen Nationalen Archiv« verwendet, um Akten der Volkszählung gemäß den Namen der Bürger lautbezogen zu ordnen. Der Algorithmus bezieht sich damit zwar mehr auf die englische Sprache, allerdings hat er sich auch im deutschen Sprachraum bewährt.

Wie setzen wir diesen Algorithmus nun in SQL um? Zunächst erzeugen wir eine Funktion, der wir den Namen »Klang« geben wollen. Als Übergabeparameter wird das Wort übermittelt, für das der Soundex-Code berechnet werden soll. Entsprechend gibt die Funktion dann den berechneten Soundex-Code als Zeichenkette zurück.

```
CREATE FUNCTION Klang ( Wort VARCHAR(250) )
RETURNS VARCHAR(250)
BEGIN
...
```

Zuerst werden die Variablen deklariert, die in der Funktion verwendet werden. In diesem Fall wird die Variable »Klang« für den Rückgabewert benötigt. Weiterhin deklarieren wir eine Variable, die den Index auf das aktuell zu bearbeitende Zeichen enthält und eine weitere, die jeweils ein einzelnes Zeichen des Wortes speichern soll.

```
   DECLARE Klang   VARCHAR(250);
   DECLARE Index   INTEGER;
   DECLARE Zeichen VARCHAR;
...
```

Nun folgen die eigentlichen SQL-Anweisungen der Funktion. Zuerst werden überflüssige Leerzeichen entfernt und das erste Zeichen des Wortes in der Rückgabevariablen »Klang« gespeichert:

```
   -- Leerzeichen entfernen und in Großbuchstaben wandeln
   SET Wort = TRIM( BOTH ' ' FROM Wort );
   SET Wort = UPPER( Wort );

   -- erstes Zeichen des Wortes bleibt erhalten
   SET Index = 1;
   SET Klang = SUBSTRING( Wort FROM Index FOR 1);
...
```

Darauf folgt die eigentliche WHILE-Schleife, die solange ausgeführt wird, bis das letzte Zeichen bearbeitet ist oder bis der Soundex-Code aus maximal 4 Zeichen besteht:

```
   -- Wort konvertieren
   WHILE Index <= LEN( Wort ) AND LEN( Klang ) < 4  DO
...
```

Danach wird die Variable »Index« auf das nächste zu bearbeitende Zeichen gesetzt, also um 1 erhöht, und das nächste Zeichen ermittelt:

```
      SET Index = Index + 1;

      -- nächstes Zeichen bearbeiten und entsprechende
      -- Codezahl hinterlegen
      SET Zeichen = SUBSTRING( Wort FROM Index FOR 1 )
...
```

Es folgt die Ermittlung der entsprechenden Zahl, die dem aktuellen Buchstaben zugeordnet werden soll. Über die CASE-Anweisung wird in der Variablen »Zeichen« die dazugehörige Zahl gespeichert:

```
      SET Zeichen =
      CASE
        WHEN Zeichen IN('B','P','F','V')                THEN '1'
        WHEN Zeichen IN('C','S','G','J','K','Q','X','Z') THEN '2'
        WHEN Zeichen IN('D','T')                        THEN '3'
        WHEN Zeichen IN('L')                            THEN '4'
        WHEN Zeichen IN('M','N')                        THEN '5'
        WHEN Zeichen IN('R')                            THEN '6'
        ELSE                                                '' 
      END CASE;
...
```

Schließlich wird die ermittelte Zahl an die Variable »Klang« angehängt und die WHILE-Bedingung erneut überprüft.

```
      SET Klang = Klang + Zeichen;
   END WHILE;
...
```

Zum Schluss wird getestet, ob die Zeichenkette, die zurückgeliefert werden soll, auch aus vier Zeichen besteht. Ist dies nicht der Fall, so werden die übrigen Stellen mit der Zahl 0 aufgefüllt. Darauf wird das Ergebnis mit der Anweisung RETURN zurückgeliefert.

```
   -- evtl. den Rest mit 0 auffüllen
   WHILE LEN( Klang ) < 4 DO
      SET Klang = Klang + '0';
   END WHILE

   RETURN( Klang );
END
```

Bevor wir das Kapitel beenden, wollen wir natürlich noch einmal sehen, ob unsere Funktion »Klang« auch das gewünschte Ergebnis liefert. Dazu fügen wir zunächst mit Hilfe der INSERT-Anweisung drei neue Kunden ein:

```
INSERT INTO Kunde VALUES
       (5, 'Meier',  'Petra',  'W', 'Magnusweg',  '61', 22222)
INSERT INTO Kunde VALUES
       (6, 'Meyer', 'Paula', 'W', 'Parkstr.', '81', 22222)
INSERT INTO Kunde VALUES
       (7, 'Maier', 'Pieter', 'M', 'Hollgang', '86', 22222)
```

Um nun alle Kunden zu finden, deren Name so ähnlich klingt wie »Mayer«, können wir folgende SELECT-Anweisung verwenden und erhalten dann alle drei eingefügten Kunden:

```
SELECT *
FROM   Kunde
WHERE  Klang(Name) = Klang('mayer')
```

12.6 Zusammenfassung

In diesem Kapitel haben wir die prozeduralen Sprachelemente von SQL kennen gelernt. Ursprünglich ist SQL als rein deklarative Programmiersprache konzipiert worden. Mit der Zeit und dem Erfolg prozeduraler Programmiersprachen wie C oder Pascal wurden in die RDBMS-Produkte und schließlich auch im ANSI SQL-Standard die wichtigsten prozeduralen Sprachelemente übernommen. Prozedurale Sprachelemente sind vor allem in Zusammenhang mit der Verwendung von Routinen sinnvoll. Routinen dienen dazu, SQL-Anweisungen unter einem bestimmten Namen vom RDBMS zu speichern. Über diesen Namen kann dann der komplette Anweisungsblock aufgerufen werden. Um Routinen flexibel einsetzen zu können, kann man Werte an die Routinen übergeben

und sich Werte zurückliefern lassen. Werte, die als Eingabeparameter an die Routine übergeben werden, kennzeichnet man mit dem Schlüsselwort IN. Werte, die als Ausgabeparameter an die Routine übergeben werden, kennzeichnet man dagegen mit dem Schlüsselwort OUT. Ein Übergabeparameter kann sowohl als Eingabe- und Ausgabeparameter dienen. Dann wird er mit dem Schlüsselwort INOUT gekennzeichnet.

Bei Routinen unterscheidet man zwischen Funktionen und Prozeduren. Funktionen liefern immer einen Wert über die Anweisung RETURN zurück, dessen Datentyp man bei Anlegen der Funktion angeben muss. Dadurch können Funktionen innerhalb von SQL-Anweisungen verwendet werden. In Kapitel 8 haben wir bereits »Built-in«-Funktionen kennen gelernt. »Built-in«-Funktionen sind vordefinierte Funktionen des RDBMS und entsprechen vom Aufruf eigenen selbst erstellten Funktionen.

Prozeduren können im Gegensatz zu Funktionen nicht innerhalb von SQL-Anweisungen verwendet werden und liefern auch keinen Wert zurück (es sei denn über einen Ausgabeparameter). Um Prozeduren aufzurufen, verwendet man die SQL-Anweisung CALL.

Der allgemeine Aufbau, um eine Prozedur anzulegen, sieht wie folgt aus:

```
CREATE PROCEDURE <routine name> ( parameter1, ..., parameter n)
```

Der allgemeine Aufbau, um eine Funktion anzulegen, sieht wie folgt aus:

```
CREATE FUNCTION <routine name> ( parameter1, ..., parameter n)
RETURNS <datentyp>
```

Um mehrere SQL-Anweisungen innerhalb einer Anweisung zu einem Block zusammenzufassen, verwendet man die Schlüsselwörter BEGIN…END.

Über die IF- oder die CASE-Anweisung kann festgelegt werden, welche Anweisungen bei Eintreten einer bestimmten Bedingung ausgeführt werden sollen.

