Datenbank-Anwendungen mit PostgreSQL

PEARSON Education

Markt+Technik

Datenbank-Anwendungen mit PostgreSQL

Einführung in die Programmierung mit SQL, Java, C/C++, Perl, PHP und Delphi

Ewald Geschwinde
Hans-Jürgen Schönig

new technology

Markt+Technik Verlag

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

10 9 8 7 6 5 4 3 2 1

06 05 04 03 02

ISBN 3-8272-6394-8

ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachkorrektorat: Dr. Michael Meskes, Jülich
Korrektorat: Brigitta Keul, München, Brigitte Hamerski, Willich
Umschlaggestaltung: adesso21, Thomas Arlt, München
Herstellung: Ulrike Hempel, uhempel@pearson.de
Satz: Hilmar Schlegel, Berlin — gesetzt in Monotype Times New Roman, Linotype Optima, Letter Gothic
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer

Printed in Germany

Inhaltsverzeichnis

Widmung

Dieses Buch ist einem aufgehenden Stern am Datenbankhimmel gewidmet, der in Kürze das Licht der Welt erblicken wird. Ewald Geschwinde wird Vater.

Danksagung

Wir möchten uns von ganzem Herzen bei allen bedanken, die dieses Buch möglich gemacht und mitgewirkt haben, es fertig zu stellen und zu dem zu machen, was es ist. Besonderer Dank geht an alle Mitarbeiter von Pearson sowie die zahlreichen Entwickler von PostgreSQL. Ohne deren Arbeit und Support wäre es unmöglich, heute ein Buch über moderne Datenbankentwicklung in den Händen zu halten und PostgreSQL in vollen Zügen nutzen und genießen zu können.

Die Arbeit der Freiwilligen hat es möglich gemacht, dass Firmen und Privatleute auf dem gesamten Erdball eine zuverlässige und hochentwickelte Datenbank in den Händen halten ohne die die Datenbankwelt zweifelsohne ein Stückchen ärmer wäre.

Besonderer Dank geht auch bei diesem, unserem dritten Buch an Patricia Barnes, die uns den Einstieg ermöglicht und uns die Chance gegeben hat, mit dem Schreiben zu beginnen.

Abschließend möchten wir uns noch bei Michael Meskes und Boris Karnikowski bedanken, die uns während der Verfassung dieses Buches mit Rat und Tat zur Seite gestanden haben.

Kapitel 1

Überblick

Nach Monaten harter und schwieriger Arbeit ist es nun vollbracht: Das erste vollständige Handbuch für Programmierer zum Thema PostgreSQL ist fertig gestellt. Da ich kein Freund großer Worte bin und Sie nicht mit ewigen lähmenden Einleitungen quälen will, schlage ich vor, sich gleich ins praktische Geschehen zu stürzen und mit einer kurzen Übersicht über PostgreSQL zu beginnen.

1.1 Über PostgreSQL

Bei vielen Terminen werden wir oft gefragt, was denn PostgreSQL sei und was der Vorteil dieser Datenbank gegenüber anderen vergleichbaren Produkten sei. Die Antwort auf diese Frage artet meist in einen sich endlos ziehenden Monolog aus, der nur schwer wieder zu stoppen ist. Es gibt zahlreiche Gründe, PostgreSQL für vielerlei Anwendungen zu nutzen und Anwendungen auf PostgreSQL zu optimieren. In dieser kurzen Einleitung wollen wir versuchen, die wesentlichen Punkte kurz zusammenzufassen und zu erläutern.

PostgreSQL ist freie Software, was bedeutet, dass sie frei und ohne anfallende Kosten verwendet und verteilt werden darf. Speziell bei großen Installationen ist das ein wesentlicher Punkt, der es dem Benutzer ermöglicht, die Gesamtkosten eines Projektes signifikant zu senken. Man denke etwa an diverse kommerzielle Produkte, die nur gegen horrende Lizenzgebühren verfügbar und erhältlich sind. Genau dieses Geld können sie verwenden, um Ihre Applikationen und Produkte zu verbessern.

Im professionellen Umfeld sind die Anschaffungskosten einer Software jedoch nicht das einzige Kriterium. Ein wesentlicher Bestandteil einer Entscheidung ist der Vergleich der so genannten Total Costs of Ownership, also die gesamten während der Nutzung einer Software anfallenden Kosten. Je nach Zuverlässigkeit und Mächtigkeit von Produkten können diese Kosten stark variieren und in vielen Fällen die Anschaffungskosten in den Schatten stellen. Durch die einfache Administration, die große Flexibilität und Zuverlässigkeit ist es beim Einsatz von PostgreSQL möglich, diese Gesamtnutzungskosten Ihrer Datenbanksysteme auf ein vertretbares Minimum zu senken. Jeder Anwender wünscht sich eine Datenbank, die Monate lang ohne ein einziges Problem funktioniert und arbeitet und genau das ist es, was PostgreSQL Ihnen bieten kann: Zuverlässigkeit, Stabilität, Langlebigkeit und sorgenfreies Arbeiten. In Kombination mit UNIX-Betriebssystemen können die Ausfallzeiten extrem gering und Ihre IT-Landschaft am Leben erhalten werden.

Ein Punkt, der immer wieder hervorgehoben wird und auch hervorgehoben werden muss ist der Begriff »Freie Software«. Bisher haben wir uns weitgehend auf die Kosten der Software beschränkt, was den Kern der Idee nicht ganz trifft. Kosten sind ein wesentlicher Punkt, aber freie Software ist noch wesentlich mehr: Frei bedeutet, dass die Software frei modifiziert werden kann. Das Recht, Ihre Software

zu modifizieren, beschränkt Sie nicht in Ihrer geistigen Freiheit. Das ist einer der wesentlichen Punkte, die es in diesem Zusammenhang gibt. »Free Software is not free beer« — Freibier mag zwar den Geist erweitern, hat aber letztendlich nichts mit Freiheit zu tun. Bei Software ist das ein wenig anders. Wenn Sie Ihr Programm modifizieren können, kann niemand Ihre Denk- und Arbeitsweisen einschränken oder in einer Weise prägen, die Ihnen vielleicht nicht angenehm ist.

Ein großer Vorteil von PostgreSQL ist dessen Flexibilität. Es ist problemlos möglich, mit geringem Aufwand einfache oder auch komplexere Erweiterungen zu schreiben, die das tägliche Arbeiten erleichtern. Ähnlich wie bei anderen Datenbanken wie etwa Oracle stehen eingebettete Sprachen zur Verfügung, um neue Funktionen zu schreiben, die direkt in SQL verwendet werden können. Diese Sprachen erleichtern auch das Portieren von Anwendungen nach PostgreSQL, da von anderen Systemen bereitgestellte SQL-Funktionen sehr leicht nachgebildet werden können.

ANSI SQL ist der gängige Datenbankstandard. Das erklärte Ziel der PostgresSQL-Entwickler ist es, sich möglichst genau an den Standard zu halten, um ein größtmögliches Maß an Kompatibilität zu erreichen. Dadurch ist es sehr leicht möglich, mit anderen Datenbanken zu interagieren und hybride Systeme herzustellen. In vielen modernen IT-Umgebungen tummeln sich eine Vielzahl von Systemen und mit ein wenig Sorgfalt kann es Ihnen gelingen, PostgreSQL als Bindeglied zwischen den Welten einzusetzen.

Dieses Buch soll einen Beitrag dazu schaffen, dass Sie lernen PostgreSQL effizienter einzusetzen und die Macht der Datenbank effizient zu nutzen. Wir hoffen, dass dieses Buch zur Verbreitung der Technologie beitragen und Ihre tägliche Arbeit mit der Datenbank erleichtern und bereichern kann.

1.2 Lizenz

PostgreSQL ist unter der so genannten BSD-Lizenz verfügbar. Diese Lizenz besagt, dass der User keine Einschränkungen seiner Rechte in irgendwelcher Art zu fürchten hat und dass der Source Code sowie Binaries von PostgreSQL lizenzfrei kopiert werden können. Die BSD-Lizenz ist wesentlich freier als die populäre GPL-Lizenz und schränkt Benutzer sowie Entwickler in einem geringeren Maße ein.

Das nächste Listing enthält den gesamten Code der BSD-Lizenz:

```
PostgreSQL Database Management System (formerly known as Postgres,
then as Postgres95)

Portions Copyright (c) 1996–2001, The PostgreSQL Global Development
Group
```

Kapitel 2

Installation

Der erste Schritt, um mit PostgreSQL zu beginnen, ist die Installation der Datenbank. In diesem Kapitel werden Sie sehen, wie PostgreSQL auf verschiedensten Systemen installiert werden kann und worauf Sie dabei aufpassen müssen.

Im Wesentlichen ist PostgreSQL für zwei Gruppen von Betriebssystemen verfügbar: Auf der einen Seite tummelt sich eine Heerschar von UNIX-basierten Betriebssystemen, die sich wiederum in Untergruppen wie BSD oder System V unterteilen.

Auf der Gegenseite Microsofts Betriebssysteme: Windows-Systeme eignen sich für den Einsatz von PostgreSQL, stellen den Anwender aber vor eine Vielzahl von Unannehmlichkeiten und Problemen.

Dieses Buch soll Ihnen den Umgang mit beiden Betriebssystemen näher bringen.

2.1 Unterstützte Betriebssysteme

PostgreSQL kann im Prinzip überall dort eingesetzt werden, wo ein ANSI C Compiler verfügbar ist. Das Faktum, dass sich PostgreSQL sehr stark an Standards orientiert, macht die Plattform überaus flexibel und leicht einsetzbar — es ist nahezu egal, welches Betriebssystem Sie verwenden — PostgreSQL tut brav seinen Dienst und wird Ihnen eine verlässliche Datenbank sein.

Das folgende Listing enthält eine Zusammenstellung aller von PostgreSQL 7.2 unterstützten Plattformen. Unterstützt bedeutet, dass alle so genannten Regression Tests problemlos funktioniert haben und das System daher auf allen Systemen korrekte Ergebnisse liefert:

- AIX auf IBM pSeries (RS/6000)
- BeOS auf x86
- BSD/OS auf x86
- FreeBSD auf x86 und Alpha
- HP-UX auf PA-RISC
- IRIX auf SGI Mips
- Linux auf Alpha, armv4l, SGI Mips, Playstation 2, PPC74xx, S/390, Sparc, PA-RISC, M68K, IA64 und x86
- MacOS X auf PPC (zusätzliches Compiler-Flag wegen Namespaces notwendig)
- NetBSD auf Alpha, arm32, m68k, PPC, Sparc, VAX, x86

- OpenBSD auf Sparc, x86
- Open UNIX auf x86
- QNX 4 RTOS auf x86
- Solaris auf Sparc, x86
- SunOS 4 auf Sparc
- True64 UNIX auf Alpha
- Windows auf x86 (Cygwin)

Zusätzlich zu den genannten gibt es noch eine lange Liste von Systemen, die in früheren Versionen bereits explizit als unterstützt vermerkt waren, aber bei denen für PostgreSQL 7.2 kein offizieller Regression- Test vorliegt. In der Regel funktioniert PostgreSQL auf solchen Systemen genauso problemlos wie mit x86 CPUs, dennoch gibt es auch manche Systeme, die aus verschiedensten Gründen nicht als funktionsfähig eingestuft worden sind.

2.2 Installation auf UNIX-Systemen

Die Installation von PostgreSQL unter UNIX geht in der Regel sehr schnell und ohne gröbere Probleme vonstatten. Da PostgreSQL in ANSI C implementiert worden ist, lässt sich die Datenbank prinzipiell auf jedem System, auf dem ein ANSI C Compiler zur Verfügung steht, kompilieren. Die meisten Anwender vertrauen bei der Installation auf den GNU C Compiler, da sich dieser als freie Alternative zu kommerziellen Compilern auf den meisten Plattformen durchgesetzt hat.

Auf manchen Systemen ist es nicht notwendig, selbst den Compiler anzuwerfen, da bereits vorgefertige Binärpakete zur Verfügung stehen. In solchen Fällen lässt sich das Datenbanksystem meist bequem mit einem Shell-Kommando beziehungsweise einer grafischen Oberfläche installieren.

2.2.1 Installation und Deinstallation von RPM-Paketen unter Linux

Ein wesentlicher Teil der gegenwärtig auf dem Markt vertretenen Linux-Systeme basiert auf RPM, dem RedHat Package Manager. RPM-Pakete beruhen intern auf einem Cpio-Archiv und auf Header-Informationen, die für die Installation benötigt werden. Mittels Kommandozeile können die Pakete dann leicht installiert werden und genau das werden wir uns nun etwas genauer im Detail ansehen:

Als Allererstes sollen Sie eine der Mirrorsites (etwa `www.de.postgresql.org`) von PostgreSQL aufsuchen, um die fertigen RPM-Pakete zu downloaden. In der Regel

sind RPM-Files für x86 CPUs verfügbar; auf verschiedenen anderen Sites sollen aber auch schon Alpha- und PowerPC-Versionen gesichtet worden sein.

Nach dem Download der Dateien (am besten nehmen Sie dazu ein Tool wie `wget`) ist es im nächsten Schritt ratsam, nachzusehen, ob PostgreSQL bereits auf Ihrem System installiert ist. Zu diesem Zwecke können Sie `rpm` mit einigen Flags aufrufen:

```
[root@duron hs]# rpm -qva | grep post
postgresql-perl-7.1.3-2
postgresql-odbc-7.1.3-2
postgresql-docs-7.1.3-2
postgresql-tcl-7.1.3-2
postgresql-devel-7.1.3-2
postgresql-python-7.1.3-2
postgresql-server-7.1.3-2
postgresql-contrib-7.1.3-2
postgresql-tk-7.1.3-2
postgresql-jdbc-7.1.3-2
postgresql-libs-7.1.3-2
postgresql-7.1.3-2
```

Der `rpm`-Befehl liefert eine Liste aller auf dem System installieten Pakete. Diese Liste kann in weiterer Folge sehr leicht mit `grep` gefiltert werden. Übrig bleibt eine Liste aller Pakete, die zu PostgreSQL gehören. Da es sich in diesem Fall um eine ziemlich veraltete Version der Datenbank handelt, empfiehlt es sich, diese Version zu deinstallieren. Um langes Tippen zu vermeiden, kann man einen kleinen Shell-Befehl basteln:

```
[root@duron hs]# rpm -e `rpm -qva | grep postgr |
sed -e 's/-7.*//gi' ` --nodeps
```

Wie Sie sehen können, wird `rpm` zweimal aufgerufen. Innerhalb der Backtips wird eine Liste der Paketnamen ohne Versionsnummer generiert. Nach dem Generieren dieser Liste wird diese an den äußeren `rpm`-Befehl weitergeleitet, der die entsprechenden Pakete deinstalliert. `-nodeps` stellt sicher, dass `rpm` nicht nach Abhängigkeiten sucht.

Nachdem Sie die alten Pakete (falls vorhanden) vom System entfernt haben, können Sie die aktuellen Versionen einspielen. Sofern Sie die aktuellen RPM-Pakete bereits heruntergeladen haben, können Sie diese installieren:

```
[root@duron root]# rpm -i postgresql-*rpm
```

Sind keine Fehler aufgetreten, haben Sie PostgreSQL erfolgreich installiert.

2.2.2 Installation auf Debian-basierten Systemen

Auf Debian-basierten Systemen funktioniert die Installation ähnlich wie bei Systemen, die auf `rpm` aufsetzen. Im Gegensatz zu `rpm`-Files besteht ein Debian-Archiv intern aus einem `ar`-Archiv, das wiederum aus drei Teilen besteht. Der erste Teil ist eine Dummy-Datei, deren Name als `Magic` dient. Der zweite Teil enthält ein `tar`-Archiv mit den Nutzdaten. Am Ende des Archives findet sich wieder ein `tar`-Archiv, das Hilfsprogramme für die Installation enthält. Diese Information ist interessant, weil sie es Ihnen leicht ermöglicht, Debian-Archive auf anderen, binärkompatiblen Systemen einzusetzen. Natürlich können Sie auch `Alien` verwenden, das Ihnen diese Arbeit abnimmt. Ein grundlegendes Verständnis der Paketverwaltung kann jedoch nicht schaden.

Nach diesem Exkurs in die Welt der Paketverwaltung sehen wir uns an, wie ein Paket installiert werden kann:

```
1 [root@debian deb]# dpkg -i *deb
```

Verwenden Sie einfach das Programm `dpkg`, um PostgreSQL zu installieren. Sollte es zu keinen Problemen gekommen sein, können Sie PostgreSQL nun ohne Bedenken verwenden. Für Freunde grafischer Oberflächen empfehlen wir, `dselect` zu verwenden.

Eine weitere Möglichkeit, PostgreSQL auf Ihr System zu spielen, ist `apt-get`.

2.2.3 Sourceinstallation unter Linux und FreeBSD

In vielen Fällen ist es sinnvoll, den Source Code anstatt Binaries zu installieren. Das ist besonders dann wichtig, wenn Sie planen, Modifikationen oder Erweiterungen zu implementieren.

Source-Installationen haben den Ruf, sehr kompliziert zu sein und viel Wissen zu erfordern. Das ist nur bedingt richtig, da der Installationsprozess gut dokumentiert ist und nur wenige Befehle notwendig sind.

Der erste Schritt bei der Installation ist, den Source-Code zu entpacken. Das kann sehr leicht mit `tar` gemacht werden:

```
1 [postgres@duron src]# tar xvfz postgresql-7.2.tar.gz
```

Das `z`-Flag von `tar` ist in den meisten Binärdistributionen verfügbar. Sollte das nicht der Fall sein, lässt sich das Problem auch elegant über eine Pipe lösen, wie das im nächsten Beispiel dargestellt ist:

```
1 [postgres@duron src]# gunzip -c postgresql-7.2.tar.gz | tar xvf -
```

Sobald die Sourcen entpackt sind, kann man in das soeben erstellte Verzeichnis wechseln und den Installationsvorgang beginnen. Im ersten Schritt muss `configure` gestartet werden. Dieses File dient dazu, die notwendigen Makefiles zu generieren, die anschließend zum Kompilieren der Software notwendig sind. Configure bietet eine Möglichkeit, alle Konfigurationsparameter gesammelt auszugeben.

Im Folgenden sind die wichtigsten Konfigurationsparameter kurz erläutert:

`–prefix`: Dieses Flag wird grob gesagt dazu benutzt, um das Installationsverzeichnis von PostgreSQL festzulegen. Sofern keine anderen Pfade festgelegt werden, wird der gesamte Baum relativ zu `$PREFIX` installiert.

`–datadir`: Dieses Flag definiert den Pfad zu den Datenbanktemplates, die notwendig sind, um einen neuen Datenbankcluster anzulegen. Wenn dieses Flag nicht gesetzt wird, wird `$PREFIX/share` verwendet. Was wichtig zu bemerken ist: das ist nicht das Verzeichnis, in dem die Files, aus denen Ihre Datenbanken bestehen, gespeichert werden — in diesem Verzeichnis liegen nur die Templates.

`–enable–locale`: Für Benutzer, die Support für Sprachen außer Englisch benötigen, empfiehlt es sich, dieses Flag einzuschalten. Hierbei ist zu bedenken, dass es zu geringfügigen Performance-Einbußen kommt.

`–enable–recode`: Dieses Flag ist notwenig, um mit Kyrillisch und dergleichen zu arbeiten. Unterstützt werden Zeichensätze, die pro Zeichen ein Byte allozieren.

`–enable–multibyte`: Verschiedene Zeichensätze (etwa Unicode) benötigen pro Zeichen zwei Bytes. Unterstützung für solche Zeichensätze kann mittels `–enable–multibyte` aktiviert werden.

`–with–tcl`: PostgreSQL unterstützt `Tcl`. Wenn dieses Flag eingeschaltet ist, werden `PL/Tcl` und einige andere `Tcl`-Komponenten installiert. Dieses Flag schaltet auch die `Tk`-Unterstützung ein.

`–without–tk`: Sofern die `Tcl`-Unterstützung eingeschalten ist, kann auf Wunsch `Tk` ausgeschalten werden.

`–with–perl`: Der Support für `Perl` besteht aus zwei Komponenten. Die erste Komponente ist eine an die C-Schnittstelle angelehnte `Perl`-Bibliothek. Der zweite Teil ist `PL/Perl`, eine eingebettete Sprache, die verwendet werden kann, um Extensions für PostgreSQL zu schreiben.

`–with–python`: PostgreSQL stellt Bibliotheken für Python zur Verfügung, die mit diesem Flag eingeschaltet werden können.

`–with–java`: Um Java verwenden zu können, ist dieses Flag zu aktivieren. Bei Java ist es notwendig, einige Vorkehrungen zu treffen, die in einem eigenen Unterpunkt beschrieben werden.

`–with–krb5`: PostgreSQL unterstützt Kerberos-Authentifizierung in den Versionen 4 und 5. Es kann nur jeweils eine der beiden Versionen aktiviert sein.

`–with–pam`: `PAM` ist das Standardmodul zur Authentifizierung unter Linux. `PAM` wird seit PostgreSQL 7.2 unterstützt.

`–with–openssl`: In sicherheitskritischen Bereichen kann verschlüsselte Übertragung sinnvoll sein. Zu diesem Zwecke bietet PostgreSQL eine Schnittstelle zu `OpenSSL`, die hiermit eingeschaltet werden kann. Um die Schnittstelle zu nutzen, ist es notwendig, dass `OpenSSL` auf Ihrem System installiert ist.

`–enable–odbc`: `ODBC` wird häufig in der Windows-Welt verwendet. PostgreSQL unterstützt zwei Implementierungen von UNIX ODBC (`unixODBC` und `iODBC`).

`–with–CXX`: Die C++-Schnittstelle setzt direkt auf die C-Schnittstelle auf und kann durch Aktivierung dieses Flags kompiliert werden.

Nach diesem kurzen Überblick über die Konfigurationsparameter ist es möglich, ein kleines Script zu schreiben, das PostgreSQL konfiguriert und kompiliert. Ziel ist es, den Installationsvorgang möglichst schnell und sinnvoll durchzuführen. Ein Script hat hier verschiedene Vorteile. Viele Leute kompilieren Programme ohne Scripts. Das hat zur Folge, dass Sie wenige Tage nach der Kompilierung nie mehr nachvollziehen können, was Sie gemacht haben und welche Komponenten aktiviert worden sind. Das ist ein wichtiger Punkt, dessen Bedeutung nicht unterschätzt werden soll. Im Fall von PostgreSQL ist das weniger tragisch, aber im Fall anderer Pakete können gewisse Flags das Verhalten der Software stark beeinflussen und es ist daher für die Applikationsentwicklung wichtig, nachvollziehen zu können, welches System und wie es installiert worden ist.

Wenn Sie PostgreSQL auf FreeBSD installieren wollen, müssen Sie `GNU make` installieren — das mit BSD gelieferte `Make` reicht nicht aus.

Das nächste Listing zeigt ein kurzes Programm, das alle gewünschten Aufgaben erfüllt:

```
1 #!/bin/sh
2
3 CFLAGS=' –march=athlon –O3 ' ./configure \
4         ––prefix=/usr/local/postgresql \
5         ––enable–locale \
6         ––enable–recode \
7         ––enable–multibyte \
8         ––with–tcl \
```

```
        --with-perl \
        --with-python \
        --with-pam \
        --with-openssl=/usr/include/openssl \
        --enable-odbc \
        --with-unixodbc \
        --with-CXX

# Kompilieren und Installation der Sourcen
make && make install
```

Sollte kein Fehler aufgetreten sein, ist PostgreSQL erfolgreich installiert. Im nächsten Schritt ist es notwendig, einen so genannten Datenbankcluster zu bauen. Im Falle von PostgreSQL ist ein Datenbankcluster eine Ansammlung von Datenbanken. Diese Datenbanken liegen allesamt auf einer Maschine und werden von einem so genannten Postmasterprozess verwaltet. Das Wort Cluster sollte hier nicht irreführen — es handelt sich nur um eine Maschine.

Als Erstes bei einem Datenbankcluster ist der Befehl `initdb` zu verwenden. Sollten Sie das gezeigte Script verwendet haben, befindet sich `initdb` unter `/usr/local/postgresql/bin`. Es empfiehlt sich, dieses Unterverzeichnis in den Pfad aufzunehmen.

Schauen wir uns nun kurz an, wie der Datenbankcluster angelegt werden kann:

```
bash-2.05$ /usr/local/postgresql/bin/initdb -D /data/postgres_db/
The files belonging to this database system will be owned by user
"postgres".
This user must also own the server process.

Fixing permissions on existing directory /data/postgres_db/... ok
creating directory /data/postgres_db//base... ok
creating directory /data/postgres_db//global... ok
creating directory /data/postgres_db//pg_xlog... ok
creating directory /data/postgres_db//pg_clog... ok
creating template1 database in /data/postgres_db//base/1... ok
creating configuration files... ok
initializing pg_shadow... ok
enabling unlimited row size for system tables... ok
creating system views... ok
loading pg_description... ok
vacuuming database template1... ok
copying template1 to template0... ok

Success. You can now start the database server using:

    /usr/local/postgresql/bin/postmaster -D /data/postgres_db/
or
   /usr/local/postgresql/bin/pg_ctl -D /data/postgres_db/ -l logfile
   start
```

Sie können leicht erkennen, dass der Datenbankcluster in diesem Beispiel unter `/data/postgres_db` angelegt wird. Der Ort dieses Verzeichnisses bleibt absolut Ihnen überlassen. Wichtig dabei ist, dass der User, den Sie verwenden, um PostgreSQL zu starten, der Eigentümer dieses Verzeichnisses ist. PostgreSQL darf aus Sicherheitsgründen nicht als Root gestartet werden und es empfiehlt sich daher, einen eigenen User anzulegen, der sich üblicherweise `postgres` nennt. Das ist eine Art Konvention, aber sicherlich kein Zwang. In diesem Fall operieren wir als User `postgres`.

Am Ende des Listings können Sie bereits sehen, wie der Datenbankdaemon gestartet werden kann. Das angegebene Beispiel zeigt, wie PostgreSQL ohne Unterstützung von TCP/IP gestartet werden kann. Das ist ein sehr wichtiger Punkt. Die angegebene Variante unterstützt nur UNIX-Sockets und kein TCP/IP. Mit anderen Worten, es gibt keine Möglichkeit, sich von einer entfernten Maschine zum Datenbankserver zu verbinden, weil PostgreSQL nicht auf Netzwerkverbindungen hört. Aus Sicherheitsgründen ist das sinnvoll, kann aber vielen Neulingen etwas Kopfzerbrechen bereiten.

Zum Starten von PostgreSQL kann entweder der Postmaster direkt gestartet oder auch auf `pg_ctl` zurückgegriffen werden, was in der Regel die etwas elegantere Variante ist. `pg_ctl` bietet eine Reihe von Möglichkeiten, die man sich mit `pg_ctl –help` leicht ansehen kann.

Starten wir nun PostgreSQL:

```
1 [postgres@duron src]# pg_ctl –D /data/postgres_db/ –l /tmp/logfile
2 –o "–i" start
3 postmaster successfully started
```

Das Flag `–o` sorgt dafür, dass das Flag `–i` an den Postmaster durchgeschleust wird. Das ist notwendig, um TCP/IP einzuschalten. Das ist bei lokalen Verbindungen nicht notwendig, aber es sei hier an dieser Stelle noch einmal mit Nachdruck erwähnt, dass es für Verbindungen von anderen Rechnern zu Ihrer Datenbank von essentieller Wichtigkeit ist.

Des Weiteren ist es sinnvoll, ein Logfile anzulegen. Das ermöglicht Ihnen zu verfolgen, was sich innerhalb der Datenbank abspielt und was auf Ihrem System passiert. Zur Fehleranalyse können Logfiles sehr wichtig sein. Um die Logfiles nicht unnötig anwachsen zu lassen, können Sie Apaches `logrotate` verwenden, was bei längeren Systemlaufzeiten viel Speicherplatz sparen kann und nicht unnötig Ihre Platte zumüllt.

Um herauszufinden, ob PostgreSQL aktiv ist, können Sie `ps` und `grep` verwenden. `ps` liefert eine Liste aller Prozesse, die auf Ihrem System laufen und mithilfe von `grep` können Sie diese nach `postgres` durchsuchen. Das folgende Listing zeigt, wie das erreicht werden kann:

```
[postgres@duron pgsql]# ps ax | grep post
31926 tty5     S      0:00 su - postgres
32118 tty5     S      0:00 /usr/bin/postmaster -i
32119 tty5     S      0:00 postgres: stats buffer process
32121 tty5     S      0:00 postgres: stats collector process
32123 tty5     S      0:00 grep post
```

Um die Datenbank wieder zu stoppen, können Sie ebenfalls `pg_ctl` verwenden. Zum Stoppen der Datenbank bietet PostgreSQL drei Möglichkeiten:

`smart`: Diese Methode erlaubt es, dass alle derzeit mit der Datenbank verbundenen User noch fertig arbeiten dürfen und die Datenbank dann sauber heruntergefahren wird. Laufende Queries werden davon nicht beinträchtigt.

`fast`: Diese Option sorgt dafür, dass alle Verbindungen sofort aber dennoch sauber geschlossen werden.

`immediate`: Wenn Sie PostgreSQL unverzüglich und unsauber beenden möchten, steht Ihnen `immediate` zur Verfügung.

Wenn Sie die Datenbank unsauber und hart beenden, kann es nicht zu Problemen oder zu Inkonsistenzen kommen, weil PostgreSQL intern auf unerwartete Unterbrechungen vorbereitet ist und die Files auf der Platte daher keinen Schaden nehmen können. Das ist sehr wichtig, weil es dem Benutzer höchste Datensicherheit garantiert und Ausfälle effizient zu verhindern hilft.

Nach diesem kurzen Überblick über die zur Verfügung stehenden Möglichkeiten wollen wir uns kurz ansehen, wie ein Shutdown praktisch funktioniert:

```
[postgres@duron src]# pg_ctl -D /data/postgres_db/ -m smart stop
waiting for postmaster to shut down.....done
postmaster successfully shut down
```

Vergessen Sie nicht, beim Shutdown die Position des Datenbankclusters anzugeben, da die Möglichkeit besteht, mehrere Postmaster simultan zu betreiben, die jeweils auf verschiedene TCP-Ports hören und auf verschiedene Datenbankcluster zugreifen.

2.2.4 Sourceinstallation mit Java Support unter Linux

Auf manchen Systemen wie RedHat 7.2 kann die Installation von PostgreSQLs Java-Modulen etwas mühsam sein, weil in der Standardinstallation ein Tool namens `ant` fehlt.

`Ant` ist eine Software, die als Ersatz für `Make` konzipiert worden ist und gänzlich auf XML-Basis arbeitet. Glaubt man den Entwicklern von `Ant`, hat das einige Vorteile, die ich hier nicht im Detail erläutern möchte.

Fügt man `–with–java` zum Aufruf von `Configure` hinzu, kann es zu Fehlermeldungen kommen, wie sie im folgenden Listing dargestellt sind:

```
checking whether to build Java/JDBC tools... yes
checking for jakarta–ant... no
checking for ant... no
checking for ant.sh... no
checking for ant.bat... no
checking whether  works... no
configure: error: ant does not work
```

`Ant` konnte nicht gefunden werden und daher ist es unmöglich, die Java-Schnittstellen von PostgreSQL zu kompilieren. Um das Problem zu lösen, muss `Ant` nachträglich installiert werden, was ein wenig mühsam sein kann.

2.2.5 Sourceinstallation unter Solaris

Im Reigen der kommerziellen UNIX-Systeme spielt Sun Microsystems eine gewichtige Rolle. Sun Solaris hat sich vor allem im Businessbereich gut etabliert und stellt für viele Unternehmen das Rückgrad der IT dar. Solaris-Systeme gelten als verhältnismäßig ausfallsicher und glänzen auch sonst durch eine Vielzahl interessanter und zum überwiegenden Teil sinnvoller Features.

Um die Vorzüge von PostgreSQL auf Solaris auskosten zu dürfen, bedarf es einiger Vorarbeiten, die wir in diesem Kapitel kurz erläutern möchten. Alle Schritte wurden auf Solaris 8 (10/2001) durchgeführt.

Um PostgreSQL kompilieren zu können, benötigen Sie einen C-Compiler. Sie können entweder die Sun Compiler Suite oder ganz einfach den GNU C-Compiler verwenden, wie wir das in diesem Beispiel machen werden. Zusätzlich zu einem C-Compiler sind auch noch einige andere Tools wie etwa `Gzip` und `GNU Make` notwendig. `Gzip` wird benötigt, um das Source-Archiv zu entpacken und `GNU Make` wird zum Abarbeiten der Makefiles gebraucht. Zusätzlich empfiehlt es sich noch, `Readline` und einige andere Tools zu installieren. Üblicherweise werden noch die Pakete `Perl`, `ncurses`, `patch`, `binutils` und `fileutils` installiert. Zum Generieren der Makefiles sind `autoconf` und `automake` vonnöten. Sollten Sie einen CVS Snapshot installieren wollen, werden Sie `Bison` und `Flex` benötigen, da diese zum Generieren des Parsers gedacht sind. Sollten Sie das `Tar`-Archiv mit den Sources statt dem CVS-Auszug installieren wollen, wird der Parser nicht mehr generiert und `Bison` und `Flex` sind somit keine Bedingung mehr.

Zum Installieren der oben genannten Pakete suchen Sie am besten `www.sunfreeware.com` auf und laden sich die entsprechende Software herunter. Danach können Sie die Pakete mittels `gunzip` entpacken und mithilfe von `pkgadd` (`pkgadd —d`) installieren. Sofern es dabei zu keinen Problemen gekommen ist, können Sie PostgreSQL wie gewohnt und wie oben bereits erklärt installieren.

2.3 Installation unter Windows

Um PostgreSQL unter Windows zum Laufen zu bringen, ist etwas mehr Aufwand als unter UNIX notwendig. Die Gründe dafür sind vielfältig und werden an dieser Stelle nicht erläutert.

Als Basis einer PostgreSQL-Installation dient das `Cygwin`-Paket, das kostenlos von `www.redhat.com` heruntergeladen werden kann. Im Prinzip ist das Paket eine Zusammenstellung von UNIX-Software in einer Windows-Umgebung. Mithilfe von `Cygwin` lassen sich viele Projekte realisieren, die bei Einsatz von Windows-Tools in dieser Form nicht möglich wären.

Die Installation von `Cygwin` läuft weitgehend selbstständig ab und ist gut dokumentiert beziehungsweise selbsterklärend.

Nach der Installation von `Cygwin` sollte sich PostgreSQL bereits auf Ihrem System befinden. Um einen Datenbankcluster anzulegen und den Daemon zu starten, sind jedoch noch einige Arbeiten notwendig. Zuallererst müssen Sie `cypipc` installieren. Das Paket ist notwendig, da Windows nicht die gewünschte Art von Shared Memory zur Verfügung stellt. Das Paket kann unter `http://www.neuro.gatech.edu/users/cwilson/cygutils/V1.1/cygipc/` gratis heruntergeladen werden. Bitte stellen Sie sicher, dass Sie eine Version verwenden, die neuer oder gleich 1.04 ist, da es andernfalls zu Problemen kommen kann.

