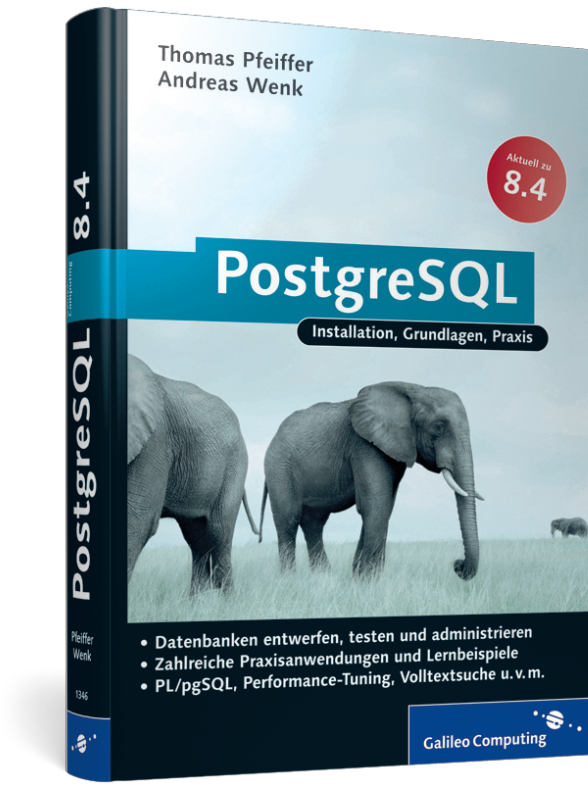
Thomas Pfeiffer, Andreas Wenk

PostgreSQL

Das Praxisbuch



Galileo Press

Auf einen Blick

Inhalt

Wie kommt man zur PostgreSQL? Wer hilft? Und: Was erwartet Sie in diesem Buch? Ein kurzer Abschnitt, der Ihnen das Weiterlesen schmackhaft machen soll.

1 Einleitung

BEGIN WORK;

Ziele

Fangen wir damit an, was dieses Buch nicht ist. Dieses Buch ist keine Referenz! Der Grund dafür ist relativ einfach: Referenzen zu schreiben ist langweilig, und Referenzen zu lesen ist noch viel langweiliger.

Unser Ziel ist es, dass Sie als Leser die Vorzüge der PostgreSQL kennen lernen – mit oder ohne SQL-Erfahrung. Wenn Sie bereits Erfahrungen mit Datenbanken gesammelt haben auch gut. Wir hoffen, Sie entdecken Dinge, die Sie bislang noch nicht kannten. Um Sie auf Ihrer Forschungsreise zu unterstützen, geben wir Ihnen mit diesem Buch einen Praxisleitfaden an die Hand.

Wir haben für Sie eine Sammlung anschaulicher Beispiele zusammengestellt. Auch wenn diese vielleicht nicht jedermanns Geschmack treffen, der eine will mehr, der andere weniger, so sollten sich die Beispiele doch recht einfach an Ihre eigenen Bedürfnisse adaptieren lassen. Damit kommen dann sowohl Einsteiger als auch fortgeschrittene Benutzer auf ihre Kosten.

Wege zur PostgreSQL

Vielleicht haben Sie ja einen ähnlichen *Leidensweg* wie die Autoren hinter sich. Grundsätzlich im Thema Webentwicklung angesiedelt, haben wir lange Zeit MySQL als Datenbank eingesetzt. Dies ergab sich (wie auch für viele andere Webentwickler) aufgrund der weiten Verbreitung von MySQL bei den Hosting-Anbietern fast automatisch. Allerdings, wie das nun mal so ist: Man wächst mit der Erfahrung. Irgendwann sind wir ein-

fach an den Punkt gekommen, an dem wir *»ein bisschen mehr Datenbank«* brauchten.

Nun sind wir grundsätzlich der Meinung, dass es sinnvoller ist, Geld in das Wissen der Mitarbeiter anstatt in teure Lizenzen zu investieren, das heißt, bei unseren Projekten bevorzugen wir, wenn möglich, Open-Source-Technologien und -Software. Damit fiel unsere Wahl recht schnell auf die PostgreSQL-Datenbank – eine Entscheidung, die wir bislang nicht bereut haben.

PostgreSQL ist cool!

Tatsächlich erhalten Sie mit der PostgreSQL ein mächtiges Datenbanksystem. Damit verbunden ist an so mancher Stelle natürlich eine höhere Komplexität, insbesondere bei der Konfiguration und bei der Erweiterung des Datenbanksystems. Auf der anderen Seite erhalten Sie aber auch eine höhere Flexibilität, oder salopp gesagt: PostgreSQL hat eine Menge Schrauben, an denen Sie drehen können.

Kein Grund zur Panik – in diesem Buch zeigen wir Ihnen, wie Sie sicher durch die vermeintlichen Untiefen des Datenbanksystems manövrieren. Und um gleich mal mit einem verbreiteten Vorurteil aufzuräumen: Ja, es dauert ein bisschen länger mit der PostgreSQL warm zu werden, aber es ist nicht zu kompliziert. Ganz im Gegenteil: Sie werden sehen, dass ein PostgreSQL-Cluster genauso schnell installiert ist wie andere Datenbanksysteme. Auch die Konfiguration stellt keine allzu großen Hürden dar.