Mit REPEAT oder WHILE kann ein Anweisungblock mehrmals hintereinander durchlaufen werden. Dabei wird über eine Abbruchbedingung festgelegt, wie oft der Anweisungsblock ausgeführt wird. Bei der REPEAT-Anweisung erfolgt die Überprüfung der Abbruchbedingung am Ende des Anweisungsblockes, bei der WHILE-Anweisung am Anfang. Um einen Anweisungsblock zu verlassen, bevor die Abbruchbedingung eintritt, kann man die LEAVE-Anweisung verwenden.

12.7 Aufgaben

Wiederholungsfragen

1) Worin besteht der wesentliche Unterschied zwischen einer Prozedur und einer Funktion?
2) Welche Unterschiede bestehen inhaltlich zwischen den beiden folgenden Prozeduren?

```
CREATE PROCEDURE beispiel(IN param1, OUT param2, INOUT param3)
CREATE PROCEDURE beispiel(param1, OUT param2, param3)
```

3) Worin besteht der Unterschied zwischen der CASE- und der IF-Anweisung? Kann jede IF-Anweisung auch mit einer CASE-Anweisung geschrieben werden bzw. umgekehrt?
4) Wozu dient die WHILE-Anweisung?
5) Worin besteht der Unterschied zwischen der REPEAT- und der WHILE-Anweisung?
6) Geben Sie ein Beispiel zur Verwendung der LEAVE-Anweisung!

Übungen

1) Ändern Sie die Prozedur »KindErfassen« aus Abschnitt 12.2, so dass anstelle einer IF-Anweisung die CASE-Anweisung verwendet wird!
2) Ändern Sie die Funktion »Klang« aus Abschnitt 12.5 so, dass anstelle einer CASE- die IF-Anweisung und anstelle der WHILE- die REPEAT-Anweisung verwendet wird!
3) Ändern Sie die Funktion »Klang« aus Abschnitt 12.5 so, dass auch der Buchstabe ‚ß' als ‚ss' im Algorithmus berücksichtigt wird!
4) Erweitern Sie die Funktion »Potenz« aus Abschnitt 12.3.4, so dass auch negative Zahlen und die Zahl 0 als Exponent übergeben werden können!
5) Schreiben Sie eine Prozedur »ErhoeheGehalt«, die das Gehalt aller Mitarbeiter so lange um jeweils immer 1% erhöht, bis das niedrigste Gehalt auf über 2.500 Eur gestiegen ist?
6) Schreiben Sie eine Funktion, die den durchschnittlichen Preis eines Werbeartikels zurückliefert!
7) Verwenden Sie die Funktion aus Übung 6., um eine SELECT-Anweisung zu schreiben, die die Beschreibung aller Werbeartikel mit ihrem Preis ausgibt und die Differenz zum Durchschnittspreis!
8) Schreiben Sie eine Prozedur »KundenBonus«, die ermittelt, welcher Kunde die insgesamt höchste Bestellsumme hat und fügen Sie dann für jede Bestellung, die dieser Kunde getätigt hat, einen Bestellposten mit dem Werbeartikel 1081 (Menge 1) hinzu!
9) Schreiben Sie eine Funktion mit dem Namen Klangunterschied, der zwei Wörter als Übergabeparameter übergeben werden. Die Funktion soll dann den Soundex-Code für beide Wörter ermitteln und dann die Differenz zwischen den beiden Zahlen (also nach dem ersten Buchstaben des Soundex-Codes) zurückliefern! (Hinweis: Verwenden Sie die »Built-in«-Funktion CAST).

13 Trigger

In Kapitel 13 sollen folgende Fragen geklärt werden:

- Wie können komplizierte Integritätsbedingungen unterstützt werden?
- Wie kann man automatisch Prozeduren aufrufen, wenn ein Datensatz aus einer Tabelle gelöscht, eingefügt oder geändert wird?
- Was sind Trigger?

13.1 Motivation

Frau Kart hat inzwischen Prozeduren geschrieben, über die man Bestellungen und die Kinder von Kunden erfassen kann. Obgleich sie Herrn Klein gezeigt hat, wie er die Prozeduren verwenden soll, trägt Herr Klein dennoch manchmal Datensätze in die Kind-Tabelle über die INSERT-Anweisung ein. Dadurch haben sich einige Fehler eingeschlichen: So existieren mehrere Einträge in der Kind-Tabelle mit einem Geburtsdatum, dass in der Zukunft liegt.

Bei der Durchsicht der Tabellen fallen Frau Kart die Fehler auf. Als vorbildliche Vorgesetzte ermahnt sie Herrn Kart, die von ihr geschriebenen Prozeduren zu verwenden. Dennoch ruft sie wieder Herrn Fleissig von der Unternehmensberatung an, der ihr bisher immer gute Ratschläge geben konnte. Auch diesmal kennt Herr Fleissig eine bessere Lösung für ihr Problem.

Er rät Frau Kart Folgendes: »Neben Prozeduren, die man explizit aufrufen muss, kann man Prozeduren schreiben, die bei Eintreten eines bestimmten Ereignisses für eine Tabelle automatisch ausgeführt werden. Solche Ereignisse sind das Einfügen, Ändern oder Löschen von Datensätzen. Man bezeichnet solche automatisch aufgerufenen Prozeduren als »Trigger« In ihrem Fall können Sie eine Trigger-Prozedur schreiben, die immer dann ausgeführt wird, wenn in die Kind-Tabelle ein Datensatz eingefügt oder geändert wird.«

13.2 »Trigger«

Bisher haben wir Datenbankensysteme als mehr oder weniger passive Softwaresysteme kennen gelernt. Eine Aktion wird von der Datenbank dann ausgeführt, wenn man es ihr explizit mitteilt. Trigger bieten nun die Möglichkeit, SQL-Anweisungen automatisch ausführen zu lassen, wenn in eine Tabelle ein neuer Datensatz eingefügt oder ein Datensatz geändert oder gelöscht wird. In diesem Fall übernimmt das Datenbanksystem eine aktive Rolle. Deshalb unterscheidet man häufig auch zwischen akti-

ven und passiven Datenbanksystemen, je nachdem ob das Datenbanksystem auch selbstständig SQL-Anweisungen ausführen kann oder nicht.

Der Begriff »Trigger« stammt aus dem Englischen und heißt übersetzt »Gewehrabzug«. Man spricht häufig davon, dass ein RDBMS »Trigger abfeuert«. Diese sehr dramatische Wortwahl soll verdeutlichen, dass ein RDBMS auf Ereignisse reagiert und Prozeduren selbstständig ausführt. Dadurch bieten Trigger die Möglichkeit, komplizierte Integritätsbedingungen in einer Datenbank abzubilden. Daneben können Trigger auch eingesetzt werden, um Änderungen an Tabellen zu protokollieren.

13.3 Erzeugen eines »Triggers«

Um eine Trigger-Prozedur zu erzeugen, verwendet man die SQL-Anweisung CREATE TRIGGER und um diese wieder zu löschen, natürlich die SQL-Anweisung DROP TRIGGER.

Beim Erzeugen gibt man dem Trigger einen Namen und bestimmt, für welche Tabelle dieser Trigger aufgerufen werden soll. Nach Angabe des Tabellennamens wird über die Schlüsselwörter BEFORE und AFTER festgelegt, ob die Prozedur vor der Datenänderung oder danach ausgeführt werden soll. Darauf folgen schließlich eines oder mehrere der Schlüsselwörter INSERT, DELETE oder UPDATE, je nachdem aufgrund welchen Ereignisses die Trigger-Prozedur aufgerufen werden soll.