Um das Paket zu installieren, müssen Sie es erst entpacken:

```
$ tar -C / -xjf cygipc-${version}.tar.bz2
```

Stellen Sie weiter sicher, dass das `bin`-Verzeichnis von `Cygwin` vor den Windows-Pfaden gereiht ist — das ist notwendig, um zu garantieren, dass die Version `sort.exe` von `Cygwin` verwendet wird. Nach diesem Schritt können Sie den IPC-Daemon starten, was folgendermaßen geschehen kann:

```
ipc-daemon &
```

Wenn Sie den Daemon bereits als Service installiert haben, können Sie ihn so starten:

```
net start ipc-daemon
```

Vergessen Sie auf keinen Fall, den IPC-Daemon zu aktivieren, da PostgreSQL sonst de facto nutzlos ist.

In der Regel läuft PostgreSQL auch unter Windows ohne Probleme und PostgreSQL selbst ist als stabil anzusehen, wie das auch unter UNIX der Fall ist. Die Performance der Datenbank unter Windows ist aber schlecht.

2.4 Testen der Datenbank und Infos zum Frontend

Nachdem Sie PostgreSQL installiert und den Daemon gestartet haben, können Sie die ersten Schritte mit der Datenbank wagen. Um herauszufinden, ob die Datenbank funktioniert, können Sie sich eine Liste der am System installierten Datenbanken ausgeben lassen. Das kann mit `psql` erreicht werden:

```
[postgres@duron pgsql]$ psql -l
        List of databases
   Name    |  Owner   | Encoding
-----------+----------+-----------
 template0 | postgres | SQL_ASCII
 template1 | postgres | SQL_ASCII
(2 rows)
```

Obigem Listing ist zu entnehmen, dass defaultmäßig zwei Datenbanken am System verfügbar sind. Die Datenbanken `template0` und `template1` sind Schablonen. `template1` wird geklont, sobald eine neue Datenbank angelegt wird. Das impliziert, dass alle Funktionen, Tabellen und Datentypen, die in `template1` verfügbar sind, auch in die neue Datenbank übernommen werden. Das ist wichtig zu wissen, weil es Ihnen ermöglicht, gewisse Erweiterungen für alle in der Zukunft erstellten Datenbanken gesammelt dem System hinzuzufügen.

Nachdem Sie nun gesehen haben, welche Datenbanken am System verfügbar sind, wollen wir versuchen, uns mit einer Datenbank zu verbinden. Hierfür gibt es das Programm `psql`. Es unterstützt eine Vielzahl von Optionen und Argumenten, die sehr flexibles Arbeiten erlauben. Das nächste Listing enthält eine Übersicht über alle verfügbaren Möglichkeiten, die von PostgreSQL 7.2 angeboten werden:

```
[hs@duron hs]$ psql --help

This is psql, the PostgreSQL interactive terminal.

Usage:
  psql [options] [dbname [username]]

Options:
  -a              Echo all input from script
  -A              Unaligned table output mode (-P format=unaligned)
  -c COMMAND      Run only single command (SQL or internal) and exit
  -d DBNAME       Specify database name to connect to (default: hs)
  -e              Echo commands sent to server
  -E              Display queries that internal commands generate
  -f FILENAME     Execute commands from file, then exit
  -F STRING       Set field separator (default: "|") (-P fieldsep=)
  -h HOSTNAME     Specify database server host (default: local
                  socket)
```

```
  -H              HTML table output mode (-P format=html)
  -l              List available databases, then exit
  -n              Disable enhanced command line editing (readline)
  -o FILENAME     Send query results to file (or |pipe)
  -p PORT         Specify database server port (default: 5432)
  -P VAR[=ARG]    Set printing option 'VAR' to 'ARG' (see \pset
                  command)
  -q              Run quietly (no messages, only query output)
  -R STRING       Set record separator (default: newline) (-P
                  recordsep=)
  -s              Single step mode (confirm each query)
  -S              Single line mode (end of line terminates SQL
                  command)
  -t              Print rows only (-P tuples_only)
  -T TEXT         Set HTML table tag attributes (width, border) (-P
                  tableattr=)
  -U NAME         Specify database user name (default: hs)
  -v NAME=VALUE   Set psql variable 'NAME' to 'VALUE'
  -V              Show version information and exit
  -W              Prompt for password (should happen automatically)
  -x              Turn on expanded table output (-P expanded)
  -X              Do not read startup file (~/.psqlrc)

For more information, type "\?" (for internal commands) or "\help"
(for SQL commands) from within psql, or consult the psql section in
the PostgreSQL documentation.

Report bugs to <pgsql-bugs@postgresql.org>.
```

Wie Sie sehen können, bietet sich eine Fülle von Möglichkeiten. Es ist sogar möglich, sich mit einer Datenbank zu verbinden, die nicht auf einer lokalen Maschine liegt. Das ist besonders wichtig, wenn man die Verbindung zu einem anderen Rechner testen will, beziehungsweise wenn man eine Datenbank irgendwo im Netz administrieren soll.

Um sich zur Datenbank zu verbinden, kann man `psql` mit dem Namen der Datenbank aufrufen:

```
[postgres@duron pgsql]$ psql template1
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

template1=#
```

Nach dem Start von `psql` landet man in einer komfortablen Shell, die dem Benutzer keine Wünsche offen lässt. Um die Funktionsweise der Datenbank zu testen, können Sie eine einfache Berechnung durchführen:

```
template1=# SELECT 1+1;
 ?column?
----------
        2
(1 row)
```

Sollte es bei diesem einfachsten aller SQL-Statements zu keinem Problem gekommen sein, funktioniert PostgreSQL einwandfrei.

Im nächsten Schritt zahlt es sich aus, die interaktive Shell ein wenig genauer anzusehen. Mittels des Befehls `\?` können Sie sich eine Liste aller von der Shell unterstützten Befehle ausgeben lassen. Das ist hilfreich, wenn Sie sich einen Überblick über die unterstützten Datentypen, Funktionen oder die in der Datenbank vorhandenen Tabellen verschaffen wollen. Wenn Sie sich etwa in der aktuellen Datenbank eine Liste aller vorhandenen Tabellen ausgeben lassen wollen, können Sie das mit dem Befehl `\d` machen:

```
template1=# \d
No relations found.
```

Wie Sie erkannt haben, sind keine Tabellen vom Benutzer definiert worden. Es gibt Tabellen in der Datenbank, aber dabei handelt es sich um Systemtabellen, die in dieser Ansicht nicht angezeigt werden.

Ein wesentlicher Punkt bei der täglichen Arbeit mit `psql` ist die Art, wie mit dem Ende von Befehlen umgegangen wird. Es reicht nicht, ein SQL-Statement einfach mit einem Zeilenumbruch abzuschließen. Sollte der String nicht mit einem Strichpunkt terminiert werden, nimmt `psql` an, dass der Befehl weitergeht. Speziell bei mehrzeiligen Befehlen kann das gewisse Vorteile haben, weil es in bestimmten Fällen die Übersichtlichkeit steigert. In der praktischen Arbeit sieht das dann beispielsweise so aus:

```
template1=# SELECT 1
template1-# +1
template1-# ;
 ?column?
----------
        2
(1 row)
```

Sollte Ihnen dieses Verhalten der Tools nicht angenehm sein, können Sie es bequem beim Starten von `psql` mit dem Flag -S abschalten. In diesem Fall würde das Zeilenende auch automatisch das Ende des Befehles bedeuten.

```
template1^# SELECT 1+1;
 ?column?
----------
        2
(1 row)

template1^# SELECT 1+1
 ?column?
----------
        2
(1 row)
```

Aus dem Listing geht hervor, dass der zweite `SELECT`-Befehl sofort abgeschlossen wurde, obwohl kein Strichpunkt am Ende des Kommandos zu finden ist.

Soweit ein erster Einblick in das Standard-Frontend von PostgreSQL. Um das Programm wieder zu beenden, verwenden Sie einfach den Befehl `\q`.

Kapitel 3

SQL

Um mit einer Datenbank vernünftig interagieren zu können, benötigt man eine Sprache. Sobald eine Datenbank als Serverdatenbank einsetzbar ist, bedeutet das, dass Sprachen notwendig sind, die sowohl der Client als auch der Server verstehen.

SQL ist eine Sprache zur Kommunikation mit Datenbanken. Wie bei allen Sprachen gibt es auch bei SQL eine Vielzahl von Dialekten und Varianten, die untereinander nicht wirklich hundertprozentig zusammenpassen. Um diesem Problem einen Riegel vorzuschieben, ist 1986 beziehungsweise 1987 der erste Entwurf von standardisiertem SQL veröffentlicht worden — die Geburts von SQL 1.

1992 — schließlich — hat die Zeit von SQL2 (auch SQL-92 oder SQL/92) begonnen. SQL 92 ist derzeit der de facto-Standard und es wird noch ein wenig dauern bis SQL 3 (verabschiedet am 26. Juli 1995) von einer noch größeren Zahl von Anbietern unterstützt werden wird.

Das oberste Ziel des PostgreSQL-Teams ist es, eine möglichst flexible und ANSI SQL-kompatible Datenbank zur Verfügung zu stellen. Im Gegensatz zu einigen anderen, ebenfalls frei verfügbaren, Datenbanken hält sich PostgreSQL so weit wie möglich an den Standard, um so kompatibel wie möglich zu sein. Nur durch Kompatibilität ist es möglich, am Markt auf Dauer zu bestehen und eine größere Anzahl von Usern hinter sich zu sammeln.

In diesem Kapitel werden wir uns mit SQL beschäftigen und die Implementierung von PostgreSQL genau durchleuchten und lernen, diese effizient und schnell zu benutzen.

3.1 Datenbanken anlegen und löschen

Bevor wir uns ins Geschehen stürzen, legen wir sinnvollerweise eine neue Datenbank an, um `template1` nicht zu verschmutzen. Das kann mit dem Befehl `createdb` gemacht werden. Hier sehen Sie einen Überblick über die Syntax des Befehles:

```
[postgres@duron pgsql]$ createdb --help
createdb creates a PostgreSQL database.

Usage:
  createdb [options] dbname [description]

Options:
  -D, --location=PATH            Alternative place to store the database
  -T, --template=TEMPLATE        Template database to copy
  -E, --encoding=ENCODING        Multibyte encoding for the database
  -h, --host=HOSTNAME            Database server host
  -p, --port=PORT                Database server port
```

```
  -U, --username=USERNAME         Username to connect as
  -W, --password                  Prompt for password
  -e, --echo                      Show the query being sent to the backend
  -q, --quiet                     Don't write any messages

By default, a database with the same name as the current user is created.

Report bugs to <pgsql-bugs@postgresql.org>.
```

Der Befehl ist ungeheuer mächtig und erlaubt es, verschiedenste Attribute der neuen Datenbank zu definieren. Ein wichtiges Attribut ist zweifelsohne der verwendete Zeichensatz. Defaultmäßig wird SQL_ASCII verwendet, was in der Regel für die allermeisten Anwendungen reichen sollte. Das folgende Listing zeigt, wie Sie eine Datenbank namens `buch` anlegen können, die wir in diesem Buch verwenden werden:

```
[postgres@duron pgsql]$ createdb buch
CREATE DATABASE
```

In diesem Falle wird die Datenbank im mit `initdb` definierten Datenbankcluster abgelegt. Als Zeichensatz wird SQL_ASCII verwendet. Versuchen wir nun eine Datenbank anzulegen, die auf Unicode basiert.

```
[postgres@duron db]$ createdb -E unicode
CREATE DATABASE
```

Ingesamt haben wir zwei Datenbanken generiert. Um abzufragen, welche Datenbanken nun im System sind, kann wieder `psql -l` verwendet werden:

```
[postgres@duron db]$ psql -l
List of databases
   Name    |  Owner   | Encoding
-----------+----------+-----------
 buch      | postgres | SQL_ASCII
 postgres  | postgres | UNICODE
 template0 | postgres | SQL_ASCII
 template1 | postgres | SQL_ASCII
(4 rows)
```

Da wir keinen Namen für die neue Datenbank angegeben haben, trägt sie den Namen des Benutzers, der sie angelegt hat. Hin und wieder kann dieses Verhalten erwünscht sein, aber wir empfehlen trotzdem, den Namen besser explizit vorzugeben. Der Zeichensatz ist wie gewünscht Unicode.

Um die Datenbank wieder zu löschen, kann der Befehl `dropdb` verwendet werden:

```
[postgres@duron db]$ dropdb postgres
DROP DATABASE
```

PostgreSQLs-Tools stellen in der Regel eine Option zur Verfügung, die anzeigt, welche Befehle tatsächlich während einer Operation an die Datenbank geschickt werden. Das kann hilfreich sein, um die Datenbank besser zu verstehen und sich ab und zu auch ein wenig zeigen zu lassen, wie gewisse Dinge funktionieren. Die Option, von der ich spreche, nennt sich -e. Das nächste Listing zeigt, was an die Datenbank geschickt wird, wenn eine Datenbank angelegt beziehungsweise gelöscht wird:

```
1 [postgres@duron db]$ createdb -E unicode -e; dropdb -e postgres
2 CREATE DATABASE "postgres" WITH ENCODING = 'unicode'
3 CREATE DATABASE
4 DROP DATABASE "postgres"
5 DROP DATABASE
```

Es ist zu sehen, dass es sich um reinen SQL-Code handelt, der von der Datenbank interpretiert wird wie jeder andere Code auch. Das ist wichtig zu wissen, weil es Ihnen ermöglicht, alle Operationen auch mit Hilfe eines Tools durchzuführen, das nichts anderes tut als SQL Code an PostgreSQL zu schicken.

3.2 Einfache SQL-Operationen

SQL ist eine Sprache, die relativ schnell erlernt werden kann. Einfache Abfragen können sehr schnell realisiert werden und es ist leicht, die Idee von SQL zu verstehen. In diesem Abschnitt werden wir uns ein wenig genauer mit einfachen Abfragen und Möglichkeiten der Modifizierung von Daten beschäftigen.

3.2.1 Tabellen anlegen und löschen

Die Grundbausteine von (objekt)-relationalen Datenbanken sind Tabellen. Tabellen sind im Wesentlichen Listen, die zusammenhängende Daten in strukturierter Form speichern. Nehmen wir etwa an, Sie wollen Daten über Personen speichern. Personen haben verschiedene Merkmale wie etwa Namen, Geschlecht oder Haarfarbe. Alle diese Daten können in einer Tabelle gespeichert werden. Für jede Person wird in der Regel eine Zeile angelegt. Tabellen bestehen aus Spalten. Abhängig von den Daten, die Sie in der Tabelle speichern wollen, müssen Sie jedem Feld einen entsprechenden Datentyp zuordnen. Der Name einer Person könnte z.B. als Zeichenkette abgelegt werden. Das Einkommen wäre beispielsweise eine Gleitkommazahl und das Feld, das anzeigt, ob eine Person verheiratet ist oder nicht, könnte als so genannter Boolescher Wert abgelegt werden (Ja/Nein). Datentypen sind notwendig, um Daten effizient zu speichern. Natürlich ist es auch möglich, alles als Textwerte zu speichern, aber das würde die Performance Ihrer Datenbank massiv drosseln und auch sonst große Nachteile mit sich bringen.

PostgreSQL unterstützt eine große Zahl bereits vordefinierter Datentypen. Das nächste Listing zeigt eine Liste der wichtigsten, von PostgreSQL 7.2 unterstützten, Datentypen sowie eine kurze Erläuterung zum jeweiligen Datentyp:

`integer`: Integer wird auch als `int4` bezeichnet und kann eine 32-Bit-Zahl speichern. Der Wertebereich reicht von etwa −2 Mrd. bis +2 Mrd.

`bigint`: Der Zahlenbereich von 64-Bit-Integer-Werten umfasst 18 Stellen.

`numeric`: Wenn Sie Zahlen beliebiger Genauigkeit und Länge speichern wollen, ist `numeric` der richtige Datentyp. Wenn Sie etwa `numeric(10,3)` verwenden, bedeutet das, dass die Zahl in Summe zehn Stellen lang ist, von denen drei Stellen als Nachkommastellen verwendet werden.

`char`: `Character` sind Zeichenketten fixer Länge. `char(5)` ist beispielsweise für das Speichern von genau fünf Zeichen ausgelegt. Der intern verbrauchte Speicher ist immer derselbe — egal, ob Sie das Feld mit nur zwei Zeichen befüllen oder nicht.

`varchar`: Wenn Sie Zeichenketten variabler Länge speichern wollen, bietet sich varchar an. `varchar(15)` beispielsweise ermöglicht es Ihnen, Zeichenketten mit einer maximalen Länge von 15 Zeichen zu speichern. Befüllen Sie das Feld mit lediglich vier Zeichen, wird natürlich dementsprechend weniger Speicher verbraucht.

`text`: Wenn Sie Texte beliebiger Länge speichern wollen, empfiehlt es sich, `text` zu verwenden, weil Texte keinerlei Längeneinschränkungen aufweisen, wie das bei `varchar` der Fall ist.

`double [precision]`: PostgreSQLs double-Werte sind ein Gegenstück zu denen, die Sie vermutlich von C kennen werden. Es werden acht Bytes Storage verbraucht.

`time without time zone`: Dieser Datentyp dient zum Speichern von Zeitwerten, die nicht zeitzonenbezogen sind. Es werden acht Bytes Storage alloziert.

`time with time zone`: Dieser Datentyp dient zum Speichern von Zeitwerten, die zeitzonenbezogen sind. Es werden zwölf Bytes Storage alloziert.

`date`: Mit nur vier Bytes Speicher können Sie einen Datumswert abspeichern.

`timestamp with time zone` und `timestamp without time zone`: Diese beiden Datentypen sind eine Kombination aus Datums- und Zeitbereichen.

`interval`: Intervalle dienen zum Abspeichern von Zeitdifferenzen. Ein Bereich kann von −178 000 000 bis +178 000 000 reichen. Es werden zwölf Bytes Storage alloziert.

Außer bei Tagen beträgt die Genauigkeit von Datentypen, die auf Zeit bezogen sind, eine Mikrosekunde.

Wenn Sie eine Übersicht aller im System vorhandenen Datentypen benötigen, können Sie `\dT` in Ihrer `psql`-Shell verwenden. PostgreSQL wird daraufhin eine ansehnliche Liste präsentieren.

Wollen wir nun eine Tabelle zum Speichern von Tabelleninformationen anlegen:

```
1 CREATE TABLE person (
2         name varchar(30),
3         geschlecht char(1),
4         einkommen numeric(10, 2)
5 );
6
7
```

Wenn Sie den Code aus dem vorherigen Listing an die Datenbank schicken wollen, können Sie das entweder direkt mittels der interaktiven Shell machen oder ganz einfach Ihren Kommandozeileninterpreter benutzen, wie das im nächsten Beispiel dargestellt ist:

```
1 [postgres@duron db]$ psql buch < code.sql
2 CREATE
```

Sollte es zu keinem Problem gekommen sein, haben Sie die Tabelle erfolgreich angelegt.

Um Informationen über die Tabelle abzufragen, können Sie `psql` und `\d` verwenden:

```
1 buch=# \d person
2                   Table "person"
3    Column   |         Type          | Modifiers
4 ------------+-----------------------+-----------
5  name       | character varying(30) |
6  geschlecht | character(1)          |
7  einkommen  | numeric(10,2)         |
```

Wie Sie sehen können, werden alle Spaltennamen sowie der Datentyp aufgelistet. Die Ausgabe der Daten erfolgt in Form einer Tabelle. Der Grund dafür ist denkbar einfach: Alle von `psql` abgesetzten Kommandos sind reiner SQL-Code, der direkt von PostgreSQL ausgeführt werden kann.

Wie Sie im letzten Beispiel gesehen haben, können Sie den Befehl `CREATE TABLE` verwenden, um Tabellen zu erzeugen. Der Befehl `CREATE TABLE` ist sehr mächtig

und unterstützt eine Vielzahl von Features. Speziell im Hinblick auf Datenintegrität bleibt kaum ein Wunsch offen und eine breite Palette von Möglichkeiten kann mit PostgreSQL ohne Probleme und zuverlässig abgedeckt werden. Das folgende Listing zeigt eine Übersicht der von PostgreSQL unterstützten Syntax:

```
buch=# \h CREATE TABLE
Command:     CREATE TABLE
Description: define a new table
Syntax:
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
    { column_name data_type [ DEFAULT default_expr ]
    [ column_constraint [, ... ] ]
    | table_constraint }  [, ... ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL |
  MATCH PARTIAL ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
    IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) |
  PRIMARY KEY ( column_name [, ... ] ) |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ (
    refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL ] [ ON DELETE action ] [ ON UPDATE
    action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
    IMMEDIATE ]

```

Mithilfe von `\h` kann eine Übersicht über die Syntax des jeweiligen Befehles generiert werden. Wichtig dabei ist, das Ergebnis auch korrekt interpretieren zu können. Für diejenigen unter Ihnen, die mit kontextfreien Grammatiken, Regular Expressions oder Formalsprachen im Allgemeinen vertraut sind, sollte das kein Problem

darstellen. Für alle anderen seien hier die wesentlichen Punkte kurz zusammengefasst:

Eckige Klammern bedeuten, dass der geklammerte Teil optional ist. Das Wort `LOCAL` beispielsweise ist optional und muss nicht angegeben werden. Senkrechte Striche (|), so genannte Pipe-Symbole, bedeuten ODER. Es kann entweder die Option links oder rechts von der Pipe angegeben werden. Im Falle von `TEMP` und `TEMPORARY` bedeutet das, dass entweder die eine oder die andere Option angegeben werden kann. Da der gesamte Block in eckigen Klammern geschrieben ist, muss jedoch keines der beiden angegeben werden: Der ganze Block ist optional. Geschwungene Klammern werden verwendet, um Blöcke zu verbinden. Wenn Sie sich die Syntaxübersicht ein wenig genauer ansehen, werden Sie sehr schnell verstehen, wie das System funktioniert, und Sie werden diese Art der kompakten Darstellung lieben lernen. Wir werden uns im Laufe dieses Buches noch sehr intensiv mit dem Befehl `CREATE TABLE` beschäftigen und ich werde Sie immer wieder auf die mittels `\h` erzeugten Übersichten hinweisen.

Da Sie nun gesehen haben, wie Tabellen angelegt werden können, ist es auch an der Zeit zu zeigen, wie Tabellen zu löschen sind. Mit dem Befehl DROP TABLE ist es einfach, eine Tabelle aus der Datenbank zu entfernen. Jetzt folgt ein Blick auf die Syntaxübersicht:

```
1 buch=# \h DROP TABLE
2 Command:     DROP TABLE
3 Description: remove a table
4 Syntax:
5 DROP TABLE name [, ...]
```

Wieder verwenden wir `\h`, um die Übersicht zu erhalten. Wie Sie feststellen, ist es einfach, den Befehl zu benutzen. Fügen Sie einfach den Namen der Tabelle und einen Strichpunkt zu `DROP TABLE` hinzu und führen Sie den Befehl aus. PostgreSQL wird dann die Tabelle für Sie entfernen.

3.2.2 Daten einfügen und abfragen

Wenn Sie es geschafft haben, eine Tabelle anzulegen, können Sie beginnen, Daten in diese einzufügen beziehungsweise einfache Abfragen zu tätigen.

Das Einfügen von Daten kann mit dem Befehl `INSERT` sehr einfach durchgeführt werden. Hier ist wieder ein Überblick über die Syntax von `INSERT`. Mithilfe des anschließenden Beispieles werden Sie eine Syntaxübersicht sehr schnell und sehr viel leichter verstehen lernen:

```
buch=# \h INSERT
Command:     INSERT
Description: create new rows in a table
Syntax:
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( expression [, ...] ) | SELECT query }
```

Und hier ein Beispiel:

```
buch=# INSERT INTO person VALUES ('Hans', 'm', '45000');
INSERT 16563 1
```

Der Befehl beginnt mit `INSERT INTO`. Danach kann in Klammern eine Liste von Spalten angegeben werden, was wir hier nicht gemacht haben. Nach dem Schlüsselwort `VALUES` werden die zur Tabellen hinzuzufügenden Werte aufgelistet. Hierbei werden die einzelnen Werte in einfachen Anführungszeichen gelistet. Die jeweiligen Spalten sind mit Beistrich getrennt. Sofern das Schlüsselwort `INSERT`, die Object ID der Datenzeile sowie die Zahl der vom `INSERT` betroffenen Zeilen zurückgegeben worden sind, hat das `INSERT`-Statement funktioniert.

Um die Daten in der Tabelle abzufragen, können Sie ein einfaches `SELECT`-Kommando verwenden:

```
buch=# SELECT * FROM person;
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
(1 row)
```

Es wird genau eine Zeile zurückgegeben. Der `*` bedeutet, dass alle Spalten der Tabelle im Ergebnis aufscheinen sollen. Wenn Sie nur die ersten beiden Spalten wollen, muss die Query wie folgt aussehen:

```
buch=# SELECT name, geschlecht FROM person;
 name | geschlecht
------+------------
 Hans | m
(1 row)
```

Die Spalten sind explizit aufzuzählen. Selbiges gilt für `INSERT`-Statements. Wenn Sie in einer Tabelle nur einige wenige Spalten anfüllen möchten, müssen Sie die gewünschten Spalten auflisten, wie es im nächsten Beispiel zu sehen ist:

```
buch=# INSERT INTO person (name, geschlecht) VALUES ('Paul', 'm');
INSERT 24748 1
```

Nach dem Namen für die Tabelle werden die einzelnen Spalten angeführt. Die Reihenfolge der Spalten ist dabei von Bedeutung, da sie der Anordnung der Werte entsprechen muss.

Wenn Sie nun sehen wollen, wie die Werte in der Datenbank gelistet werden, können Sie ein einfaches `SELECT`-Statement absetzen, wie Sie das auch schon zuvor gesehen haben:

```
buch=# SELECT * FROM person;
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
 Paul | m          |
(2 rows)
```

In diesem Fall befinden sich zwei Werte in der Tabelle. In der Regel wollen Sie nicht immer alle Daten aus einer Tabelle abfragen. Zu diesem Zwecke ist es notwendig, Selektionen vorzunehmen. Das nächste Beispiel zeigt alle Personen, die Hans heißen:

```
buch=# SELECT * FROM person WHERE name='Hans';
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
(1 row)
```

Die Selektion erfolgt durch eine so genannte `WHERE` Clause. Die genannte Bedingung ist, dass der Name in der Spalte gleich Hans sein muss. Zu beachten ist, dass der Wert unter einfachen Anführungszeichen übergeben wird. Wenn Sie darauf vergessen, kommt es zu einem Fehler, wie er im nächsten Listing gezeigt wird:

```
buch=# SELECT * FROM person WHERE name=Hans;
ERROR:  Attribute 'hans' not found
ERROR:  Attribute 'hans' not found
```

Die Spalte names hans kann nicht gefunden werden. Der Grund dafür ist einfach: Da sowohl der String name als auch der String Hans ohne Anführungszeichen an die Datenbank übergeben werden, ist es unmöglich, Werte von Spaltennamen zu unterscheiden. Das ist ein wichtiger Punkt, den Sie bei der täglichen Arbeit mit der Datenbank im Kopf behalten sollten.

Die nächste Abfrage zeigt, wie Sie Bedingungen verknüpfen können. Diesmal werden alle Personen extrahiert, die Hans heißen und zusätzlich noch weiblich sind:

```
buch=# SELECT * FROM person WHERE name='Hans' AND geschlecht='f';
 name | geschlecht | einkommen
------+------------+-----------
(0 rows)
```

Wie man am Ergebnis sehen kann, gibt es keine Personen in der Datenbank, die beide Bedingungen erfüllen. Wenn Sie alle finden wollen, bei denen entweder die eine oder die andere Bedingung zutrifft, können Sie `OR` verwenden:

```
buch=# SELECT * FROM person WHERE name='Hans' OR geschlecht='f';
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
(1 row)
```

In diesem Fall wird genau ein Datensatz geliefert, da es nur diesen einen gibt, der alle Kriterien der Abfrage erfüllt.

Was müssen Sie tun, wenn Sie alle Werte finden wollen, die kein Einkommen haben? Hier ist ein Beispiel für einen nicht funktionierenden Ansatz:

```
buch=# SELECT * FROM person WHERE einkommen = '';
ERROR:  Bad numeric input format ''
ERROR:  Bad numeric input format ''
```

Eine Zahl ist kein leerer String und daher scheitert die Abfrage mit einem Syntaxfehler. Leere Werte werden als so genannte `NULL`-Werte bezeichnet. Es ist wichtig zu wissen, dass `NULL` nichts mit der Zahl 0 zu tun hat. `NULL` ist `NULL` und nichts anderes, wie die nächste Abfrage einfach beweist:

```
buch=# SELECT * FROM person WHERE einkommen = 0;
 name | geschlecht | einkommen
------+------------+-----------
(0 rows)
```

Da kein Einkommen nicht bedeutet, dass das Einkommen 0 ist, werden keine Werte zurückgegeben. Um nach `NULL`-Werten zu suchen, ist eine Abfrage wie die im folgenden Listing notwendig:

```
buch=# SELECT * FROM person WHERE einkommen IS NULL;
 name | geschlecht | einkommen
------+------------+-----------
 Paul | m          |
(1 row)
```

Hier wird das richtige Ergebnis generiert. Bis PostgreSQL 7.1 gilt auch die folgende Variante, die in PostgreSQL 7.2 jedoch nicht mehr zwingend funktionieren

muss. Wir werden im Kapitel über Serveradministration sehen, welche Modifikationen vorzunehmen sind, um dieses alte Verhalten einzuschalten. Vorher kommen wir jedoch zu unserem Listing:

```
buch=# SELECT * FROM person WHERE einkommen=NULL;
 name | geschlecht | einkommen
------+------------+-----------
(0 rows)
```

Deutlich sichtbar ist das Ergebnis falsch, weil die Standardkonfiguration von PostgreSQL das alte Verhalten nicht unterstützt.

Wenn Sie abfragen wollen, welche Einkommen nicht `NULL` sind, können Sie folgende Query verwenden:

```
buch=# SELECT * FROM person WHERE einkommen IS NOT NULL;
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
(1 row)
```

Das Ergebnis enthält genau einen Wert.

Da Sie nun gesehen haben, wie man alle Werte finden kann, die nicht `NULL` sind, werden Sie jetzt erfahren was man machen kann, um alle Werte zu finden, die nicht einen bestimmten Wert haben. Dazu benötigen Sie einen Operator, der für die Negation zuständig ist:

```
buch=# SELECT * FROM person WHERE name <> 'Hans';
 name | geschlecht | einkommen
------+------------+-----------
 Paul | m          |
(1 row)
```

In diesem Fall haben Sie alle Personen gefunden, die nicht Hans heißen.

Ein wichtiger Punkt, wenn Sie mit dem Ungleich-Operator arbeiten, ist die Behandlung von `NULL`-Werten. Nehmen wir an, Sie wollen alle Einträge von Personen finden, die nicht 45.000 Geldeinheiten verdienen. Intuitiv würde man folgende Lösung versuchen:

```
buch=# SELECT * FROM person WHERE einkommen <> 45000;
 name | geschlecht | einkommen
------+------------+-----------
(0 rows)
```

In der Regel erwartet man, dass genau ein Datensatz im Ergebnis zu finden sein wird. Das ist in diesem Fall nicht so und es handelt sich definitiv um keinen Bug in

PostgreSQL. Die Wurzel dieses Verhaltens liegt in der ANSI SQL-Spezifikation. Laut Definition liefert ein Vergleich immer `FALSE`, außer Sie testen explizit auf `NULL`-Werte. Um es nochmals auf den Punkt zu bringen: Das ist keine Schwäche von PostgreSQL, sondern ein Beweis für dessen ANSI SQL-Kompatibilität und die Qualität der Software.

Korrekt müsste die Abfrage folglich etwa so aussehen:

```
SELECT *
        FROM person
        WHERE einkommen <> 45000 OR einkommen IS NULL;
```

Um das gewünschte Ergebnis zu erhalten, ist es notwendig, explizit auf `NULL` zu testen. Wie im folgenden Listing zu erkennen ist, enthält das Ergebnis nun den gewünschten Datensatz:

```
 name | geschlecht | einkommen
------+------------+-----------
 Paul | m          |
(1 row)
```

3.2.3 Daten modifizieren

Nachdem Sie gelernt haben, wie man Tabellen anlegen und Daten in eine Tabelle einfügen kann, werden wir uns mit der Modifikation von Daten beschäftigen. Der einfachste Weg, Daten zu manipulieren, ist, den Befehl `UPDATE` zu verwenden. `UPDATE` ist ein mächtiges Tool, das es erlaubt, komplexe Veränderungen durchzuführen. Das nächste Listing zeigt eine kurze Übersicht über die Syntax des Befehles:

```
buch=# \h UPDATE
Command:     UPDATE
Description: update rows of a table
Syntax:
UPDATE [ ONLY ] table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Es ist offensichtlich, dass die Syntax des Befehles sehr einfach ist und `UPDATE` daher sehr leicht verwendet werden kann.

Das nächste Beispiel zeigt, wie das Einkommen von Paul definiert werden kann:

```
buch=# UPDATE person SET einkommen=32000 WHERE name='Paul';
UPDATE 1
```

Sie müssen zuallererst spezifizieren, welche Tabelle modifiziert werden soll. In weiterer Folge können Sie den Wert der entsprechenden Spalte auf den gewünschten neuen Wert setzen. Im Falle eines Falles können Sie auch noch eine `WHERE` Clause zum Befehl hinzufügen, um die zu modifizierenden Werte zu selektieren. In unserem Fall haben wir Paul selektiert.

Interessant zu bemerken ist der Rückgabewert des Befehles. `UPDATE` bedeutet, dass eine `UPDATE`-Operation durchgeführt worden ist. Die im Anschluss daran genannte Zahl ist die Zahl der durch den Befehl modifizierten Zeilen in der Datenbank. Wäre dieser Wert 0, wäre zwar der Befehl korrekt ausgeführt jedoch wären keine Daten modifiziert worden. Dieses Verhalten ist sehr wichtig, da man auf diesem Wege leicht feststellen kann, wie sich ein Befehl auf die Datenbank ausgewirkt hat.

Nachdem Sie wissen, wie der Datensatz modifiziert werden kann, können wir einen Blick auf die Tabelle werfen:

```
1 buch=# SELECT * FROM person;
2  name | geschlecht | einkommen
3 ------+------------+-----------
4  Hans | m          |  45000.00
5  Paul | m          |  32000.00
6 (2 rows)
```

Wie erwartet enthält sie den neuen Wert des Datensatzes.

In manchen Fällen kann es sinnvoll sein, den Wert einer Spalte auf einen neuen Wert zu setzen, der aus dem alten Wert errechnet werden kann. Auch das ist mit `UPDATE` sehr leicht möglich. Nehmen wir etwa an, Paul bekommt eine Gehaltserhöhung — dieses Problem wäre so zu lösen:

```
1 buch=# UPDATE person SET einkommen=einkommen+1000 WHERE name='Paul';
2 UPDATE 1
```

Nehmen Sie einfach die Spalte und addieren Sie die entsprechende Zahl. PostgreSQL wird die Berechnung für Sie vornehmen. Das nächste Listing zeigt, dass die Operation funktioniert hat:

```
1 buch=# SELECT * FROM person;
2  name | geschlecht | einkommen
3 ------+------------+-----------
4  Hans | m          |  45000.00
5  Paul | m          |  33000.00
6 (2 rows)
```

In vielen Fällen ist es notwendig, mehrere Felder auf einmal zu updaten und wie das funktioniert, demonstriert dieses Listing:

```
buch=# UPDATE person SET name='Paul Huber', einkommen='31500' WHERE
name='Paul';
UPDATE 1
```

Das folgende Listing zeigt, dass wirklich beide Spalten modifiziert worden sind:

```
buch=# SELECT * FROM person;
    name    | geschlecht | einkommen
------------+------------+-----------
 Hans       | m          |  45000.00
 Paul Huber | m          |  31500.00
(2 rows)
```

3.2.4 Daten löschen

Nachdem Sie Daten in eine Datenbank eingefügt haben, kann es sinnvoll sein, diese auch wieder zu löschen. Um Daten aus der Datenbank zu entfernen, kann der Befehl `DELETE` verwendet werden. Die Syntax von `DELETE` ist ähnlich wie die von `SELECT` und `UPDATE`. Der wesentliche Punkt ist, dass Daten selektiert werden, die in weiterer Folge zu löschen sind. Um Daten zu selektieren, ist eine `WHERE` Clause an den Befehl anzufügen.

Die komplette Syntax kann dem nächsten Listing entnommen werden:

```
buch=# \h DELETE
Command:     DELETE
Description: delete rows of a table
Syntax:
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

Im nächsten Beispiel ist zu sehen, wie alle Personen, deren Einkommen kleiner als 40.000 ist, entfernt werden können:

```
buch=# DELETE FROM person WHERE einkommen < 40000;
DELETE 1
```

Zuerst ist die Tabelle zu definieren, aus der Sie Daten löschen wollen. In Weiteren dient die `WHERE` Clause dazu, die gewünschten Daten zu selektieren. In unserem Fall werden alle Personen mit einem zu geringen Einkommen aus der Tabelle entfernt.