Aber nicht vergessen: Dies ist keine Referenz. Das ist auch nicht notwendig, denn wer Referenzmaterial benötigt (und das werden Sie), wird leicht im Internet fündig. PostgreSQL hat eine extrem gute Dokumentation zu bieten (*http://www.postgresql.org*). Wir ermutigen Sie bereits jetzt, sich die Seite ausgiebig anzusehen und die entsprechende Dokumentation, passend zu der von Ihnen verwendeten Version (bevorzugt natürlich die 8.4) zu bookmarken.

Lizenz

Die PostgreSQL-Datenbank ist vollständig Open Source und steht unter der BSD-Lizenz (*Berkley Software Distribution*, *http://de.wikipedia.org/wiki/BSD-Lizenz*). Das bedeutet, dass Sie die Software kopieren, verändern oder vertreiben dürfen – so wie Sie es für nötig halten, solange Sie

einen Copyright-Hinweis in Ihrer Distribution belassen (*http://www.postgresql.org/about/licence*). Diese Lizenz gibt Ihnen die größtmögliche Freiheit und Flexibilität für Ihre Projekte, zwingt Sie aber auch, die Auflage und ihre Regeln zu respektieren.

Community

Hinter der PostgreSQL steht wie bei so vielen Open-Source-Projekten eine Community – eine große, hilfsbereite und viel gelobte Community – eine wirklich coole Community. Wir möchten Sie ermutigen, sich an dieser zu beteiligen. Der einfachste Einstiegspunkt ist das Mitlesen der unterschiedlichen Mailinglisten (*http://www.postgresql.org/community/lists/*). Stellen Sie Fragen. Beantworten Sie Fragen. Beteiligen Sie sich an der Weiterentwicklung, indem Sie Vorschläge besprechen, oder übermitteln Sie selbst einen *Patch* (englisch *Flicken*, also eine Nachbesserung des Programms). Es stehen Ihnen alle Wege offen. Nur so wächst die Software und damit die Verbreitung von PostgreSQL.

Außerdem möchten wir auf zwei deutsche Webseiten hinweisen: das deutsche PostgreSQL-Forum (*http://www.pg-forum.de/*) und die Informations- und Linksammlung unter *http://www.postgresql.de/*. Ein Vorbeisurfen lohnt sich alle mal obwohl die Websites nicht soooo aktiv sind (bis auf das Forum). Die absoluten PostgreSQL-Geeks treffen Sie allerdings im IRC Chat auf *freenode.net*: *#postgresql* (*irc://irc.freenode.net/postgresql*) und *#postgresql-de* (*irc://irc.freenode.net/postgresql-de*).

ROLLBACK?

Sie halten die erste Auflage unseres Buches in den Händen. Sie können uns glauben – wir haben alles auf Herz und Nieren getestet. Aber nobody is perfect. Wenn sich also Fehler eingeschlichen haben, bitten wir Sie uns diese mitzuteilen, damit wir diese in der zweiten Auflage (die ja bestimmt kommt …) korrigieren können. Senden Sie uns doch einfach eine E-Mail an info@pg-praxisbuch.de. Oder noch einfacher haben Sie es, wenn Sie die Website zum Buch unter *http://www.pg-praxisbuch.de* besuchen. Dort können Sie mit uns in Kontakt treten, einen Kommentar hinterlassen oder uns mit Fragen löchern … Die Beispiele aus dem Buch finden Sie übrigens dort oder auf der Website des Verlages: *http://www.galileocomputing.de/2008*.

Danke

Ohne ein paar nette Menschen, auf deren Unterstützung wir zählen können und konnten, wäre es schwer gewesen, das Buch zu schreiben. Da wären zum Beispiel Klara, Kiana, Ole, Meike und Verena, die auf uns verzichten mussten und uns mit gewährter Zeit unterstützt haben. Annette und Wolfgang Wenk danken wir für das Schreibcamp im März 2009 und viele hilfreiche Tipps zum Thema *Schreiben*. Außerdem danken wir Andreas Putzo und Arne Böttger, die quergelesen, viele sehr hilfreiche Kommentare gegeben und Fehler isoliert haben (klingt nicht so schlimm wie »entdeckt haben«). Stefan Reimers, seines Zeichens selbst Autor beim Galileo-Computing-Verlag, ist der Fachgutachter dieses Buchs und hat zu allen Kapiteln Kommentare, Fragen und Verbesserungsvorschläge geliefert – danke.

Tja – und dann wollen wir uns sehr herzlich bei dem Mann bedanken, der dieses Buch überhaupt erst möglich gemacht hat, weil er uns im letzten Jahr gefragt hat, ob wir Lust haben ein Buch über die PostgreSQL-Datenbank zu schreiben: unser Lektor Jan Watermann.