Der allgemeine Aufbau zum Erzeugen eines Triggers sieht folgendermaßen aus:

```
CREATE TRIGGER <trigger name>
   BEFORE | AFTER
   INSERT | DELETE | UPDATE[ OF (Spaltenliste)]
   ON <Tabelle> [REFERENCING {OLD | NEW} {ROW |TABLE} [AS] <name>
   [ FOR EACH { ROW | STATEMENT ]
   BEGIN
...
```

Wie man dem Aufbau entnehmen kann, folgen nach der Angabe der Ereignisse die Schlüsselwörter FOR EACH ROW oder FOR EACH STATEMENT. Wenn man z. B. mit der SQL-Anweisung DELETE Datensätze löscht, so kann es sein, dass davon mehrere Datensätze betroffen sind, also mehrere Datensätze gelöscht werden. Gibt man FOR EACH STATEMENT an, so wird der Trigger nur einmal aufgerufen, egal wie viele Datensätze gelöscht werden sollen. Wird dagegen FOR EACH ROW verwendet, so wird der Trigger für jeden zu löschenden Datensatz aufgerufen.

Um zu ermitteln, welche Datensätze eingefügt, gelöscht oder geändert werden, erzeugt das RDBMS eine scheinbare Tabelle, in der diese Datensätze enthalten sind. Über das Schlüsselwort REFERENCING kann auf diese Datensätze zugegriffen werden, indem man einen Namen für diese scheinbare Tabelle vergibt.

Sehen wir uns hierzu nun ein Beispiel an. Immer wenn in die Tabelle »Kind« ein Datensatz eingefügt wird, soll überprüft werden, ob das angegebene Geburtsdatum des Kindes auch nicht in der Zukunft liegt.

```
CREATE TRIGGER KindPruefGeburtsdatum
BEFORE INSERT ON Kind REFERENCING NEW ROW AS neuerSatz
FOR EACH ROW
BEGIN
    IF neuerSatz.Geburtsdatum >= CURRENT_TIMESTAMP THEN
        ROLLBACK;
    END IF;
END
```

Über CREATE TRIGGER erzeugen wir einen Trigger mit dem Namen »KindPruefGeburtsdatum«. Die Trigger-Prozedur soll bei jedem Einfügen eines Datensatzes in die Tabelle »Kind« ausgeführt werden. Über das Schlüsselwort BEFORE legen wir fest, dass der Datensatz zum Zeitpunkt des Aufrufes noch nicht in die Datenbank eingefügt wurde. Dadurch haben wir die Möglichkeit über ROLLBACK das Einfügen zu verhindern. Da wir in diesem Trigger ausschließlich auf das Einfügen von Datensätzen reagieren, kann hinter REFERENCING nur das Schlüsselwort NEW folgen. OLD wird dann verwendet, wenn man auf die Daten eines zu löschenden Datensatzes zugreifen möchte, also bei Eintreten des Ereignisses DELETE. Reagiert man dagegen auf eine Änderung eines Datensatzes, so gibt NEW die geänderten Daten an und OLD die bisherigen Daten.

Sehen wir uns hierzu ein weiteres Beispiel an: Wir wollen jede Änderung am Geburtsdatum in der Tabelle Kind in der Tabelle Protokoll mitprotokollieren. Die Tabelle Protokoll soll die Spalten GeburtsdatumAlt, GeburtsdatumNeu, Benutzer und Datum enthalten. In den ersten beiden Spalten soll das bisherige und das neue Geburtsdatum festgehalten werden. In der Spalte Benutzer soll der Name desjenigen, der die Änderung vorgenommen hat und in der Spalte Datum der Zeitpunkt der Änderung gespeichert werden. Da wir nur protokollieren und die Daten nicht auf Gültigkeit prüfen, soll der Trigger erst nach Ausführen der Änderung aufgerufen werden. Das Erzeugen des Triggers sieht also folgendermaßen aus:

```
CREATE TRIGGER ProtokolliereAenderungGeburtsdatum
AFTER UPDATE ON Kind REFERENCING NEW ROW AS neuerSatz
                                 OLD ROW AS alterSatz
FOR EACH ROW
BEGIN
...
```

In »neuerSatz« befindet sich der Datensatz nach der Änderung und in »alterSatz« der Datensatz vor der Änderung. Also verwenden wir die INSERT-Anweisung, um das bisherige und das neue Geburtsdatum in der Tabelle »Protokoll« zu speichern:

```
    INSERT INTO Protokoll
    VALUES (alterSatz.Geburtsdatum, neuerSatz.Geburtsdatum,
            CURRENT_USER, CURRENT_TIMESTAMP);
END
```

Immer wenn FOR EACH ROW verwendet wird, kann man hinter REFERENCING auch nur auf eine Zeile referenzieren, da ja der Trigger für jede Zeile einzeln aufgerufen wird. Verwendet man dagegen FOR EACH STATEMENT, so muss man bei REFERENCING entsprechend NEW bzw. OLD TABLE angeben. In diesem Fall verweist man auf alle geänderten bzw. gelöschten Datensätze. Betrachten wir hierzu das Beispiel noch einmal, wollen aber diesmal den Trigger nur für jede Anweisung einmal aufrufen lassen. Die Trigger-Prozedur würde dann folgendermaßen aussehen:

```
CREATE TRIGGER ProtokolliereAenderungGeburtsdatum
AFTER UPDATE ON Kind REFERENCING NEW TABLE AS neueSaetze
                                 OLD TABLE AS alteSaetze
FOR EACH STATEMENT
BEGIN
  INSERT INTO Protokoll(GeburtsdatumAlt, GeburtsdatumNeu,
                        Benutzer, Datum)
    SELECT alteSaetze.Geburtsdatum, neueSaetze.Geburtsdatum,
           CURRENT_USER, CURRENT_TIMESTAMP
    FROM alteSaetze JOIN neueSaetze USING(Kundennummer, Vorname);
END
```

Im Gegensatz zu unserem vorherigen Trigger wird die Prozedur diesmal für jede Anweisung nur einmal aufgerufen. Trigger, die pro Datensatz aufgerufen werden, sind zwar einfacher zu programmieren, dafür beanspruchen sie in der Regel aber auch das RDBMS mehr.

Da Trigger nicht explizit aufgerufen werden, sollte man darauf achten, Trigger so zu programmieren, dass ihre Ausführungszeit möglichst kurz bleibt. Werden beim Löschen eines Datensatzes langwierige Berechnungen innerhalb eines Triggers von mehreren Minuten durchgeführt, so dauert natürlich auch das Löschen dieses einen Datensatzes mehrere Minuten. Außerdem kann ein Trigger, der nach Ausführen einer Aktion (also mit AFTER deklariert ist) aufgerufen wird, natürlich nicht zur Überprüfung von Daten verwendet werden. Dementsprechend kann man bei einem AFTER-Trigger kein ROLLBACK verwenden.

13.4 Zusammenfassung

In diesem Kapitel haben wir gesehen, dass ein RDBMS durch Trigger auch selbst aktiv werden kann. Trigger sind auf dem RDBMS-Server gespeicherte Prozeduren, die bei einer Datenänderung einer Tabelle automatisch vom RDBMS aufgerufen werden. Der Hauptzweck von Triggern dient vorwiegend der Abbildung komplizierter Geschäftsregeln, die man über normale Integritätsbedingungen nicht mehr ohne weiteres abbilden kann.

Der allgemeine Aufbau der Anweisung zum Erzeugen eines Triggers sieht folgendermaßen aus:

```
CREATE TRIGGER <trigger name>
   BEFORE | AFTER
   INSERT | DELETE | UPDATE[ OF (Spaltenliste)]
   ON <Tabelle> [REFERENCING {OLD | NEW} {ROW |TABLE} [AS] <name>
   [ FOR EACH { ROW | STATEMENT ]
```

Neben einer Bezeichnung für den Trigger muss angegeben werden, ob der Trigger vor oder nach Ausführen der Datenänderung aufgerufen werden soll (BEFORE oder AFTER). Weiterhin muss spezifiziert werden, bei welchen DML-Anweisungen der Trigger aufgerufen werden soll (INSERT, DELETE, UPDATE). Soll der Trigger bei einem UPDATE aufgerufen werden, so kann der Trigger entweder auf die Änderung aller oder ganz bestimmter Spalten reagieren. Dazu führt man hinter dem Schlüsselwort UPDATE die gewünschten Spalten auf. Über das Schlüsselwort ON gibt man an, auf welche Tabelle der Trigger sich bezieht. Über OLD und NEW schließlich kann auf die neuen bzw. die bisherigen Werte zugegriffen werden.