Sofern es beim Ausführen des Befehles zu keinen Problemen gekommen ist, befindet sich nur mehr ein Datensatz in der Liste und das ist genau der erwünschte Zustand:

```
buch=# SELECT * FROM person;
 name | geschlecht | einkommen
------+------------+-----------
 Hans | m          |  45000.00
(1 row)
```

Eine weitere Möglichkeit, Daten aus einer Tabelle zu löschen, ist, `TRUNCATE` zu verwenden. Ein wesentlicher Unterschied zwischen `DELETE` und `TRUNCATE` ist, dass es `TRUNCATE` nicht erlaubt, Daten zu selektieren. Die zu löschende Tabelle wird auf dem schnellsten Wege geleert. Hier sehen Sie die Syntax von `TRUNCATE`:

```
buch=# \h TRUNCATE
Command:     TRUNCATE
Description: empty a table
Syntax:
TRUNCATE [ TABLE ] name
```

Das nächste Listing zeigt ein Beispiel, wie `TRUNCATE` praktisch verwendet werden kann, um Daten zu löschen:

```
buch=# TRUNCATE person;
TRUNCATE
```

Die Tabelle `Person` wird zur Gänze geleert. Interessant ist anzumerken, dass `TRUNCATE` nicht einmal mehr die Zahl der betroffenen Datensätze notiert, was `DELETE` sehr wohl macht.

Das folgende Listing zeigt, dass die Daten wirklich aus der Tabelle entfernt worden sind:

```
buch=# SELECT * FROM person;
 name | geschlecht | einkommen
------+------------+-----------
(0 rows)
```

`TRUNCATE` dient dazu, Daten so schnell wie möglich vom System zu entfernen. Im Gegensatz zu `DELETE` wird das von `TRUNCATE` auf schnellstem Wege und ohne Rücksicht gemacht. Speziell bei größeren Datenmengen kann `TRUNCATE` spürbar schneller sein als `DELETE`, weil ein wenig Overhead eingespart wird.

3.2.5 Daten verknüpfen

Bisher haben Sie kennen gelernt, wie einfache Tabellen und Listen geführt werden können. Für sehr triviale Anwendungen ist das teilweise ausreichend, aber was passiert, wenn es darum geht, Daten miteinander zu verknüpfen? In diesem Falle

ist es notwendig, eine Verbindung zwischen den Tabellen herzustellen. Im relationalen Datenbankkonzept bezeichnet man das als so genannten Join. Nehmen wir also an, wir haben zwei Tabellen. Die erste Tabelle speichert Informationen über Eltern. Die zweite Tabelle enthält Informationen über deren Kinder. Eine derartige Beziehung wird als 1:n-Relation bezeichnet, da ein Elternpaar (1) mehrere Kinder (n) haben kann.

Das nächste Listing zeigt, wie die Tabellen angelegt werden können. Des Weiteren werden einige Datensätze in die Datenbank eingetragen:

```
CREATE TABLE eltern (
        id              serial,         -- Laufnummer
        vater           text,           -- Name des Vaters
        mutter          text            -- Name der Mutter
);

CREATE TABLE kind (
        eltern_id       int4,
        name            text,
        gesch           char(1)
);

INSERT INTO eltern (vater, mutter) VALUES ('Ewald', 'Christina');
INSERT INTO eltern (vater, mutter) VALUES ('Hans', 'Nura');
INSERT INTO eltern (vater, mutter) VALUES ('Alex', 'Nadine');

INSERT INTO kind (eltern_id, name, gesch) VALUES (1, 'Lukas');
INSERT INTO kind (eltern_id, name, gesch) VALUES (1, 'Wolfgang');
INSERT INTO kind (eltern_id, name, gesch) VALUES (3, 'Paul');
INSERT INTO kind (eltern_id, name, gesch) VALUES (3, 'Hans');
INSERT INTO kind (eltern_id, name, gesch) VALUES (3, 'Alex');

```

Zwei Tabellen und acht Datensätze sollten sich jetzt in der Datenbank befinden. Im nächsten Schritt soll eine Tabelle erzeugt werden, die Kinder und die Namen Ihrer Eltern enthält:

```
buch=# SELECT * FROM eltern, kind WHERE eltern.id = kind.eltern_id;
 id | vater |  mutter   | eltern_id |   name   | gesch
----+-------+-----------+-----------+----------+-------
  1 | Ewald | Christina |         1 | Lukas    |
  1 | Ewald | Christina |         1 | Wolfgang |
  3 | Alex  | Nadine    |         3 | Paul     |
  3 | Alex  | Nadine    |         3 | Hans     |
  3 | Alex  | Nadine    |         3 | Alex     |
(5 rows)
```

Hier wird für jedes Kind genau ein Datensatz ausgegeben. Die Eltern der Kinder werden quasi in der Tabelle ergänzt. Wichtig zu bemerken ist, dass die Eltern mit der Nummer 2 nicht gelistet sind. Der Grund dafür ist einleuchtend: Sie haben keine Kinder.

Sehen wir uns nun die Abfrage ein wenig genauer an. Alle Spalten in den Tabellen `eltern` und `kind` werden selektiert. Es sind alle Spalten auszuwählen, in denen die Nummer (Id) der Eltern mit der des jeweiligen Kindes korrespondiert. Über dieses gemeinsame Merkmal können die Tabellen verknüpft werden. Die `WHERE` Clause dieser Query ist entscheidend. Wenn Sie auf die `WHERE` Clause vergessen, berechnet PostgreSQL (genauer gesagt macht das jede SQL-Datenbank) das kartesische Produkt der beiden Tabellen und das würde dann so aussehen:

```
buch=# SELECT * FROM eltern, kind;
 id | vater |  mutter   | eltern_id |   name   | gesch
----+-------+-----------+-----------+----------+-------
  1 | Ewald | Christina |         1 | Lukas    |
  1 | Ewald | Christina |         1 | Wolfgang |
  1 | Ewald | Christina |         3 | Paul     |
  1 | Ewald | Christina |         3 | Hans     |
  1 | Ewald | Christina |         3 | Alex     |
  2 | Hans  | Nura      |         1 | Lukas    |
  2 | Hans  | Nura      |         1 | Wolfgang |
  2 | Hans  | Nura      |         3 | Paul     |
  2 | Hans  | Nura      |         3 | Hans     |
  2 | Hans  | Nura      |         3 | Alex     |
  3 | Alex  | Nadine    |         1 | Lukas    |
  3 | Alex  | Nadine    |         1 | Wolfgang |
  3 | Alex  | Nadine    |         3 | Paul     |
  3 | Alex  | Nadine    |         3 | Hans     |
  3 | Alex  | Nadine    |         3 | Alex     |
(15 rows)
```

Das kartesische Produkt enthält das Ergebnis, aber eben auch eine größere Zahl nicht relevanter Datensätze. Im Wesentlichen ist das Ergebnis eines korrekt durchgeführten Joins nichts anderes als die Teilmenge aller möglichen Kombinationen. Das ist sehr wichtig zu wissen, denn wenn man bei einem Join eine `WHERE` Clause vergisst, muss es nicht heißen, dass die Query nicht funktioniert — Sie werden nur erst auf den zweiten Blick erkennen, dass Sie viel zu viele Daten von der Datenbank zurückbekommen. Speziell beim Testen von Applikationen kann das für viele Neueinsteiger in SQL ein Problem sein, weil man während eines Betatests in der Regel mit wesentlich weniger Datensätzen arbeitet als im Echtbetrieb. Auf diese Weise können sich sehr leicht Fehler einschleichen, die dann im Echtbetrieb latent werden. Aus diesem Grund ist besonderes Augenmerk auf die Join-Bedingungen zu legen.

In vielen Fällen wollen Sie nicht alle Spalten eines Ergebnisses haben. Dann müssen Sie nur die entsprechenden Spalten vor der `FROM` Clause selektieren und die Query ausführen:

```
buch=# SELECT eltern.vater FROM eltern, kind WHERE eltern.id =
kind.eltern_id;
vater
────────
 Ewald
 Ewald
 Alex
 Alex
 Alex
(5 rows)
```

In unserem Fall haben Sie eine Liste aller Väter generiert. Es fällt sofort auf, dass manche Väter mehrmals im Ergebnis erscheinen. Das Ergebnis ist korrekt, aber es ist möglicherweise nicht das, was Sie sich unter dem Ergebnis vorgestellt haben. Aus diesem Grund können Sie das Schlüsselwort `DISTINCT` verwenden, das die mehrfache Auflistung von gleichen, im Ergebnis vorkommenden Datensätzen unterdrücken kann.

Das nächste Listing zeigt die gleiche Abfrage, aber diesmal unter Verwendung von `DISTINCT`:

```
buch=# SELECT DISTINCT eltern.vater FROM eltern, kind WHERE eltern.id
= kind.eltern_id;
 vater
────────
 Alex
 Ewald
(2 rows)
```

Jetzt ist das Ergebnis diesmal nur zwei Zeilen lang, was eher den Anforderungen entspricht.

Das nächste Listing zeigt eine Liste der Mütter und deren Kinder:

```
buch=# SELECT eltern.mutter, kind.name FROM eltern, kind WHERE
eltern.id = kind.eltern_id;
   mutter   |   name
────────────+──────────
 Christina | Lukas
 Christina | Wolfgang
 Nadine    | Paul
 Nadine    | Hans
 Nadine    | Alex
(5 rows)
```

In den letzten Beispielen haben Sie gesehen, wie zwei Tabellen miteinander verknüpft werden können. Was für zwei Tabellen möglich ist, funktioniert folglich auch für jede beliebige Anzahl von Relationen. Wenn Sie nicht gerade das kartesische Produkt berechnen wollen, benötigen Sie für jede Relation mindestens eine Join-Bedingung.

3.2.6 Sortieren

In diesem Abschnitt werden wir uns kurz mit Sortierungen beschäftigen.

Das Ziel der nächsten Abfrage ist es, die Daten sortiert auszugeben. Das erste Sortierkriterium ist der Name der Mutter. Als Zweites wird der Name des Kindes berücksichtigt:

```
SELECT eltern.mutter, kind.name
        FROM eltern, kind
        WHERE eltern.id = kind.eltern_id
        ORDER BY mutter, name;
```

Das Ergebnis enthält dieselben Daten wie die Abfrage, die Sie zuvor gesehen haben, nur mit dem Unterschied, dass diesmal der Name sortiert ist:

```
  mutter   |   name
-----------+----------
 Christina | Lukas
 Christina | Wolfgang
 Nadine    | Alex
 Nadine    | Hans
 Nadine    | Paul
(5 rows)
```

Wenn Sie die Daten absteigend sortieren wollen, können Sie den folgenden SQL-Code benutzen:

```
SELECT eltern.mutter, kind.name
        FROM eltern, kind
        WHERE eltern.id = kind.eltern_id
        ORDER BY mutter, name DESC;
```

Der SQL-Code unterscheidet sich nur durch das Einfügen von `DESC`. Das folgende Listing enthält das Ergebnis der Abfrage:

```
  mutter   |   name
-----------+----------
 Christina | Wolfgang
 Christina | Lukas
```

```
 Nadine    | Paul
 Nadine    | Hans
 Nadine    | Alex
(5 rows)
```

In sehr speziellen Fällen ist es notwendig, auf unterschiedliche Spalten unterschiedliche Sortierungen anzuwenden. Auch derart ausgefallene Wünsche können mit ANSI SQL erfüllt werden. Das nächste Beispiel zeigt, wie die erste Spalte absteigend und die zweite aufsteigend sortiert werden kann:

```
SELECT eltern.mutter, kind.name
        FROM eltern, kind
        WHERE eltern.id = kind.eltern_id
        ORDER BY mutter DESC , name ASC;
```

Das Schlüsselwort ASC ist optional und in diesem Fall nur aus Gründen der Lesbarkeit in Verwendung. Das folgende Listing zeigt das Ergebnis:

```
   mutter   |   name
------------+----------
 Nadine    | Alex
 Nadine    | Hans
 Nadine    | Paul
 Christina | Lukas
 Christina | Wolfgang
(5 rows)
```

Wie nicht anders zu erwarten, ist die erste Spalte absteigend, die zweite Spalte jedoch aufsteigend sortiert. Die Spalte, die bei der Sortierung priorisiert werden soll, muss als erste Spalte genannt werden. Im Folgenden wird gezeigt, wie die zweite Spalte favorisiert werden kann:

```
SELECT eltern.mutter, kind.name
        FROM eltern, kind
        WHERE eltern.id = kind.eltern_id
        ORDER BY name ASC, mutter DESC;
```

Das Ergebnis der Abfrage unterscheidet sich durch die Sortierung der Daten:

```
   mutter   |   name
------------+----------
 Nadine    | Alex
 Nadine    | Hans
 Christina | Lukas
 Nadine    | Paul
 Christina | Wolfgang
(5 rows)
```

Die Möglichkeit, die Sortierung von Daten zu verändern, ist eine wichtige Grundfunktion von ANSI SQL. Bei komplexeren Anwendungen kann es sogar notwendig sein, die Sortierung noch weiter zu verfeinern. Zu diesem Zwecke ist es möglich, Vergleichsfunktionen zu definieren, die die Sortierreihenfolge explizit festlegen, doch dazu werden wir erst im Weiteren kommen.

3.2.7 Sequenzen

Sequenzen sind ein rudimentärer Bestandteil relationaler Datenbanksysteme. Sequenzen sind nichts Anderes als fortlaufende Nummern, die es ermöglichen, mit eindeutige Numerierungen zu arbeiten.

Um eine Sequenz anzulegen, können Sie den Befehl `CREATE SEQUENCE` verwenden. Das nächste Listing zeigt die Syntaxübersicht des Befehles:

```
buch=# \h CREATE SEQUENCE
Command:     CREATE SEQUENCE
Description: define a new sequence generator
Syntax:
CREATE [ TEMPORARY | TEMP ] SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Wie Sie leicht erkennen können, bietet der Befehl eine Reihe von Möglichkeiten, mit Sequenzen zu arbeiten. Bevor wir uns auf diese Fülle von Features stürzen, wollen wir versuchen, eine Sequenz anzulegen:

```
buch=# CREATE SEQUENCE seq_test;
CREATE
```

Sofern der Befehl korrekt funktioniert hat, exisitiert jetzt eine Sequenz namens `seq_test`. Jede Sequenz verfügt über eine Reihe von Attributen, die in Tabellenform abgefragt werden kann. Um eine Liste dieser Attribute zu extrahieren, können Sie `\d` verwenden, wie es im nächsten Listing zu sehen ist:

```
buch=# \d seq_test
   Sequence "seq_test"
    Column     |  Type
---------------+---------
 sequence_name | name
 last_value    | bigint
 increment_by  | bigint
 max_value     | bigint
 min_value     | bigint
 cache_value   | bigint
 log_cnt       | bigint
```

```
 is_cycled      | boolean
 is_called      | boolean
```

Die Namen der einzelnen Attribute sprechen für sich und es ist unschwer zu erraten, was die einzelnen Werte bedeuten. Im ersten Schritt sind die folgenden Spalten von Bedeutung:

```
buch=# SELECT last_value, increment_by FROM seq_test;
 last_value | increment_by
------------+--------------
          1 |            1
(1 row)
```

Der aktuelle Wert der Sequenz ist 1. Sofern die Sequenz abgefragt wird, inkrementiert PostgreSQL ihren Wert automatisch um 1. Um den nächsten Wert einer Sequenz zu generieren, kann die Funktion `next_val` verwendet werden:

```
buch=# SELECT nextval('seq_test');
 nextval
---------
       1
(1 row)

buch=# SELECT nextval('seq_test');
 nextval
---------
       2
(1 row)
```

In diesem Fall haben wir die Sequenz zweimal erhöht. PostgreSQL stellt sicher, dass ein Wert niemals doppelt vergeben werden kann. Auf diese Weise ist die Laufnummer einer Sequenz eindeutig.

In vielen Fällen kann eine Laufnummer als Index einer Tabelle verwendet werden. Dann ist es sinnvoll, dass PostgreSQL Werte automatisch einfügt und die Sequenz eigenständig behandelt. Für diese Zwecke hat man Datentypen namens `serial` bzw. `serial8` geschaffen. Das nächste Beispiel zeigt, wie dieser Datentyp verwendet werden kann:

```
buch=# CREATE TABLE fehler(id serial, code text);
NOTICE:  CREATE TABLE will create implicit sequence 'fehler_id_seq'
for SERIAL column 'fehler.id'
NOTICE:  CREATE TABLE / UNIQUE will create implicit index
'fehler_id_key' for table 'fehler'
CREATE
```

Das SQL-Kommando im vorigen Listing erstellt eine Tabelle zum Speichern von Fehlerinformationen. Jeder Fehler soll automatisch eine fortlaufende Nummer erhalten. Zu diesem Zwecke kommt eine Sequenz zum Einsatz. Bevor wir beginnen, Daten einzufügen, können wir uns kurz die Definition der Tabelle ansehen:

```
buch=# \d fehler
                            Table "fehler"
 Column |  Type   |                     Modifiers
--------+---------+---------------------------------------------------
 id     | integer | not null default nextval('"fehler_id_seq"'::text)
 code   | text    |
 Unique keys: fehler_id_key
```

Die erste Spalte verwendet als Default-Wert die nächste Zahl in der Sequenz. Auf diese Weise wird sichergestellt, dass jede Zeile einen eindeutigen Wert enthält.

Wenn Sie nun Werte in die zweite Spalte einfügen, wird die erste Spalte automatisch vom System ergänzt.

In vielen Fällen ist es notwendig oder wünschenswert, den aktuellen Wert einer Sequenz explizit zu modifizieren. In solchen Fällen kann der Befehl `setval` verwendet werden. Das folgende Listing zeigt, wie der Wert der Sequenz auf 100 gesetzt werden kann:

```
buch=# SELECT setval('fehler_id_seq', 100);
NOTICE:  fehler_id_seq.setval: sequence was re-created
 setval
--------
    100
(1 row)
```

Wie wir vorher bereits angedeutet haben, können mit einer Sequenz vielerlei vorteilhafte Dinge realisiert werden. Wenn Sie etwa eine Sequenz benötigen, die sich nicht immer um 1, sondern um einen beliebigen Wert erhöht, so kann das leicht erreicht werden, indem Sie beim Erstellen der Sequenz einen Parameter mitgeben. Das nächste Listing zeigt, wie eine maßgeschneiderte Sequenz angelegt werden kann:

```
buch=# CREATE SEQUENCE seq_special INCREMENT 10 MINVALUE 5
MAXVALUE 1000;
CREATE
```

In diesem Fall beginnt die Sequenz bei 5 und wird automatisch um 10 inkrementiert, sobald mit `nextval` ein neuer Wert abgefragt wird. Der maximale Wert wird auf 1.000 gesetzt. Das nachfolgende Listing demonstriert, was passiert, wenn mehrmals auf die Sequenz zugegriffen wird:

```
buch=# SELECT nextval('seq_special');
 nextval
─────────
       5
(1 row)

buch=# SELECT nextval('seq_special');
 nextval
─────────
      15
(1 row)
```

Wie unschwer zu erraten ist, ist der zweite von der Sequenz zurückgegebene Wert um genau 10 höher als der erste Wert.

Temporäre Sequenzen sind ein mächtiges Feature. Temporär bedeutet, dass eine Sequenz nach dem Beenden einer Connection (entspricht einem Backend-Prozess) automatisch gelöscht wird. Zusätzlich dazu sind temporäre Sequencz nur für den Benutzer beziehungsweise innerhalb der Datenbankverbindung sichtbar, die benutzt worden ist, um die Sequenz anzulegen. Aus diesem Grund können temporäre Sequenzen niemals mit anderen temporären Sequenzen kollidieren. Das ist sehr wichtig, wenn Ihr Datenbanksystem eine Vielzahl konkurrierender Datenbankverbindungen abarbeitet.

Das nächste Listing zeigt, wie eine temporäre Sequenz angelegt werden kann:

```
buch=# CREATE TEMPORARY SEQUENCE seq_tmp;
CREATE
```

Fügen Sie einfach das Schlüsselwort `TEMP` beziehungsweise `TEMPORARY` zu Ihrem Befehl `CREATE SEQUENCE` hinzu und PostgreSQL macht den Rest für Sie. Sobald eine Datenbankverbindung terminiert wird, können Sie sicher sein, dass alle temporären Sequenzen, die während dieser Session generiert worden sind, automatisch vom System entfernt werden.

3.2.8 Aggregierungen

Datenbanken sind nicht nur dazu da, um Daten abzufragen. In den allermeisten Fällen ist es notwendig, Daten zu aggregieren und zusammenzufassen. Das geschieht mithilfe so genannter Aggregierungsfunktionen sowie mit Aggregierungsoperatoren. In diesem Teil des Buches werden wir einen genauen Blick darauf werfen, wie Daten aggregiert werden können.

Eine der wichtigsten Funktionen, die im Umgang mit Aggregierungen immer wieder vorkommt, ist die `COUNT`-Funktion. `COUNT` dient dazu, Datensätze, die bestimmen Kriterien entsprechen, zu zählen. Nehmen wir etwa an, wir wollen die Zahl

der in der Tabelle `eltern` gespeicherten Personen zählen. Eine Abfrage, die diesen Zweck erfüllt, würde etwa so aussehen:

```
buch=# SELECT COUNT(*) FROM eltern;
 count
-------
     3
(1 row)
```

Es befinden sich genau drei Datensätze in der Tabelle.

Aggregierungsoperationen können auch in Kombination mit Joins eingesetzt werden. Nehmen wir etwa an, wir wollen die Zahl der Kinder abfragen, bei denen die Eltern bekannt sind (also die Spalte `eltern_id` einen in der Spalte `id` vorhandenen Wert hat). Die gewünschte Abfrage sieht wie folgt aus:

```
SELECT COUNT(eltern.id)
        FROM eltern, kind
        WHERE eltern.id=kind.eltern_id;
```

Das Ergebnis ist fünf, weil alle Kinder auch Eltern haben:

```
 count
-------
     5
(1 row)
```

Im nächsten Beispiel wollen wir abfragen, wie viele Kinder ein Elternpaar hat. Wir machen einen ersten Versuch:

```
SELECT eltern.vater, eltern.mutter, COUNT(kind.name)
        FROM eltern, kind
        WHERE kind.eltern_id = eltern.id;
```

Wir listen also die Namen der Eltern und die Anzahl der Kinder auf. Mithilfe von `COUNT` soll die Zählung durchgeführt werden. Die Query hat jedoch einen Schönheitsfehler, der sich durch eine Fehlermeldung äußert.

```
ERROR:  Attribute eltern.vater must be GROUPed or used in an aggregate
function
```

Die oben gelistete Fehlermeldung ist für jeden, der intensiver mit SQL zu tun hat, ein Begriff. Der Grund für den Fehler ist denkbar einfach zu erklären. Es ist für PostgreSQL so gut wie unmöglich, zu erkennen, was gezählt werden soll. Aus diesem Grund ist eine `GROUP BY` Clause notwendig. Alles, was nicht in der `GROUP BY` Clause aufscheint, wird gezählt.

Hier ist die korrekte Query:

```
SELECT eltern.vater, eltern.mutter, COUNT(kind.name)
        FROM eltern, kind
        WHERE kind.eltern_id = eltern.id
        GROUP BY eltern.vater, eltern.mutter;
```

Aus dem Ergebnis ist ersichtlich, dass nur zwei Familien Kinder haben:

```
 vater |  mutter   | count
-------+-----------+-------
 Alex  | Nadine    |     3
 Ewald | Christina |     2
(2 rows)
```

Im nächsten Schritt ist es wieder möglich, die Sortierung der Datensätze zu beeinflussen. Die `ORDER BY` Clause ist hierbei ans Ende des Codes zu stellen, wie es im nächsten Listing zu sehen ist:

```
SELECT eltern.vater, eltern.mutter, COUNT(kind.name)
        FROM eltern, kind
        WHERE kind.eltern_id = eltern.id
        GROUP BY eltern.vater, eltern.mutter
        ORDER BY eltern.mutter, eltern.vater;
```

Diesmal sieht das Ergebnis aufgrund der Sortierung ein wenig anders aus:

```
 vater |  mutter   | count
-------+-----------+-------
 Ewald | Christina |     2
 Alex  | Nadine    |     3
(2 rows)
```

Nach diesem kurzen Überblick über die `COUNT`-Funktion kommen wir zur nächsten wichtigen Funktion. Die `SUM`-Funktion dient zum Addieren von Werten in einer Spalte. Rufen wir uns noch einmal kurz den Inhalt der Tabelle `person` in Erinnerung:

```
buch=# SELECT * FROM person;
    name    | geschlecht | einkommen
------------+------------+-----------
 Hans       | m          |  45000.00
 Paul Huber | m          |  31500.00
(2 rows)
```

Die Tabelle enthält Informationen über Name, Geschlecht und das Einkommen von Personen. Ziel ist es nun, die Einkommen aller Personen in der Tabelle zu summieren:

```
buch=# SELECT SUM(einkommen) FROM person;
   sum
─────────
 76500.00
(1 row)
```

Wenn Sie detailliertere Informationen über eine Gruppe von Werten benötigen, können Sie wie mit der Funktion `COUNT` arbeiten. Ziel des nächsten Beispieles ist es, die Summe der Einkommen nach Geschlechtern differenziert zu berechnen. Um das Beispiel ein wenig spektakulärer zu machen, fügen wir noch einen Wert in die Tabelle ein:

```
buch=# INSERT INTO person VALUES ('Jeanny', 'f', 39000);
INSERT 24840 1
```

Es sollte nun ein einfaches Unterfangen sein, die Summe der Einkommen nach Geschlechtern differenziert zu berechnen. Die folgende Abfrage führt zum Ziel:

```
SELECT geschlecht, SUM(einkommen)
        FROM person
        GROUP BY geschlecht;
```

Das Ergebnis enthält für jedes Geschlecht genau eine Zeile mit den entsprechenden Werten:

```
 geschlecht |   sum
────────────+─────────
 f          | 39000.00
 m          | 76500.00
(2 rows)
```

In vielen Kursen zum Thema SQL werde ich gefragt, ob es möglich sei, mehrere Aggregierungsfunktionen in einer Abfrage zu verwenden. Die Antwort ist ein klares Ja. Das nächste Beispiel zeigt, wie die Funktionen `COUNT` und `SUM` zugleich verwendet werden können:

```
SELECT geschlecht, COUNT(einkommen), SUM(einkommen)
        FROM person
        GROUP BY geschlecht;
```

Das Ergebnis ist eine zusätzliche Spalte, die die Anzahl der gezählten Werte enthält:

```
 geschlecht | count |   sum
────────────+───────+─────────
 f          |     1 | 39000.00
 m          |     2 | 76500.00
(2 rows)
```

Im Folgenden werden wir uns noch mit einer Funktion namens AVG beschäftigen, die zum Berechnen des Mittelwertes herangezogen werden kann. Das folgende Beispiel zeigt, wie der Mittelwert der Einkommen der beiden Geschlechter berechnet werden kann.

```
SELECT geschlecht, AVG(einkommen)
        FROM person
        GROUP BY geschlecht;
```

Wie man unschwer erkennt, kann die Berechnung analog zu den vorherigen Abfragen durchgeführt werden. Das Ergebnis ist nicht weiter überraschend:

```
 geschlecht |       avg
------------+------------------
 f          | 39000.0000000000
 m          | 38250.0000000000
(2 rows)
```

In vielen Fällen ist die Beschriftung der Tabellen ein wesentlicher Punkt. Die Bedeutung der Formatierung der Tabelle ist bei weitem nicht mehr so groß, wie das einmal war, bei textbasierenden Anwendungen ist es immer noch wichtig. Um die Überschrift der Tabelle zu verändern, können Sie das Keyword AS verwenden:

```
SELECT geschlecht, AVG(einkommen) AS Mittelwert
        FROM person
        GROUP BY geschlecht;
```

Das folgende Listing zeigt den veränderten Titel. Es ist festzustellen, dass die Groß- und Kleinschreibung nicht beachtet worden ist:

```
 geschlecht |    mittelwert
------------+------------------
 f          | 39000.0000000000
 m          | 38250.0000000000
(2 rows)
```

Abschließend sehen wir uns noch an, wie das Maximum und das Minimum einer Spalte berechnet werden können. Zu diesem Zwecke können die Funktionen MIN und MAX verwendet werden:

```
buch=# SELECT MAX(einkommen), MIN(einkommen) FROM person;
   max    |   min
----------+---------
 45000.00 | 31500.00
(1 row)
```

Aggregierungsfunktionen sind wichtig beim Arbeiten mit relationalen Datenbanken. Zusätzliche Funktionen sind relativ leicht zu implementieren, da PostgreSQL einfache Interfaces zur Verfügung stellt. Das ist Thema weiterer Kapitel.

3.2.9 Views

Im Gegensatz zu anderen frei verfügbaren Datenbanksystemen unterstützt PostgreSQL seit geraumer Zeit so genannte Views. Views sind Sichten, die es ermöglichen sollen, Daten aus einem anderen Blickwinkel zu erfassen. Soweit die Theorie — um es mit den einfachen Worten auszudrücken — Views sehen aus wie Tabellen und enthalten das Ergebnis eines SQL-Statements, das jedes Mal ausgeführt wird, wenn jemand Daten aus der View abfragt. Im nächsten Listing sehen Sie, wie eine View angelegt werden kann:

```
buch=# \h CREATE VIEW
Command:     CREATE VIEW
Description: define a new view
Syntax:
CREATE VIEW view [ ( column name list ) ] AS SELECT query
```

Die Syntax von `CREATE VIEW` ist denkbar einfach und sollte keine Probleme mehr machen. Versuchen wir nun also eine View anzulegen:

```
CREATE VIEW view_person AS
        SELECT name, geschlecht, einkommen * '0.86' AS dollars
        FROM person;
```

Die Definition einer View kann wie die einer Tabelle mithilfe von `\d` abgefragt werden:

```
buch=# \d view_person
               View "view_person"
   Column   |         Type          | Modifiers
------------+-----------------------+-----------
 name       | character varying(30) |
 geschlecht | character(1)          |
 dollars    | numeric               |
View definition: SELECT person.name, person.geschlecht,
(person.einkommen * '0.86'::"numeric") AS dollars FROM person;
```

Dem Listing kann man den SQL-Code entnehmen, der der View zugrunde liegt. Sie werden sich vermutlich wundern, dass der SQL-Code etwas anders aussieht als der, den wir beim Anlegen der View verwendet haben. Der Grund dafür liegt in den Tiefen von PostgreSQL. Die im Listing gezeigten Queries sind von PostgreSQL neu geschrieben worden. Aus internen Gründen ist das sinnvoll und notwendig, um eine Abfrage effizient und zufällig abarbeiten zu können.

Wenn Sie nun alle Daten in der View abfragen wollen, können Sie ein einfaches `SELECT`-Statement verwenden. Wie Sie im folgenden Listing erkennen, werden die Daten wie gewohnt von PostgreSQL retourniert:

```
buch=# SELECT * FROM view_person;
    name    | geschlecht |  dollars
------------+------------+------------
 Hans       | m          | 38700.0000
 Paul Huber | m          | 27090.0000
 Jeanny     | f          | 33540.0000
(3 rows)
```

Das Ergebnis entspricht genau dem Ergebnis des `SELECT`-Statements, auf dem die View basiert.

Viele Leute fragen mich, wozu Views notwendig sind. Die Antwort auf diese Frage ist nicht immer trivial und teilweise ein wenig philosophisch. Im Prinzip gibt es keine Anwendung, die nicht auch ohne Views realisierbar wäre. Der Vorteil von Views ist, dass es möglich wird, mehrschichtige Applikationen und Abfragen zu bauen, die die Lesbarkeit einer Anwendung erhöhen. Lesbarkeit und Reproduzierbarkeit sind zwei wesentliche Punkte, die nicht unterschätzt werden sollen. Views werden Ihnen helfen, die Komplexität einzelner SQL-Statements zu senken und dadurch wird es wiederum möglich, schneller zu Ihrem Ziel zu kommen. Bedenken Sie, dass der interne Aufwand der Datenbank dadurch nicht geringer wird — Views helfen lediglich die Komplexität des SQL-Codes zu reduzieren. Durch zu extensives Verwenden von Views kann es auch passieren, dass Ihre Anwendungen ein wenig langsamer werden.

Da Views nach außen wie Tabellen aussehen, ist es ohne Probleme möglich, sie zu schachteln und auf Basis von anderen Views zu definieren. Das nächste Beispiel zeigt eine View, die nur mehr Informationen über Männer enthält:

```
CREATE VIEW view_mann AS
        SELECT *
                FROM view_person
                WHERE geschlecht = 'm';
```

Dieses Listing zeigt den Inhalt der View:

```
buch=# SELECT * FROM view_mann;
    name    | geschlecht |  dollars
------------+------------+------------
 Hans       | m          | 38700.0000
 Paul Huber | m          | 27090.0000
(2 rows)
```

3.2.10 Mengenoperationen mit `UNION`, `INTERSECT` und `EXCEPT`

SQL bietet eine reichhaltige Syntax und eine Vielzahl von Möglichkeiten, effizient mit Daten umzugehen. Bei vielen Anwendungen ist es notwendig, Daten zusammenzufassen, Schnittmengen zu bilden oder herauszufinden, welche Datensätze

nur in dem einen oder anderen Datenbestand enthalten sind. Zu diesem Zwecke stellt SQL die drei Schlüsselwörter `UNION`, `INTERSECT` und `EXCEPT` zur Verfügung.

Wenn Sie Daten miteinander vereinigen möchten, können Sie `UNION` verwenden. Das folgende Beispiel zeigt, wie die Personen, die mehr als 39.000 Geldeinheiten verdienen, mit denen vereint werden können, die weniger als 35.000 Geldeinheiten verdienen:

```
SELECT *
        FROM person
        WHERE einkommen > 39000
UNION SELECT *
        FROM person
        WHERE einkommen < 35000;
```

Das Ergebnis wird aus zwei Abfragen zusammengesetzt. Das Ergebnis des SQL-Codes kann dem folgenden Listing entnommen werden:

```
    name    | geschlecht | einkommen
------------+------------+-----------
 Hans       | m          |  45000.00
 Paul Huber | m          |  31500.00
(2 rows)
```

Wichtig zu beachten ist, dass die Anzahl der Spalten, die von den einzelnen Komponenten der Query zurückgegeben werden, übereinstimmt, weil anderenfalls es der Datenbank nicht möglich ist, die Ergebnisse sinnvoll zu vereinen.

Hier sehen Sie ein Beispiel für eine nicht funktionierende Abfrage:

```
SELECT name
        FROM person
        WHERE einkommen > 39000
UNION SELECT *
        FROM person
        WHERE einkommen < 35000;
```

Der erste Teil der Query liefert genau eine Spalte, während die zweite Teilkomponente drei Spalten zurückgibt. Da die Spaltenanzahl nicht übereinstimmt, wird ein Fehler ausgegeben:

```
ERROR:  Each UNION query must have the same number of columns
```

Im letzten Beispiel haben wir gesehen, wie die Personen mit dem gewünschten Einkommen extrahiert werden können. Das folgende Beispiel liefert dasselbe Ergebnis. Im Unterschied zu vorher wird `UNION` nicht benötigt:

```
SELECT *
        FROM person
        WHERE einkommen > 39000
                OR einkommen < 35000;
```

Das Ergebnis ist gleich:

```
    name    | geschlecht | einkommen
------------+------------+-----------
 Hans       | m          |  45000.00
 Paul Huber | m          |  31500.00
(2 rows)
```

Dieses Beispiel zeigt, wie einfache Probleme auf verschiedenste Art gelöst werden können. Das ist ein wichtiger Punkt, da es Ihnen eine gewisse Freiheit beim Verfassen Ihrer SQL-Statements gibt. Je nach Art der Abfrage wird auch die Geschwindigkeit mehr oder weniger stark variieren.

Nachdem Sie jetzt wissen, wie `UNION` verwendet werden kann, können wir uns kurz ansehen, wie `EXCEPT` zu verwenden ist. Das Ziel der nächsten Abfrage ist es, alle Personen zu finden, die nicht in der View `view_mann` gefunden werden können:

```
buch=# SELECT * FROM view_person EXCEPT SELECT * FROM view_mann;
  name  | geschlecht |  dollars
--------+------------+------------
 Jeanny | f          | 33540.0000
(1 row)
```

Es wird genau ein Datensatz zurückgegeben. Da alle Datensätze, die Männer enthalten, nicht im Ergebnis sind, wird der Datensatz der einzigen Frau in der Datenbank angezeigt.

Wenn Sie die Schnittmenge der beiden Views berechnen wollen, können Sie `INTERSECT` benutzen:

```
buch=# SELECT * FROM view_person INTERSECT SELECT * FROM view_mann;
    name    | geschlecht |  dollars
------------+------------+------------
 Hans       | m          | 38700.0000
 Paul Huber | m          | 27090.0000
(2 rows)
```

Das Ergebnis enthält alle Datensätze, die in `view_mann` enthalten sind, weil diese genau die Schnittmenge der beiden Resultate bilden.

3.2.11 Datenstrukturen modifizieren

Wenn Sie Datenstrukturen erst einmal definiert haben, kann es in vielen Fällen notwendig sein, Veränderungen vorzunehmen. Der wichtigste Befehl in diesem Zusammenhang ist der Befehl `ALTER TABLE`, der für zahlreiche Operationen herangezogen werden kann. In diesem Abschnitt werden wir uns ein wenig näher mit `ALTER TABLE` beschäftigen und sehen, für welche Zwecke der Befehl verwendet werden kann. Bevor wir zum Beispielcode kommen, ist es notwendig, einen kurzen Blick auf die Syntaxdefinition zu werfen:

```
buch=# \h ALTER TABLE
Command:     ALTER TABLE
Description: change the definition of a table
Syntax:
ALTER TABLE [ ONLY ] table [ * ]
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE [ ONLY ] table [ * ]
    ALTER [ COLUMN ] column SET STATISTICS integer
ALTER TABLE [ ONLY ] table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
ALTER TABLE table
    ADD table constraint definition
ALTER TABLE [ ONLY ] table
        DROP CONSTRAINT constraint { RESTRICT | CASCADE }
ALTER TABLE table
        OWNER TO new owner
```

Wie Sie sehen, bietet der Befehl eine nahezu endlose Fülle an Möglichkeiten.

Eine der wichtigsten Operationen ist das Anfügen einer zusätzlichen Spalte. Das nächste Beispiel zeigt, wie eine Spalte zum Speichern des Familienstandes zur Tabelle `person` hinzugefügt werden kann:

```
buch=# ALTER TABLE person ADD COLUMN verheiratet boolean;
ALTER
```

Die Syntax des Befehles ist verhältnismäßig einfach. Sie müssen lediglich die zu modifizierende Tabelle, den Namen der neuen Spalte und den Datentyp definieren.

Dieses Listing zeigt die modifizierte Datenstruktur:

```
buch=# \d person
                  Table "person"
   Column    |         Type          | Modifiers
-------------+-----------------------+-----------
 name        | character varying(30) |
 geschlecht  | character(1)          |
 einkommen   | numeric(10,2)         |
 verheiratet | boolean               |
```

Wenn Sie nach dem Anlegen der Spalte erkennen, dass Sie noch einen Default-Wert benötigen, können Sie diesen sehr einfach definieren. PostgreSQL ermöglicht das nächträgliche Setzen so genannter Constraints. Im nächsten Beispiel sehen Sie, wie der Default-Wert der Spalte `verheiratet` auf `False` gesetzt werden kann:

```
buch=# ALTER TABLE person ALTER COLUMN verheiratet SET DEFAULT 'f';
ALTER
```

Zum Generieren statistischer Informationen über eine Spalte kann `SET STATISTICS` verwendet werden. Diese Einstellung kann geringfügige Einflüsse auf die Art haben, wie eine Anfrage abgearbeitet wird.

`ALTER` kann auch zum nächträglichen Ändern von Spaltennamen herangezogen werden. Das nächste Listing zeigt, wie der Name der Spalte `verheiratet` modifiziert werden kann:

```
buch=# ALTER TABLE person RENAME verheiratet TO vergeben;
ALTER
```

Der Befehl `ALTER TABLE` ist ein mächtiges Tool und erlaubt das schnelle und einfache Bearbeiten von Datenstrukturen. Wir werden in den nächsten Kapiteln noch öfter auf diesen Befehl zurückkommen.

3.2.12 Subselects

In manchen Applikationen beruhen Berechnungen auf dem Ergebnis vorangegangener Abfragen. Zur Lösung derartiger Probleme können so genannte Subselects (Unterabfragen) herangezogen werden. Um die Problemstellung ein wenig zu verdeutlichen, ist es notwendig, sich ein einfaches Beispiel vor Augen zu führen:

Die Selektion aller Personen, die mehr verdienen als der Durschnitt, ist ein klassisches Beispiel für ein Subselect. Um nicht ganz aus der Reihe zu tanzen, haben wir uns entschlossen, dieses Beispiel hier einzubauen.

Sehen wir uns noch einmal kurz den Inhalt der Tabelle `person` an:

```
buch=# SELECT * FROM person;
     name     | geschlecht | einkommen | vergeben
--------------+------------+-----------+----------
 Hans         | m          |  45000.00 |
 Paul Huber   | m          |  31500.00 |
 Jeanny       | f          |  39000.00 |
(3 rows)
```

Wenn Sie alle Personen finden wollen, die mehr als der Durchnitt verdienen, ist es zu allererst notwendig, das durchschnittliche Einkommen zu berechnen. Das kann relativ leicht erreicht werden:

```
buch=# SELECT AVG(einkommen) FROM person;
        avg
--------------------
 38500.0000000000
(1 row)
```

Auch diese Abfrage kann sehr leicht realisiert werden:

```
buch=# SELECT * FROM person WHERE einkommen > 38500;
  name  | geschlecht | einkommen | vergeben
--------+------------+-----------+----------
 Hans   | m          |  45000.00 |
 Jeanny | f          |  39000.00 |
(2 rows)
```

Das Ergebnis enthält genau zwei Datensätze, was absolut korrekt ist. Aber was passiert, wenn jemand die Datenbank verändert, nachdem wir den Mittelwert berechnet haben? Stellen wir uns ein einfaches `INSERT`-Statement vor, das irgendein Benutzer nach unserer Mittelwertberechnung ausführt:

```
buch=# INSERT INTO person VALUES ('Evil', 'f', '100000', 'f');
INSERT 24847 1
```

Nachdem der fremde User die Datenbank modifizert hat, berechnen wir die gewünschten Datensätze. Das Ergebnis enthält auf einmal drei Werte:

```
buch=# SELECT * FROM person WHERE einkommen > 38500;
  name  | geschlecht | einkommen | vergeben
--------+------------+-----------+----------
 Hans   | m          |  45000.00 |
 Jeanny | f          |  39000.00 |
 Evil   | f          | 100000.00 | f
(3 rows)
```

Dieses Ergebnis erscheint auf den ersten Blick korrekt, aber durch das Einfügen des neuen Wertes hat sich der Mittelwert geändert:

```
buch=# SELECT AVG(einkommen) FROM person;
        avg
--------------------
 53875.0000000000
(1 row)
```

Durch den höheren Mittelwert sollten zwei Datensätze eigentlich nicht mehr im Ergebnis sichtbar sein. Manche Leser denken jetzt vielleicht, dass dieser Sachverhalt logisch und vertretbar ist. Im Falle komplexer Berechnungen ist das jedoch definitiv nicht der Fall. Wenn Sie minderwertige Datenbanksysteme, die keine Subselects unterstützen, verwenden, ist das eben gezeigte Verfahren eventuell noch vertretbar, aber nicht im Falle von PostgreSQL.

Beim Durchführen einer derartigen Operation ist es wichtig, dass alle Operationen auf einer in sich konsistenten Datenbasis durchgeführt werden und genau das können Sie mit Subselects machen.

Sehen wir uns kurz die Lösung für unser Problem an:

```
SELECT *
        FROM person
        WHERE einkommen >
                (SELECT AVG(einkommen) FROM person);
```

Das durchschnittliche Einkommen wird in ein und der gleichen Abfrage berechnet und zur Laufzeit eingesetzt. Das stellt sicher, dass PostgreSQL zu jeder Zeit auf den gleichen, konsistenten Datenbestand zugreift und während der Abfrage keine störenden Operationen durchgeführt werden können.

Das Ergebnis der obigen Abfrage sollte etwa wie folgt aussehen:

```
 name | geschlecht | einkommen | vergeben
------+------------+-----------+----------
 Evil | f          | 100000.00 | f
(1 row)
```

Diesmal wird nur ein Datensatz zurückgegeben, was dem korrekten Ergebnis nach dem Einfügen des neuen Datensatzes entspricht.

Die letzte Abfrage hat gezeigt, wie Sie mit genau einem vom Subselect gelieferten Wert arbeiten können. In vielen Fällen reicht das nicht aus, da das Subselect eine ganze Liste von Werten zurückgibt. Auch in diesem Fall liefert PostgreSQL eine einfache Möglichkeit, die Anforderung zu lösen.

Das nächste Listing zeigt eine Abfrage, die das maximale Einkommen von Männern und Frauen zurückgibt. Es sollen alle Personen gelistet werden, die entweder das maximale Männer- oder das maximale Fraueneinkommen erhalten:

```
SELECT *
        FROM person
        WHERE einkommen IN
                (SELECT MAX(einkommen)
                        FROM person
                        GROUP BY geschlecht);
```

In diesem Beispiel die Abfrage diesmal ein klein wenig komplexer. Mithilfe des Schlüsselwortes `IN` werden alle Werte, die vom Subselect zurückgegeben werden, einzeln verglichen. Das Ergebnis der Abfrage sieht wie folgt aus:

```
 name | geschlecht | einkommen | vergeben
------+------------+-----------+----------
 Hans | m          |  45000.00 |
 Evil | f          | 100000.00 | f
(2 rows)
```

Sehen wir uns das Subselect ein wenig näher an. Das folgende Listing zeigt, dass es genau zwei Werte an die aufrufende Abfrage zurückgibt:

```
buch=# SELECT MAX(einkommen) FROM person GROUP BY geschlecht;
    max
-----------
 100000.00
  45000.00
(2 rows)
```

Die Abfrage von vorher kann natürlich auch ohne Verwendung des Schlüsselwortes `IN` realisiert werden:

```
SELECT *
        FROM person
        WHERE einkommen = (SELECT MAX(einkommen)
                FROM person
                WHERE geschlecht = 'm')
        OR einkommen = (SELECT MAX(einkommen)
                FROM person
                WHERE geschlecht = 'f');
```

Das Ergebnis der Query ist identisch mit dem Ergebnis, das wir vorher gesehen haben:

```
 name | geschlecht | einkommen | vergeben
------+------------+-----------+----------
 Hans | m          |  45000.00 |
 Evil | f          | 100000.00 | f
(2 rows)
```

Der Vorteil von `IN` ist, dass die vom Subselect zurückgegebene Liste beliebig lang sein kann. Im Falle von `OR` muss die Query erst mühsam generiert werden. Mithilfe von `IN` kann die Datenmenge variabel gehalten werden und Ihre SQL Statements werden flexibler.

Wenn Sie `IN` verwenden, müssen Sie bedenken, dass sehr lange Listen zu schlechterem Laufzeitverhalten führen können, weil sehr viele Vergleiche durchzuführen sind.

Eine weiteres Feature, das beim Arbeiten mit Subselects von großem Vorteil sein kann, ist das Keyword `EXISTS`. Sofern die Subquery gültige Daten zurückgibt, ist die `WHERE` Clause der Mutterabfrage erfüllt und Daten werden zurückgegeben. Um diesen Vorgang ein wenig anschaulicher zu gestalten, haben wir ein einfaches Beispiel eingefügt:

```
SELECT *
        FROM person
        WHERE EXISTS
                (SELECT 1
                        FROM person
                        WHERE geschlecht = 'x');
```

Die Subquery sucht alle Werte, die das Geschlecht `x` aufweisen. Sofern Datensätze gefunden werden, ist die `WHERE` Clause erfüllt. In diesem Fall gibt es keine Personen, die diesem Geschlecht zugeordnet sind, und das Ergebnis der Abfrage ist daher leer:

```
 name | geschlecht | einkommen | vergeben
------+------------+-----------+----------
 (0 rows)
```

Die Situation sieht anders aus, wenn Sie versuchen, nach Männern zu suchen:

```
SELECT *
        FROM person
        WHERE EXISTS
                (SELECT 1
                        FROM person
                        WHERE geschlecht = 'm');
```

In diesem Fall liefert die Subquery Daten und die WHERE Clause ist somit erfüllt. Das führt dazu, dass die gesamte Tabelle im Ergebnis zu finden ist:

```
    name    | geschlecht | einkommen | vergeben
------------+------------+-----------+----------
 Hans       | m          |  45000.00 |
 Paul Huber | m          |  31500.00 |
 Jeanny     | f          |  39000.00 |
 Evil       | f          | 100000.00 | f
(4 rows)
```

Im Falle einer Negation verwenden Sie statt EXISTS einfach NOT EXISTS. EXISTS und NOT EXISTS sind gute Möglichkeiten, effizient zu prüfen, ob bestimmte Daten vorhanden sind und um Queries darauf abzustimmen.

3.3 Import und Export von Daten

Nachdem Sie wissen, wie Tabellen angelegt und einfache Abfragen durchgeführt werden, ist es notwendig zu lernen, wie größere Mengen von Daten in PostgreSQL importiert werden können. Der Datenimport wird von vielen Leuten eher als Stiefkind gesehen und es wird ihm keine große Bedeutung eingeräumt. In vielen Fällen ist die Migration beziehungsweise der Import von Daten von zentraler Bedeutung. Vor allem beim Umstieg auf PostgreSQL ist es notwendig, die alten Datenstrukturen umzubauen oder in einfachen Fällen nur nachzubilden. Durch die leicht unterschiedlichen Datenformate kann es bei solchen Migrationen sehr schnell zu Problemen kommen, die den Erfolg eines IT-Projektes ernsthaft gefährden können.

In diesem Abschnitt wollen wir versuchen, einen Überblick über die möglichen Vorgehensweisen und Möglichkeiten beim Import von Daten zu geben. Wir hoffen, dass es Ihnen dadurch leichter fallen wird, die Basisdaten für Ihre Applikationen schnell und sicher zu verarbeiten.

3.3.1 COPY

Bisher wurde gezeigt, wie Daten mittels INSERT einer Datenbank hinzugefügt werden können. Durch den relativ großen internen Aufwand sind INSERT-Kommandos bei großen Datenmengen vergleichsweise langsam. Das kann bei Millionen von Datensätzen ein echtes Problem darstellen. Diesem Problem kann mithilfe des COPY-Befehles schnell und einfach begegnet werden. Bei größeren Datenmengen kann man als Faustregel sagen, dass COPY mindestens um Faktor 6 schneller ist als ein Set von INSERT-Statements. Die Gründe dafür sind naheliegend:

INSERT ist ein sehr mächtiger Befehl, der verschiedenste Möglichkeiten bietet. Durch diese Macht entsteht intern jedoch ein nicht zu vernachlässigender Mehraufwand.

Durch die mächtige Syntax von INSERT entsteht ein erheblicher Parsingaufwand. Die Syntaxanalyse von Millionen von INSERT-Statements dauert eine gewisse Zeit. Im Vergleich dazu entstehen bei COPY nur sehr geringe Kosten.

Die Unterstützung von PostgreSQL für Transaktionen und das hochentwickelte Locking System sind hauptverantwortlich für den Performanceunterschied. Im Vergleich mit anderen Open Source-Datenbanken wird oft hervorgehoben, dass PostgreSQL vergleichsweise langsam ist. Dasselbe gilt auch für Oracle, das in vielen Bereichen wesentlich schlechter abschneidet als bestimmte nicht so hoch entwickelte Open Source-Datenbanken. Der Grund dafür ist relativ leicht erklärbar. Je höher entwickelt eine Datenbank ist und je mehr Features unterstützt werden, desto mehr ist intern zu beachten und desto komplexer werden die internen Vorgänge, was in der Regel zu verminderter Performance führen kann. Im Allgemeinen ist das jedoch kein Problem und ein Opfer, das im Sinne von moderner und ANSI SQL kompatibler Datenbankentwicklung bedenkenlos erbracht werden soll. Speziell bei großen Datenbanken wendet sich das Blatt sehr schnell zu Gunsten qualitativ höherwertiger Datenbanken.

Nach diesem kleinen Exkurs in die Welt der Performance wollen wir uns nun mit der Syntax von COPY beschäftigen. Das nächste Listing zeigt eine Übersicht über die möglichen Syntaxelemente des Befehles:

```
1  buch=# \h COPY
2  Command:     COPY
3  Description: copy data between files and tables
4  Syntax:
5  COPY [ BINARY ] table [ WITH OIDS ]
6      FROM { 'filename' | stdin }
7      [ [USING] DELIMITERS 'delimiter' ]
8      [ WITH NULL AS 'null string' ]
9  COPY [ BINARY ] table [ WITH OIDS ]
10     TO { 'filename' | stdout }
11     [ [USING] DELIMITERS 'delimiter' ]
12     [ WITH NULL AS 'null string' ]
```

Wie Sie auf den ersten Blick erkennen können, unterstützt der COPY-Befehl bei PostgreSQL binäre Daten. Das ist besonders wichtig, wenn Sie Ihre Daten nicht gerade als Plain-Text verbreiten wollen. Des Weiteren können binäre Daten den Vorteil haben, dass sie nicht mehr in das interne Format bei PostgreSQL konvertiert werden müssen, was mehr Performance bringt.

Bevor wir uns weiter in die Features von COPY vertiefen, können wir uns ein einfaches Beispiel ansehen. Sollte die Tabelle person noch in Ihrer Datenbank sein, bitte ich Sie, diese zu löschen, bevor Sie fortfahren.

Das folgende Listing zeigt ein Stück Beipielcode, das eine Tabelle anlegt und Daten einfügt:

```
CREATE TABLE person (
        vorname         text,
        nachname        text,
        geburtsdatum    date,
        familienstand   text
);

COPY person FROM stdin;
Hans    Schoenig        1978/08/09      ledig
Ewald   Geschwinde      1976/06/21      verheiratet
Nadine  Blauhaus        1954/12/01      geschieden
\.
```

Die Daten im Befehl COPY sind mit Tabulator getrennt. Der Befehl liest die Daten von Standard-Input. Um das Ende des Datenstromes zu definieren, müssen Sie \. verwenden.

Um den Datenbestand an die Datenbank zu schicken, können Sie psql verwenden:

```
[postgres@duron code]$ psql buch < copy.sql
CREATE
```

Sofern kein Fehler aufgreten ist, hat PostgreSQL die Daten erfolgreich importiert.

Im vorangegangenen Beispiel haben Sie gesehen, wie Daten auf dem einfachsten Weg importiert werden können. In diesem Beispiel haben wir die Daten direkt von Standard-Input gelesen und die Spalten mittels Tabulator getrennt. In vielen Fällen ist es praktisch, das Trennzeichen selbst festzulegen und die Daten aus einem File einzulesen. Es ist ohne weiteres möglich, hinter diesem File eine Named Pipe zu verstecken, die gegebenenfalls für das Generieren der Daten verantwortlich ist. Das folgende Beispiel zeigt, wie Daten, die mit einem Strichpunkt getrennt sind, von einem File gelesen werden können.

Zuerst legen wir eine einfache Tabelle an, in der wir die Daten speichern werden:

```
CREATE TABLE speisekarte (id int4, produkt text, preis numeric(9,2));
```

Im nächsten Schritt sehen wir uns das zu importierende File an:

```
1;Hamburger;1.22
2;Cheeseburger;1.19
3;Mineralwasser;1.98
4;Käsebrot;1.49
```

Um die Daten in die Datenbank aufzunehmen, können wir den `COPY`-Befehl mit einigen Parametern aufrufen. Wichtig ist, dass die verwendeten Trennzeichen unter Anführungszeichen definiert werden. Gleiches gilt für den Namen der zu importierenden Datei (in unserem Fall `data.sql`):

```
buch=# COPY speisekarte FROM 'data.sql' DELIMITERS ';';
COPY
```

Sofern es zu keinen Problemen gekommen ist, können Sie nun die Daten aus der Tabelle selektieren:

```
buch=# SELECT * FROM speisekarte;
 id |    produkt    | preis
----+---------------+-------
  1 | Hamburger     |  1.22
  2 | Cheeseburger  |  1.19
  3 | Mineralwasser |  1.98
  4 | Käsebrot      |  1.49
(4 rows)
```

In der Regel sind `NULL`-Werte beim Datenimport besonders kritisch. Um dem Problem aus dem Weg zu gehen, erlaubt es PostgreSQL, zu definieren, welche Werte in der Datenbank als `NULL` dargestellt sind. Das ist wichtig, da leere Felder sonst nicht als `NULL`, sondern als leere Felder interpretiert werden.

Mithilfe der Option `WITH NULL AS` können Sie einen String definieren, den Sie als `NULL` deuten wollen.

Nachdem wir gezeigt haben, wie Daten mithilfe von `COPY` importiert werden können, sehen wir uns kurz an, wie Daten in der Datenbank in eine File exportiert werden können. Auch dafür eignet sich der `COPY`-Befehl hervorragend, wie dieses Listing beweist:

```
buch=# COPY speisekarte TO '/tmp/data.txt';
COPY
```

Hier haben wir die Daten in der Tabelle `Speisekarte` in das File `/tmp/data.txt` exportiert. Der Inhalt des Files ist nicht weiter überraschend:

```
[hs@notebook hs]$ cat /tmp/data.txt
1       Hamburger       1.22
2       Cheeseburger    1.19
3       Mineralwasser   1.98
4       Käsebrot        1.49
```

Die verschiedenen Spalten werden mit einem Tabulator getrennt ausgegeben, sodass es ohne Probleme möglich ist, das erzeugte File in die Datenbank zu reimportieren.

3.3.2 Datenverarbeitung mittels Shell

Arbeiten mit Datenbanken bedeutet nicht nur, mit SQL oder anderen Sprachen zu arbeiten. Das effektive Abfragen der Daten ist nicht der erste Schritt in der Kette der Verarbeitung. Der Import und die Konvertierung von Daten ist in der Regel wesentlich aufwändiger und fehleranfälliger als das Auswerten der Daten im zweiten Schritt. Die Erfahrung zeigt, dass speziell die Migration der Daten immer wieder zu Problemen führen kann und sehr viel Zeit in Anspruch nimmt. Außerdem verwenden viele Datenbankentwickler viel zu ineffiziente Algorithmen zum Konvertieren der Daten, die teilweise sehr viel Rechenzeit benötigen.

Da das richtige Aufbereiten von Daten einen wichtigen Platz im Leben eines Datenbankprofis einnimmt, haben wir uns entschlossen, einige einfache Wege der Datenverarbeitung aufzuzeigen.

Die zahlreichen UNIX-Shells stellen seit Jahren und Jahrzehnten eine zuverlässige und mächtige Umgebung dar. Viele Leute verwenden für die Aufbereitung Ihrer Daten komplexe Software, obwohl der überwiegende Teil der gängigsten Aufgaben wesentlich schneller mittels einfacher Shell Programme gelöst werden kann. Die meisten dieser Shell-Programme können sogar relativ unabhängig von der tatsächlich verwendeten Shell implementiert werden und stellen daher eine weitgehend plattformunabhängige Alternative zu anderer Software dar.

Das Ziel der nächsten Beispiele ist es, einige Zeilen aus `/etc/passwd` in eine PostgreSQL-Datenbank zu importieren. Um das Beispiel ein wenig zu verkürzen, speichern wir die letzten fünf Zeilen von `/etc/passwd` in `users.txt`:

```
[hs@notebook code]$ tail -n5 /etc/passwd > users.txt
[hs@notebook code]$ cat users.txt
mailman:x:41:41:List Manager:/var/mailman:/bin/false
ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
pvm:x:24:24::/usr/share/pvm3:/bin/bash
sfr:x:500:500:Charles O'Connor:/home/sfr:/bin/bash
hs:x:501:501::/home/hs:/bin/bash
```

Im folgenden Schritt können wir bereits die für den Import notwendigen Datenstrukturen in der Datenbank anlegen. Dieser Schritt erfolgt mit einem einfachen `CREATE TABLE`-Befehl:

```
CREATE TABLE password (
        login_name      text,
        password        text,
        uid             int4,
        gid             int4,
        user_name       text,
        directory       text,
        shell           text);
```

Der einfachste Weg, Daten aufzubereiten, ist, `sed` zu verwenden. `Sed` (stream editor) ist ein altes, sehr zuverlässiges UNIX-Tool zur nicht interaktiven Textverarbeitung. `Sed` ermöglicht es, einfache und komplexe Muster zu erkennen beziehungsweise weiterzuverarbeiten. Die Mustersuche basiert auf so genannten Regular Expressions, die es erlauben, sehr schnell komplexe Muster zusammenzustellen und durch andere Zeichenketten zu ersetzen. Auf Nicht-UNIX-Systemen gibt es de facto kein einziges derart einfaches Tool, um solche Operationen schnell und performant durchzuführen.

Das nächste Beispiel zeigt, wie die Doppelpunkte in unserem File durch Tabulatoren ersetzt werden können:

```
cat users.txt | sed –e "s/:/    /gi"
```

Hierbei genügt ein einfacher Befehl, um die Ersetzung durchzuführen. Wenn Sie sich den Output des Befehles ansehen, werden Sie erkennen, dass die Daten die gewünschte Struktur haben.

Natürlich können Sie auch wesentlich komplexere Operationen durchführen, aber das ist in diesem Fall nicht notwendig. Das mithilfe von `Sed` erstellte File kann direkt mit `COPY` importiert werden.

Ein weiteres, sehr mächtiges Tool ist `Awk`. Der Name `Awk` leitet sich von dessen Schöpfern Aho, Kernighan und Weinberger ab. Mittlerweile gehört der `Awk` zur Basisausstattung jedes UNIX-Betriebssystems und wird häufig für Transformationen jeder Art herangezogen. Das folgende Beispiel zeigt, wie der `Awk` zum Generieren von SQL-Code verwendet werden kann:

```
#!/bin/ksh

cat users.txt | awk "
   BEGIN { print \"COPY password FROM stdin USING DELIMITERS ':';\" }
   { print \$0 }
   END { print \"\\\.\" } " | psql buch
```

Am Beginn des Programmes wird der Inhalt von `users.txt` an den `Awk` geschickt. Das `Awk`-Programm selbst teilt sich in drei Abschnitte: Der Block `BEGIN` sorgt dafür, dass `Awk` sofort nach dessen Start eine Zeile ausgibt. In unserem Fall wird der Kopf von `COPY` dargestellt. Der nächste Block ist das Hauptprogramm. Es gibt `$0` aus. `$0` bezeichnet die erste Spalte, die zu verarbeiten ist. Da wir kein Trennsymbol definiert haben, ist die erste Spalte die gesamte Zeile.

Der letzte Block ist der `END`-Block, der aufgerufen wird, nachdem alle Datenzeilen abgearbeitet worden sind. Dieser Block tut nichts Anderes als `\.` auszugeben. Der gesamte Output von `Awk` wird an `psql` geschickt.

Sehen wir uns kurz an, was `Awk` produziert und was an `psql` geschickt wird:

```
1 COPY password FROM stdin USING DELIMITERS ':';
2 mailman:x:41:41:List Manager:/var/mailman:/bin/false
3 ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
4 pvm:x:24:24::/usr/share/pvm3:/bin/bash
5 sfr:x:500:500:Charles O'Connor:/home/sfr:/bin/bash
6 hs:x:501:501::/home/hs:/bin/bash
7 \.
```

Der Input kann von PostgreSQL direkt verarbeitet werden. Nach dem Import der Daten sind die einzelnen Zeilen in der Datenbank zu finden. Das nächste Listing zeigt (aus Platzgründen) einen Ausschnitt des Datenbestandes:

```
1 buch=# SELECT login_name, user_name FROM password;
2  login_name |    user_name
3 ------------+-----------------
4  hs         |
5  ldap       | LDAP User
6  mailman    | List Manager
7  pvm        |
8  sfr        | Charles O'Connor
9 (5 rows)
```

In diesem Beispiel hat der Import problemlos funktioniert.

In vielen Fällen stehen die zu importierenden Daten in einer Struktur mit fixer Satzlänge zur Verfügung. Speziell bei älteren Systemen sind fixe Satzstrukturen gängige Formate. Da der `COPY`-Befehl solche Daten nicht direkt importieren kann, müssen diese vorher entsprechend aufbereitet werden. Auch für solche Operationen bieten UNIX-Shells alles, was man für das tägliche Leben benötigt und in der Regel sogar noch etliches mehr.

Sehen wir uns das nächste Beispiel an. Das folgende File enthält Daten einer Speisekarte:

```
0001         Burger   1.99
0002   Cheeseburger   2.19
0003    Wurstsemmel   1.29
0004     Currywurst   3.49
```

Ziel ist es nun, die Spalten entsprechend zu trennen und die Spaltenfüller zu eliminieren. Das so erzeugte File soll bequem mit `COPY` importiert werden können. Da `Awk` auch Ersetzungen und Mustererkennung durchführen kann, ist die Operation mit sehr geringem Aufwand möglich, wie das nächste File verdeutlicht:

```
{
        string1 = substr($0, 1, 4)
        string2 = substr($0, 5, 15)
        string3 = substr($0, 20, 7)

        gsub("^(0)+", "", string1)
        gsub("^( )+", "", string2)
        gsub("^( )+", "", string3)
        print string1 ";" string2 ";" string3
}
```

Jede Zeile wird mittels `substr` in Ihre Teilkomponenten zerlegt. Im Folgenden werden die voranstehenden Nullen sowie überflüssige Blanks eliminiert. Die Ersetzungen erfolgen mit dem Befehl `gsub`. Der erste Parameter dieser Funktion enthält das Suchmuster, der zweite den neuen Text und der dritte Parameter enthält den zu modifizierenden String. In unserem Beispiel werden die Daten nach den Ersetzungen einfach inklusive Trennzeichen ausgegeben. Bedenken Sie, dass Sie in diesen Beispiel auch ohne `substr` auskommen können, indem Sie mit `FS` arbeiten.

Sehen wir also kurz an, was an Output produziert wird:

```
[hs@duron sql]$ awk -f awkprog.awk data.txt
1;Burger;1.99
2;Cheeseburger;2.19
3;Wurstsemmel;1.29
4;Currywurst;3.49
```

Das vorliegende File kann direkt importiert werden.

Wenn Sie planen, mit größeren Datenmengen zu arbeiten, die über eine externe Schnittstelle zu importieren sind, zahlt es sich auf alle Fälle aus, sich mit den wichtigsten UNIX-Tools vertraut zu machen, weil die Kommandozeile in den meisten Fällen immer noch das effizenteste zu Gebot stehende Mittel ist, Daten zu verarbeiten.

3.3.3 Arbeiten mit Binärdaten

In vielen Fällen kann es vorteilhaft sein, mit Binärdaten zu arbeiten. Speziell bei der Datensicherung können Binärdaten gewisse Vorteile gegenüber Plain-Text haben. Da Binärdaten nicht direkt lesbar sind, können sie einen überwiegenden Teil von Benutzern problemlos daran hindern, Ihre Daten zu lesen. Zusätzlich dazu sind Binärdaten in vielen Fällen schneller les- und schreibbar, weil keine Konvertierungen auf das interne Format durchgeführt werden müssen.

Zuallererst legen wir eine Tabelle an, in der wir die zu verarbeitenden Daten speichern werden:

```
1 buch=# CREATE TABLE essen (id int4, speise text, preis numeric(9,2));
2 CREATE
```

Dann importieren wir die im letzten Abschnitt generierten Daten (mit `Awk`):

```
1 buch=# COPY essen FROM '/tmp/data.ready' DELIMITERS ';';
2 COPY
```

Sofern beim Import der Daten keine Katastrophen passiert sind, können Sie die Tabelle nun mit `SELECT` abfragen:

```
1 buch=# SELECT * FROM essen;
2  id |    speise    | preis
3 ----+--------------+-------
4   1 | Burger       |  1.99
5   2 | Cheeseburger |  2.19
6   3 | Wurstsemmel  |  1.29
7   4 | Currywurst   |  3.49
8 (4 rows)
```

Um den Datenbestand im Binärformat zu exportieren, können Sie `COPY` mit dem Parameter `BINARY` aufrufen:

```
1 buch=# COPY BINARY essen TO '/tmp/data.binary';
2 COPY
```

Um die binären Daten wieder zu importieren, können Sie wieder `COPY FROM` verwenden:

```
1 buch=# COPY BINARY essen FROM '/tmp/data.binary';
2 COPY
```

Das Arbeiten mit Binärdaten funktioniert einfach und in der Regel auch absolut problemlos. Abhängig von der Form Ihrer Anwendung kann es vorteilhaft sein, Daten entweder als Plain-Text oder eben als Binärdaten abzulegen.

3.3.4 Arbeiten mit Fremdsystemen und Byteorder

Obwohl die Verbreitung von PCs mittlerweile ein gigantisches Ausmaß erreicht hat, sind PCs nicht die einzige Plattform für die Datenverarbeitung. Im professionellen Bereich kommen zahlreiche andere Systeme zum Einsatz, die jeweils ihre spezifischen Vor- und Nachteile haben. In Rechenzentren von Banken werden Sie auf IBM zSeries (S/390), IBM pSeries (RS/6000), iSeries (AS/400), SGIs, HP-UX und eine Reihe anderer Maschinen treffen. Unterschiedlichste Technologien kommen zum Einsatz, die teilweise den Datenimport ein wenig erschweren. Dieser Abschnitt soll Ihnen einen kurzen Überblick über die möglicherweise auftretenden Schwierigkeiten geben und Ihnen helfen, Probleme leichter zu lösen beziehungsweise den Problemkreis einzugrenzen.

Der gravierendste Unterschied zwischen den Systemen, der ausschließlich beim Arbeiten mit Binärdaten zu Tage tritt, ist die so genannte Byteorder. Wer glaubt, dass Zahlen auf allen System gleich abgespeichert werden, der irrt. Im Wesentlichen werden zwei Arten von Systemen unterschieden: Little Endian Systeme (x86, Alpha, …) speichern das signifikanteste Byte am Ende eines Speicherfeldes. Als fiktives Beispiel schauen wir uns die Zahl 245 an — im Falle von Little Endian wäre die Zahl intern als 542 gespeichert. Diese Vorgehensweise hat gewisse Vor- aber auch Nachteile, die hier nicht erläutert werden. Im Gegensatz dazu speichert ein Big Endian-System die Zahl auch intern als 245. Mit anderen Worten, das most significant byte (das Byte mit der größten Wertigkeit) ist vorne. Die wichtigsten Vertreter von Big Endian-Systemen sind Sparcs und PowerPCs.

Beim Import von Binärdaten ist der Big Endian unbedingt zu beachten. Das Ordnen der Bytes im Falle einer Konvertierung wird allgemein als Byte-Swapping bezeichnet.

Ein weiterer Unterschied zwischen verschiedenen Systemen liegt im verwendeten Zeichensatz. Auf PC-Systemen ist ASCII sehr verbreitet. Im Gegensatz dazu wird auf Großrechenanlagen vorwiegend EBCDIC-Code verwendet. Um EBCDIC-Code bei PC-Systemen auf ASCII konvertieren zu können, gibt es mehrere Möglichkeiten. Die einfachste davon ist, den guten alten `dd` zu verwenden. `dd` ist ein mächtiges UNIX-Tool, das für eine Vielzahl von Konvertierungen herangezogen werden kann.

Das folgende Beispiel zeigt, wie ein File von EBCDIC auf ASCII konvertiert werden kann:

```
1 [postgres@duron data]$ dd if=data.ebcdic of=/tmp/data.ascii
2 conv=ascii
3 6+1 records in
4 6+1 records out
```

Der umgekehrte Weg ist ebenfalls möglich, wie dieses Listing zeigt:

```
[postgres@duron data]$ dd if=data.ascii of=/tmp/data.ebcdic
conv=ebcdic
6+1 records in
6+1 records out
```

Das Ergebnis der Konvertierung können wir leider nicht darstellen, weil der Code für das Auge eines PC-Benutzers nicht sinnvoll lesbar ist.

Für jede verbreitete Programmiersprache sind Schnittstellen zu Mainframedaten verfügbar und die Datenkonvertierung wird mit der entsprechenden Satzstruktur kein großes Problem darstellen.

Wenn Sie eine vollständige EBCDIC-Tabelle benötigen, möchte in an dieser Stelle auf das englischsprachige »PostgreSQL Developer's Handbook« verweisen, das bei SAMS Publishing erhältlich ist.

3.3.5 Laufzeitüberlegungen

Beim Import von Daten sind Laufzeitüberlegungen von entscheidenter Bedeutung. Speziell bei der Übernahme großer Datenmengen kann ein wenig Wissen über die interne Verarbeitung der Daten nicht schaden und das Arbeiten mit PostgreSQL wesentlich erleichtern.

Je höher die zu importierende Datenmenge ist, desto schneller werden Sie erkennen, dass `INSERT` eher ein ungeeignetes Werkzeug ist. Der Grund dafür liegt in den Untiefen der Datenbank verborgen. Im Falle von `INSERT` muss jedes Statement geparst und analysiert werden. Zusätzlich entsteht intern ein nicht unerheblicher Administrationsaufwand. Dieses Laufzeitverhalten ist kein Manko von PostgreSQL, sondern eine natürliche Gegebenheit. Datenbanken, die über eine breite Palette von Features verfügen, müssen bei bestimmten Operationen geringfügig langsamer sein, wie Datenbanken, die nur über geringen Funktionsumfang verfügen, da intern wesentlich mehr Möglichkeiten beachtet werden müssen.

Im Vergleich mit anderen Open Source-Datenbanken schneidet PostgreSQL nicht allzu schlecht ab; wenn man jedoch bedenkt, welch großes Pool an Features bereitgestellt wird, die das Arbeiten auf Applikationslevel vereinfachen, sieht man die wahren Vorteile der Datenbank.

Kommen wir nun kurz auf den Datenimport zurück. Wenn Sie also Daten importieren, empfehlen wir den `COPY`-Befehl zu verwenden, da er bei größeren Datenmengen in der Regel mindestens um Faktor 6 schneller ist. Der Performancegewinn mit `COPY` ist sehr stark von der Beschaffenheit der Daten abhängig und kann daher nicht eindeutig bestimmt werden. Fest steht jedoch, dass der Import spürbar schneller vonstatten geht.

3.4 Indizierung

Beim Thema Performance stolpert man immer wieder über Indices. Allgemein formuliert sind Indices dazu da, um Daten in einer Tabelle schnell und effizient wiederzufinden. Intern werden dabei die Eigenschaften so genannter Bäume genutzt. Ein Baum ist eine Datenstruktur, die im Prinzip einer sortierten Liste mit Verweisen auf die entsprechenden Datensätze entspricht. Dadurch wird es möglich, Suchoperationen schnell durchzuführen.

Daten, die in zufälliger Reihenfolge in einem Speicherfeld zu finden sind, müssen im Allgeimen zur Gänze gelesen werden, um jedes Auftreten eines bestimmten Wertes zu finden. Im Falle von Indices hier weiterer Folge kurz erklärt sei. Dieser Vorgang ist essentiell und es ist wichtig, ihn zu verstehen, um effizient mit Datenbanken im Allgemeinen arbeiten zu können.

Stellen wir uns eine sortierte Liste von Werten vor (etwa ein Telefonbuch). Wenn Sie einen bestimmten Namen suchen, können Sie im Prinzip auf zwei Arten vorgehen. Sie können die Datenbank beispielsweise von vorne bis hinten durchlesen bis Sie den entsprechenden Wert gefunden haben. Wenn Sie 1.000.000 Datensätze in der Datenbank haben, werden Sie im Mittel etwa 500.000 Suchschritte benötigen, um den korrekten Datensatz zu finden. Wenn sich die zu verwaltende Datenmenge verdoppelt, verdoppelt sich auch die Laufzeit einer Suche. Im Falle von einer Milliarde Datensätzen wären das dann schon 500.000.000 Suchschritte. Wie man sich leicht denken kann, führt das zu viel zu langen Abfragezeiten.

Um der Situation Herr zu werden, können effizientere Suchalgorithmen herangezogen werden. Kehren wir wieder zu unserem Telefonbuch zurück. Wir schlagen das Buch in der Mitte auf und überprüfen, ob der gesuchte Name vor oder nach der aufgeschlagenen Seite zu finden ist. Alleine durch diesen ersten Suchschritt haben wir das Problem um die Hälfte verkleinert, da ein Teil des Telefonbuches den gesuchten Wert nicht mehr enthält. Nehmen wir also die verbleibende Hälfte und sehen wieder in der Mitte nach, wo der gesuchte Datensatz zu finden ist. Da wieder 50 Prozent wegfallen, haben wir nach zwei Suchschritten bereits drei Viertel der Daten ausgeschlossen. Die Halbierung der Daten wird so lange wiederholt, bis der gesuchte Wert gefunden ist. Im Falle von einer Milliarde Datensätzen können wir auf diese Weise nach etwa 30 Suchschritten den richtigen finden. Wenn wir die Datenmenge noch einmal vertausendfachen, werden wir aber lediglich zehn Suchschritte mehr benötigen und nicht wie befürchtet die 1.000-fache Rechenzeit. Ein derartiges Laufzeitverhalten bezeichnet man als logarithmisch, weil die Anzahl der maximal benötigten Suchschritte dem Logarithmus Dualis der Anzahl der Datensätze entspricht.

Im Falle von hochentwickelten Datenbanken werden nicht nur sortierte Listen verwendet. Um gewisse physikalische Eigenschaften der Hardware besser ausnützen zu können, werden heutzutage so genannte B+-Bäume verwendet. An dieser

Stelle wollen wir nicht genau auf die exakte Funktionsweise von PostgreSQL-Indexstrukturen eingehen, da das den Rahmen dieses Buches leider sprengen würde. Wichtig zu wissen ist, wie Indices prinzipiell funktionieren. Des Weiteren werden in der Praxis so genannte Mehrwegbäume verwendet (d.h. jeder Knoten hat mehr als zwei Nachfolger). Das führt dazu, dass der Logarithmus nicht zur Basis 2 berechnet wird. An der grundlegenden Funktionsweise eines Indexes ändert das aber nichts.

Nach diesem kurzen Ausflug in die Welt der Therorie wollen wir uns ansehen, wie Spalten indiziert werden können. Dazu legen wir eine kleine Tabelle an und fügen einige Datensätze in diese Tabelle ein:

```
1 buch=# CREATE TABLE zahl (id int4);
2 CREATE
```

Zum Generieren von Beispieldaten können Sie ein kleines Perl-Script schreiben. In unserem Falle verwenden wir ein kleines Perl-Script, das eine Million Zahlen generiert und sie in einem File speichert. Aus Gründen der Einfachheit besteht der Datenbestand nur aus einer Spalte:

```
1 #!/usr/bin/perl
2
3 open(FILE, "> data.sql") or die "kann file nicht öffnen\n";
4 print FILE "COPY zahl FROM stdin;\n";
5
6 for     ($i = 0; $i < 1000000; $i++)
7 {
8         print FILE      "$i\n";
9 }
10
11 print FILE '\.';
12 close(FILE);
```

Zum Generieren der Daten können Sie das Script direkt in der Shell aufrufen. Hier haben wir das Programm mit `time` gestartet, um zu sehen, wie lange die Generierung der Daten dauert:

```
1 [postgres@duron code]$ time ./gendata.pl
2
3 real    0m9.298s
4 user    0m8.720s
5 sys     0m0.180s
```

Wie Sie sehen können, geht die Generierung sehr schnell vor sich (die verwendete Maschine: AMD 750, 384 MB Ram, 120 GB IBM Platte).

Nach dem Generieren der Daten können Sie diese sehr schnell in die Datenbank importieren:

```
[postgres@duron code]$ time psql buch < data.sql

real    0m21.725s
user    0m1.540s
sys     0m0.070s
```

Der Import der Daten dauert etwas länger als deren Generierung, da PostgreSQL die Daten intern organisieren muss. Wenn Sie dieselbe Operation mit `INSERT`-Operationen durchführen, werden Sie bemerken, dass `COPY` wesentlich schneller ist.

Im nächsten Schritt werden wir versuchen herauszufinden, wie lange es dauert, einen speziellen Wert aus der Tabelle zu selektieren:

```
[postgres@duron code]$ time psql -c "SELECT * FROM zahl WHERE id=300"
-d buch
 id
-----
 300
(1 row)


real    0m2.110s
user    0m0.020s
sys     0m0.000s
```

Die Abfrage dauer etwa zwei Sekunden. Intern wird die gesamte Tabelle gelesen und jeder einzelne Wert mit der gesuchten Zahl verglichen.

Um die Abfrage zu beschleunigen, kann ein Index angelegt werden. Das kann mit dem Befehl `CREATE INDEX` erfolgen. Die Syntax des Befehles ist im folgenden Listing zu finden:

```
buch=# \h CREATE INDEX
Command:     CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( column [ ops_name ] [, ...] )
    [ WHERE predicate ]
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( func_name( column [, ... ]) [ ops_name ] )
    [ WHERE predicate ]
```

Aus dem Listing geht hervor, dass der Befehl sehr mächtig ist. In der Regel wird ein Index jedoch nur in zwei Varianten verwendet. Eine Möglichkeit ist, Indices zu erstellen, bei denen jeder Wert beliebig oft vorkommen kann. Eine andere Möglichkeit ist die Erstellung eines `UNIQUE`-Indexes. Ein `UNIQUE`-Index stellt sicher,

dass es in der Spalte keine doppelten Werte geben kann. Weiterhin ist ein UNIQUE-Index geringfügig schneller als einer, bei dem Werte öfter vorkommen können.

Im Beispiel legen wir einen normalen Index an, da wir dem Benutzer die Möglichkeit geben wollen, weitere, bereits vorhandene, Werte zur Datenbank hinzuzufügen.

Der nächste Befehl zeigt, wie ein Index erstellt werden kann:

```
buch=# CREATE INDEX idx_zahl_id ON zahl(id);
CREATE
```

In diesem Fall wird die einzige vorhandene Spalte indiziert. Bei der Indizierung wird intern ein Suchbaum aufgebaut. Dafür müssen die Daten sortiert und in die Baumstruktur eingetragen werden.

Nach dem Erstellen des Indexes versuchen wir, dieselbe Anfrage wie zuvor an die Datenbank zu schicken:

```
[postgres@duron code]$ time psql -c "SELECT * FROM zahl WHERE id=300"
-d buch
 id
-----
 300
(1 row)


real    0m0.047s
user    0m0.020s
sys     0m0.000s
```

Dabei ist die Abfrage signifikant schneller als die Abfrage ohne Index. Der Geschwindigkeitsgewinn beträgt nicht nur einige Prozent, sondern ist gewaltig. Je höher die Datenmenge in der Datenbank ist, desto gravierender fällt der Unterschied aus.

Um einen Index von der Datenbank zu entfernen, können Sie den Befehl DROP INDEX verwenden, der in der nachfolgenden Übersicht dargestellt ist:

```
buch=# \h DROP INDEX
Command:     DROP INDEX
Description: remove an index
Syntax:
DROP INDEX index_name [, ...]
```

Ein wichtiger Punkt beim Arbeiten mit Indices ist der richtige Zeitpunkt der Indexgenerierung. Nehmen wir an, Sie bauen eine neue Datenbank, in die Sie eine

größere Datenmenge importieren wollen. Stellen Sie sicher, dass die Indices nach dem Import der Daten erstellt werden. Das hat den Vorteil, dass die Daten nur einmal sortiert werden müssen. Wenn Sie den Index bereits vor dem Import erstellen, muss PostgreSQL alle vorhandenen Indices beim Import updaten und das dauert insgesamt länger als die Indices nach dem Import aufzubauen.

Das folgende Beispiel zeigt, wie der Datenbestand von zuvor noch einmal importiert wird. In diesem Fall sind bereits Daten und ein Index in der Datenbank:

```
[postgres@duron code]$ time psql buch < data.sql

real    1m1.884s
user    0m1.500s
sys     0m0.110s
```

Die Laufzeit ist wesentlich höher als beim Import ohne Index.

Indices bringen in der Regel Performancegewinne. Manchmal kann es jedoch auch vorkommen, dass ein Index zu Geschwindigkeitsverlust führt. Nehmen wir an, auf einer Tabelle ist eine Vielzahl von Indices definiert. Fügt nun jemand einen Wert ein, müssen alle Indices geupdatet werden. Das dauert logischerweise länger als das bloße Anhängen von Daten an eine Tabelle. Bei Anwendungen, bei denen viele `INSERT`- und `UPDATE`-Operationen durchgeführt werden müssen, kann das übermäßige Indizieren zu Problemen führen. Als Faustregel kann man sagen, dass nur Spalten indiziert werden sollen, in denen nach Werten gesucht wird. Außerdem ist es sinnlos, Spalten zu indizieren, bei denen zu viele gleiche Werte vorkommen. Es ist sinnlos, einen Index auf eine Spalte zu setzen, in der nur gleiche Werte zu finden sind — ein derartiges Vorgehen bringt keinen Vorteil beim Suchen.

Oft kommt es vor, dass gewisse Paare von Spalten abgefragt werden. Sehen wir uns die nächste Tabelle an:

```
buch=# CREATE TABLE person (vorname text, nachname text);
CREATE
```

Vielfach wird nach dem kompletten Namen gesucht. Um eine derartige Anfrage schnell abarbeiten zu können, gibt es zwei Möglichkeiten: Entweder Sie definieren auf jeder der beiden Spalten einen Index oder Sie definieren einen Index, der beide Spalten umfasst. Diese Vorgehensweise hat den Vorteil, dass nur ein Index abgefragt werden muss, was schneller ist. Der Nachteil dabei ist, dass der Index nur dann weiterhilft, wenn Sie nach beiden Spalten suchen — er bringt keinen Vorteil, wenn Sie nur nach dem Vornamen oder dem Nachnamen suchen. Das ist ein wesentlicher Punkt, der sehr oft nicht beachtet wird.

Im nächsten Listing ist zu sehen, wie zwei Spalten mit nur einem Index indiziert werden können:

```
1 buch=# CREATE INDEX idx_person_vornach ON person (vorname, nachname);
2 CREATE
```

Geben Sie einfach eine Liste der zu indizierenden Spalten an und PostgreSQL wird automatisch den entsprechenden Index aufbauen.

Zusätzlich zu B-Bäumen unterstützt PostgreSQL noch Hashes und R-Bäume. In vielen Fällen kann es sinnvoll sein, Hashes statt Bäumen zu verwenden, aber darauf wollen wir an dieser Stelle nicht eingehen.

3.5 Geometrische Daten

PostgreSQL ist nicht nur eine Datenbank zum Speichern von Standarddaten wie Text oder Zahlen. Die Möglichkeiten der Datenbank gehen weit über diese einfachen Funktionen hinaus und erlauben den Einsatz der Datenbank in vielen Nischenmärkten und Anwendungsgebieten. Eines der bedeutensten Einsatzgebiete von PostgreSQL ist zweifelsohne der geografische Bereich. Da es vergleichsweise einfach ist, neue Datentypen und Schnittstellen zu PostgreSQL hinzuzufügen, ergeben sich nahezu grenzenlose Möglichkeiten für den Einsatz der Datenbank. PostgreSQL verfügt über eine Vielzahl von geografischen Datentypen, die sogar über eigene Indexstrukturen angesprochen werden können. Aus diesem Grund können Geodaten sehr effizient bearbeitet und abgefragt werden.

Um mit geografischer Information umgehen zu können, werden intern so genannte R-Bäume verwendet. Der Vorteil von R-Bäumen gegenüber B+-Bäumen ist, dass die Datenstruktur es ermöglicht, räumliche Abfragen durchzuführen. Es ist beispielsweise möglich, mittels Index abzufragen, ob ein Punkt in einem bestimmten Umkreis eines anderen Punktes ist. Das ist mit B+-Bäumen in dieser Form nicht möglich, da diese es lediglich erlauben, Datensätze als Intervall oder einzelne Objekte abzufragen. Speziell bei größeren Datenmengen ist PostgreeSQL daher im Umgang mit Geodaten nahezu unschlagbar schnell und flexibel.

In diesem Kapitel werden wir uns ein wenig eingehender mit geometrischer Information und deren Verarbeitung beschäftigen.

3.5.1 Geometrische Datentypen

Wie bereits kurz erwähnt, stellt PostgreSQL eine Vielzahl von Datentypen zur Verfügung. Diese Datentypen sind bereits im Standardpaket enthalten und müssen nicht erst mühsam nachinstalliert werden. Sollten Sie mit den bereits vordefinierten Schnittstellen nicht das ausreichend versorgt sein, möchten wir Sie auf ein Zusatzpaket namens PostGIS verweisen, das weitere zusätzliche Datentypen enthält.

Wenden wir uns jedoch den Standarddatentypen zu:

`point`: Punkte sind der einfachste Datentyp, weil er nur aus zwei Komponenten besteht (aus x- und y-Koordinate).

`line`: Eine Linie ist ein aus zwei Punkten zusammengefügter Datentyp. Je ein Punkt kennzeichnet den Beginn und das Ende des Punktes.

`lseg`: Liniensegmente beschreiben Teile von Linien und bestehen ebenfalls aus zwei gegenüber liegenden Punkten.

`box`: Rechtecke werden durch zwei Eckpunkte beschrieben. Derzeit ist es nicht möglich, einen Winkel zu definieren, um das Rechteck in eine Schieflage zu bringen — dafür muss auf Streckenzüge respektive Polygone zurückgegriffen werden.

`circle`: Ein Kreis besteht aus einem Mittelpunkt und einem Radius.

`polygon`: Ein Polygon besteht aus einem Set von Punkten.

`path`: Ein Streckenzug besteht aus einem Set von Punkten.

Nach diesem Überblick über die vorhandenen Datentypen sehen wir uns kurz an, welche Eingabeformate PostgreSQL für die einzelnen Datentypen akzeptiert.

point

Punkte sind das Herzstück der vorher angeführten Datentypen. Sie erlauben es, einzelne Paare von Koordinaten in einem kartesischen Koordinatensystem zu identifizieren und zu lokalisieren.

Mehrere Eingabeformate werden akzeptiert:

```
1 buch=# SELECT '1, 0'::point, '(3, 2)'::point;
2  point | point
3 -------+-------
4  (1,0) | (3,2)
5 (1 row)
```

Zwei Inputformate werden akzeptiert. Wie im letzten Listing gezeigt wurde, akzeptiert PostgreSQL Eingaben mit und ohne Klammern.

`::point` kennzeichnet einen so genannten Cast. Casten bedeutet, dass der Datentyp des Ergebnisses explizit modifiziert wird. In unserem Falle soll das Ergebnis den Datentyp `point` aufweisen. In vielen Fällen wird automatisch gecastet — hier ist es jedoch notwendig, diese Operation explizit durchzuführen, weil die Datenbank den Input sonst als String deutet, wie das im nächsten Listing gezeigt wird:

```
buch=# SELECT '1, 0';
 ?column?
----------
  1, 0
(1 row)
```

Punkte benötigen 16 Bytes Speicherplatz.

line

In der vorliegenden Version 7.2 sind Linien noch nicht vollständig implementiert:

```
buch=# SELECT '1, 1, 2, 2'::line;
ERROR:  line not yet implemented
```

Um mit Linien zu arbeiten, ist es notwendig, den Datentyp `lseg` zu verwenden. Linien benötigen 32 Bytes Speicherplatz.

lseg

Liniensegmente sind der eigentliche Datentyp zum Verarbeiten von Linien. Im Wesentlichen basieren Liniensegmente auf einem Paar von Punkten. Aus diesem Grund werden folgende Inputformate akzeptiert:

```
buch=# SELECT '1, 1, 2, 2'::lseg, '(1, 1), (2, 2)'::lseg;
      lseg      |     lseg
----------------+---------------
 [(1,1),(2,2)] | [(1,1),(2,2)]
(1 row)

buch=# SELECT '(1, 1, 2, 2)'::lseg, '[(1, 1), (2, 2)]'::lseg;
      lseg      |     lseg
----------------+---------------
 [(1,1),(2,2)] | [(1,1),(2,2)]
(1 row)
```

Da ein Liniensegment aus Punkten besteht, werden alle Formate akzeptiert, die bei Punkten auch akzeptiert werden. Liniensegmente benötigen 32 Bytes Speicherplatz.

box

Der Datentyp `box` dient zum Speichern von Rechtecken. Ein Rechteck definiert sich über den linken oberen und den rechten unteren Eckpunkt. Die folgenden Datenformate werden als Input akzeptiert:

```
buch=# SELECT '1, 1, 2, 2'::box, '(1, 1), (2, 2)'::box;
     box     |     box
-------------+-------------
 (2,2),(1,1) | (2,2),(1,1)
(1 row)

buch=# SELECT '(1, 1, 2, 2)'::box, '((1, 1), (2, 2))'::box;
     box     |     box
-------------+-------------
 (2,2),(1,1) | (2,2),(1,1)
(1 row)
```

Rechtecke benötigen 32 Bytes Speicherplatz.

circle

Ein Kreis besteht aus einem Mittelpunkt und einem Radius. Die folgenden Beispiele zeigen, wie ein Kreis mit dem Mittelpunkt (1, 1) und einem Radius von 10 angegeben werden kann:

```
buch=# SELECT '1, 1, 10'::circle, '((1, 1), 10)'::circle;
   circle   |   circle
------------+------------
 <(1,1),10> | <(1,1),10>
(1 row)

buch=# SELECT '<(1, 1), 10>'::circle;
   circle
------------
 <(1,1),10>
(1 row)
```

Kreise benötigen 24 Bytes Speicherplatz.

path

Der Datentyp `path` eignet sich zum Ablegen von Streckenzügen. Im Wesentlichen werden zwei Arten unterschieden: Offene Streckenzüge sind eine Ansammlung von Linien. Geschlossene Streckenzüge sind solche, bei denen es eine Verbindung zwischen dem letzten und dem ersten Punkt gibt. Im Prinzip beschreiben geschlossene Streckenzüge eine Fläche.

Mehrere Inputformate werden von PostgreSQL akzeptiert:

```
buch=# SELECT '1,1, 2,3, 4,6'::path, '(1,1), (2,3), (4,6)'::path;
        path         |        path
---------------------+---------------------
 ((1,1),(2,3),(4,6)) | ((1,1),(2,3),(4,6))
(1 row)

buch=# SELECT '(1,1, 2,3, 4,6)'::path, '((1,1), (2,3), (4,6))'::path;
        path         |        path
---------------------+---------------------
 ((1,1),(2,3),(4,6)) | ((1,1),(2,3),(4,6))
(1 row)

buch=# SELECT '[(1,1), (2,3), (4,6)]'::path;
        path
---------------------
 [(1,1),(2,3),(4,6)]
(1 row)
```

Eckige Klammern zeigen an, dass es sich hierbei um einen offenen Streckenzug handelt. Um einen Pfad explizit als geschlossen beziehungsweise als offen zu markieren, können die Funktionen `popen` und `pclose` verwendet werden. Um zu prüfen, ob ein Streckenzug offen oder geschlossen ist, stehen die Funktionen `isopen` und `isclosed` zur Verfügung:

```
buch=# SELECT isopen('[(1,1), (2,3), (4,6)]'::path);
 isopen
--------
  t
(1 row)

buch=# SELECT isopen('((1,1), (2,3), (4,6))'::path);
 isopen
--------
    f
(1 row)
```

In diesem Beispiel zeigen runde Klammern an, dass es sich beim vorliegenden Datensatz um einen geschlossenen Streckenzug handelt.

Ein Streckenzug benötigt 4 Bytes für Stammdateninformation sowie 32 Bytes pro Knoten.

polygon

Polygone können als geschlossene Streckenzüge angesehen werden. Intern werden Polygone jedoch anders behandelt und PostgreSQL stellt spezielle Funktionen zum Abarbeiten derselben zur Verfügung.

Zur Eingabe von Polygonen werden leicht andere Inputformate als bei Streckenzügen akzeptiert:

```
buch=# SELECT '1,1, 2,2, 4,4'::polygon, '(1,1), (2,2),
(4,4)'::polygon;
       polygon       |       polygon
---------------------+---------------------
 ((1,1),(2,2),(4,4)) | ((1,1),(2,2),(4,4))
(1 row)

buch=# SELECT '(1,1, 2,2, 4,4)'::polygon,
'((1,1), (2,2), (4,4))'::polygon;
       polygon       |       polygon
---------------------+---------------------
 ((1,1),(2,2),(4,4)) | ((1,1),(2,2),(4,4))
(1 row)
```

Alle Inputformate führen zum gleichen Ergebnis.

Ein Streckenzug benötigt 4 Bytes für Stammdateninformation sowie 32 Bytes pro Knoten.

3.5.2 Operatoren für geometrische Abfragen

Um mit geometrischen Datentypen effizient arbeiten zu können, benötigt man ein Set spezieller Operatoren. Wie wir bereits erwähnt haben, bietet PostgreSQL ein außerordentlich weitreichendes Spektrum an Möglichkeiten, das großteils auf effizienten Operatoren beruht. In diesem Abschnitt werden wir uns ein wenig näher mit den von PostgreSQL zur Verfügung gestellten Möglichkeiten beschäftigen. Danach werden wir die wesentlichsten Operatoren kurz detailiert vorstellen:

+

Der +-Operator dient zum Durchführen einfacher Additionen und ist für eine Vielzahl von Kombinationen von Datentypen definiert. Deshalb sei eine Liste von Beispielen zur Verwendung dieses Datentypes angeführt:

```
buch=# SELECT '1,1'::point + '4,3'::point;
 ?column?
----------
  (5,4)
(1 row)
```

Wenn zwei Punkte addiert werden, bedeutet das nichts anderes, als dass beide Koordinaten einfach addiert werden.

Sehen wir uns das nächste Beispiel an, bei denen wir versuchen, ein Rechteck und einen Punkt zu addieren:

```
buch=# SELECT '1,1'::point + '(4,3), (2,2)'::box;
 ?column?
----------
 (4,3.5)
(1 row)

buch=# SELECT '(4,3), (2,2)'::box + '1,1'::point;
  ?column?
------------
 (5,4),(3,3)
(1 row)
```

Das erste Ergebnis ist denkbar überraschend — die zweite Berechnung liefert jedoch das zu erwartende Ergebnis. Dieses Beispiel zeigt sehr eindrucksvoll, dass geometrische Operatoren nicht unbedingt kommutativ sein müssen und es daher nicht egal ist, welcher Wert zu welchem zuerst addiert wird. Das ist ein sehr wesentlicher Punkt, der Sie durch den restlichen Abschnitt begleiten wird.

Das folgende Beispiel zeigt, wie ein Punkt zu einem Kreis addiert werden kann:

```
buch=# SELECT '4,3, 1'::circle + '10,10'::point;
  ?column?
-------------
 <(14,13),1>
(1 row)
```

Hier entspricht das Ergebnis wieder unseren Erwartungen — wir empfehlen aber trotzdem, alle Berechnungen, die Sie in Applikationen einbauen, vorher in der Shell zu prüfen und sicherzustellen, dass Ihre Berechnung von der Datenbank in Ihrem Sinne gedeutet wird. Alle bisher gezeigten Ergebnisse sind absolut korrekt — wenn das Ergebnis einmal nicht in Ihrem Erwartungsbereich liegen sollte, liegt das immer an einer falscher Formulierung der Berechnung und sicherlich nicht an der Software. Im Prinzip ist diese Regel überall gültig; als Faustregel kann man sagen, dass die Datenbank immer Recht hat (das hat sich in unserer langjährigen Praxis nur zu oft bestätigt).

–

Der −-Operator ist das exakte Gegenstück des +-Operators. Hier einige Beispiele:

```
buch=# SELECT '1,1'::point - '3,3'::point;
 ?column?
-----------
 (-2,-2)
(1 row)

buch=# SELECT '1,1, 10,10'::lseg - '3,3'::point;
   ?column?
--------------
 (-7.5,-7.5)
(1 row)
```

Die Ergebnisse entsprechen den Erwartungen. Die verwendeten Konzepte entsprechen denen des +-Operators.

*****

Der *-Operator dient zum Durchführen geometrischer Skalierungen und Rotationen. Vorab möchten wir einmal klarstellen, wofür der Operator nicht verwendet werden kann:

```
buch=# SELECT '3, 4'::point * 3;
ERROR:  Unable to identify an operator '*' for types 'point' and
'integer'
        You will have to retype this query using an explicit cast
```

Das Ergebnis ist nicht »9, 12«, sondern undefiniert. Das ist ein wichtiger Punkt, der von vielen Leuten oft vergessen wird.

Wie bereits erwähnt, dient der Operator für Skalierungen sowie Rotationen. Die folgenden Beispiele sollen die Arbeitsweise des Operators ein wenig erläutern:

```
buch=# SELECT box '((1,1),(0,0))' * point '(2,0)';
   ?column?
--------------
 (2,2),(0,0)
(1 row)

buch=# SELECT box '((1,1),(0,0))' * point '(2,1)';
   ?column?
--------------
 (1,3),(0,0)
(1 row)
```

```
12
13 buch=# SELECT box '((1,1),(2,2))' * point '(2,1)';
14   ?column?
15 -------------
16  (2,6),(1,3)
17 (1 row)
```

Um die Funktionsweise des Operators vollständig zu verstehen, sind gewisse geometrische Grundkenntnisse vonnöten, auf die wir hier aus Platzgründen jedoch nicht eingehen wollen.

/

Wie der *-Operator ist auch der /-Operator für Skalierungen bestimmt. Die Funktionsweise des Operators unterscheidet sich nicht von der des *-Operators.

#

Zur Berechnung von Überschneidungen kann der #-Operator herangezogen werden. Die folgenden beiden Beispiele zeigen wie die Schnittmenge zweier Objekte berechnet werden kann:

```
1 buch=# SELECT '<(1,1), 3>'::circle # '0,0, 2,2'::box;
2   ?column?
3 -------------
4  (2,2),(0,0)
5 (1 row)
6
7 buch=# SELECT '<(1,1), 3>'::circle # '100,100, 200,200'::box;
8  ?column?
9 ----------
10
11 (1 row)
```

Die erste Berechnung liefert ein Ergebnis. Die beiden Objekte, die in der zweiten Abfrage verwendet werden, haben nichts Gemeinsames und das Ergebnis ist daher leer.

Der #-Operator kann nicht nur zum Berechnen von Schnittmengen herangezogen werden. Um die Anzahl der Elemente in einem Polygon zu berechnen, leistet der Operator ebenfalls gute Dienste. Das nächste Beispiel zeigt einen Anwendungsfall:

```
1 buch=# SELECT # '(4,6), (1,17), (3,3)'::polygon;
2  ?column?
3 ----------
4          3
5 (1 row)
```

##

Wenn Sie den Punkt mit dem geringsten Abstand berechnen wollen, können Sie den ##-Operator einsetzen. Das folgende Listing zeigt, wie der Operator eingesetzt werden kann; in diesem Beispiel sehen Sie, wie mit Kreisen gearbeitet werden kann:

```
buch=# SELECT '<(1,1), 3>'::circle ## '<(5,8), 4>'::circle;
               ?column?
-------------------------------------
 (2.17157287525381,5.17157287525381)
(1 row)
```

&&

PostgreSQL bietet eine Möglichkeit, herauszufinden, ob sich zwei Objekte überlappen. Derartige Berechnungen werden mit dem &&-Operator durchgeführt. Das nächste Listing enthält zwei einfache Beispiele:

```
buch=# SELECT '<(1,1), 3>'::circle && '<(5,5), 4>'::circle;
 ?column?
----------
 t
(1 row)

buch=# SELECT '<(1,1), 3>'::circle && '<(5,5), 1>'::circle;
 ?column?
----------
 f
(1 row)
```

Die ersten beiden Objekte überlappen sich. Die Daten in der zweiten Abfrage haben keine Schnittmenge.

&< und &>

&< und &> prüfen, ob sich zwei Objekte links oder rechts überlappen.

<< und >>

Der <<-Operator gibt true zurück, wenn sich ein Objekt links von einem anderen befindet.