Nicht vergessen dürfen wir die Core-Entwickler und Contributor der PostgreSQL-Datenbank. Zur ersten Kategorie gehört unter anderem Peter Eisentraut. An ihn geht ein dickes Dankeschön für das sehr gelungene Vorwort. Ohne diese Menschen und deren unermüdlichem Einsatz gäbe es die PostgreSQL nicht, und wir müssten auf eine Menge Spaß und die *most advanced open source database* verzichten.

So, bei uns in Hamburg sagt man, wenn's losgehen soll: »Nu ma ran an' Speck!« Viel Erfolg und Spaß wünschen Ihnen

Hamburg,
Thomas Pfeiffer und Andreas Wenk

COMMIT;

Wir werden hier den ersten produktiven Kontakt mit der Datenbank aufnehmen, und obwohl das Kapitel »Praxis 1« heißt, bleibt es uns nicht erspart, doch noch einen kurzen Ausflug zur theoretischen Seite zu machen.

3 Praxis 1: Die Grundlagen

In diesem Kapitel wollen wir uns zunächst mit den folgenden grundlegenden Fragen befassen: Wie kommen die Daten in die Datenbank, und wie kommen sie wieder heraus? Bevor wir nun unsere Datenbank erstellen und jede Menge SQL-Statements produzieren, werden wir uns zunächst ein praktisches Beispiel ausdenken, an dem wir das Gelernte festzurren.

3.1 Herausforderung und Modell: Unsere kleine Firma

Gehen wir von folgender Problematik aus: Wir haben ein Unternehmen mit einer gewissen Anzahl von Mitarbeitern. Dieses Unternehmen bietet Produkte und/oder Dienstleistungen zum Verkauf an. Die Herausforderung, der wir uns nun gegenüber sehen, ist die Erstellung einer Datenbank, in der Informationen über Mitarbeiter, Produkte, Kunden, Bestellungen und daraus resultierende Rechnungen hinterlegt werden können.

Dieses kleine Modell wird uns dann als Grundlage für die vorgestellten Funktionen der Datenbank genügen, erhebt aber natürlich keinen Anspruch auf Vollständigkeit. Wir werden der Übersicht halber viele Tabellendefinitionen etwas kleiner halten, also zum Beispiel bei den Kunden auf zusätzliche Felder für die Lieferanschrift oder Bankverbindung verzichten, da es sich bei solchen Felder eher um Wiederholungen eines bereits vorgestellten Typs handelt. Es ist Ihnen natürlich freigestellt, den Tabellen weitere Felder nach Belieben (sofort oder später) hinzuzufügen.

Außerdem wird der Kern unserer Datenbank aus einer vergleichsweise übersichtlichen Zahl von Tabellen bestehen. Erweiterungen sind auch

hier denkbar, etwa für die Bereiche »Lagerverwaltung«, »Mahnwesen« oder vielleicht sogar »Customer-Relationship-Management«. Diese Themen würden sich allerdings besser in einem Buch über Applikationsentwicklung machen, das wir hier jedoch nicht schreiben.

Nun zurück zur Aufgabe: Die ersten Daten die wir in unserer Datenbank hinterlegen möchten, sind die unserer Mitarbeiter. Wir sehen hier Felder für die gängigen Informationen wie »Vorname«, »Nachname«, »Telefon« und so weiter vor. Eine Struktur für die Tabelle `mitarbeiter` könnte also etwa wie folgt aussehen:

Information	Typ des Werts
ID	Mitarbeiter-Nr. / eindeutiger Schlüssel
Anrede	nur die Werte »herr« oder »frau«
Vorname	Zeichenkette, max. 100 Zeichen
Nachname	Zeichenkette, max. 100 Zeichen
E-Mail	Zeichenkette, max. 150 Zeichen
Position	Zeichenkette, max. 150 Zeichen
Telefon	Zeichenkette, max. 100 Zeichen
Mobiltelefon	Zeichenkette, max. 100 Zeichen
Straße	Zeichenkette, max. 100 Zeichen
PLZ	Zeichenkette, max. 10 Zeichen
Ort	Zeichenkette, max. 100 Zeichen
Bemerkungen	beliebig lange Zeichenkette für einen Freitext
Gehalt	Zahl mit zwei Nachkommastellen
Geburtsdatum	Datum

Tabelle 3.1 Struktur für die Tabelle »mitarbeiter«

Was wir wollen ist folgendes: Die Tabelle `mitarbeiter` soll wie übrigens die meisten unserer Tabellen eine eigene *eindeutige ID* bekommen. Grundsätzlich könnte man hier eine vielleicht bereits vorhandene Personalnummer verwenden, wir möchten jedoch erreichen, dass das Datenbanksystem für jeden neu eingefügten Datensatz eine eigene ID vergibt und werden deswegen später den Datentyp `serial` verwenden, der genau dies gewährleistet. Wir werden im Folgenden `serial` zwar als Datentyp

bezeichnen, doch es sei schon im Vorfeld angemerkt, dass es sich hierbei nur um einen Alias für die Verwendung einer *Sequenz* handelt. Wer es jetzt schon genauer wissen möchte, springt einfach schnell mal zu Abschnitt 3.8.4, »Selbstzählende Datentypen«.