Ein Trigger wird bei einer Datenänderung entweder für jeden geänderten Datensatz einzeln (FOR EACH ROW) oder für eine einzelne DML-Anweisung (FOR EACH STATEMENT) aufgerufen.

13.5 Aufgaben

Wiederholungsfragen

1) Wie muss die REFERENCING-Klausel lauten, wenn FOR EACH STATEMENT angegeben wurde?
2) Wie muss die REFERENCING-Klausel lauten, wenn FOR EACH ROW angegeben wurde?
3) Sie wollen einen Trigger schreiben, der auf das Löschen von Datensätzen reagiert. Wie können Sie im Trigger auf die zu löschenden Daten zugreifen? Geben Sie ein Beispiel für die REFERENCING-Klausel!
4) Sie wollen einen Trigger schreiben, der auf das Einfügen von Datensätzen reagiert. Wie können Sie im Trigger auf die einzufügenden Daten zugreifen? Geben Sie ein Beispiel für die REFERENCING-Klausel!
5) Worin bestehen Vor- und Nachteile der Schlüsselwörter FOR EACH ROW bzw. FOR EACH STATEMENT?
6) Was ist mit einem aktiven Datenbanksystem gemeint?

Übungen

1) Erweitern Sie den Trigger »KindPruefGeburtsdatum« so, dass auch überprüft wird, ob das Geburtsdatum des Kindes aktueller als das des dazugehörigen Elternteils ist!
2) Schreiben Sie einen Trigger »KindPruefGeburtsdatum2«, der bei Änderung eines Datensatzes in der Kind-Tabelle das zu ändernde Geburtsdatum auf Plausibilität überprüft!

3) Schreiben Sie einen Trigger »PruefSitzplatzFrei«, der bei Einfügen eines Bestellpostens überprüft, ob der gewünschte Sitzplatz frei ist. Ist der Sitzplatz nicht als »frei« gekennzeichnet, soll das Einfügen verhindert werden!
4) Schreiben Sie einen Trigger »PruefVorstellungsTermin«, der bei Einfügen eines Bestellposten überprüft, ob der Termin der gewünschten Vorstellung nicht in der Vergangenheit liegt!
5) Verhindern Sie über einen Trigger, dass bei Einfügen oder Ändern eines neuen Mitarbeiter-Datensatzes das Gehalt des Mitarbeiters nicht höher ist als das Gehalt des Geschäftsführers, Herrn Kowalski!
6) Überprüfen Sie über einen Trigger bei Einfügen oder Ändern eines Datensatzes in die Tabelle »Sitzplatz«, ob der Preis gleich ist, wenn es sich bei einer Vorstellung um die gleiche Veranstaltung im gleichen Haus handelt. Ist der Preis nicht gleich, so soll in die Spalte Beschreibung der Tabelle »Problem« eine Zeichenkette eingefügt werden, die das Problem beschreibt! (Hinweis: Legen Sie vorher eine Tabelle »Problem« an, die aus einer Spalte des Datentyps VARCHAR(250) besteht.)

14 »User Defined Types« (UDT)

In Kapitel 14 sollen folgende Fragen geklärt werden:

- Was ist mit objektorientiert und objektrelational gemeint?
- Wie sieht die Zukunft relationaler Datenbanksysteme aus?
- Was sind benutzerdefinierte Typen (»User Defined Types«)?
- Was sind Methoden?

14.1 Motivation

Frau Kart ist zwar zufrieden mit der enstandenen Datenbank, dennoch bemängelt sie, dass sie oder ihre Kollegen direkt SQL-Anweisungen eingeben müssen. Also vereinbart sie einen Termin mit Herrn Fleissig von der Unternehmensberatung und ihrem Geschäftsführer.

Aus dem Gespräch ergibt sich, dass die Unternehmensberatung einen neuen Auftrag zur Entwicklung eines Programmes erhalten soll. Dieses Programm soll unter dem Betriebssystem Windows lauffähig sein und die Funktionalitäten eines Abrechnungssystems für Eintrittskarten enthalten.

Herr Fleissig, erfreut darüber, einen neuen Auftrag aquiriert zu haben, benachrichtigt nach dem Gespräch seinen Geschäftsführer: »Herr Warner, wir sollen für »KartoFinale« ein Programm zur Abrechnung erstellen. Ich würde folgenden Ablauf empfehen: Zunächst ermittle ich in den nächsten vier Wochen die Anforderungen an das Programm und werde unser UML-Datenmodell um Funktionen erweitern, so dass wir Daten- und Funktionsmodell dann vorliegen haben. Danach werde ich mit dem Entwicklungsleiter unserer Softwareabteilung sprechen und ihm empfehlen, die meisten der Funktionen auf dem RDBMS selbst zu entwickeln. Inzwischen bietet SQL nämlich mit dem neuen Standard hervorragende Eigenschaften, um objektorientiert mit SQL zu entwickeln. Das Windows-Programm selbst sollte unser neuer Kollege, Herr Klever, in C++ entwickeln. Über C++ kann er dann die Funktionen auf dem RDBMS-Server direkt aufrufen.«

14.2 Objektorientierung

Die Kapitel 6 bis 13 stellen auch die geschichtliche Entstehung von SQL dar. Zuerst wurde SQL als reine Datenbankabfragesprache entworfen. Sie sollte dazu dienen, dass Anwender selbst SQL-Anweisungen eingeben und Ergebnisse vom RDBMS zurücker-

halten. In diesem ersten Schritt war SQL eine rein deklarative Programmiersprache, zwar leistungsfähig, aber teilweise nicht besonders flexibel.

Das änderte sich 1996 durch nachträgliches Hinzufügen prozeduraler Sprachelemente zum SQL-92 Standard. SQL war nun sowohl eine deklarative als auch eine prozedurale Sprache. SQL wurde jetzt selten von Endanwendern selbst verwendet. Vielmehr wurde es zur Entwicklung von Softwareprogrammen benutzt. Dazu wurden SQL-Anweisungen in andere Programmiersprachen wie C, C++, Java, Pascal oder Visual Basic eingebettet und von dem Anwendungsprogramm an den RDBMS-Server gesendet, um von diesem eine Ergebnistabelle zu erhalten.

Inzwischen hatte sich aber auch schon ein anderes Programmierkonzept durchgesetzt, die objektorientierte Programmierung. Die meisten Änderungen bzw. Erweiterungen am SQL-Standard im Jahr 1999 betrafen deshalb vorwiegend die Erweiterung von SQL um objektorientierte Eigenschaften. SQL als objektorientierte Sprache ist für den Anwenungsentwickler sicherlich von größerer Bedeutung als für den Endanwender, zumal Endanwender SQL heutzutage in der Regel nicht mehr direkt nutzen. Zu diesem Zeitpunkt gab es bereits rein objektorientierte Datenbanksysteme. Diese enthielten eigene Datenbanksprachen zum Abfragen der Datenbank. Da man mit SQL:1999 auch weiterhin deklarativ und prozedural entwickeln kann, werden Datenbanksysteme, die auf relationalen Datenbanken basieren, deshalb zur Unterscheidung nicht als objektorientierte, sondern als objektrelationale Datenbanksysteme bezeichnet. Die häufig anzutreffende Meinung, SQL:1999 sei deshalb nicht objektorientiert, ist schlichtweg falsch. SQL:1999 enthält alle wesentlichen Eigenschaften, die eine objektorientierte Sprache ausmachen. Dennoch ist zu anzumerken, dass alle bekannten RDBMS-Produkte die objektorientierten Eigenschaften von SQL:1999 bis zum jetzigen Zeitpunkt noch gar nicht oder nicht vollständig abdecken (am nächsten kommt IBM DB2 hier sicherlich dem SQL:1999 Standard). Dieses Kapitel soll deshalb mehr als Überblick dienen, wie die objektorientierten Eigenschaften zukünftig in den RDBMS-Produkten aussehen werden. Bevor wir uns dies in SQL selbst ansehen, folgt eine kurze Einführung in das Thema Objektorientierung. Wer sich bereits mit objektorientierten Konzepten auskennt, kann den Rest dieses Abschnitts also überspringen.