```
buch=# SELECT '1,1'::point << '2,1'::point;
 ?column?
----------
 t
(1 row)

buch=# SELECT '1,1'::point << '0,1'::point;
 ?column?
----------
 f
(1 row)
```

Im Beispiel liefert der Operator unterschiedliche Ergebnisse abhängig davon, wo sich die beiden Objekte, die betrachtet werden, befinden.

Das Gegenstück von << ist der >>-Operator, der prüft, ob ein Objekt rechts von einem anderen Objekt ist; hier zeigen wir ein Beispiel:

```
buch=# SELECT '1,1'::point >> '0,1'::point;
 ?column?
----------
  t
(1 row)
```

<->

Zur Berechnung der Distanz zwischen zwei Objekten wird der <–>-Operator zur Verfügung gestellt. Das folgede Beispiel zeigt, wie man den Abstand zwischen zwei Punkten berechnen kann:

```
buch=# SELECT '1,1'::point <-> '4,6'::point;
     ?column?
------------------
 5.8309518948453
(1 row)
```

Intern wird zur Berechnung der Distanz der Lehrsatz des Pythagoras herangezogen, was sehr schnell zu einem Ergebnis führt.

@

Um zu sehen, ob ein Objekt ein Teil eines anderen Objektes ist, stellt PostgreSQL den @-Operator zur Verfügung. Das folgende Beispiel zeigt, wie herausgefunden werden kann, ob ein Punkt innerhalb eines Kreises ist oder nicht:

```
buch=# SELECT '1,1'::point @ '1,3, 4'::circle;
 ?column?
----------
 t
(1 row)
```

3.5.3 Eine Übersicht über alle geometrischen Operatoren

Nach diesem etwas detaillierteren Überblick über die Funktionsweise der wichtigsten geometrischen Operatoren wenden wir uns nun einer Liste aller Operatoren zu:

`+`	Addition (Translation)	`-`	Subtraktion (Translation)
`*`	Multiplikation (Scaling/Rotation)	`/`	Division (Scaling/Rotation)
`#`	Schnittmenge beziehungsweise Zahl der Elemente in einem Polygon	`##`	Punkt mit dem geringsten Abstand
`&&`	Prüfen auf Überlappungen	`&<`	Prüfen auf linke Überlappung
`&>`	Prüfen auf rechte Überlappung	`<->`	Abstand zwischen zwei Objekten
`<<`	Prüfen, ob ein Objekt links von einem anderen Objekt ist	`?#`	Prüfen, ob sich zwei Objekte schneiden oder überlappen
`?-`	Prüfen, ob zwei Objekte aufeinander horizontal stehen	`?-|`	Prüfen, ob zwei Objekte aufeinander senkrecht stehen
`@-@`	Berechnen des Umfangs eines Objektes	`?|`	Prüfen, ob zwei Objekte vertikal aufeinander stehen
`?||`	Prüfen auf Parallelität	`@`	Prüfen, ob ein Objekt ein anderes enthält
`@@`	Berechnen des Mittelpunktes eines Kreises		

3.6 Speichern von Netzwerkinformationen

In manchen Fällen kann es sinnvoll sein, Netzwerkinformationen abzulegen. Zu diesem Zwecke stellt PostgreSQL einige auf das Speichern von Netzwerkinformationen spezialisierte Datentypen bereit. Im Prinzip kann jegliche Information auch als Text abgelegt werden aber im Gegensatz zu Text bieten spezialisierte Datentypen einige wichtige Vorteile. Der größte Vorteil ist wohl, dass die Lesefunktion eines Datentyps sicherstellt, dass keine ungültigen Werte eingegeben

werden können; wir werden uns im Laufe dieses Abschnittes noch eingehend damit beschäftigen.

Bevor wir uns detailliert mit jedem Datentyp beschäftigen, wenden wir uns einer kleinen Übersicht zu:

`cidr`: IP-Adresse/Netmask, Netzwerk-Adresse

`inet`: IP-Adresse mit optionaler Netmask

`macaddr`: MAC-Adresse

Im Folgenden wollen wir uns kurz ansehen, wie diese Datentypen effizient eingesetzt werden können.

`cidr` und `inet`

`cidr` und `inet` sind zwei sehr ähnliche Datentypen. Der Hauptunterschied liegt darin, dass `inet` Bits rechts von der Netmask akzeptiert, die ungleich 0 sind.

Das nächste Listing zeigt die wichtigsten von PostgreSQL akzeptierten Inputformate:

```
buch=# SELECT '192.168.1.128/25'::cidr, '192.168'::cidr;
       cidr       |      cidr
------------------+---------------
 192.168.1.128/25 | 192.168.0.0/24
(1 row)

buch=# SELECT '192.168.1.128/25'::inet, '192.168.1.1'::inet;
       inet       |    inet
------------------+------------
 192.168.1.128/25 | 192.168.1.1
(1 row)
```

macaddr

PostgreSQL stellt einen eigenen Datentyp zum Speichern von MAC-Adressen zur Verfügung. Die folgenden Beispiele zeigen die Funktionsweise des Datentyps:

```
buch=# SELECT '35302b:e15203'::macaddr, '35302b-e15203'::macaddr;
      macaddr      |      macaddr
-------------------+-------------------
 35:30:2b:e1:52:03 | 35:30:2b:e1:52:03
(1 row)

buch=# SELECT '3530.2be1.5203'::macaddr, '35-30-2b-e1-52-03'::macaddr;
```

```
       macaddr       |       macaddr
---------------------+---------------------
35:30:2b:e1:52:03 | 35:30:2b:e1:52:03
(1 row)

buch=# SELECT '35:30:2b:e1:52:03'::macaddr;
       macaddr
-------------------
 35:30:2b:e1:52:03
(1 row)
```

Wie unschwer zu erkennen ist, wird eine Vielzahl von Eingabeformaten akzeptiert.

Um effizient mit Netzwerkdatentypen arbeiten zu können, stellt PostgreSQL ein Set von Operatoren zur Verfügung. Ein grundlegendes Set von Operatoren ist für jeden Datentyp notwendig, da es nur so möglich wird, mit dem Datentypen zu arbeiten und Abfragen zu formulieren. Das folgende Listing zeigt zwei ganz einfache Operationen:

```
buch=# SELECT '3530.2be1.5203'::macaddr = '35302b-e15203'::macaddr;
 ?column?
----------
 t
(1 row)

buch=# SELECT '3530.2be1.5203'::macaddr > '35302b-e15203'::macaddr;
 ?column?
----------
 f
(1 row)
```

3.7 Arbeiten mit Datum

Seit jeher sind Berechnungen mit Datum und Uhrzeit das Schreckgespenst eines jeden Programmierers. Das Arbeiten mit Zeiten zeichnet sich durch eine Fülle von Ausnahmen aus und ist alles andere als leicht zu bewerkstelligen. PostgreSQL stellt ein extrem ausgereiftes und zuverlässiges System zum Arbeiten mit Zeiten und Zeitbereichen aller Art zur Verfügung; in diesem Abschnitt werden wir uns eingehend damit beschäftigen.

Beginnen wir mit den wichtigsten eingebauten Datentypen.

3.7.1 `date`

Zum Abspeichern einfacher Datumsinformationen kann der Datentyp `date` verwendet werden. Wie alle Datentypen, die zur Verarbeitung von Zeit dienen, ist

date sehr flexibel und akzeptiert eine unglaubliche Vielzahl von Ein- und Ausgabeformaten. Um Ihnen einen guten Überblick über die vorhandenen Möglichkeiten zu geben, werden wir zu jedem Inputformat ein kleines Beispiel anführen. Ziel ist es, den 11. September 2001 in möglichst vielen häufig verwendeten Formaten anzuzeigen:

```
buch=# SELECT 'September 11, 2001'::date, '2001-09-11'::date;
    date    |    date
------------+------------
 2001-09-11 | 2001-09-11
(1 row)

buch=# SELECT '09/11/2001'::date, '20010911'::date, '010911'::date;
    date    |    date    |    date
------------+------------+------------
 2001-09-11 | 2001-09-11 | 2001-09-11
(1 row)
```

Es werden noch einige weitere Formate akzeptiert, die jedoch in der Regel nicht verwendet werden. Im Folgenden haben wir bewusst auf das europäische Eingabeformat (11/09/2001) verzichtet, weil es aus Gründen der Lesbarkeit und Eindeutigkeit nicht verwendet werden sollte. Vor allem im Hinblick auf Sortierung kann das europäische Format sehr unvorteilhaft sein.

3.7.2 time

Zum Ablegen von Uhrzeiten stellt PostgreSQL einen Datentyp namens time zur Verfügung, den es in zwei Varianten gibt: time with time zone und time without time zone. Diese »Zweiteilung« des Datentyps erlaubt große Flexibilität und eine Applikation kann unter Verwendung von Zeitzonen völlig unabhängig vom Einsatzort implementiert werden.

Wie bei den vorangegangenen Datentypen wollen wir uns auch hier die erlaubten Eingabeformate kurz ansehen:

```
buch=# SELECT '16:32:05.74564'::time with time zone,
'16:32:05.74564'::time;
       timetz       |      time
--------------------+-----------------
 16:32:05.745640+01 | 16:32:05.745640
(1 row)
```

Um anzugeben, dass Zeitzonen in Betracht gezogen werden sollen, muss »with time zone« explizit angegeben werden, da sonst die Variante ohne Zeitzonen verwendet wird. Um die Zeitzone explizit wegzulassen, können Sie auch time without time zone verwenden.

3.7.3 interval

Um mit Zeitbereichen arbeiten zu können, stellt PostgreSQL einen Datentyp namens `interval` zur Verfügung. Dieser Datentyp erlaubt komplexe Berechnungen ohne viel Aufwand und kann flexibel eingesetzt werden. Sehen wir uns anhand eines Beispieles an, wie der Datentyp eingegeben werden kann:

```
buch=# SELECT '2 centuries, 54 years, 9 months, 2 days, 3 hours, 9
minutes, 2 seconds'::interval;
              interval
-----------------------------------
 254 years 9 mons 2 days 03:09:02
(1 row)
```

`Interval` erlaubt es dem Benutzer, Zeitbereiche mit Worten zu beschreiben. Der eingegebene Wert wird analysiert und gegebenfalls neu formuliert. Wenn Sie etwa mit Jahrhunderten rechnen, werden diese immer automatisch auf Jahre umgewandelt. Das hat den Vorteil der größeren Übersichtlichkeit und der besseren Lesbarkeit. Dasselbe gilt auch für Jahrzehnte:

```
buch=# SELECT '2 decade'::interval, '2 DecadeS'::interval;
 interval | interval
----------+----------
 20 years | 20 years
(1 row)
```

Wichtig zu bemerken ist, dass es egal ist, ob Sie die Schlüsselwörter groß oder klein schreiben; Singular und Plural sind ebenfalls nicht von Bedeutung, weil PostgreSQL den Wert automatisch richtig interpretiert.

3.7.4 Datumsberechnungen

Mit Datum und Uhrzeit arbeiten bedeutet mehr als nur das Abspeichern von Informationen. PostgreSQL unterstützt eine Reihe von Operatoren, die Berechnungen jeder Art zu einem Kinderspiel werden lassen.

Um uns ein wenig in die Materie einzudenken, beginnen wir mit einigen einfachen Beispielen:

```
buch=# SELECT now();
              now
-------------------------------
 2002-05-01 09:32:24.121312+02
(1 row)
```

`now` fragt die aktuelle Transaktionszeit ab. Wir möchten an dieser Stelle noch einmal ausdrücklich darauf hinweisen, dass `now` immer den Zeitpunkt der aktuellen

Transaktion bestimmt — innerhalb einer Transaktion ist dieser konsistent und folglich identisch.

Wenn Sie Datentypkonvertierungen durchführen wollen, kann das leicht mithilfe des ::-Operators geschehen:

```
buch=# SELECT now()::date, now()::time with time zone;
    now     |        now
------------+-------------------
 2002-05-01 | 09:33:13.663387+02
(1 row)
```

Beginnen wir gleich mit der ersten Berechnung:

```
buch=# SELECT now(), now() - '1 day'::interval;
               now               |           ?column?
---------------------------------+-------------------------------
 2002-05-01 09:31:02.005541+02 | 2002-04-30 09:31:02.005541+02
(1 row)
```

Mit dieser Berechnung ist es möglich, einen Tag vom aktuellen Zeitstempel abzuziehen. Die Berechnung wirkt klar aber Berechnungen dieser Art sollten niemals auf diese Weise durchgeführt werden, wie am folgenden Beispiel zu erkennen ist:

```
buch=# SELECT ('2002-10-26'::date + '1 day'::interval)::date;
    date
------------
 2002-10-27
(1 row)
```

Die Welt scheint noch in Ordnung sein, da der 27. offensichtlich ein Tag nach dem 26. kommt — versuchen wir eine ähnliche Berechnung:

```
buch=# SELECT ('2002-10-27'::date + '1 day'::interval)::date;
    date
------------
 2002-10-27
(1 row)
```

Das riecht verdächtig nach Bug aber weit gefehlt — die Berechnung ist absolut korrekt, wie das nächste Beispiel gleich zeigen wird:

```
buch=# SELECT '2002-10-27'::date + '1 day'::interval;
        ?column?
------------------------
 2002-10-27 23:00:00+01
(1 row)
```

»1 day« bedeutet 24 Stunden. Da aber am 26. Oktober eine Stunde Zeitverschiebung (Stichwort »Daylight Saving Time«) berücksichtigt werden muss, ist die Verwendung von 24 Stunden fatal. Um das Problem eliminieren zu können, muss die Abfrage wie folgt formuliert werden:

```
buch=# SELECT '2002-10-27'::date + 1;
  ?column?
------------
 2002-10-28
(1 row)
```

In diesem Fall wird der Integerwert als Tag interpretiert und die Berechnung läuft korrekt ab.

Also ist die Zahl der möglichen Fehler, die Ihnen unterlaufen können, unendlich. Dieses Beispiel illustriert auch, wie hoch die Qualität der Software letztendlich ist und welche Fülle von Möglichkeiten sich bieten, wenn Sie das System erst einmal verstanden haben.

Bei der täglichen Arbeit mit PostgreSQL werden Sie sehr schnell erkennen, dass bestimmte Operatoren nicht definiert sind:

```
buch=# SELECT now() + now();
ERROR:  Unable to identify an operator '+' for types 'timestamp with
time zone' and 'timestamp with time zone'
        You will have to retype this query using an explicit cast
```

Wenn Sie sich ein wenig Zeit nehmen und Ihre Abfrage noch einmal in Ruhe durchdenken, werden Sie erkennen, dass das in der Regel einen Grund hat. In unserem Beispiel kann man sehen, dass die Addition von zwei Zeitstempeln einfach keinen Sinn macht und es daher auch keinen Operator dafür gibt.

3.8 Transaktionen

Jede hoch entwickelte Datenbank unterstützt so genannte Transaktionen. Im Gegensatz zu anderen Open Source-Datenbanken unterstützt PostgreSQL seit langer Zeit Transaktionen sehr gut. Da Transaktionen im Kern der Datenbank fest enthalten sind, ist es nicht notwendig, dubiose Zusatzpakete oder Datenbankformate zu installieren, um in den Genuss von Transaktionen zu kommen. In diesem Abschnitt werden wir uns ein wenig näher mit der Implementation von Transaktionen bei PostgreSQLs beschäftigen und sehen, wie Transaktionen eingesetzt werden können, um Fehler zu vermeiden und Ihre Applikationen klarer zu gestalten.

3.8.1 Grundlagen

Bevor wir uns tiefer in das praktische Geschehen stürzen, ist es notwendig, sich ein wenig genauer anzusehen, was Transaktionen sind und wozu sie verwendet werden können:

In vielen Fällen schickt der User ein Set von SQL-Statements an den Server, der diese dann auswertet. Nehmen wir also an, ein User will Daten importieren und sendet Tausende von Datensätzen an die Datenbank. Nach Hunderten oder Tausenden Datensätzen tritt plötzlich ein Fehler auf oder die Applikation des Users kollabiert. Was passiert mit den bereits eingefügten Datensätzen? Zuerst würde man meinen, dass die bereits veränderten Datensätze modifiziert bleiben sollten, aber was passiert, wenn der Benutzer nun seinen Datenimport neu startet? Auf einmal werden mehrere Kopien desselben Datensatzes existieren und niemand wird mehr wissen, wer was wann eingefügt hat. Mit anderen Worten, ein unendliches Durcheinander ist vorprogrammiert, nicht jedoch im Falle von Transaktionen. Wenn ein User die Daten in einer einzigen Transaktion an den Server schickt, werden die Daten erst sichtbar, wenn der letzte Datensatz korrekt eingefügt worden ist. Sollte während des Imports ein Fehler auftreten, sorgt die Datenbank dafür, dass alle bisher eingefügten Datensätze niemals sichtbar werden. Das hat den Vorteil, dass man sich nicht darum kümmern muss, Daten nach einer verpatzten Operation wieder zu löschen oder gar zu modifizieren. Das ist ein unsagbarer Vorteil, der vor allem im professionellen Einsatz wichtig ist, wo es nicht passieren darf, dass auf einmal Datensätze unmotiviert auftauchen und wieder verschwinden.

Im Falle von konkurrierenden (also gleichzeitig laufenden) Transaktionen stellt PostgreSQL sicher, dass ein User während einer gesamten Transaktion immer ein konsistentes Bild der Datenbank hat und niemand das Ergebnis »halbfertiger« Transaktionen lesen kann. Auf diese Weise werden Transaktionen voneinander unabhängig und der gesamte Verwaltungsaufwand wird an die Datenbank delegiert. Wie wir bereits erwähnt haben, sind Transaktionen fest im Kern von PostgreSQL integriert und der User muss lediglich dafür sorgen, dass die gewünschten Transaktionen gestartet und wieder beendet werden. Diese Arbeit kann nicht von der Datenbank erledigt werden, da diese nicht wissen kann, welche Daten in welcher Form zusammenzufassen sind.

Drei Operationen sind beim Arbeiten mit Transaktionen von Bedeutung:

- `BEGIN` dient zum Starten einer Transaktion.
- `COMMIT` schließt eine Transaktion ab und macht deren Änderungen wirksam.
- `ROLLBACK` sorgt dafür, dass die von der aktuellen, offenen Transaktion durchgeführten Änderungen rückgängig gemacht werden und die Transaktion beendet wird.

Alle drei Operationen können mittels einfacher SQL-Befehle einfach durchgeführt werden. Im Folgenden sehen wir uns kurz die Syntax der drei Befehle an. Beginnen wir mit dem `BEGIN`-Kommando:

```
buch=# \h BEGIN
Command:     BEGIN
Description: start a transaction block
Syntax:
BEGIN [ WORK | TRANSACTION ]
```

Drei Arten, die allesamt zum gleichen Resultat führen, werden unterstützt. Beim Arbeiten mit Transaktionen ist zu beachten, dass im Falle von PostgreSQL noch keine geschachtelten Transaktionen unterstützt werden:

```
buch=# BEGIN TRANSACTION;
BEGIN
buch=# BEGIN TRANSACTION;
NOTICE:  BEGIN: already a transaction in progress
BEGIN
```

Geschachtelte Transaktionen werden Bestandteil weiterer Entwicklungsarbeiten sein und PostgreSQL um weitere Features bereichern.

Um eine Transaktion abzuschließen, kann `COMMIT` verwendet werden. Hier ist eine kleine Übersicht über die Syntax des Befehles:

```
buch=# \h COMMIT
Command:     COMMIT
Description: commit the current transaction
Syntax:
COMMIT [ WORK | TRANSACTION ]
```

Wie bei `BEGIN` sind die Schlüsselwörter `WORK` und `TRANSACTION` optional und müssen daher nicht angegeben werden. Statt `COMMIT` kann auch `END` verwendet werden.

Das nächste Listing zeigt, wie die zuvor gestartete Transaktion wieder geschlossen werden kann:

```
buch=# COMMIT;
COMMIT
```

Wenn Sie während einer Transaktion erkennen, dass Operationen rückgängig gemacht werden müssen, können Sie `ROLLBACK` verwenden:

```
buch=# \h ROLLBACK
Command:     ROLLBACK
Description: abort the current transaction
Syntax:
ROLLBACK [ WORK | TRANSACTION ]
```

Beim Arbeiten mit Transaktionen wird sehr oft von ACID gesprochen. ACID ist ein Grundbegriff beim Umgang mit modernen Datenbanksystemen und sollte an dieser Stelle nicht unerwähnt bleiben.

ACID ist ein Kürzel für vier Begriffe der Transaktionskontrolle:

- Atomic: Eine Transaktion ist atomar, also unteilbar. Entweder führt die Datenbank alle oder keine Transaktion durch.
- Consistent: Es ist möglich, in sich konsistente Abbildungen eines Datenbestandes zu definieren.
- Isolated: Transaktionen dürfen nicht von nicht abgeschlossenen Transaktionen beeinflusst sein.
- Durable: Auch im Falle von Ausfällen müssen einmal beendete Transaktionen beendet bleiben.

Um das Arbeiten mit Transaktionen ein wenig zu veranschaulichen, wollen wir uns dem nächsten Beispiel zuwenden, das uns zeigt, was passiert, wenn Transaktionen explizit gestartet und beendet werden. Starten wir also eine Transaktion und legen eine temporäre Tabelle an:

```
buch=# BEGIN;
BEGIN
buch=# CREATE TEMPORARY TABLE interpret (id int4, name text);
CREATE
```

Im nächsten Schritt fügen wir einige Datensätze ein:

```
buch=# INSERT INTO interpret VALUES (1, 'Hansi Hinterseher');
INSERT 3041947 1
buch=# INSERT INTO interpret VALUES (2, 'Heino');
INSERT 3041948 1
```

Wie nicht anders zu erwarten ist, finden Sie die zwei Zeilen in der Tabelle wieder:

```
buch=# SELECT * FROM interpret;
 id |       name
----+------------------
  1 | Hansi Hinterseher
  2 | Heino
(2 rows)
```

Sollten die eingefügten Datensätze nicht stimmen oder ist eine andere Operation daneben gegangen, können Sie die Transaktion stoppen und alle Operationen rückgängig machen. Das kann mit einem einfachen ROLLBACK bewerkstelligt werden:

```
buch=# ROLLBACK;
ROLLBACK
```

Nach dem `ROLLBACK` gibt es die Tabelle nicht mehr und alle Daten sind automatisch vernichtet worden:

```
buch=# SELECT * FROM interpret;
ERROR:  Relation "interpret" does not exist
```

Dieses Beispiel verdeutlicht, wie Transaktionen funktionieren und wie mit ihnen gearbeitet werden kann.

Ein wesentlicher Punkt, der nicht oft genug hervorgehoben werden kann, ist, dass Änderungen, die innerhalb einer Transaktion durchgeführt werden, erst dann sichtbar werden, wenn der User die Transaktion beendet hat. Befindet sich also User A in einer Transaktion, die Daten einfügt, wird User B diese Daten erst sehen und verwenden können, wenn User A die Transaktion abgeschlossen hat. Auf diese Weise wird verhindert, dass User B Daten von unfertigen Transaktionen liest, die eventuell schon gar nicht mehr gültig sind. Dieses Verhalten der Datenbank impliziert, dass in der Regel jeder Benutzer auf einem anderen Schnappschuß der Datenbank operiert. Das ist notwendig, damit sich unfertige Transaktionen nicht in die Quere kommen. Auf den ersten Blick mag dieses Verhalten seltsam anmuten — bei etwas genauerem Hinsehen werden Sie erkennen, dass die Vorgehensweise von PostgreSQL sehr sinnvoll ist und hilft, viele Probleme des täglichen Arbeitens mit der Datenbank zu umgehen oder gänzlich zu lösen.

Ein wesentlicher Vorteil des Transaktionsmanagementes von PostgreSQL ist, dass sich die Entwickler streng an die ANSI-Vorgaben gehalten haben und Sie daher davon ausgehen können, dass sich auch andere Datenbanken wie PostgreSQL verhalten werden.

Nachdem Sie gesehen haben, wie eine Transaktion rückgängig gemacht werden kann, wollen wir nun auch in einem kurzen Beispiel zeigen, wie eine Transaktion erfolgreich abgeschlossen werden kann:

```
buch=# BEGIN;
BEGIN
buch=# CREATE TEMPORARY TABLE interpret (id int4, name text);
CREATE
buch=# INSERT INTO interpret VALUES (1, 'Rolling Stones');
INSERT 3041954 1
buch=# INSERT INTO interpret VALUES (2, 'U2');
INSERT 3041955 1
buch=# COMMIT;
COMMIT
buch=# SELECT * FROM interpret;
```

```
 id |      name
----+----------------
  1 | Rolling Stones
  2 | U2
(2 rows)
```

In diesem Fall sind die Daten auch nach dem Beenden der Transaktion noch verfügbar. Sofern Sie die Daten nicht in eine temporäre Tabelle einfügen, werden auch andere Benutzer auf den Datenbestand zugreifen können.

Beim Arbeiten mit Transaktionen sind einige Dinge von großer Bedeutung. Sobald ein Benutzer eine Transaktion startet, friert die Datenbank implizit den Datenbestand für den Benutzer ein. Das heißt, dass er nur mehr Modifikationen sieht, die er auch selbst vorgenommen hat. Das bedeutet, dass die Datenbank im Falle konkurrierender Transaktionen möglicherweise mehrere Versionen des gleichen Datenbestandes zu verwalten hat. Das ist ein wesentlicher Punkt, der auch erklärt, warum die intern benötigte Speichermenge im Falle von Transaktionen steigt. Wichtig zu wissen ist jedoch auch, dass PostgreSQL sicherstellt, dass Transaktionen zu keinem Datenverlust irgendwelcher Art führen; die Datenbank verwaltet alle intern vorhandenen Versionen des Datenbestandes automatisch, sodass sich der Benutzer darum keine Sorgen machen muss.

Nehmen wir an, Sie führen ein SQL Statement aus, das eine sehr große Tabelle liest. Während Ihre Abfrage die Tabelle liest, verändert ein anderer Prozess den Datenbestand. Was wird passieren? Wird Ihr Statement von anderen Prozessen beeinflusst und das Ergebnis gar falsch sein? Die Antwort auf diese Fragen ist natürlich Nein. Jede Abfrage wird automatisch in einer eigenen Transaktion ausgeführt. Jede der Operationen sieht einen eigenen, in sich konsistenten Schnappschuß der Daten. Das Ergebnis einer Abfrage ist daher richtig und andere Prozesse greifen nicht in eine laufende Abfrage ein. Das ist ein wichtiger Punkt, da so sichergestellt werden kann, dass es zu keinen falschen Abfragen kommt.

3.8.2 Transaktionen und Sequenzen

Wie bereits erwähnt, sieht ein Benutzer innerhalb einer Transaktion immer denselben, konsistenten Schnappschuss eines Datenbestandes. In der Regel ist dieser Schnappschuss von anderen Transaktionen unabhängig. Änderungen werden nur dann sichtbar, wenn andere Transaktionen bereits korrekt abgeschlossen worden sind. Im Falle von Sequenzen ist das nicht der Fall, da es dabei zu Problemen kommen würde. Per definitionem stellen Sequenzen eine durchgehende und eindeutige Nummerierung dar. Dieses Konzept bleibt beim Arbeiten mit Transaktionen jedoch nicht ohne Folgen wie uns das nächste Beispiel zeigt:

```
buch=# CREATE SEQUENCE test_seq;
CREATE
buch=# SELECT nextval('test_seq');
 nextval
---------
       1
(1 row)

buch=# BEGIN;
BEGIN
buch=# SELECT nextval('test_seq');
 nextval
---------
       2
(1 row)
```

Am Beginn des Beispiels wird eine Sequenz angelegt. Danach fragen wir den ersten Wert der Sequenz ab und starten eine Transaktionen. Innerhalb der Transaktion wird der nächste Wert in der Sequenz abgefragt. Bisher haben wir noch keine Unregelmäßigkeiten feststellen können. Was passiert jedoch, wenn wir ein `ROLLBACK` durchführen?