Unterschied natürlicher und technischer Primärschlüssel

Beachten Sie, dass eine bereits vorhandenene Personalnummer einen *natürlichen Primärschlüssel* darstellt. Das ist also ein Schlüssel, der eindeutig ist – bei einer Personalnummer in einem Unternehmen sollte das sichergestellt sein, da die Personalabteilung (hoffentlich) fortlaufende Nummern nutzt.

Im Gegensatz dazu ist der als `serial` generierte Primärschlüssel ein *technischer Schlüssel*. Diesen Schlüssel lassen wir von der Datenbank erstellen (er ist selbstverständlich eindeutig und fortlaufend), und er ist völlig unabhängig von irgendwelchen Eigenschaften des Unternehmens im *real life*.

Im Feld für die Anrede möchten wir nur die Werte »herr« oder »frau« abspeichern. Das mag etwas merkwürdig anmuten – man könnte hier auch nur »0« oder »1« für eine männlich/weiblich-Unterscheidung oder aber einen Freitext für größtmögliche Flexibilität zulassen. Wir möchten anhand dieses Feldes jedoch den ersten Kontakt mit dem Regelwerk (Constraints) der Datenbank aufnehmen.

Die Felder `Vorname` bis `Ort` sind dann wieder weitgehend selbsterklärend. Hier sollen nur beliebige Zeichenketten mit einer Maximallänge gespeichert werden. Übrigens legen wir hier auch Postleitzahlen und Telefonnummern als Zeichenketten ab. Darüber kann man zwar geteilter Meinung sein, aber wir entscheiden uns hier dafür, um zum Beispiel Probleme mit der Speicherung führender Nullen zu vermeiden (etwa bei der Postleitzahl 01099).

Die letzten drei Felder haben dann noch einmal andere Datentypen: Im Feld für die Bemerkungen möchten wir einen beliebig langen Text erfassen, brauchen also eine maximale Obergrenze hier nicht.

Das Gehalt möchten wir als Dezimalzahl mit zwei Nachkommastellen speichern. Als Letztes wollen wir dann noch das Geburtsdatum des Mitarbeiters hinterlegen können und werden auch hier den dafür bestimmten Datentyp benutzen.

Wir gehen mal davon aus, dass es sich bei dem Unternehmen nicht um eine »Zweimannklitsche« handelt, sondern dass wir eine größere Anzahl von Mitarbeitern verwalten. Üblicherweise sind diese Mitarbeiter gewis-

sen Abteilungen zugeordnet. Wir erfassen hier zunächst ganz einfach die Abteilungen des Unternehmens:

Information	Typ des Werts
ID	eindeutiger Schlüssel der Abteilung
Abteilungs-Nr.	Ganzzahl (Integer)
Abteilung	Zeichenkette, max. 50 Zeichen

Tabelle 3.2 Struktur für die Tabelle »abteilungen«

Damit haben wir die Struktur für einen zweiten »Informations-Container«. Noch wissen die beiden Objekte `abteilungen` und `mitarbeiter` nichts voneinander. Das werden wir aber in Kürze ändern, wenn wir in der Tabelle `mitarbeiter` jedem Datensatz (also Mitarbeiter) sagen, zu welcher Abteilung er gehört. Praktisch bedeutet das, dass wir der Struktur für die Tabelle `mitarbeiter` noch ein weiteres Feld hinzufügen, in dem wir einen Verweis auf einen Eintrag in der Tabelle `abteilungen` speichern können.

Damit sieht die gewünschte Struktur zwischen den beiden zu erstellenden Tabellen wie folgt aus:

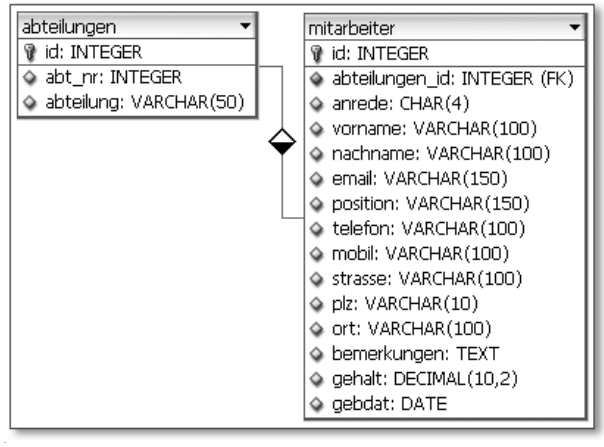


Abbildung 3.1 Mitarbeiter und Abteilungen im fertigen Modell

Wir haben beide Tabellen also mit jeweils einem sogenannten *Primärschlüssel* (*primary key*) versehen.

Eine Grundanforderung an ein relationales Datenbanksystem ist, dass jeder Datensatz einer Tabelle (Relation) über einen Schlüssel, der sich nicht ändert, eindeutig identifizierbar sein muss.