Bis zum Kapitel 11 haben wir gesehen, wie man in SQL Daten einfügt, ändert und abfragt. Wir haben SQL bis dahin ausschließlich von der Datensicht betrachtet. In Kapitel 12 wurde die Entwicklung von Routinen in SQL vorgestellt. In diesem Fall haben wir die Datensicht vernachlässigt und und uns auf die Funktionssicht konzentriert. Objektorientierung fasst nun Daten und Funktionen zu einer Einheit, einem Objekt, zusammen. Ein Objekt entspricht, wie wir es bereits in ersten Kapiteln zur Datenmodellierung kennen gelernt haben, weitgehend einer Entität. Im Gegensatz zur Entität wird ein Objekt aber eben nicht nur durch Daten bzw. Attribute beschrieben, sondern auch durch Funktionen, die dieses Objekt ausführen kann. Funktionen, die an ein Objekt gebunden sind, werden zur Unterscheidung als Methode bezeichnet.

In Kapitel 3 haben wir zur Datenmodellierung bereits die UML kennen gelernt. Dort wurde auch erwähnt, dass ein Objekt bzw. eine Klasse durch eine Rechteck dargestellt wird, das in drei Bereiche aufgeteilt ist. Im oberen Bereich erscheint der Name der Klasse, im mittleren Bereich die Attribute und im unteren Bereich die Methoden. UML ist die einzige Modellierung, die wir kennen gelernt haben, mit der man also sowohl

Daten- als auch Funktionsmodellierung vornehmen kann. Um die Konzepte der Objektorientierung zu verstehen, verwenden wir die Notation der UML. Betrachten wir hierzu aus unserem Fallbeispiel den Objekttyp bzw. die Klasse Kunde. Ein Kunde besteht aus mehreren Attributen, wie »Name«, »Vorname«, »Strasse«, »Hausnummer«, »Ort« usw. Aus der realen Welt wissen wir aber, dass ein Kunde auch bestimmte Aktionen ausführt: Er kann eine Bestellung aufgeben, er kann ein Kind bekommen, er kann umziehen und damit eine neue Adresse erhalten usw. Objektorientiert heißt also: Man betrachtet nicht nur die Attribute eines Objektes, sondern auch dessen Funktionen. In einem UML-Klassendiagramm würde das dann folgendermaßen aussehen:

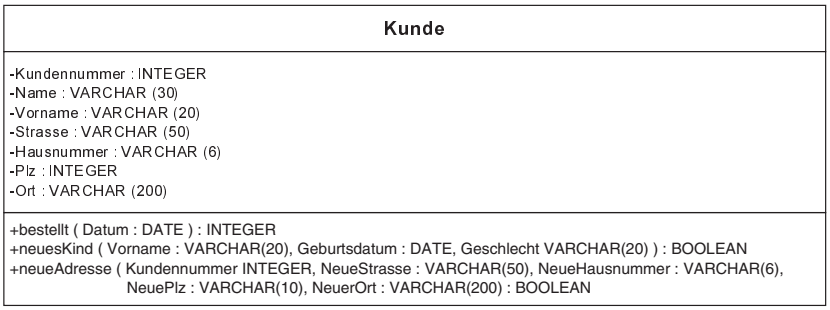


Abbildung 14.1: Klasse »Kunde«mit Attributen und Methoden

Methoden einer Klasse verändern nun in der Regel die zur Klasse gehörigen Attribute. Über eine objektorientierte Sprache kann man Attribute als privat deklarieren, so dass diese nur von Methoden der Klasse selbst geändert werden können. Wollen Sie also den Namen oder Vornamen eines Kunden ändern, so müssten Sie eine Methode »AendereName« erstellen. Diese Methode kann dann auf die klasseneigenen Attribute zugreifen.

Doch warum so umständlich, warum kann man nicht direkt auf die Attribute zugreifen und diese verändern?

Zwei wesentliche Gründe sprechen dagegen. Nehmen wir einmal an, Sie haben ein Anwendungsprogramm erstellt, in der an mehreren Stellen direkt auf das Attribut Postleitzahl des Kunden zugegriffen wird. Bisher haben wir dieses Attribut als Ganzzahl deklariert. Nun soll das Attribut Plz aber 8-stellig sein und alphanumerisch. Da Zeichenketten in SQL in Anführungsstrichen dargestellt werden, müssen nun überall dort, wo auf Plz verwiesen wird, Anführungsstriche verwendet werden. Das kann bei einer komplexen Anwendung sehr aufwändig und fehleranfällig sein.

Betrachten wir nun den anderen Fall: Änderungen an der Postleitzahl erfolgen nur in der Methode »neueAdresse« selbst. Diesmal müssen Sie ihr Anwendungsprogramm nur an einer Stelle ändern, nämlich in der Methode »neueAdresse«. Sie sehen also, dass der Änderungsaufwand wesentlich geringer und daher auch bei weitem nicht so fehleranfällig ist. Gerade bei sehr komplexen Anwendungen wird der Wartungsaufwand eines Anwendungsprogramms erheblich minimiert. Der zweite Grund, Attribute nicht direkt zu ändern, besteht in der Lesbarkeit des Anwendungsprogramms. Verwendet

man eine Methode mit einem dem Inhalt entsprechenden Namen wie »neueAdresse«, so ist dem Programmierer in der Regel klar, dass mit dieser Methode die Adresse des Kunden geändert wird. Das Programm wird dadurch besser verständlich und damit auch leichter wartbar.

Doch das ist noch nicht alles, was Objektorientierung uns zu bieten hat! Schließlich könnten wir das oben beschriebene Problem ja auch durch Verwendung einer gespeicherten Prozedur, wie wir sie in Kapitel 12 kennen gelernt haben, lösen. Routinen wie in Kapitel 12 sind jedoch nicht an bestimmte Objekttypen gebunden. Dadurch neigt man dazu, Funktionen zu schreiben, die sich auf mehrere Objekttypen beziehen. Dies ist deshalb nachteilig, weil man dann Programmzeilen erzeugt, die in der Regel nicht wiederverwendbar sind, also von anderen Anwendungsprogrammen nicht eingesetzt werden können. Ein wesentlicher Punkt bei der Entstehung objektorientierter Sprachen ist also die Wiederverwendung und Wartbarkeit von Anwendungsprogrammen. Neben der Zusammengehörigkeit von Daten und Funktionen (Datenkapselung) wird dies auch erreicht durch die so genannte Vererbung. Betrachten wir hierzu wieder unser Fallbeispiel: Neben der Klasse »Kunde« haben wir eine weitere Klasse »Mitarbeiter«. Vergleichen wir diese beiden Klassen miteinander, so stellen wir fest, dass es viele Übereinstimmungen zwischen den Attributen und Methoden gibt, aber auch einige Spezialisierungen.

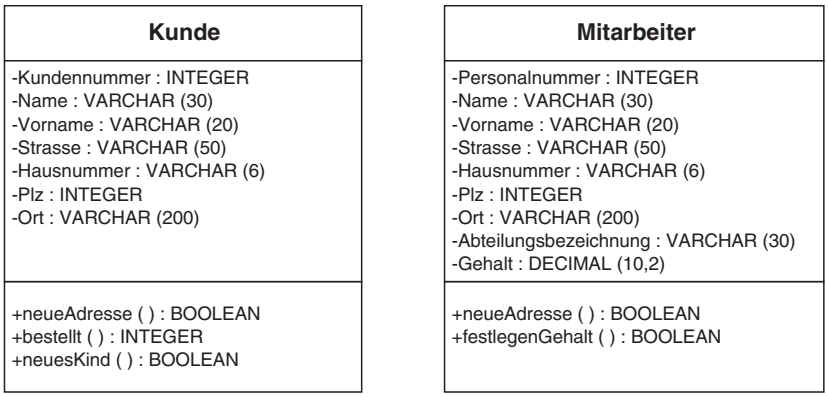


Abbildung 14.2: Gleiche Attribute und Methoden zwischen »Kunde« – »Mitarbeiter«

So kommen die Attribute »Kundennummer« bzw. »Personalnummer«, »Name«, »Vorname« und die Attribute für die Adresse in beiden Klassen vor. »Mitarbeiter« hat dagegen zwei Attribute, die in der Klasse »Kunde« nicht vorkommen, nämlich »Abteilungsbezeichnung« und »Gehalt«. Bei den Methoden existiert »neueAdresse« in beiden Klassen, ansonsten haben beide Klassen weitere spezielle Methoden.

Bereits bei der Datenmodellierung haben wir so etwas schon einmal gesehen, nämlich bei Sub- und Supertypen. Da viele übereinstimmende Merkmale zwischen den Klassen existieren, können wir also eine neue so genannte Oberklasse bilden, die wir Person nennen. Person enthält nun alle Attribute und Methoden, die sowohl in »Kunde« als auch in »Mitarbeiter« auftreten.

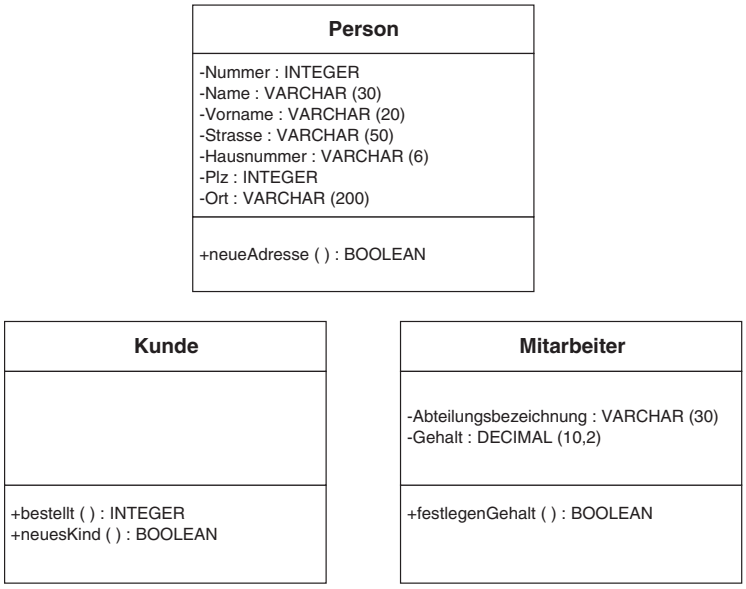


Abbildung 14.3: »Kunde« und »Mitarbeiter« »erben« von »Person«

Die Klassen »Kunde« und »Mitarbeiter« »erben« also die Eigenschaften ihrer Oberklasse »Person«. Da die Methode »neueAdresse« nur noch in »Person« vorkommt, muss diese nur einmal gespeichert werden und nicht wie zuvor redundant bei »Kunde« und »Mitarbeiter«. Vererbung dient also dazu, Verallgemeinerungen abzubilden und dadurch den Programmieraufwand zu minimieren.

Neben der Datenkapselung und der Vererbung besteht die letzte wesentliche Eigenschaft einer objektorientierten Sprache in der Unterstützung des Polymorphismus (»Vielgestaltigkeit«). Wir wollen uns den Polymorphismus wieder an unserem Fallbeispiel ansehen. Kunden und Mitarbeiter sollen eine Methode »kuendigen« besitzen. Bei Aufruf der Methode »kuendigen« der Klasse »Kunde« soll der Kunde aus der Datenbank gelöscht werden, inklusive der Datensätze der Kinder und seiner Bestellungen. Bei einem Mitarbeiter dagegen, soll die Methode »kuendigen« den Datensatz aus der Tabelle »Mitarbeiter« löschen und den Fremdschlüssel in der Tabelle »Bestellung« auf NULL setzen. Obgleich die Methode den gleichen Namen hat, ist ihre Funktionalität unterschiedlich bzw. polymorph. Für unser UML-Diagramm bedeutet dies, dass wir eine zusätzliche Methode »kuendigen« in »Person« eintragen, aber auch in »Kunde« und »Mitarbeiter«. Der Programmcode dieser Methode sieht bei beiden Klassen natürlich unterschiedlich aus.

Wir haben damit die wichtigsten Eigenschaften der Objektorientierung kennen gelernt:

- Datenkapselung
- Vererbung
- Polymorphismus

Wir wollen uns im Folgenden ansehen, wie SQL:1999 diese Eigenschaften umgesetzt hat.

14.3 Klassen

Klassen werden in SQL:1999 im Gegensatz zu den meisten anderen objektorientierten Sprachen nicht als Klassen, sondern als benutzerdefinierte Typen (»User Defined Types« – UDT) bezeichnet. UDT's werden über die SQL-Anweisung CREATE TYPE erzeugt und über DROP TYPE wieder gelöscht. Zum Anlegen der Klasse »Person« lautet die SQL-Anweisung also folgendermaßen:

```
CREATE TYPE cPerson AS
(
...
```

Hier wurde bewusst das kleine »c« vor den Klassennamen gesetzt, um zwischen der Klassendefinition und der eigentlichen späteren Tabelle, in der die Daten gespeichert werden sollen, zu unterscheiden. Innerhalb der CREATE TYPE-Anweisungen werden dann die Attribute definiert, so wie wir es von CREATE TABLE her kennen:

```
CREATE TYPE cPerson AS
(
    Nummer     INTEGER,
    Name       VARCHAR (30),
    Vorname    VARCHAR (20),
    Strasse    VARCHAR (50),
    Hausnummer VARCHAR (6),
    Plz        INTEGER,
    Ort        VARCHAR (200),
...
```

Bisher unterscheidet sich das Anlegen eines benutzerdefinierten Typs nicht sonderlich vom Anlegen einer Tabelle. Ein wesentlicher Unterschied besteht jedoch jetzt schon: Das Anlegen eines UDT's erzeugt ausschließlich eine »Schablone« zum eigentlichen Anlegen einer Tabelle. Letztendlich arbeitet ein RDBMS nur mit Tabellen, ein UDT kommt also immer innerhalb einer CREATE TABLE-Anweisung vor. Bevor wir uns das allerdings ansehen, müssen wir erst einmal unsere CREATE TYPE-Anweisung zu Ende schreiben. Eine wesentliche Eigenschaft der Klasse fehlt noch, nämlich seine Methoden. In unserem Fall hat »cPerson« nur die Methode »neueAdresse«:

```
CREATE TYPE cPerson AS
(
    Nummer     INTEGER,
    Name       VARCHAR (30),
    Vorname    VARCHAR (20),
    Strasse    VARCHAR (50),
    Hausnummer VARCHAR (6),
    Plz        INTEGER,
    Ort        VARCHAR (200),
    METHOD neueAdresse( Nr INTEGER,
                        NeueStrasse VARCHAR(50),
```

```
                        NeueHausnummer VARCHAR(6),
                        NeuePlz INTEGER, NeuerOrt VARCHAR(200) )
                      RETURNS BOOLEAN
)
```