Ein Primärschlüssel ist eine *eindeutige Identifikation* einer Zeile in einer Tabelle. Ein Wert einer als Primärschlüssel definierten Spalte darf in der Tabelle nur einmal vorkommen, (sonst hat sich's was mit Eindeutigkeit) und es darf nur einen Primärschlüssel pro Tabelle geben.

Technisch gesehen ist der Primärschlüssel eine Verknüpfung von `UNIQUE Constraint` und `NOT NULL Constraint`, aber wir wollen der Geschichte nicht vorgreifen, später erfahren Sie mehr dazu.

In Abbildung 3.1 sehen Sie ebenfalls die bereits angekündigte Beziehung zwischen den beiden Tabellen. Es handelt sich hierbei um eine *1:n-Beziehung*, das heißt, eine Abteilung kann beliebig viele Mitarbeiter enthalten.

Dafür haben wir der Tabelle `mitarbeiter` die Spalte `abteilungen_id` gegönnt. Diese ist vom gleichen Typ wie die Spalte `id` der Tabelle `abteilungen` (Integer). Die Angabe `FK` nach dem Spaltentyp besagt, dass es sich hierbei um einen sogenannten *foreign key* (*Fremdschlüssel*) handelt. Das bedeutet, dass Werte dieser Spalte aus der Schlüsselspalte einer anderen Tabelle stammen.

Jetzt zu den Dingen, die wir verkaufen möchten. Das können entweder Produkte und/oder Dienstleistungen sein. Allen gemeinsam dürfte sein, dass sie eine Bezeichnung und einen Preis haben. Auf das Wesentlichste beschränkt könnte die Struktur unserer Produkttabelle etwa so aussehen:

Information	Typ des Werts
ID	eindeutiger Schlüssel des Produkts
Artikel-Nr.	Zeichenkette, max. 100 Zeichen (zum Beispiel EAN-Code)
Bezeichnung	Zeichenkette, max. 200 Zeichen
Beschreibung	beliebig lange Zeichenkette für Produktbeschreibung
Preis	Zahl mit zwei Nachkommastellen
Steuersatz	Zahl mit drei Nachkommastellen (zum Beispiel 0.190)

Tabelle 3.3 Struktur für die Tabelle »produkte«

In dieser Tabelle haben wir uns dazu entschlossen, die Artikelnummer als Zeichenkette abzulegen und nicht wie in der Tabelle `abteilungen` die Abteilungsnummer als Integer-Wert. Das ist letztlich Geschmacksache, erlaubt in der Produkttabelle aber auch die Speicherung alphanumerischer Zeichenketten. Der Preis des jeweiligen Produkts soll als Nettobetrag mit zwei Nachkommastellen gespeichert werden, der Steuersatz hingegen als Zahl mit drei Nachkommastellen (man denke an Frankreich mit zur Zeit 19,6 % Mehrwertsteuer).

Gönnen wir den Kunden unseres Unternehmens auch noch eine eigene Tabelle. Hier werden zunächst nur einfache Informationen wie Name und Anschrift gespeichert. Wir weichen an dieser Stelle nur von unserer Definition des Felds `anrede` ab, so wie wir es für die Tabelle `mitarbeiter` beschrieben haben, da wir bei unseren Kunden mit der Anrede gegebenenfalls auch so etwas wie »Herr Prof. Dr.« abspeichern möchten.

Information	Typ des Werts
ID	eindeutiger Schlüssel, Kundennummer
Anrede	Zeichenkette, max. 50 Zeichen
Vorname	Zeichenkette, max. 100 Zeichen
Nachname	Zeichenkette, max. 100 Zeichen
Straße	Zeichenkette, max. 100 Zeichen
PLZ	Zeichenkette, max. 10 Zeichen
Ort	Zeichenkette, max. 100 Zeichen
Telefon	Zeichenkette, max. 100 Zeichen
E-Mail	Zeichenkette, max. 200 Zeichen

Tabelle 3.4 Struktur für die Tabelle »kunden«

Kunden können Bestellungen aufgeben, deshalb werden wir diese in einer dafür bestimmten Tabelle speichern. Für den ersten Entwurf genügt es, wenn wir uns zu einer Bestellung die Kundennummer, das Bestell-, Liefer- und Rechnungsdatum merken. Daraus ergibt sich folgende Möglichkeit einer Struktur:

Information	Typ des Werts
ID	eindeutiger Schlüssel, Bestellnummer
Kunden-Nr.	Verweis auf eine ID in der Tabelle `kunden`
Bestelldatum	Zeitstempel
Lieferdatum	Zeitstempel
Rechnungsdatum	Zeitstempel

Tabelle 3.5 Struktur für die Tabelle »bestellungen«

Im Feld für die Kundennummer müssen wir später sicherstellen, dass hier nur Werte gespeichert werden können, die auch als ID in der Tabelle `kunden` vorhanden sind. Außerdem müssen wir noch festlegen, was mit einem Eintrag in der Tabelle `bestellungen` passieren soll, wenn ein dazu korrespondierender Eintrag in der Tabelle `kunden` gelöscht wird. Hier werden Sie eine weitere Begegnung mit dem Regelwerk der Datenbank haben.

Für das Bestell-, Liefer- und Rechnungsdatum, das wir uns in dieser Tabelle merken möchten, wollen wir einen sogenannten Zeitstempel (engl. timestamp) verwenden, das heißt, wir können mit einer vollständigen Zeitangabe wie etwa »2009-12-24 14:00:00« arbeiten.

Nun reicht es natürlich nicht aus, sich einfach nur zu merken, dass ein bestimmter Kunde eine Bestellung aufgegeben hat, sondern wir wollen natürlich auch wissen, was dieser Kunde im Einzelnen bestellt hat.

Da sich eine Bestellung aus mehreren Positionen zusammensetzen kann, wollen wir hier eine weitere und zunächst letzte Tabelle anlegen. Sie soll die Bestellungen mit den Produkten verknüpfen. In dieser Tabelle benötigen wir eigentlich erst mal nur drei Informationen: die ID der Bestellung, die ID des bestellten Produkts und die Menge, die von diesem Produkt bestellt wurde.

Information	Typ des Werts
Bestell-Nr.	Verweis auf eine ID in der Tabelle `bestellungen`
Produkt-Nr.	Verweis auf eine ID in der Tabelle `produkte`
Menge	Ganzzahl, Vorgabewert 1

Tabelle 3.6 Struktur für die Tabelle »bestellungen_produkte«

Es fällt auf, dass diese Tabelle keine ID-Spalte wie die vorherigen Tabellen hat. Wir gehen mal davon aus, dass kein Produkt in einer Bestellung doppelt auftaucht, sondern bei Mehrfachbestellungen eines Produkts einfach die Menge erhöht wird. Deshalb ist die Nummer der Bestellung in Kombination mit der ID des Produkts eine Möglichkeit, den Datensatz eindeutig zu identifizieren. Wir werden hier später einen verketteten Primärschlüssel einsetzen.

Damit haben wir für unsere Beispieldatenbank zunächst sechs Tabellen zur Verfügung, um zumindest das Basissystem zu implementieren. Später werden wir dieses Modell dann noch an die erweiterten Anforderungen anpassen.

Im grafischen Modell sieht die zusätzliche Struktur der vier neuen Tabellen dann so aus:

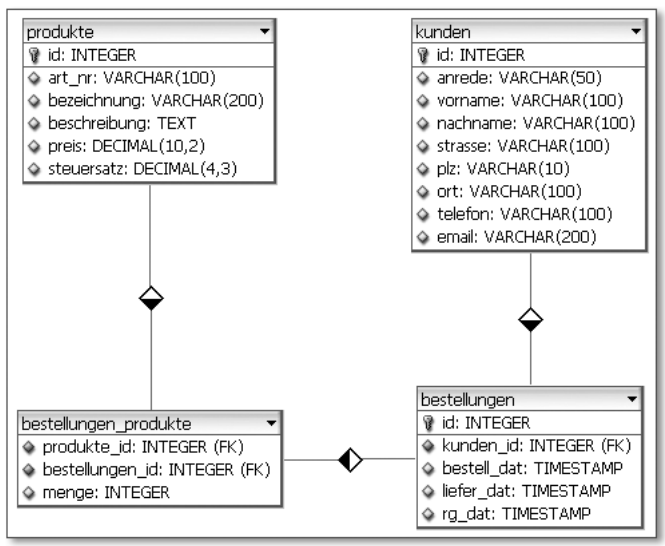


Abbildung 3.2 Produkte, Kunden und Bestellungen im fertigen Modell

Auch hier zeigen sich wieder die Beziehungen zwischen den einzelnen Tabellen. So kann ein Kunde beliebig viele Bestellungen tätigen (1:n-Beziehung).

Das kennen wir schon aus der Beziehung zwischen den Tabellen `mitarbeiter` und `abteilungen`. Aber wir finden hier auch etwas Neues: eine sogenannte *n:m-Beziehung*: Das bedeutet, eine Bestellung kann nicht nur ein, sondern beliebig viele Produkte enthalten, ebenso wie ein Produkt

nicht nur in einer, sondern in ebenfalls beliebig vielen Bestellungen vorkommen kann.

Den »Kitt« zwischen den Tabellen `bestellungen` und `produkte` liefert uns die Tabelle `bestellungen_produkte`.

Den soeben angekündigten verketteten Primärschlüssel können Sie jetzt auch gut erkennen. Nehmen wir einmal an, wir hätten (unter anderem) Produkte mit den IDs 10, 20 und 30. Des Weiteren haben wir Bestellungen mit den IDs 1, 2, und 3 vorliegen.

In Bestellung 1 wurden die Produkte mit den IDs 10 und 20 jeweils einmal bestellt. In Bestellung 2 wurden die Produkte mit den IDs 20 und 30 jeweils zweimal bestellt. Schließlich wurden in Bestellung 3 einmal das Produkt mit der ID 10, zweimal das Produkt mit der ID 20 und dreimal das Produkt mit der ID 30 bestellt. Dann sieht der Inhalt der Tabelle `bestellungen_produkte` wie folgt aus:

produkte_id	bestellungen_id	menge
10	1	1
20	1	1
20	2	2
30	2	2
10	3	1
20	3	2
30	3	3

Tabelle 3.7 Daten der Tabelle »bestellungen_produkte«

Wie Sie sehen ist die Kombination von `produkte_id` und `bestellungen_id` eindeutig und genügt uns somit in der Kombination als primärer Schlüssel.