Damit haben wir den UDT erzeugt. Allerdings haben wir noch nicht beschrieben, wie der Programmcode der Methode »neueAdresse« denn nun aussieht. Eine Methode wird ähnlich wie eine Funktion erzeugt. Schließlich sind Funktionen und Methoden sehr ähnlich, mit dem Unterschied, dass Methoden immer an einen UDT gebunden sind. Zum Erzeugen einer Methode verwendet man die SQL-Anweisung CREATE METHOD. Die SQL-Anweisungen dieser Methode könnten folgendermaßen aussehen:

```
CREATE METHOD neueAdresse(Nr INTEGER,
                          NeueStrasse VARCHAR(50),
                          NeueHausnummer VARCHAR(6),
                          NeuePlz INTEGER, NeuerOrt VARCHAR(200) )
RETURNS BOOLEAN FOR cPerson
BEGIN
    UPDATE Person SET
        Strasse = NeueStrasse,
        Hausnummer = NeueHausnummer,
        Plz = NeuePlz,
        Ort = NeuerOrt
    WHERE Nummer = Nr;
END;
```

Zunächst wird nach CREATE METHOD noch einmal der Aufbau der Methode mit seinem Rückgabewert deklariert. Danach wird über das Schlüsselwort FOR festgelegt, an welchen UDT die Methode gebunden werden soll. Schließlich folgt der Anweisungsblock, der den Programmcode der Methode darstellt. In diesem Fall besteht die Methode nur aus einer SQL-Anweisung, die über UPDATE den Datensatz ändert.

Bisher haben wir ja nur eine »Schablone« erzeugt, die man über CREATE TABLE verwenden kann, um eine Tabelle physikalisch zu erzeugen. Will man eine Tabelle erzeugen, die auf einem UDT basiert, so gibt man einfach an:

```
CREATE TABLE Person OF cPerson;
```

Damit haben wir eine Tabelle »Person« erzeugt, die die Attribute der Klasse »cPerson« und dessen Methoden enthält.

14.4 Zugriff auf Attribute und Methoden

Doch wie fügt man nun Daten in eine solche erzeugte Tabelle ein und wie kann man diese Daten über SELECT wieder ausgeben?

Zum Einfügen von Daten in diese Tabelle muss man zunächst eine Variable der Klasse cPerson deklarieren. Dieser Variablen weist man die Daten zu, die eingefügt werden sollen und verwendet diese Variable dann beim Einfügen:

```
DECLARE p cPerson;

SET p = cPerson();
SET p.Nummer    = 8;
SET p.Name      = 'Petri';
SET p.Vorname   = 'Paul';
SET p.Strasse   = 'Malerstr.';
SET p.Hausnummer= '26';
SET p.Plz       = 22222;
SET p.Ort       = 'Karlstadt';

INSERT INTO Person VALUES (p);
```

Es gibt noch eine etwas einfachere Variante über das Schlüsselwort NEW. Damit sieht das Ganze folgendermaßen aus:

```
INSERT INTO Person VALUES (NEW( 8, 'Petri', 'Paul', 'Malerstr.',
                               '26', 22222, 'Karlstadt')
```

14.5 Vererbung

Als Letztes wollen wir uns ansehen, wie Vererbung im aktuellen SQL Standard umgesetzt wurde. Hierzu gibt es ein einziges Schlüsselwort, nämlich UNDER. Um den UDT »cMitarbeiter« zu erzeugen und alle Eigenschaften des UDT »cPerson« auf diese zu vererben, erzeugt man den UDT wie folgt:

```
CREATE TYPE cMitarbeiter UNDER cPerson
(
    Abteilungsbezeichnung VARCHAR(30),
    Gehalt DECIMAL(10,2),
    METHOD festlegenGehalt( neuesGehalt DECIMAL(10,2) )
                          RETURNS DECIMAL(10,2)
)
```

Über das Schlüsselwort UNDER erhält der UDT »cMitarbeiter« automatisch alle Attribute und Methoden des UDT »cPerson«.

Das Schlüsselwort UNDER lässt sich auch ohne die Verwendung von UDT's direkt für Tabellen einsetzen. Zunächst erzeugen wir die Tabelle »Person«, so wie wir es in Kapitel 6 kennen gelernt haben und anschließend die Tabelle »Mitarbeiter«. Dabei wird über UNDER auf die Tabelle »Person« verwiesen. Beim Anlegen der Tabelle »Mitarbeiter« werden alle speziellen Attribute dieser angegeben:

```
CREATE TABLE Person
(
    Nummer     INTEGER,
    Name       VARCHAR (30),
    Vorname    VARCHAR (20),
    Strasse    VARCHAR (50),
    Hausnummer VARCHAR (6),
    Plz        INTEGER,
    Ort        VARCHAR (200),
)

CREATE TABLE Mitarbeiter UNDER Person
(
    Abteilungsbezeichnung VARCHAR(30),
    Gehalt DECIMAL(10,2)
)
```

14.6 Zusammenfassung

Im letzen Kapitel haben wir uns angesehen, wie sich die Verwendung von SQL demnächst verändern wird bzw. teilweise schon verändert hat. Mit der Verbreitung objektorientierter Konzepte wurden diese auch im aktuellen SQL Standard übernommen, so dass SQL heutzutage sowohl eine deklarative und prozedurale als auch eine objektorientierte Sprache darstellt.

Im ersten Abschnitt wurde die Philosophie, die hinter Objektorientierung steht, erläutert. Danach haben wir gesehen, wie die einzelnen Eigenschaften der Objektorientierung in SQL umgesetzt wurden. In benutzerdefinierten Typen (UDT) werden sowohl Daten als auch Funktionen (Methoden) als eine Einheit gespeichert. Auf Basis eines UDT wird eine Tabelle erzeugt, in der die Daten gespeichert werden, auf die man über die Methoden zugreift.

Einen UDT erzeugt man über die SQL-Anweisung CREATE TYPE. Hierbei gibt man zunächst die Attribute und die Methoden an. Danach müssen die einzelnen Methoden über CREATE METHOD mit »Leben« gefüllt werden.

Über das Schlüsselwort UNDER kann die Idee der Vererbung realisiert werden. Sowohl UDT's, als auch Tabellen können damit in einer Hierarchie abgebildet werden.

14.7 Aufgaben

Wiederholungsfragen

1) Worin besteht der Unterschied zwischen Prozedur, Funktion und Methode?
2) Wie erzeugt man einen benutzerdefinierten Typ in SQL?
3) Wie erzeugt man eine Methode?

4) Wie unterstützt SQL das objektorientierte Konzept der Vererbung?
5) Nennen Sie Beispiele für Methoden der Klasse »Bestellung«!
6) Nennen Sie Beispiele für Methoden der Klasse »Vorstellung«!
7) Was ist mit »Polymorphismus« gemeint? Geben Sie ein Beispiel!
8) Was könnte mit »Klassenhierachie« gemeint sein?

Übungen

1) Erzeugen Sie einen UDT zum Speichern einer Adresse unter dem Namen »cAdresse«. Der UDT soll die Spalten »Strasse«, »Hausnummer«, »Plz« und »Ort« enthalten.
2) Erzeugen Sie eine Tabelle »Person«, die aus den Spalten »Name«, »Vorname« und dem UDT »cAdresse« besteht!
3) Erzeugen Sie eine Tabelle »Musiker«, die aus der Spalte »Musikinstrument« besteht und alle weiteren Spalten von der Tabelle »Person« aus Übung 2 erbt!

Literaturverzeichnis

[Achilles97] Achilles, Albrecht: »SQL«, Oldenbourg, 1997

[ANSI99-1] ANSI/ISO/IEC: » 9075-1-1999 Database Languages – SQL – Part 1: Framework (SQL/Framework)«, http://www.ansi.org, 1999

[ANSI99-2] ANSI/ISO/IEC: » 9075-2-1999 Database Languages – SQL – Part 2: Foundation (SQL/Foundation)«, http://www.ansi.org, 1999

[ANSI99-3] ANSI/ISO/IEC: » 9075-3-1999 Database Languages – SQL – Part 3: Call-Level Interface (SQL/CLI)«, http://www.ansi.org, 1999

[ANSI99-4] ANSI/ISO/IEC: » 9075-4-1999 Database Languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM)«, http://www.ansi.org, 1999

[ANSI99-5] ANSI/ISO/IEC: » 9075-3-1999 Database Languages – SQL – Part 5: Host Language Bindings (SQL/Bindings)«, http://www.ansi.org, 1999

[Barker90] Barker, R: »CASE*Method: Tasks and Deliverables, Addison-Wesley, 1990

[Batini92] Batini, Carlo / Ceri, Stefano / Navathe, Shamkant B.: »Conceptual Database Design«, Benjamin Cummings Publishing, 1992

[Booch99] Booch, Grady / Rumbaugh, Jim / Jacobson, Ivar: »Das UML-Benutzer-handbuch«

[Carlis01] Carlis, John V. / Maguire, Joseph D.: »Mastering Data Modeling«, Addison-Wesley, 2001

[Chen76] Chen, Peter P.: »The Entity-Relationship Model – Toward a Unified View of Data«, ACM Trans. Database System 1, No. 1, 1976

[Codd70] Codd, E. F.: »A Relational Model for large Shared Data Banks«, Comm. ACM 13, No. 6, 1970

[Date95] Date, Chris J.: »An Introduction to Database Systems«, Addison Wesley, 1995

[Date98] Date, Chris J. / Darwen, Hugh: »SQL – Der Standard«, Addison Wesley, 1998

[Delaney99] Delaney, Karen / Soukoup Ron: »Inside Microsoft SQL Server 7.0«, Microsoft Press, 1999

[Eirund00] Eirund, Helmut / Kohl, Ullrich: »Datenbanken – leicht gemacht«, Teubner, 2000

[Elmasri00] Elmasri, Ramez / Navathe, Shamkant B.: »Fundamentals of Database Systems«, Addison Wesley, 2000

[Fortier99] Fortier, Paul J.: »SQL-3 – Implementing the Object-Relational Database«, McGraw-Hill, 1999

[Fowler00]	Fowler, M. / Scott, K.: »UML konzentriert«, Addison Wesley, 2000
[Gorman00]	Gorman, Michal M.: »Great News: The Relational Model is dead«, http://www.tdan.com, 2000
[Gorman01]	Gorman, Michal M.: »Is SQL a real Standard anymore«, http://www.tdan.com, 2001
[Groff99]	Groff, James R. / Weinberg, Paul N.: »SQL: The Complete Reference«, McGraw-Hill, 1999
[Gulutzan99]	Gulutzan, Peter / Pelzer, Trudy: »SQL-99 Complete, Really «, R&D Books, 1999
[Halpin99]	Halpin, Terry: »Entity Relationship Modeling from an ORM Perspective: Part 1«,http://www.inconcept.com, 1999
[Halpin00-1]	Halpin, Terry: »Entity Relationship Modeling from an ORM Perspective: Part 2«,http://www.inconcept.com, 2000
[Halpin00-2]	Halpin, Terry: »Entity Relationship Modeling from an ORM Perspective: Part 3«,http://www.inconcept.com, 2000
[Halpin00-3]	Halpin, Terry: »Entity Relationship Modeling from an ORM Perspective: Part 4«,http://www.inconcept.com, 2000
[Halpin00-4]	Halpin, Terry: »Entity Relationship Modeling from an ORM Perspective: Part 5«,http://www.inconcept.com, 2000
[Hay99-1]	Hay, David C.: »A Comparison of Data Modeling Techniques«, Essential Strategies Inc., 1999
[Hay99-2]	Hay, David C.: »Object Orientation and Information Engineering: Object and Data Modeling«, Essential Strategies Inc., 1999
[Heuer97]	Heuer, Andreas / Saake, Gunter: »Datenbanken«, Thomson Publishing, 1997
[Hotek00]	Hotek, Michael: »Introduction to TSQL«, http://www.mssqlserver.com, 2000
[Kleinschmidt97]	Kleinschmidt, Peter / Rank, Christian: »Relationale Datenbanksysteme«, Springer, 1997
[Kuhlmann99]	Kuhlmann, Gregor / Müllmerstadt, Friedrich: »SQL«, Rowohlt, 1999
[Lackes01]	Lackes, R. / Brandl, W. / Siepermann, M.: »Handling von Informationssystemen mit SQL«, Springer, 2001
[Mattos99]	Mattos, Nelson M. u.a.: »SQL99, SQL/MM and SQLJ: An Overview of the SQL Standards«, http://www.wiscorp.com, 1999
[Matthiessen97]	Mathiessen, Günter / Unterstein, Michael: »Relationale Datenbanken und SQL«, Addison Wesley, 1997
[Meier01]	Meier, Andreas: »Relationale Datenbanken«, Springer, 2001
[Melton01]	Melton, Jim / Simon, Alan R.: »SQL:1999«, Morgan Kaufmann, 2001
[Melton99]	Melton, Jim: »SQL:1999 – A Tutorial«, http://www.wiscorp.com, 2001

[Microsoft99-1] Microsoft Corporation, »Transact-SQL Sprachverzeichnis«, Microsoft Press, 1999

[Microsoft99-2] Microsoft Corporation, »Microsoft SQL Server 7.0 – Die technische Referenz«, Microsoft Press, 1999

[Microsoft99-3] Microsoft Corporation, »Microsoft SQL Server 7.0 – Datenbank-implementierung«, Microsoft Press, 1999

[Moos97] Moos, Alfred / Daues, Gerhard: »Datenbank-Engineering«, vieweg, 1997

[Muller99] Muller, Robert J.: »Database for Smarties : using UML for data modeling«, Morgan Kaufmann, 1999

[Oracle99] Oracle Corporation, »Oracle Designer Tutorial Release 6.0«, Oracle, 1999

[Panny00] Panny, Wolfgang: »Einführung in den Sprachkern von SQL-99«, Springer, 2000

[Rational01] Rational Company: »Introducing Rational Suite AnalystStudio for the Data Analyst – Whitepaper«, http://www.rational.com/products/whitepaper, 2001

[Rational00] Rational Company: »The UML and Data Modeling«, http://www.rational.com/products/whitepaper, 2000

[Rauh97] Rauh, Otto / Stickel, Eberhard: »Konzeptuelle Datenmodellierung«, B. G. Teubner, 1997

[Riordan00] Riordan, Rebecca M.: »Microsoft SQL Server 2000-Programmierung – Schritt für Schritt«, Microsoft Press, 2000

[Scheer91] Scheer, August-Wilhelm: »Wirtschaftsinformatik – Informationssysteme im Industriebetrieb«, Springer, 1991

[Scheer97] Scheer, August-Wilhelm: »Wirtschaftsinformatik – Referenzmodelle für industrielle Geschäftsprozesse«, Springer, 1997

[Scheer99] Scheer, August-Wilhelm: »ARIS – Vom Geschäftsprozess zum Anwendungssystem«, Springer, 1999

[Schicker99] Schicker, Edwin: »Datenbanken und SQL«, B. G. Teubner, 1999

[Schmidt01] Schmidt, Meinhardt / Demmig, Thomas: »SQL GE-PACKT«, mitp, 2001

[Sparks01] Sparks, Geoffrey: »Database Modeling in UML«, Method & Tools, 2001

[Shlaer96] Shlaer, Sally / Mellor, Stephen J.: »Objektorientierte Systemanalyse«, Hanser, 1996

[Unterstein96] Unterstein, Michael: »Unternehmensübergreifende Modellierung von Datenstrukturen«, Deutscher Universitäts Verlag, 1996

[Vetter91] Vetter, Max: »Aufbau betrieblicher Informationssysteme«, B. G. Teubner, 1991

[Vossen94] Vossen, Gottfried: »Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme«, Addison Wesley, 1994

[Vossen00] Vossen, Gottfried: »Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme«, Oldenbourg, 2000

Stichwortverzeichnis