3.2 Theorie und Praxis: Was ist SQL?

3.2.1 SQL – Structured Query Language

Wir haben ja schon im vorigen Kapitel mit der PostgreSQL gesprochen. Wie war das gleich? `INSERT`, `SELECT`, `CREATE DATABASE` ...

Sobald wir als Mensch mit der Maschine »reden« wollen, muss eine Sprache her, die uns das ermöglicht. So gibt es in der Softwareentwicklung Sprachen der ersten (Assembler), zweiten (Cobol) und mittlerweile dritten Generation (Java, PHP, Ruby), die nichts anderes tun, als es dem Anwender so einfach wie möglich zu machen, mit der Maschine zu kommunizieren. *SQL* (*Structured Query Language*) ermöglicht das Gleiche, wobei SQL an sich keine Programmiersprache im klassischen Sinn ist – wie der Name ja auch sagt.

Wir müssen also die Syntax erlernen und letztlich verstanden haben. Danach sollte es uns möglich sein, Operationen auszuführen, wie etwa eine Datenbank zu erstellen oder Tabellen zu erstellen, Daten in die Datenbank einzufügen und Daten wieder aus der Datenbank auszulesen.

Wer das Inhaltsverzeichnis bereits überflogen hat, wird festgestellt haben, dass wir in Abschnitt 3.4, »Die Umsetzung«, eine Einführung in die SQL-Syntax geben. Im weiteren Verlauf des Buchs werden wir Ihnen dann reichlich Gelegenheit geben, Ihre Kenntnisse weiter zu vertiefen.

3.2.2 Wie fing es an?

Im Zeitalter von Wikis und der »Wikisierung« all unseren Wissens ist es kein Problem, reichlich Informationen über SQL zu erhalten. Für uns soll hier ein kleiner Rückblick auf die Entstehungsgeschichte ausreichend sein.

Eine Firma namens IBM hat SQL für ihre Forschungen an relationalen Datenbanksystemen bereits Mitte der 70er-Jahre entwickelt. Daran maßgeblich beteiligt war der Computerwissenschaftler Edgar F. Codd. Er gilt als der Erfinder des relationalen Datenbanksystems. Bei seiner Forschung hat er für die Abfrage der Datenbank den Vorläufer von SQL namens *SEQUEL* (*Structured English Query Language*) genutzt. Entwickelt wurde die Sprache von zwei IBM-Mitarbeitern namens Donald D. Chamberlin und Raymond F. Boyce.

Fast gleichzeitig hat auch Oracle (damals Relational Software) ein *DBMS* (*Datenbank-Managementsystem*) mit SQL als Sprache entwickelt und als Produkt vertrieben.

1986 war es dann so weit, und es wurde der erste Standard durch das *ANSI* (*American National Standards Institute*) verabschiedet: SQL-86 (auch SQL 1) war geboren. 1989 wurde der Standard dann aktualisiert und hat – wie nicht schwer zu erraten – den Namen SQL-89 verpasst bekommen.

In den Jahren 1992 und 1999 gab es wiederum Aktualisierungen des Standards. PostgreSQL unterstützt diese beiden Standards: SQL-92 (auch SQL 2) und SQL-99 (auch SQL 3).

Mittlerweile gibt es auch die Standards SQL-2003 und SQL-2006. Seit 1987 hat die *ISO* (*Internationale Organisation für Normung*) die gleichnamigen Standards aufgenommen.

Im Folgenden lesen Sie einen kleinen Überblick, welche Datenbank welche SQL-Standards unterstützt:

Datenbank	Version	SQL Standard
PostgreSQL	8.4	fast vollständig SQL-2003
MySQL	5.x	teilweise SQL-99, teilweise SQL-2003, teilweise proprietär
Oracle	10g	SQL-2003, teilweise proprietär
Ingres	2006 Release 3	SQL-92 / SQL-99
Firebird	2.1.1	SQL-92 / SQL-99, teilweise SQL-2003
IBM DB2	9.1	weitestgehend SQL-2003, teilweise proprietär
Sybase	11	teilweise SQL-2003
Informix	11.50	teilweise SQL-2003

Tabelle 3.8 Welche DB unterstützt welche Standards?

3.2.3 Der SQL-Sprachkern

SQL ist in drei Bereiche unterteilt, die jeweils unterschiedliche Datenbank-operationen abdecken. Diese Bereiche werden im Folgenden kurz vorgestellt.

Data Definition Language – DDL

Zu diesem Bereich gehören die Befehle `CREATE`, `ALTER` und `DROP`. Daraus ergibt sich fast von selbst die Funktion der DDL: Objekte erstellen, verändern oder löschen – das alles geschieht mit der DDL. Dazu gehört auch

das Erstellen der Datenbank selbst oder das Anlegen von Tabellen oder Views. Genauso gehören Operationen zum Verändern oder Löschen dieser Elemente dazu.

Data Manipulation Language – DML

Wie der Name schon sagt, geht es in diesem Bereich um das Verändern von Datenbankinhalten. Dabei kommen die Befehle `SELECT`, `INSERT`, `UPDATE` und `DELETE` zur Anwendung. Im Gegensatz zum endgültigen Befehl `DROP` sollten diese Befehle immer in eine Transaktion gekapselt werden. Was eine Transaktion ist und wie diese angewandt wird, erfahren Sie in Abschnitt 4.5, »Transaktionen: Ein paar Worte zum Thema Sicherheit«. Vorab wollen wir ein kurzes Beispiel hierzu geben, da das Thema sehr wichtig ist – wenn Ihnen Ihre Daten lieb sind.

Eine Transaktion wird eingesetzt, um die Konsistenz der Datenbank aufrechtzu erhalten. Nehmen wir die bereits vorgestellten Tabellen `kunden` und `bestellungen`. Jeder Kunde kann viele Bestellungen tätigen. Dies ist eine, wie oben schon erwähnte, *1:n-Verbindung*. Konsistent sind unsere Tabellen nur, wenn es zu jeder Bestellung in der Tabelle `bestellungen` mindestens einen Kunden in der Tabelle `kunden` gibt. Um die Tabellen mit Daten zu füllen, verwenden Sie die zwei folgenden `INSERT`-Statements:

```
INSERT INTO bestellungen (id,kunden_id) VALUES (1,24);
INSERT INTO kunden (id,vorname,nachname) VALUES
(24,'Andy','Wenk');
```

Was passiert nun, wenn das erste Statement aus irgendeinem Grund schief geht? Richtig, wir haben Leichen in der Tabelle `bestellungen`, weil es zur Bestellung mit der ID 1 keinen Kunden in der Tabelle `kunden` gibt.

Aus diesen (in größeren Applikationen fatalen) Gründen werden solche `INSERT`-Statements in eine Transaktion gekapselt. Nur wenn beide `INSERT`-Befehle erfolgreich durchgeführt werden konnten, werden Daten in die Datenbank geschrieben. In SQL sieht das dann so aus:

```
BEGIN TRANSACTION;
INSERT INTO bestellungen (id,kunden_id) VALUES (1,24);
INSERT INTO kunden (id,vorname,nachname) VALUES
(24,'Andy','Wenk');
COMMIT TRANSACTION;
```

Schreibweisen

Die Befehle `BEGIN TRANSACTION` und `COMMIT TRANSACTION` sind die ausführliche Schreibweise. `BEGIN` kann entweder von der Anweisung `WORK` oder `TRANSACTION` gefolgt werden. Sie können aber auch einfach nur `BEGIN` schreiben, da PostgreSQL selbstständig eine Transaktion durchführt, sobald mehr als ein SQL-Statement ausgeführt werden soll.

Wir sagen der PostgreSQL mit dem einleitenden `BEGIN` dass wir einen Transaktionsblock starten wollen. Mit `COMMIT` weisen wir die Datenbank an, den Transaktionsblock zu beenden. Alles, was zwischen `BEGIN` und `COMMIT` steht, wird also in einer Transaktion durchgeführt. Die Transaktion ist nur erfolgreich, wenn beide Statements erfolgreich ausgeführt werden konnten. Schlägt ein Statement fehl, führt die PostgreSQL einen `ROLLBACK` durch. Dadurch werden alle Änderungen an der Datenbank rückgängig gemacht, und unsere Datenbank bleibt in einem konsistenten Zustand. Sehr gut!

Data Control Language (DCL)

Jede PostgreSQL-Datenbank hat Benutzer (zumindest einen). Mit den Befehlen `GRANT` und `DENY` ist es uns möglich, Regeln festzulegen, welcher Benutzer Zugriff auf bestimmte Ressourcen der Datenbank (zum Beispiel auf eine Tabelle oder ein View) hat, oder welche Operationen der Benutzer ausführen darf. Damit ist uns etwa möglich, dem Benutzer *pfeiffer* nur die Befehle `SELECT`, `INSERT` und `UPDATE` in der Datenbank *db_von_andy* zu erlauben. Der Benutzer *wenk* hat dagegen auch das Recht, die Operation `DELETE` auszuführen und kann somit einfach alle Daten, die der Benutzer *pfeiffer* eingefügt hat, wieder löschen. (oh oh – das gibt Ärger ...).

Diese drei Bereiche umfassen alle möglichen Operationen, die wir in unserer Datenbank vornehmen können. Das hier Beschriebene ist natürlich nur der theoretische Ansatz. Den tieferen Einblick erhalten Sie im Abschnitt 3.4, »Die Umsetzung«. Bevor es damit losgeht, stellen wir Ihnen im nächsten Abschnitt das Modell unserer Beispielanwendung vor. In dem Zusammenhang erfahren Sie auch, was Entitiy-Relationship bedeutet und was eine relationale beziehungsweise objekt-relationale Datenbank ist.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z