```
buch=# ROLLBACK;
ROLLBACK
buch=# SELECT nextval('test_seq');
 nextval
---------
       3
(1 row)
```

Obwohl wir alle Operationen, die wir innerhalb der Transaktion durchführten, rückgängig gemacht haben, hat die Sequenz nach dem `ROLLBACK` den höheren Wert. Das hängt damit zusammen, dass Sequenzen völlig unabhängig von Transaktionen verwendet werden können, da es sonst absolut unmöglich wäre, eine durchgehende Liste von Nummern zu erzeugen. Beim Arbeiten mit Sequenzen ist das unbedingt zu beachten.

3.8.3 Transaktionen und Zeitfunktionen

Wie das Arbeiten mit Sequenzen ist auch das Arbeiten mit Zeitfunktionen von großer Bedeutung. Dabei sind einige Dinge zu beachten, die von vielen Programmierern oft vergessen werden. Wenn man innerhalb einer Transaktion die Funktion `now()` aufruft, gibt diese nicht die aktuelle Zeit an, sondern sie retourniert die so genannte Transaction Time, was so viel bedeutet wie der Zeitpunkt, zu dem die Transaktion gestartet worden ist. Das ist ein wichtiger Punkt, da es mithilfe von

now() nicht möglich ist, den genauen Zeitpunkt einer Operation zu ermitteln. Das nächste Beispiel zeigt, wie sich PostgreSQL verhält:

```
buch=# BEGIN;
BEGIN
buch=# SELECT now();
               now
-------------------------------
 2002-04-01 19:40:37.019372+02
(1 row)

buch=# SELECT now();
               now
-------------------------------
 2002-04-01 19:40:37.019372+02
(1 row)

buch=# COMMIT;
COMMIT
buch=# SELECT now();
               now
-------------------------------
 2002-04-01 19:40:47.384518+02
(1 row)
```

Innerhalb der Transaktion wird immer derselbe Zeitstempel zurückgegeben. Sobald die Transaktion zu Ende ist, wird wieder die aktuelle Zeit retourniert, weil das SELECT-Statement selbst wieder eine eigene Transaktion ist.

3.9 Constraints

In diesem Kapitel haben wir uns bisher mit den Grundzügen von SQL beschäftigt. Sie haben gelernt, wie man einfache Abfragen schnell durchführen kann und wie Tabellen miteinander verknüpft werden können. In diesem Abschnitt werden wir uns ein wenig genauer mit Datenstrukturen und deren Möglichkeiten befassen.

3.9.1 Integrity Constraints

Bei sehr komplexen Datenbeständen kann das Arbeiten mit SQL jedoch schnell sehr komplex werden. Der Grund dafür ist, dass Tabellen miteinander verknüpft werden müssen. Wie kann es also bewerkstelligt werden, dass alle Tabellen konsistent sind und keine Probleme auftreten können? Die Antwort liefern so genannte Constraints. Constraints sind Attribute Ihrer Datenstruktur, die angeben, wie Ihr Datenbestand konsistent zu halten ist. Um das Problem ein wenig zu verdeutlichen, legen wir zwei Tabellen an:

```
buch=# CREATE TABLE firma (name text);
CREATE
buch=# INSERT INTO firma VALUES ('Cybertec');
INSERT 3041967 1
buch=# CREATE TABLE rechnung (firma_name text, kunde text,
betrag int4);
CREATE
buch=# INSERT INTO rechnung VALUES ('Cybertec', 'xy inc', 3000);
INSERT 3041973 1
```

Der Datenbestand enthält zwei Tabellen mit je einem Datensatz. Die Tabelle `firma` enthält eine Liste aller Firmen. Die zweite Tabelle enthält eine Liste aller Rechnungen, die diese Firmen an ihre Kunden verschickt haben. Wie Sie sehen, hat Cybertec genau eine Rechnung an die Firma xy verschickt. So weit so gut — versuchen wir nun also alle Firmen abzufragen, die Rechnungen verschickt haben, und in der Liste der Firmen aufscheinen:

```
buch=# SELECT firma.* FROM firma, rechnung WHERE
firma.name = rechnung.firma_name;
   name
-----------
 Cybertec
(1 row)
```

Dabei wird genau ein Datensätz zurückgegeben.

Was passiert nun, wenn die Firma Cybertec Ihren Namen wechselt? Sehen wir uns das kurz an:

```
buch=# UPDATE firma SET name = 'www.postgresql.at' WHERE
name = 'Cybertec';
UPDATE 1
buch=# SELECT * FROM firma ;
        name
--------------------
 www.postgresql.at
(1 row)
```

Natürlich ist es kein Problem, den Namen der Firma zu ändern. Wir versuchen nun, die Abfrage von vorher noch einmal durchzuführen:

```
buch=# SELECT firma.* FROM firma, rechnung WHERE
firma.name = rechnung.firma_name;
 name
------
(0 rows)
```

Wie erwartet wird die Abfrage keine Datensätze zurückgeben. Der Grund dafür liegt auf der Hand. Der neue Name der Firma kann in der Tabelle mit den Rechnungen nicht mehr gefunden werden. Natürlich hätten wir den Wert in der zweiten Tabelle auch verändern können, aber was passiert im Falle von zig Tabellen mit unzähligen Abhängigkeiten? Eine einfache `UPDATE`-Operation würde eine große Zahl weiterer Operationen nach sich ziehen, was denkbar ineffizient und unkomfortabel wäre. Constraints sind eine einfache Möglichkeit, dieses Problem elegant zu umschiffen.

Wir zeigen Ihnen, wie eine neue, verbesserte Datenbankstruktur erstellt werden kann:

```
1  DROP TABLE firma;
2  DROP TABLE rechnung;
3
4  CREATE TABLE firma (
5          name text,
6          PRIMARY KEY (name)
7  );
8
9  CREATE TABLE rechnung (
10         firma_name text REFERENCES firma(name),
11         kunde      text,
12         betrag     int4
13 );
```

Nach dem Löschen der alten Tabellen wird eine neue Tabelle angelegt. Diese Tabelle enthält nun einen so genannten Primary Key (Primärschlüssel). Ein Primärschlüssel wird immer dann verwendet, wenn eine Spalte herangezogen werden kann, die einen Datensatz eindeutig kennzeichnet und identifiziert. In vielen Fällen wird dafür eine Laufnummer verwendet. Primärschlüssel sind immer eindeutig und jeder Eintrag in der Spalte kann daher nur einmal vorkommen. Schlüssel sind seit jeher ein wichtiger Bestandteil relationaler Datenbanken und das kommt auch hier zum Tragen.

Nach dem Anlegen der ersten Tabelle wenden wir uns der zweiten Relation zu. Die Tabelle referenziert die Spalte `name` in der ersten Tabelle.

Nach dem Anlegen der Datenstruktur sehen wir, wie die Tabellen intern repräsentiert sind:

```
1  buch=# \d firma
2         Table "firma"
3   Column | Type | Modifiers
4  ────────+──────+───────────
5   name   | text | not null
6  Primary key: firma_pkey
```

```
Triggers: RI_ConstraintTrigger_3042019,
          RI_ConstraintTrigger_3042021

buch=# \d rechnung
         Table "rechnung"
   Column   |  Type   | Modifiers
------------+---------+-----------
 firma_name | text    |
 kunde      | text    |
 betrag     | integer |
Triggers: RI_ConstraintTrigger_3042017
```

In diesem Listing sind alle Constraints auf so genannten Triggern aufgebaut. Trigger sind eine Vorrichtung, um bestimmte Funktionen beim Eintreten von Ereignissen selbstständig auszuführen. Trigger werden uns noch den Rest des Buches begleiten und wir werden noch sehr genau darauf eingehen.

Nachdem wir die Datenstruktur angelegt haben, können wir erkennen, wie mit den Tabellen gearbeitet werden kann:

```
buch=# INSERT INTO firma VALUES ('Cybertec');
INSERT 3042023 1
buch=# INSERT INTO rechnung VALUES ('Cybertec', 'X Company', '4000');
INSERT 3042024 1
buch=# SELECT * FROM rechnung;
 firma_name |   kunde   | betrag
------------+-----------+--------
 Cybertec   | X Company |   4000
(1 row)
```

Nach dem Eingeben von Daten versuchen wir, den Datenbestand zu modifizieren:

```
buch=# UPDATE firma SET name = 'www.postgresql.at';
ERROR:  <unnamed> referential integrity violation - key in firma still
referenced from rechnung
buch=# UPDATE rechnung SET firma_name = 'www.postgresql.at';
ERROR:  <unnamed> referential integrity violation - key referenced
from rechnung not found in firma
```

Diese Operation wird fehlschlagen, weil die Integritätsbedingungen verletzt sind — eine Modifikation würde zu inkonsistenten Daten führen. Referenzen und Verweise auf andere Tabellen sind wichtige Features, die herangezogen werden können, um die Sicherheit Ihrer Daten zu gewährleisten.

Dabei ist es jedoch auf Basis der vorliegenden Konfiguration nicht möglich, den Datenbestand zu verändern, weil der so genannte Foreign Key alle Änderungen sofort blockiert. Um diese Sperre aufzuheben, muss PostgreSQL mitgeteilt werden, wie `UPDATE`- beziehungsweise `DELETE`-Operationen zu handhaben sind. Das

folgende Listing zeigt ein kurzes Beispiel (vergessen Sie nicht, die alte Version der Tabelle rechnung zu löschen, bevor Sie die neue Variante einspielen):

```
CREATE TABLE rechnung (
        firma_name text REFERENCES firma(name)
                        ON UPDATE CASCADE
                        ON DELETE SET NULL,
        kunde text,
        betrag int4
);

INSERT INTO firma VALUES ('Cybertec');
INSERT INTO rechnung VALUES ('Cybertec', 'X Company', 4000);
```

In diesem Fall wird die Spalte firma_name automatisch geändert, sobald sich rechnung ändert. Sofern ein Wert aus firma gelöscht wird, sorgt PostgreSQL dafür, dass der entsprechende Wert in rechnung auf NULL gesetzt wird. Auf diese Weise werden alle notwendigen Abhängigkeiten von der Datenbank selbst gemanagt.

Im folgenden Listing können Sie sehen, was passiert, wenn Werte geändert werden:

```
buch=# UPDATE firma SET name = 'www.postgresql.at';
UPDATE 1
buch=# SELECT * FROM rechnung;
    firma_name     |   kunde   | betrag
-------------------+-----------+--------
 www.postgresql.at | X Company |   4000
(1 row)
```

Wie versprochen hat PostgreSQL den Datensatz automatisch geändert. Sehen wir uns nun an, was passiert, wenn Werte gelöscht werden:

```
buch=# DELETE FROM firma;
DELETE 1
buch=# SELECT * FROM rechnung;
 firma_name |   kunde   | betrag
------------+-----------+--------
            | X Company |   4000
(1 row)
```

Wie Sie sicherlich nicht anders erwartet haben, ist die erste Spalte der Tabelle leer.

Zusätzlich zu CASCADE und SET NULL werden noch zwei weitere Möglichkeiten unterstützt:

`SET DEFAULT` sorgt dafür, dass einer Spalte der Default-Wert zugewiesen wird. Mithilfe von `NO ACTION` können Sie definieren, dass keine Operation durchgeführt werden soll. Diese Option führt zum gleichen Resultat wie das Weglassen von Constraints.

PostgreSQL bietet noch einige weitere Möglichkeiten, Constraints zu definieren. Da diese jedoch den Rahmen eines Programmierhandbuches sprengen würden, wenden wir uns nun anderen Constraints zu.

3.9.2 Beschränkung der Eingabedaten

Constraints, die zur Sicherung der Integrität der Daten dienen, sind nicht die einzigen, die von PostgreSQL zur Verfügung gestellt werden.

Eines der wichtigsten Constraints ist sicherlich das `CHECK`-Constraint. Es dient dazu, Bedingungen prüfen zu lassen. Das nächste Listing zeigt eine kleine Tabelle, in der `CHECK`-Constraints zum Tragen kommen:

```
1 CREATE TABLE cpu (
2         id              int4,
3         hersteller      text CHECK ( length(hersteller) >= 3 ),
4         mhz             int4 CHECK ( mhz > 0 )
5 );
```

Der Name des Herstellers der CPU muss in diesem Beispiel mindestens drei Zeichen lang sein. Zusätzlich kann eine CPU keine negative Taktfrequenz haben — beide Fälle werden mittels `CHECK`-Constraint abgefangen.

Im folgenden Listing können Sie die Funktionsfähigkeit der Constraints begutachten:

```
1 buch=# INSERT INTO cpu VALUES (1, 'AMD', '2100');
2 INSERT 3042256 1
3 buch=# INSERT INTO cpu VALUES (2, 'AM', '750');
4 ERROR:  ExecAppend: rejected due to CHECK constraint cpu_hersteller
5 buch=# INSERT INTO cpu VALUES (2, 'AM', '-750');
6 ERROR:  ExecAppend: rejected due to CHECK constraint cpu_mhz
```

Wie aus dem Listing hervorgeht, werden die letzten beiden SQL-Statements nicht ausgeführt, weil die Eingabedaten nicht erlaubt sind. Auf diese Weise ist es möglich, sehr genau zu definieren, welche Daten zugelassen und erwünscht sind.

Zwei Constraints, die ebenfalls von großer Bedeutung sind, sind `NOT NULL` und `UNIQUE`. `NOT NULL` stellt sicher, dass eine Spalte einen gültigen Wert enthalten muss. Mithilfe von `UNIQUE` können Sie PostgreSQL mitteilen, dass eine Spalte keine doppelten Einträge aufweisen darf.

Das folgende Beispiel zeigt eine Tabelle, die beide Constraints enthält:

```
CREATE TABLE blume (
        name    text NOT NULL,
        UNIQUE (name)
);
```

Um zu sehen, welchen Effekt die Constraints auf das Verhalten der Datenbank haben, können Sie einige SQL-Befehle absetzen:

```
buch=# INSERT INTO blume VALUES ('Rose');
INSERT 3042263 1
buch=# INSERT INTO blume VALUES (NULL);
ERROR:  ExecAppend: Fail to add null value in not null attribute name
buch=# INSERT INTO blume VALUES ('Rose');
ERROR:  Cannot insert a duplicate key into unique index blume_name_key
```

Die letzten beiden `INSERT`-Kommandos schlagen fehl, weil jeweils eines der beiden Kriterien nicht erfüllt ist.

3.10 Binäre Objekte

Wie viele andere hoch entwickelte Datenbanken auch, unterstützt PostgreSQL binäre Objekte, die direkt in der Datenbank abgelegt werden können. Im Vergleich zum Speichern von Daten im Filesystem kann das Ablegen der Daten in einer Datenbank einige Vorteile haben. Dadurch, dass Sie die ganze Datei und nicht den Dateinamen speichern, kann es bei richtiger Handhabung nicht passieren, dass Dateinamen und Dateien inkonsistent werden. Das ist ein großer Vorteil, dessen Bedeutung oft unterschätzt wird. Im Gegensatz dazu ist es schwierig, auf ein File zuzugreifen.

In diesem Abschnitt werden Sie erfahren, wie binäre Objekte mit PostgreSQL verarbeitet werden können.

3.10.1 Einfügen und Löschen von Dateien

Bevor wir Files in die Datenbank laden, legen wir eine Tabelle an, die wir zum Speichern von Informationen heranziehen werden:

```
CREATE TABLE  datei (
        name    text,
        id      oid
);
```

Binäre Objekte werden nicht direkt in einer Tabelle gespeichert — das wäre viel zu unflexibel und würde keinerlei Vorteile bringen. Im Falle von PostgreSQL enthält eine Tabelle nur Verweise auf ein Objekt. Jedes Objekt in PostgreSQL hat eine

so genannte Object Id. Das ist eine eindeutige Nummer eines Objektes in einer Datenbank. Durch das Ablegen der Object Id in einer Tabelle können Sie ein Objekt in der Datenbank wiederfinden und bearbeiten.

Sehen wir uns also an, wie ein File importiert werden kann:

```
buch=# SELECT lo_import('/etc/resolv.conf');
 lo_import
------------
    3042275
(1 row)
```

Der Befehl `lo_import` dient zum Importieren von Dateien in die Datenbank. Der Rückgabewert der Funktion enthält die Object Id des neu angelegten Objektes. Diese kann zur weiteren Verarbeitung herangezogen werden. Wichtig zu beachten ist, dass es in der Datenbank keine Dateinamen mehr gibt: Objekte werden einzig und alleine über deren Nummer referenziert. Das ist ein wesentlicher Punkt, den Anfänger gerne übersehen. Wenn Sie den Dateinamen weiterhin benötigen, ist es ratsam, ihn ebenfalls abzuspeichern, wie das im nächsten Beispiel gezeigt wird:

```
buch=# INSERT INTO datei VALUES ('/etc/resolv.conf',
lo_import('/etc/resolv.conf'));
INSERT 3042277 1
```

Der ausgeführte Befehl ist nicht kompliziert. In das erste Feld fügen wir den Namen der Datei ein. Das zweite Feld befüllen wir mit dem Rückgabewert von `lo_import`. Sehen wir uns also kurz an, welchen Datensatz die Tabelle enthält:

```
buch=# SELECT * FROM datei;
        name        |   id
--------------------+---------
 /etc/resolv.conf | 3042276
(1 row)
```

Bei diesem Import hat die Datei eine neue Object Id erhalten, da es sich intern um ein neues Objekt handelt.

Versuchen wir nun die Datei wieder zu exportieren:

```
buch=# SELECT lo_export(id, '/tmp/resolv.conf') FROM datei
WHERE id = 3042276;
 lo_export
------------
          1
(1 row)
```

Der Rückgabewert von `lo_export` ist 1, weil der Export funktioniert hat. Das nächste Listing enthält den Inhalt der exportierten Datei:

```
[hs@duron code_buch]$ cat /tmp/resolv.conf
nameserver 195.34.133.10
nameserver 194.152.178.1
nameserver 195.34.133.11
```

Dabei handelt es sich um eine normale ASCII-Datei, die eine Liste von DNS-Servern enthält.

Um zu sehen, ob die beiden Dateien wirklich gleich sind, können wir das UNIX-Kommando `diff` verwenden:

```
[hs@duron code_buch]$ diff /etc/resolv.conf /tmp/resolv.conf
```

Sofern `diff` kein Ergebnis liefert, sind die beiden Dateien identisch.

3.10.2 Löschen von Objekten

Um Objekte wieder aus der Datenbank zu entfernen, ist der Befehl `lo_unlink` zu verwenden. Das folgende Listing zeigt, wie eine BLOB entfernt werden kann:

```
buch=# BEGIN;
BEGIN
buch=# SELECT lo_unlink('3042276');
 lo_unlink
-----------
         1
(1 row)

buch=# COMMIT;
COMMIT
```

Zu beachten ist, dass der Befehl innerhalb einer Transaktion ausgeführt werden soll (das ist jedoch kein Zwang); anderenfalls wird die Operation nicht funktionieren. Das ist ein sehr wichtiger Punkt, der oft Kopfzerbrechen bereitet.

Sehen wir uns kurz an, was mit dem Inhalt der Tabelle `datei` passiert ist:

```
buch=# SELECT * FROM datei;
       name       |   id
------------------+---------
 /etc/resolv.conf | 3042276
(1 row)
```

Der Datensatz ist immer noch vorhanden und Sie müssen händisch dafür sorgen, dass der Eintrag gelöscht wird.

3.11 Fortgeschrittenes SQL

In den vorangegangenen Abschnitten haben wir uns intensiv mit grundsätzlichen SQL-Operationen beschäftigt. Für komplexere Anwendungen reichen diese Operationen jedoch nicht aus. Zu diesem Zwecke stellt PostgreSQL eine Vielzahl von hoch entwickelten Features zur Verfügung, die wir in diesem Abschnitt diskutieren werden. Sie werden sehr schnell erkennen, wie mächtig die Datenbank ist und welche Möglichkeiten sich auftun, wenn Sie tiefer in die Materie eintauchen.

3.11.1 Selektieren von Aggregatinformationen

Speziell im Umgang mit Data Warehouses sind Aggregatinformationen von großer Bedeutung. Bei fast allen Auswertungen ist es notwendig, Informationen zusammenzufassen und zu aggregieren. In vielen Fällen ist es zweckmäßig, bereits aggregierte Informationen weiter zu selektieren.

Um das Problem ein wenig zu veranschaulichen, legen wir eine Tabelle an und fügen einige Datensätze ein:

```
CREATE TABLE person (
        id      int4,
        name    text,
        geschl  char(1)
);

COPY person FROM stdin;
1       Paul    m
2       Peter   m
3       Karl    m
4       Jutta   f
\.
```

Ziel der nächsten Abfrage ist es, zu zählen, wie viele Männer und Frauen in der Tabelle zu finden sind. Die Abfrage kann dank `COUNT` und `GROUP BY` sehr leicht formuliert werden:

```
buch=# SELECT geschl, COUNT(*) FROM person GROUP BY geschl;
 geschl | count
--------+-------
 f      |     1
 m      |     3
(2 rows)
```

Um das Ergebnis der vorangegangenen Abfrage weiter zu reduzieren, kann eine so genannte `HAVING` Clause eingeführt werden, wie das im nächsten Beispiel gezeigt wird:

```
buch=# SELECT geschl, COUNT(*) FROM person GROUP BY geschl
HAVING COUNT(*) > 1;
 geschl | count
--------+-------
 m      |     3
(1 row)
```

In diesem Fall haben wir alle Gruppen selektiert, deren Teilpopulationen mehr als eine Person enthalten. Wichtig zu bemerken ist, dass die Aggregierungsfunktion in der `HAVING` Clause explizit anzuführen ist. Sehen wir uns die nächste Abfrage an:

```
buch=# SELECT geschl, COUNT(*) AS a FROM person GROUP BY geschl
HAVING a > 1;
ERROR:  Attribute 'a' not found
```

Der Versuch scheitert, weil der Alias `a` nicht gefunden werden kann. Eine Abfrage wie die soeben gezeigte wird von vielen Datenbanksystemen unterstützt. Im Falle von PostgreSQL ist das nicht so. Der Grund dafür ist in der ANSI SQL Spezifikation (ein etwa 400-seitiges, hochtechnisches Dokument) zu finden, die Aliase in der `HAVING` Clause nicht vorsieht. Aus diesem Grund wird es dieses Feature in PostgreSQL auch nie geben, solange das oberste Ziel des Projektes größtmögliche Kompatibilität mit ANSI SQL heißt. Das ist ein wichtiger Punkt und das Fehlen dieses Features sollte daher nicht als Manko, sondern als Treue zum Standard empfunden werden.

Sehr wohl unterstützt werden Subselects, die in einer `HAVING` Clause jederzeit verwendet werden können, wie das nächste Beispiel zeigt. Ziel der Abfrage ist, alle Datensätze zu zählen, deren Anzahl höher als die Zahl der Frauen ist:

```
SELECT geschl, COUNT(*)
        FROM person
        GROUP BY geschl
        HAVING COUNT(*) >
                (SELECT COUNT(*)
                        FROM person
                        WHERE geschl = 'f'
                )
```

Das Subselect berechnet die Zahl der Frauen in der Datenbank und gibt das Ergebnis an die `HAVING` Clause zurück.

Das Ergebnis der Abfrage ist nicht weiter verwunderlich:

```
 geschl | count
--------+-------
 m      |     3
(1 row)
```

3.11.2 Regular Expressions

Regular Expressions (Reguläre Ausdrücke) sind eine erprobte und uralte (für IT-Begriffe) Möglichkeit, Muster zu suchen und zu bearbeiten. Regular Expressions sind in POSIX 1003.2 definiert und zählen zu den mächtigsten Werkzeugen überhaupt. Für UNIX steht eine C-Bibliothek zur Verfügung, die alle Operationen durchführt. Diese Bibliothek zählt zu den genialsten Softwarekomponenten, die je entwickelt worden sind. Wie der Titel und die Einleitung bereits erahnen lassen, werden wir uns in diesem Abschnitt ein wenig näher mit Regular Expressions beschäftigen und sehen, wie sie eingesetzt werden können.

Zum Arbeiten mit Regular Expressions stellt PostgreSQL eine Liste von Operatoren und Funktionen zur Verfügung. Prinzipiell werden zwei Ansätze unterstützt: LIKE und POSIX Style Regular Expression; wenden wir uns zuerst den POSIX Style-Operatoren zu:

`~`	`case sensitives` (Groß- und Kleinschreibung wird berücksichtigt) Suchen	`~*`	`case insensitives` (Groß- und Kleinschreibung wird nicht berücksichtigt) Suchen
`!~`	Findet alle, die nicht auf das Muster passen (case sensitive)	`!~*`	Findet alle, die nicht auf das Muster passen (case insensitive)

Nach diesem kurzen Überblick ist es Zeit, sich einigen Beispielen zuzuwenden. Zu diesem Zweck legen wir eine Tabelle mit ein paar Datensätzen an:

```
CREATE TABLE person (
        id      int4,
        name    text
);

COPY person FROM stdin;
1       Klaus Rüdiger
2       Jochen Olaf
3       Detlef Heinrich
\.
```

Sofern die Tabelle `person` in Ihrer Datenbank bereits existiert, muss diese vorher gelöscht werden, um die neue Tabelle anzulegen.

Versuchen wir nun, einige Abfragen durchzuführen:

```
buch=# SELECT * FROM person WHERE name ~ 'd';
 id |     name
----+--------------
  1 | Klaus Rüdiger
(1 row)
```

```
buch=# SELECT * FROM person WHERE name ~* 'd';
 id |      name
----+-----------------
  1 | Klaus Rüdiger
  3 | Detlef Heinrich
(2 rows)
```

Im ersten Fall wird nur ein Datensatz gefunden, weil die Abfrage `case sensitive` formuliert ist. Das zweite Beispiel zeigt, wie die Groß- und Kleinschreibung ignoriert werden kann.

Um alle Datensätze zu finden, die nicht auf das Muster passen, ist jeweils ein `!` vor den Operator zu stellen:

```
buch=# SELECT * FROM person WHERE name !~ 'd';
 id |      name
----+-----------------
  2 | Jochen Olaf
  3 | Detlef Heinrich
(2 rows)

buch=# SELECT * FROM person WHERE name !~* 'd';
 id |    name
----+-------------
  2 | Jochen Olaf
(1 row)
```

Beim Arbeiten mit Regular Expressions geht es meist darum, komplexe Ausdrücke auszuwerten. Dafür werden in der Regel so genannte Wild Cards und Sonderzeichen verwendet, von denen es im Falle von Regular Expressions eine ganze Menge gibt.

Zwei dieser Sonderzeichen dienen zum Finden von Zeilenanfang und Zeilenende:

```
buch=# SELECT * FROM person WHERE name ~* '^d';
 id |      name
----+-----------------
  3 | Detlef Heinrich
(1 row)

buch=# SELECT * FROM person WHERE name ~* 'h$';
 id |      name
----+-----------------
  3 | Detlef Heinrich
(1 row)
```

Der Zirkumflex im ersten Beispiel dient dazu, den Beginn der Zeile zu finden. Es werden also alle Datensätze zurückgegeben, die mit einem »d« beginnen. Im zweiten Beispiel werden alle Namen gesucht, die ein »h« am Ende haben.

In vielen Fällen ist nicht ganz klar, wonach gesucht werden soll. Nehmen wir also an, Sie suchen nach einem Namen, der mit einem »k« beginnt und ein »r« enthält. Bei dieser Abfrage ist nicht klar, an welcher Stelle sich das »r« befindet; eine Abfrage könnte so aussehen:

```
buch=# SELECT * FROM person WHERE name ~* '^k.*r';
 id |     name
----+--------------
  1 | Klaus Rüdiger
(1 row)
```

Das »k« hat am Anfang des Namens zu stehen. Danach folgen beliebig viele beliebige Zeichen. Danach muss ein »r« zu finden sein. Bei Regular Expression passt ein ».« auf ein beliebiges Zeichen. Der Stern gibt an, dass das beliebige Zeichen mindestens nie vorkommen muss (also 0-mal oder öfter). Wenn Sie wissen, dass zwischen dem »k« und dem »r« mindestens ein Zeichen sein muss, wird die Abfrage so aussehen müssen:

```
buch=# SELECT * FROM person WHERE name ~* '^k.+r';
 id |     name
----+--------------
  1 | Klaus Rüdiger
(1 row)
```

Das »+« sorgt dafür, dass das beliebige Zeichen mindestens einmal auftreten muss. Wenn es notwendig ist, eine Mindestanzahl oder dergleichen festzulegen, so ist das auch möglich:

```
buch=# SELECT * FROM person WHERE name ~* '^k.{2,}r';
 id |     name
----+--------------
  1 | Klaus Rüdiger
(1 row)

buch=# SELECT * FROM person WHERE name ~* '^k.{,2}r';
 id | name
----+------
(0 rows)

buch=# SELECT * FROM person WHERE name ~* '^k.{2,4}r';
 id | name
----+------
(0 rows)
```

Das erste Beispiel zeigt, dass mindestens zwei Zeichen vor dem »r« auftreten müssen. Im zweiten Beispiel hingegen dürfen es maximal zwei Zeichen sein. Aus diesem Grund wird kein Datensatz gefunden, weil das »r« weiter hintem im Text zu finden ist. In Beispiel Nummer drei ist der Bereich auf zwei bis maximal vier Zeichen beschränkt.

In vielen Fällen ist es notwendig, Zeichenfolgen beliebig oft zu wiederholen. Zu diesem Zwecke können Klammern verwendet werden. Das folgende Beispiel zeigt, dass alle Personen gefunden werden, bei denen ein »k« von einem bis vier Ausdrücken in der Klammer gefolgt ist:

```
buch=# SELECT * FROM person WHERE name ~* '^k(la){1,4}';
 id |     name
----+--------------
  1 | Klaus Rüdiger
(1 row)
```

Da die Sequenz »la« genau einmal vorkommt, wird der Datensatz gefunden.

Wie wir bereits gesehen haben, ist der Punkt ein Symbol für ein beliebiges Zeichen. In manchen Fällen ist es aber notwendig, nur gewisse Zeichen zuzulassen. Um dieses Problem zu lösen, können eckige Klammern verwendet werden. Das folgende Beispiel zeigt, wie alle Namen gefunden werden können, die entweder mit »k« oder »d« beginnen:

```
buch=# SELECT * FROM person WHERE name ~* '^[KD]';
 id |      name
----+----------------
  1 | Klaus Rüdiger
  3 | Detlef Heinrich
(2 rows)
```

Um Zeichen auszuschließen, ist ein Zirkumflex anzuführen:

```
buch=# SELECT * FROM person WHERE name ~* '^[^KD]';
 id |    name
----+-------------
  2 | Jochen Olaf
(1 row)
```

In diesem Fall werden alle gefunden, die nicht mit »k« oder »d« beginnen. Aufmerksame Leser werden jetzt sofort sagen, dass dieses Problem auch anders gelöst werden kann:

```
buch=# SELECT * FROM person WHERE name !~* '^[KD]';
 id |    name
----+------------
  2 | Jochen Olaf
(1 row)
```

Selbstverständlich gibt es mehrere Möglichkeiten, ein und dasselbe Problem effizient zu lösen. Wir wollen ausdrücklich darauf hinweisen, da es im Falle von Regular Expressions keine alleinigen Lösungen gibt.

POSIX Style Regular Expressions bieten eine Vielzahl weiterer Möglichkeiten, die jedoch an dieser Stelle den Rahmen des Möglichen sprengen würden. Wenden wir uns nun also dem `LIKE`-Operator zu. Im Gegensatz zu POSIX Style Regular Expressions ist bei `LIKE` eine leicht andere Syntax zu verwenden. Welche Art von Regular Expressions Sie letztlich verwenden, bleibt ihnen überlassen. Sehen wir uns also kurz an, was mit `LIKE` gemacht werden kann:

```
buch=# SELECT 'Hello World' LIKE 'llo';
 ?column?
----------
 f
(1 row)
```

Im ersten Beispiel können Sie sehen, dass der reguläre Ausdruck nicht auf die Inputdaten passt, weil `LIKE` keine Substring-Suche durchführt. Um zu sehen, ob die Zeichenkette »llo« im zu prüfenden Wert enthalten ist, müssen wir vorne und hinten ein Prozentzeichen anhängen:

```
buch=# SELECT 'Hello World' LIKE '%llo%';
 ?column?
----------
 t
(1 row)
```

Prozentzeichen passen auf beliebig viele beliebige Zeichen. Sofern Sie nur ein einzelnes Zeichen nicht definieren wollen, können Sie Underscores verwenden. Das nächste Beispiel zeigt, dass der Ausdruck dann `wahr` zurückgibt, wenn genau zwei Zeichen vor dem »llo« zu finden sind:

```
buch=# SELECT 'Hello World' LIKE '__llo%';
 ?column?
----------
 t
(1 row)
```

In vielen Fällen ist es notwendig, Sonderzeichen zu maskieren. Im Falle von `LIKE` werden Sonderzeichen mittels Backslash maskiert. Nicht zu vergessen ist, dass

auch SQL als Sprache bereits Backslashes verwendet, um Zeichen zu maskieren. Wenn wir also einen Backslash ausgeben wollen, wird das etwa so aussehen:

```
buch=# SELECT 'This is a backslash: \\';
         ?column?
--------------------------
 This is a backslash: \
(1 row)
```

Also sind zwei Backslashes notwendig, um einen Backslash auszugeben. Wenn Sie jetzt nach einen Backslash suchen, müssen Sie vier Backslashes angeben. Der Grund dafür liegt auf der Hand: Ein Backslash wird verwendet, um den auszugebenden Backslash selbst zu markieren. Die anderen beiden Backslashes sind notwendig, um einen Backslash zu erzeugen, der für das Escapen des auszugebenden Backslashes notwendig ist; zusammen gibt das wie gesagt vier Backslashes und sieht dann etwa so aus:

```
buch=# SELECT 'This is a backslash: \\' LIKE '%\\\\';
 ?column?
----------
 t
(1 row)
```

Der `LIKE`-Ausdruck passt auf das Muster und wahr wird retourniert. Im Gegensatz zu POSIX Style Regular Expressions erlaubt es `LIKE`, das Fluchtsymbol selbst zu definieren. Diese Definition erfolgt mithilfe einer `ESCAPE` Clause. Das nächste Beispiel zeigt, wie »$« als Fluchtsymbol festgelegt werden kann:

```
buch=# SELECT 'This is a backslash: \\' LIKE '%$\\' ESCAPE '$';
 ?column?
----------
 t
(1 row)
```

In diesem Fall sind nur mehr zwei Backslashes notwendig, weil der Dollar als Fluchtsymbol herangezogen wird. Die anderen beiden Backslashes bleiben erhalten, weil einer der beiden ja von SQL selbst benötigt wird.

Das Gegenteil von `LIKE` ist `NOT LIKE`. `NOT LIKE` hat genau den gegenteiligen Rückgabewert und kann wie `LIKE` verwendet werden. Das folgende Listing enthält ein Beispiel:

```
buch=# SELECT 'This is a backslash: \\' NOT LIKE '%$\\' ESCAPE '$';
 ?column?
----------
 f
(1 row)
```

3.11.3 Vererbung

PostgreSQL ist eine objektrelationale Datenbank. Das bedeutet, dass die Funktionen der Datenbank über die Funktionen einer normalen relationalen Datenbank hinausgehen. Eines der bekanntesten Zusatzfeatures ist die Möglichkeit, Vererbung einzusetzen. Das Konzept der Vererbung wird Ihnen bereits von vielen objektorientierten Programmiersprachen bekannt sein. Im Falle von PostgreSQL bedeutet Vererbung, dass eine Tabelle Spalten an eine andere vererben kann. Die Kindtabelle enthält also alle Spalten der Elterntabellen sowie Ihre eigenen Spalten. Dieses Feature macht PostgreSQL sehr mächtig und erlaubt es, wesentlich komplexere und durchdachtere Funktionalitäten in eine Datenstruktur einzubauen.

Dieser Abschnitt soll Ihnen die Konzepte von PostgreSQL ein wenig näher bringen und versuchen, einige Lösungswege aufzuzeigen.

Um das Konzept der Vererbung zu verdeutlichen, sehen wir uns am besten ein kleines Beispiel an:

```
CREATE TABLE produkt (
        edvnr   int4,
        name    text
);

CREATE TABLE auto (
        hersteller      text,
        ps              int4
) INHERITS (produkt);
```

Die Tabelle `produkt` ist die Muttertabelle, die alle Informationen enthält, die über Produkte üblicherweise verfügbar sind. Da ein Auto ein spezielles Produkt ist, erbt es alle Eigenschaften (Spalten) von Produkten, verfügt aber auch über eigene Merkmale. Hier sehen wir, was die Datenbank über die beiden Tabellen zu sagen hat:

```
buch=# \d produkt
       Table "produkt"
 Column |  Type   | Modifiers
--------+---------+-----------
 edvnr  | integer |
 name   | text    |

buch=# \d auto
           Table "auto"
   Column   |  Type   | Modifiers
------------+---------+-----------
 edvnr      | integer |
 name       | text    |
 hersteller | text    |
 ps         | integer |
```

Aus der Datenstruktur selbst ist nicht zu erkennen, dass es sich bei der Tabelle `auto` um eine Tabelle handelt, die bereits von einer anderen Tabelle geerbt hat.

Versuchen wir, die Tabelle `produkt` zu erweitern:

```
buch=# ALTER TABLE produkt ADD COLUMN verfuegbar bool;
ALTER
```

Wichtig zu wissen ist, was passiert, wenn die Muttertabelle erweitert wird; wir zeigen Ihnen, welche Auswirkungen diese Änderungen auf die Tabelle `auto` haben:

```
buch=# \d auto
            Table "auto"
    Column   |  Type   | Modifiers
-------------+---------+-----------
 edvnr       | integer |
 name        | text    |
 hersteller | text    |
 ps          | integer |
 verfuegbar | boolean |
```

Wie nicht anders zu erwarten war, ist die Spalte auch in der Tochtertabelle enthalten — das ist ein wichtiger Punkt, der unbedingt bedacht werden muss.

Im nächsten Schritt wollen wir versuchen, herauszufinden, was passiert, wenn Daten aus vererbten Tabellen abgefragt werden. Zu diesem Zwecke fügen wir einige Datensätze ein:

```
buch=# INSERT INTO produkt VALUES ('234353', 'Pizzabrot', 'true');
INSERT 3042305 1
buch=# INSERT INTO auto VALUES ('234353', '206', 'Peugeot', 70,
'true');
INSERT 3042306 1
```

Wir haben in jede Tabelle je einen Datensatz eingefügt. Führen wir nun einige Abfragen durch:

```
buch=# SELECT * FROM produkt;
 edvnr  |   name    | verfuegbar
--------+-----------+------------
 234353 | Pizzabrot | t
 234353 | 206       | t
(2 rows)

buch=# SELECT * FROM auto;
 edvnr  | name | hersteller | ps | verfuegbar
--------+------+------------+----+------------
 234353 | 206  | Peugeot    | 70 | t
(1 row)
```

Wenn Daten aus der Muttertabelle abgefragt werden, bedeutet das, dass PostgreSQL alle Daten aus den Tochtertabellen mit berücksichtigt. Wird eine Tochtertabelle abgefragt, erscheinen die Daten in den Muttertabellen nicht. Das ist logisch, da Autos `produkte` sind, ein beliebiges Produkt jedoch kein Auto sein muss.

Im Gegensatz zu vielen anderen Produkten erlaubt PostgreSQL, dass eine Tabelle mehrere Eltern haben kann. In vielen objektorientierten Umgebungen ist das nicht möglich und explizit verboten. Im Falle von Datenbanken kann es jedoch sinnvoll sein, einer Tabelle mehrere Eltern zuzuordnen. Das nächste Beispiel zeigt schematisch, wie eine mehrfache Vererbung durchgeführt werden kann:

```
buch=# CREATE TABLE a (x int4);
CREATE
buch=# CREATE TABLE b (y int4);
CREATE
buch=# CREATE TABLE c (z int4) INHERITS (a, b);
CREATE
```

In diesem Beispiel hat die Tabelle `c` zwei Elterntabellen und enthält daher alle Spalten der beiden Tabellen, wie aus dem nächsten Listing eindeutig hervorgeht:

```
buch=# \d c
       Table "c"
 Column |  Type   | Modifiers
--------+---------+-----------
 x      | integer |
 y      | integer |
 z      | integer |
```

Beim Arbeiten mit vererbten Tabellen sind einige Besonderheiten zu beachten. Eine dieser zugegeben zahlreichen Besonderheiten ist für das tägliche Arbeiten von großer Bedeutung. Wenn Sie versuchen, eine Tabelle zu löschen, müssen Sie beachten, dass Sie keine Tabellen löschen können, die noch von anderen Tabellen benötigt werden. In unserem Beispiel wäre es also nicht möglich, die Tabelle `a` zu löschen, da diese von der Tabelle `c` benötigt wird:

```
buch=# DROP TABLE a;
ERROR:  Relation "c" inherits from "a"
```

Wenn Sie Tabellen löschen wollen, müssen Sie die Objekthierarchie von unten her auflösen, wie das im folgenden Listing gezeigt wird:

```
buch=# DROP TABLE c;
DROP
buch=# DROP TABLE a;
DROP
buch=# DROP TABLE b;
DROP
```

Die Tabelle `c` muss zuerst gelöscht werden, alle anderen Tabellen folgen.

3.11.4 Auto-Joins

Eine klassisches Beispiel in jedem Buch zum Thema SQL, das sich nicht gerade mit Grundlagen beschäftigt, ist ein so genannter Auto-Join. Im Gegensatz zu anderen nicht so hoch entwickelten Open Source-Datenbanken unterstützt PostgreSQL so genannte Aliase. Diese Aliase können verwendet werden, um Tabellen mit sich selbst zu joinen.

Werfen wir also einen Blick auf eine solche Tabelle:

```
1 CREATE TABLE gruppe (
2         gruppe  text,
3         person  text
4 );
5
6 COPY gruppe FROM stdin;
7 sparverein      Paul
8 sparverein      Karl
9 kegelrunde      Paul
10 kegelrunde     elise
11 \.
```

Ziel ist es, alle Personen abzufragen, die in der Kegelrunde und beim Sparverein sind. Da die einzelnen Vereine nicht durch Spalten getrennt, sondern untereinander stehen, ist diese Abfrage nicht ganz trivial.

Sehen wir uns also an, wie das Problem gelöst werden kann:

```
1 SELECT a.person
2         FROM gruppe AS a, gruppe AS b
3         WHERE a.gruppe = 'sparverein'
4                 AND b.gruppe = 'kegelrunde'
5                 AND a.person = b.person;
```

Um die Tabelle mit sich selbst zu joinen, wird ein Alias auf die Tabelle gesetzt. Dieselbe Tabelle ist nun doppelt verfügbar und die beiden Erscheinungen der Tabelle können behandelt werden, als ob es sich um zwei eigenständige Relationen handelte. Diese beiden Tabellen werden nun gejoint. Da wir alle Personen suchen, die in beiden Tabellen vorkommen (wir betrachten eine Tabelle als die, die die Kegelrunde enthält; die andere wird die Mitglieder des Sparvereines enthalten), ist die Join-Bedingung leicht zu finden.

Wir speichern den SQL-Code im File `b.sql` und senden ihn an die Datenbank, um das Ergebnis zu finden:

```
[hs@duron code_buch]$ psql buch < b.sql
 person
────────
 Paul
(1 row)
```

In unserem Beispiel ist nur Paul bei beiden Vereinen.

Auto-Joins sind ein wichtiges Feature und vor allem bei sehr komplexen Abfragen und Datenstrukturen von großem Nutzen.

3.11.5 LEFT und RIGHT Joins

Bisher haben Sie gesehen, wie Tabellen miteinader gejoint werden können. Dabei haben wir es der Datenbank überlassen, in welcher Reihenfolge die Tabellen miteinander verbunden werden. Außerdem haben wir immer nur Datensätze gesucht, die in beiden Tabellen vorkommen. Da das in vielen Fällen nicht ausreicht, um das gewünschte Ergebnis schnell zu finden, stellt PostgreSQL eine Reihe weiterer Möglichkeiten zur Verfügung, die wir in diesem Kapitel näher betrachten werden.

In diesem Abschnitt werden wir folgende Daten benutzen:

```
CREATE TABLE sportart (
        name    text
);

CREATE TABLE reporter (
        name text
);


CREATE TABLE zustaendig (
        reporter        text,
        sportart        text
);

COPY sportart FROM stdin;
boxen
skaten
radfahren
tennis
\.

COPY reporter FROM stdin;
jens
olaf
karl
```

```
erich
\.

COPY zustaendig FROM stdin;
jens    boxen
jens    skaten
karl    radfahren
\.
```

Die Datenstruktur besteht aus drei Tabellen. Die erste Tabelle enthält eine Liste von Sportarten. In der zweiten Tabelle findet sich eine Liste von Reportern. Die letzte Tabelle sagt uns, welcher Reporter von welchen Sportarten berichtet. Bisher haben wir gelernt, wie es beispielsweise möglich ist, alle Reporter und deren Sportart zu finden:

```
buch=# SELECT * FROM zustaendig;
 reporter | sportart
----------+----------
 jens     | boxen
 jens     | skaten
 karl     | radfahren
(3 rows)
```

Eine Liste von Personen und Sportarten wird ausgegeben.

Wenn Sie zusätzlich prüfen wollen, ob die genannten Reporter und Sportarten auch in den jeweiligen Basistabellen vorhanden sind, können Sie einen Join durchführen:

```
buch=# SELECT zustaendig.* FROM zustaendig, sportart, reporter
WHERE zustaendig.reporter = reporter.name AND zustaendig.sportart =
sportart.name;
 reporter | sportart
----------+----------
 jens     | boxen
 jens     | skaten
 karl     | radfahren
(3 rows)
```

Bitte beachten Sie, dass solche Checks üblicherweise automatisch von Constraints durchgeführt werden, die wir hier jedoch absichtlich nicht anwenden.

Versuchen wir nun eine Abfrage zu schreiben, die alle Sportarten und zusätzlich die Namen der Reporter ausgibt. Wir wollen alle Sportarten ausgeben — auch wenn es dazu keinen Reporter gibt. Dieses Problem kann mithilfe eines expliziten Joins gelöst werden:

```
SELECT sportart.name, zustaendig.reporter
        FROM sportart LEFT JOIN zustaendig
                ON (sportart.name = zustaendig.sportart);
```

Wir führen einen so genannten `LEFT JOIN` durch. Das bedeutet, dass alle Zeilen aus der linken Tabelle verwendet werden — egal, ob sie die Join-Bedingung erfüllen oder nicht. Aus der rechten Tabelle werden nur Datensätze aufgelistet, die die Bedingung erfüllen. Das Ergebnis enthält die gewünschten Zeilen:

```
   name    | reporter
-----------+----------
 boxen     | jens
 radfahren | karl
 skaten    | jens
 tennis    |
(4 rows)
```

Das Gegenteil von `LEFT` Joins sind so genannte `RIGHT` Joins. In diesem Falle werden alle Datensätze genommen, die in der rechten Tabelle vorkommen. Hier ein Beispiel:

```
SELECT sportart.name, zustaendig.reporter
        FROM zustaendig RIGHT JOIN sportart
                ON (sportart.name = zustaendig.sportart);
```

Dabei haben wir die Reihenfolge der Tabellen geändert und eine `RIGHT` Join durchgeführt. Das Ergebnis ist das gleiche wie zuvor.

3.11.6 Versteckte Spalten

Bisher haben wir angenommen, dass der *-Operator alle Spalten in einer Tabelle auflistet. Diese Feststellung ist jedoch nur bedingt richtig. In jeder Tabelle gibt es zusätzlich zu den definierten Spalten noch einige versteckte Felder, die in bestimmten Fällen herangezogen werden können. Hier ist ein kleines Beispiel:

```
buch=# SELECT oid, xmin, * FROM sportart WHERE name = 'boxen';
   oid   | xmin | name
---------+------+-------
 3042347 | 1822 | boxen
(1 row)
```

Wie Sie erkennen, sind diese Spalten explizit anzuführen.

Hier eine Aufzählung aller versteckten Spalten:

`oid` Die Object Id ist eine eindeutige Nummer einer jeden Zeile in einer Datenbank (das Oracle Pendant heißt `rowid`).

`tableoid` Enthält die eindeutige Nummer einer Tabelle

`xmin` Diese Spalte enthält die Nummer der Transaktion, die die Zeile eingefügt hat.

`cmin` Enthält die Befehlsnummer innerhalb einer Transaktion

`xmax` Enthält die Nummer der Transaktion, die einen Datensatz gelöscht hat. Im Falle vom ungelöschten Datensätzen enthält der Wert 0.

`cmax` Die Befehlsnummer innerhalb der löschenden Transaktion

`ctid` Enthält die Id eines Tuples innerhalb einer Tabelle. Diese Nummer kann im Falle von `VACUUM` wechseln. Die Nummer besteht aus der Nummer des Blockes sowie der Position innerhalb des Blockes.

In der Regel werden die versteckten Spalten einer Tabelle nicht verwendet; es kann jedoch in sehr speziellen Fällen sinnvoll sein, auf diese Informationen zurückzugreifen.

Kapitel 4

Administration und Tuning

Administration und Tuning sind zwei wesentliche Bestandteile im Leben eines jeden Datenbankanwenders. Auch wenn Sie selbst nicht mit der Administration eines Systemes betraut sind, ist es dennoch notwendig, die Grundzüge desselben zu kennen und in Ansätzen zu verstehen.

Dieses Kapitel soll Ihnen einen Überblick über die Administration von PostgreSQL geben und ihnen zeigen, welche Möglichkeiten die Datenbank derzeit bietet.

4.1 Laufzeitparameter

Die meisten der von PostgreSQL zur Verfügung gestellten Optionen und Parameter können zur Laufzeit verändert werden. Das ist ganz besonders wichtig, weil es ihnen dadurch möglich wird, Eingriffe in der Konfiguration Ihrer Datenbank vorzunehmen, ohne den Betrieb des Systems zu gefährden.

Weiterhin erlaubt es PostgreSQL, gewisse Parameter lokal zu setzen. Das hat den Vorteil, dass Sie bestimmte Abfragen besser tunen können und die Performance der Datenbank dadurch erhöht werden kann. Auch damit werden wir uns in diesem Kapitel beschäftigen.

4.1.1 Übersicht über die wichtigsten Parameter

Seit PostgreSQL 7.1 werden alle Parameter in einer großen Konfigurationsdatei namens `postgresql.conf` zusammengefasst. Das erleichtert es ungemein, die Datenbank zu verwalten und alle Settings im Überblick zu behalten. Die Datei `postgresql.conf` ist in Ihrem Datenbankcluster zu finden und kann leicht mithilfe eines UNIX-Editors (etwa vi) modifiziert werden.

Das nächste Listing enthält eine kurze Beschreibung der wichtigsten Konfigurationsmöglichkeiten in der Reihenfolge, in der sie in der Konfigurationsdatei vorkommen:

`tcpip_socket`: Schaltet TCP/IP ein. Wenn dieses Flag ausgeschaltet ist, werden nur UNIX-Sockets akzeptiert.

`ssl`: Schaltet verschlüsselte Verbindungen ein.

`max_connections`: Definiert die Zahl der simultan erlaubten Backend-Prozesse beziehungsweise Verbindungen zur Datenbank. Wenn dieser Wert erhöht wird, ist auch die Zahl der Shared Buffers zu erhöhen (mind. 2 Buffer pro Backend-Prozess).

`port`: Definiert den Port, auf den PostgreSQL hört.

`hostname_lookup`: Legt fest, ob IP-Adressen aufgelöst werden sollen.

`show_source_port`: Zeigt den Ausgangsport des Rechners an, der versucht, eine Verbindung zu PostgreSQL aufzubauen.

`unix_socket_directory`: Definiert das Verzeichnis für die Verwaltung der UNIX-Sockets. Standardmäßig wird `/tmp` verwendet.

`unix_socket_group`: Definiert die Gruppe, die die Sockets besitzt.

`unix_socket_permissions`: Legt die Benutzerrechte für die Sockets fest.

`virtual_host`: Arbeitet wie die von Apache bekannte Direktive.

`krb_server_keyfile`: Legt die Position der Datei mit den Schlüsseln fest, die für Kerberos benötigt wird.

`shared_buffers`: Legt die Zahl der Shared Buffers fest (ein Buffer ist 8 kb groß).

`max_locks_per_transaction`: Legt fest, wie viele Locks einer Transaktion erlaubt sind.

`wal_buffers`: Definiert die Zahl der Buffer, die für WALs (Write Ahead Logs) von PostgreSQL verwendet werden dürfen.

`sort_mem`: Definiert den maximalen für Sorts herangezogenen Arbeitsspeicher. Sind mehr Daten zu sortieren, als im Hauptspeicher sortiert werden können beziehungsweise erlaubt sind, werden temporäre Dateien angelegt. Der erlaubte Hauptspeicher wird in Kilobytes angegeben. Bedenken Sie, dass die Grenze pro Sort-Prozess festgelegt wird — im Fall mehrerer konkurrierender Sorts wird die durch diesen Parameter festgelegte Speichermenge möglicherweise mehrmals belegt.

`vacuum_mem`: Legt fest, wie viel Hauptspeicher von `VACUUM` verwendet werden darf.

`wal_files`: Legt die Zahl der für WAL-Logs verwendeten Dateien fest.

`wal_sync_method`: Legt fest, ob Daten, die WAL-Logs betreffen, sofort auf die Platte geschrieben werden oder nicht.

`commit_delay`: Um die Performance der Datenbank zu steigern, kann es sinnvoll sein, mehrere Transaktionen auf einmal auf die Festplatte zu schreiben. Mithilfe von `commit_delay` können Sie festlegen, wie lange auf weitere Transaktionen gewartet werden soll, bevor ein Schreibvorgang beginnt.

`fsync`: Um die Performance von Systemen zu erhöhen, kann es sinnvoll sein, die Ausgabe zu buffern. Derartige Dinge werden in der Regel vom Kernel vorgenommen. Sofern `fysnc` eingeschaltet wird, kann das Betriebssystem jedoch aufgefordert werden, Daten sofort auf die Festplatte zu schreiben; das erhöht die Sicherheit der Daten, kostet aber geringfügig Performance (in etwa 1–4 %).

`enable_seqscan`: Wenn die Datenbank einen so genannten Sequential Scan (oder Full Table Scan) durchführt, bedeutet das, dass eine Tabelle in ihrer Gesamtheit gelesen wird. Sequential Scans können auch vorkommen, wenn Indices definiert sind. Sofern Sequential Scans abgeschaltet sind, wird die Datenbank angewiesen, einen Index zu verwenden, wenn ein passender definiert ist.

`enable_indexscan`: Verbietet oder ermöglicht die Verwendung von Index Scans (wenn möglich).

`enable_tidscan`: Verbietet oder ermöglicht die Verwendung von Tidscans (wenn möglich).

`enable_sort`: Verbietet oder erlaubt die Sortierung von Daten (wenn möglich).

`enable_nestloop`: Schleifen innerhalb von Schleifen werden als Nested Loops bezeichnet, die ebenfalls verboten oder erlaubt sein können.

`enable_mergejoin`: Verbietet oder erlaubt die Verwendung von Merge Joins (wenn möglich).

`enable_hashjoin`: Verbietet oder erlaubt die Verwendung von Hash Joins (wenn möglich).

`ksqo`: Der Key Set Query Optimizer sorgt dafür, dass AND- und OR-Operationen so behandelt werden, wie in MS Access. Dieses Verhalten soll möglichst abgeschaltet werden.

`effective_cache_size`: Legt die Anzahl der 8 kb-Blöcke fest, die als Cache verwendet werden.

`random_page_cost`: Legt die vom Optimizer vergebenen Strafpunkte für das Lesen einer Page fest.

`cpu_tuple_cost`: Legt die vom Optimizer vergebenen Strafpunkte für das Verarbeiten eines Tuples fest.

`random_index_tuple_cost`: Legt die vom Optimizer vergebenen Strafpunkte für das Verarbeitens eines Tuples mithilfe eines Indexes fest.

`cpu_operator_cost`: Legt die vom Optimizer vergebenen Strafpunkte für das Verarbeiten eines Operators fest.

`geqo`: Schaltet den Genetic Query Optimizer ein beziehungsweise aus. Diese Art der Optimierung wird verwendet, um Abfragen abzuarbeiten, bei denen das Aufbauen des Execution-Plans zu lange dauern würde, wenn PostgreSQL alle Möglichkeiten der Verarbeitung evaluieren würde. Um GEQO zu konfigurieren, stellt PostgreSQL einige Parameter zur Verfügung, die jedoch in der Regel allesamt nicht praxisrelevant sind.

`silent_mode`: Dieser Parameter ist äquivalent mit dem `-S`-Flag des Postmasters, das dafür sorgt, dass keine Informationen auf Stdout und Stderr geschrieben wird.

`log_connections`: Stellt sicher, dass jede Verbindung, die zur Datenbank aufgebaut wird, im Logfile erscheint. Das Setzen dieses Flags ist beim Debuggen von Anwendungen sehr ratsam.

`log_timestamp`: Fügt einen Zeitstempel an jede Zeile des Logfiles an.

`log_pid`: Fügt die Nummer des Prozesses, der die Logginginformationen generiert hat, in das Logfile ein.

`debug_level`: Definiert die Menge der generierten Logginginformation. Für den praktischen Betrieb ist jedes Level, das höher oder gleich vier ist, de facto nutzlos.

`debug_print_query`: Schreibt alle an PostgreSQL gesendeten Abfragen in das Logfile.

`debug_print_parse`: Schreibt den Output des Parsers in das Logfile.

`debug_print_rewritten`: Bevor eine Abfrage ausgeführt wird, wird sie neu geschrieben und reformuliert. Das Ergebnis dieses Neuschreibens kann ebenfalls geloggt werden.

`debug_print_plan`: Sorgt dafür, dass der Execution-Plan ins Logfile geschrieben wird.

`debug_pretty_print`: Formatiert die SQL-Abfragen bei der Ausgabe.

`syslog`: `Syslog` ist ein Logging-Tool des Betriebssystems. PostgreSQL unterstützt mehrere `Syslog`-Einstellungen. 0 bedeutet, dass `Syslog` ausgeschaltet ist, 1 bedeutet, dass Nachrichten an `Standard Output` und `Syslog` geschickt werden. Sofern der Parameter auf 2 gesetzt ist, wird nur mehr `Syslog` verwendet.

`show_parser_stats`: Diese Option ermöglicht es, interne Statistiken über den Parser in das Logfile zu schreiben.

`show_planner_stats`: Diese Option ermöglicht es, interne Statistiken über den Planner in das Logfile zu schreiben.

`show_executor_stats`: Diese Option ermöglicht es, interne Statistiken über den Exekutor in das Logfile zu schreiben.

`show_query_stats`: Diese Option ermöglicht es, interne Statistiken über die aktuelle Abfrage in das Logfile zu schreiben.

`show_btree_build_stats`: Diese Option ermöglicht es, interne Statistiken über die Generierung eines Baumes (Index) zu loggen. PostgreSQL erlaubt auch

noch die Erstellung beziehungsweise die Anzeige weiterer statistischer Informationen, die jedoch in der Regel nicht praxisrelevant sind.

`trace_notify`: Erzeugt jede Menge logging-Informationen für die Befehle `LISTEN` und `NOTIFY`.

`trace_locks`: Verfolgt Locks.

`dynamic_library_path`: Setzt den Pfad für dynamisch geladene Bibliotheken.

`australian_timezones`: Das Kürzel EST kann für zwei Zeitzonen verwendet werden. Üblicherweise bedeuted EST Eastern Standard Time. Sofern das `Flag` auf True gesetzt ist, wird EST jedoch als australische Zeitzone interpretiert, deren Kürzel ebenfalls EST ist.

`authentiation_timeout`: Sorgt dafür, dass ein Authentifizierungsprozess nach einer bestimmten Zeit abgeschlossen sein muss.

`default_transaction_isolation`: Zur Abgrenzung von Transaktionen unterstützt PostgreSQL zwei Verfahren (Read Committed und Serializable). Standardmäßig ist dieser Wert auf »read committed« gesetzt.

`max_expr_depth`: Setzt die maximale Tiefe von Ausdrücken fest, die vom Parser akzeptiert werden. Auf diese Weise können unendlich tiefe Ausdrücke verhindert werden (etwa bei unendlichen Rekursionen).

`max_files_per_process`: Legt die maximale Anzahl von Dateien fest, auf die ein Prozess zugreifen darf.

`sql_inheritance`: Definiert, ob Vererbung eingeschaltet wird oder nicht

`transfor_null_equals`: Legt fest, ob »`x = NULL`« akzeptiert wird oder als »`x IS NULL`« formuliert werden muss.

Alle soeben erklärten Parameter gelten für einen gesamten Datenbankcluster.

4.1.2 Änderungen zur Laufzeit

PostgreSQL ist eine sehr flexible Datenbank und es ist daher möglich, zur Laufzeit gewisse Variablen umzudefinieren. Das hat den Vorteil, dass man bestimmte Settings sehr fein bestimmen und die Datenbank besser und dynamisch an die jeweiligen Bedürfnisse angepasst werden kann.

Zum Ändern und Bestimmen von Parametern zur Laufzeit werden zwei Befehle verwendet. `SHOW` dient zum Anzeigen von Parametern:

```
buch=# \h SHOW
Command:     SHOW
Description: show the value of a run-time parameter
Syntax:
SHOW name
```

Der Befehl `SET` dient zum Setzen eines Parameters:

```
buch=# \h SET
Command:     SET
Description: change a run-time parameter
Syntax:
SET variable { TO | = } { value | 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
```

Hier sehen wir, wie derartige Änderungen durchgeführt werden können:

```
buch=# SHOW vacuum_mem;
NOTICE:  vacuum_mem is 8192
SHOW VARIABLE
buch=# SET vacuum_mem TO 16384;
SET VARIABLE
buch=# SHOW vacuum_mem;
NOTICE:  vacuum_mem is 16384
SHOW VARIABLE
```

In diesem Fall erlauben wir es `VACUUM`, mehr Hauptspeicher zu verwenden. Das nächste Listing zeigt, wie wir den Wert wieder auf den in der Konfigurationsdatei festgelegten Wert setzen können:

```
buch=# SET vacuum_mem TO DEFAULT;
SET VARIABLE
buch=# SHOW vacuum_mem;
NOTICE:  vacuum_mem is 8192
SHOW VARIABLE
```

4.2 Netzwerkkonfiguration

Um Zugriffe über das Netzwerk zu steuern, bietet PostgreSQL einige Möglichkeiten, die in der Datei `pg_hba.conf` zusammengefasst sind. Mit Hilfe von `pg_hba.conf` können Sie genau festlegen, welche Rechner auf welche Datenbanken zugreifen dürfen und auf welche Art und Weise die Authentifizierung erfolgen soll. Die Konfiguration ist sehr einfach, weil `pg_hba.conf` großteils aus einer Dokumentation besteht, bei der alle von PostgreSQL zur Verfügung gestellten Konfigurationsmöglichkeiten ausführlich beschrieben werden.

Sehen wir uns also einen Auszug aus der Datei an:

```
# CAUTION: if you are on a multiple-user machine, the default
# configuration is probably too liberal for you. Change it to use
# something other than "trust" authentication.
#
# TYPE      DATABASE    IP_ADDRESS    MASK              AUTH_TYPE
# AUTH_ARGUMENT
local       all                                       trust
host        all         127.0.0.1     255.255.255.255   trust
```

In dieser Datei ist dem für uns relevanten Teil auch gleich eine Warnung vorangestellt, die darauf aufmerksam macht, dass die Standardkonfiguration eventuell zu liberal sein könnte. Gehen wir die Standardkonfiguration kurz durch: Die vorletzte Zeile legt fest, dass alle lokalen Verbindungen zu allen Datenbanken zugelassen sind. Unter lokalen Verbindungen verstehen wir Verbindungen, die via UNIX-Sockets erstellt werden; TCP/IP-Verbindungen sind hier ausdrücklich ausgeschlossen. Die Authentifizierungsmethode ist auf `trust` gesetzt. `trust` bedeutet, dass keine Authentifizierung notwendig ist. Diese Konfiguration ist sehr liberal und sollte in Produktionssystemen nicht eingesetzt werden. Wichtig zu beachten ist, dass nicht alle User Zugang zur Datenbank haben — der vertrauliche Zugang gilt nur für Benutzer, die im System vorhanden sind, wie das nächste Beispiel zeigt:

```
[root@notebook hs]# psql -U xy template1
psql: FATAL 1:  user "xy" does not exist
```

Der Benutzer `xy` darf sich nicht zur Datenbank `template1` verbinden, weil er nicht existiert. Im Gegensatz dazu wäre es aber dem User `postgres` erlaubt, eine Verbindung aufzubauen, weil dieser in der Regel existiert.

Die letzte Zeile der Konfiguration legt fest, dass alle Anfragen, die über das Loopback Interface erstellt werden sollen, ebenfalls auf `trust` gesetzt sind — solche Anfragen werden daher, was die Authentifizierung betrifft, genauso behandelt wie Verbindungen, die via UNIX-Sockets erstellt werden.

Folgende Authentifizierungsmethoden werden unterstützt:

`trust`: Keine Authentifizierung.

`password`: Das vom Benutzer unverschlüsselt an PostgreSQL gesendete Passwort wird mit den Einträgen in der Systemtabelle `pg_shadow` verglichen. Im Falle eines gefundenen korrekten Eintrages wird dem Benutzer der Zugriff erlaubt. Sofern als letzter Parameter in `pg_hba.conf` eine Datei angegeben wird, wird diese statt `pg_shadow` verwendet. Passwortdateien können mit dem Programm `pg_passwd` administriert werden.

`md5`: `md5` ist äquivalent zu `password`, sorgt jedoch dafür, dass verschlüsselte Passwörter übertragen werden. Sollten Sie eine Passwortdatei verwenden,

prüft `md5` nur die Benutzer, nicht jedoch die Passwörter, diese werden mithilfe von `pg_shadow` geprüft.

`crypt`: `crypt` funktioniert wie `md5`, unterstützt jedoch keine verschlüsselten Passwörter in `pg_shadow`.

`ident`: Im Falle einer TCP/IP-Verbindung können identische Verbindungen verwendet werden, um den zu benutzenden Benutzernamen direkt vom identischen Server abzufragen. Diese Vorgehensweise ist nur bei lokalen Systemen ratsam, die nicht sicherheitskritisch sind.

`krb4`: Authentifizierung mittels Kerberos IV.

`krb5`: Authentifizierung mittels Kerberos V.

`pam`: Erlaubt Authentifizierung mittels Pam.

`reject`: Untersagt eine Verbindung.

Bei dieser Beispielkonfiguration wird uns gezeigt, wie eine PostgreSQL-Datenbank konfiguriert werden kann:

```
1 local      all                                          trust
2 host       all         127.0.0.1     255.255.255.255    trust
3 host       template1   192.168.1.23  255.255.255.255    reject
4 host       template1   192.168.1.0   255.255.255.0      md5
5 host       buch        62.11.45.54   255.255.255.255    krb5
```

Die ersten beiden Zeilen der Konfiguration sind unverändert. In Zeile drei schließen wir den Rechner 192.168.1.23 explizit aus. Obwohl wir in der darauf folgenden Zeile den gesamten IP Range erlauben, bleibt der Rechner ausgeschlossen, da immer nur das Anwendung findet was zuerst festgelegt wurde. In der letzten Zeile erlauben wir es einem einzelnen Rechner, auf die Datenbank `buch` zuzugreifen. Die Authentifizierung erfolgt via Kerberos V. Beachten, dass eine PostgreSQL-Installation immer nur Kerberos IV oder Kerberos V unterstützt — es kann immer nur eine der beiden Versionen eingesetzt werden.

4.2.1 SSL

Im Falle von sicherheitskritischen Installationen kann es sinnvoll sein, via SSL auf eine Datenbank zuzugreifen. PostgreSQL unterstützt diese Möglichkeit und erlaubt verschlüsselte Kommunikation mit einem Client. Um mit SSL arbeiten zu können, sind einige Vorarbeiten notwendig, auf die wir in diesem Abschnitt eingehen wollen.

Um SSL überhaupt verwenden zu können, ist es notwendig, dies dem System beim Kompilieren mitzuteilen. Mithilfe der Option `–with–ssl` wird die SSL Unterstützung mit kompiliert und kann beim Starten des Datenbankservers aktiviert werden.

Bisher haben wir PostgreSQL immer ohne Unterstützung von SSL gestartet; hier sehen wir, wie SSL eingeschaltet werden kann:

```
pg_ctl -D /data/db -o "-i -l " start
```

Die Option -l sorgt dafür, dass SSL eingeschaltet wird — das reicht allerdings noch nicht für den Betrieb sicherer Verbindungen aus, weil es erst notwendig ist, Schlüssel zu vereinbaren, da die Datenbank sonst nicht gestartet werden kann:

```
[postgres@duron pgsql]$ pg_ctl -D /data/postgres_db/ -o "-i -l" start
postmaster successfully started
[postgres@duron pgsql]$ /usr/bin/postmaster: failed to load server
certificate (/data/postgres_db//server.crt): No such file or directory
```

Um die Schlüssel zu vereinbaren, sind einige Schritte notwendig, die im Folgenden aufgezeigt werden sollen.

Im ersten Schritt können Sie ein selbst signiertes Zertifikat erstellen. Das funktioniert am einfachsten so:

```
[postgres@duron tmp]$ openssl req -new -text -out cert.req
Using configuration from /usr/share/ssl/openssl.cnf
Generating a 1024 bit RSA private key
..........++++++
.........................................++++++
writing new private key to 'privkey.pem'
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or
a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:AT
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
```

```
29 A challenge password []:
30 An optional company name []:
```

Füllen Sie das Eingabeformular einfach aus und folgen Sie den Anweisungen der Software.

Um die Passwortsequenz zu entfernen, ist wie folgt vorzugehen:

```
1 [postgres@duron tmp]$ openssl rsa -in privkey.pem -out cert.pem
2 read RSA key
3 Enter PEM pass phrase:
4 writing RSA key
```

Im nächsten Schritt kann das Zertifikat in ein selbst signiertes Zertifikat umgewandelt werden:

```
1 [postgres@duron tmp]$ openssl req -x509 -in cert.req -text -key
2 cert.pem -out cert.cert
3 Using configuration from /usr/share/ssl/openssl.cnf
```

Nach dieser zugegeben etwas aufwändigen Prozedur, müssen Sie nur mehr die erstellten Dateien in das Hauptverzeichnis Ihres PostgreSQL-Datenbank-Clusters kopieren:

```
1 [postgres@duron tmp]$ cp cert.cert /data/postgres_db/server.crt
2 [postgres@duron tmp]$ cp cert.pem /data/postgres_db/server.key
```

Bevor wir den Datenbankserver neu starten, verfassen wir noch eine leicht modifizierte Konfiguration von `pg_hba.conf`:

```
1 hostssl    all         127.0.0.1     255.255.255.255    trust
```

Wir wollen, dass alle lokalen Verbindungen zur Datenbank via SSL abgewickelt werden.

Im Anschluss daran kann PostgreSQL gestartet werden:

```
1 [postgres@duron tmp]$ pg_ctl -D /data/postgres_db/ -o "-i -l" start
2 postmaster successfully started
3 [postgres@duron tmp]$ DEBUG:  database system was shut down at
4 2002-04-07 23:07:42 CEST
5 DEBUG:  checkpoint record is at 0/118F1480
6 DEBUG:  redo record is at 0/118F1480; undo record is at 0/0; shutdown
7 TRUE
8 DEBUG:  next transaction id: 1895; next oid: 3042360
9 DEBUG:  database system is ready
```

Wenn Sie alles richtig gemacht haben, wird es zu keinen Problemen gekommen sein. Um zu sehen, ob alles funktioniert, versuchen wir eine Verbindung zu PostgreSQL aufzubauen:

```
[postgres@duron pgsql]$ psql -h localhost buch
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

SSL connection (cipher: DES-CBC3-SHA, bits: 168)
```

An den Meldungen von `psql` ist zu erkennen, dass die Kommunikation mit der Datenbank via SSL funktioniert. Sie sollten diesen Test unbedingt durchführen, um zu wissen, ob Sie alles richtig gemacht und konfiguriert haben.

Wenn Sie PostgreSQL als Webdatenbank einsetzen, sollten Sie sich sehr genau überlegen, ob Sie SSL-Verbindungen dieser Form einsetzen wollen, da der Authentifizierungsoverhead enorm ist (bis zu Faktor 200 im Gegensatz zu normaler Authentifizierung). Es ist eher ratsam, den ganzen Port zu verschlüsseln, als SSL und PostgreSQL direkt zu verwenden.

4.3 Benutzerrechte

Wenn mehrere Benutzer auf Ihre Datenbanken zugreifen, ist es ratsam, sich Gedanken über deren Rechte zu machen. In diesem Abschnitt werden wir uns genauer mit den Möglichkeiten von PostgreSQL auseinander setzen und zeigen, was Sie machen können, um Ihre Datenbank möglichst sicher zu gestalten.

4.3.1 Benutzer- und Gruppenverwaltung

Beginnen wir mit einem Überblick über die Benutzerverwaltung in PostgreSQL. Das Kernstück der Benutzerverwaltung basiert auf so genannten Systemtabellen, über die jede ernst zu nehmende Datenbank verfügt. Im Falle von PostgreSQL sind zwei Tabellen von Relevanz. Die Tabelle `pg_shadow` enthält die in der Datenbank vorhandenen Benutzer. Sehen wir uns die Struktur der Tabelle an:

```
buch=# \d pg_shadow
         Table "pg_shadow"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 usename     | name    |
 usesysid    | integer |
 usecreatedb | boolean |
 usetrace    | boolean |
 usesuper    | boolean |
 usecatupd   | boolean |
 passwd      | text    |
 valuntil    | abstime |
Unique keys: pg_shadow_usename_index,
             pg_shadow_usesysid_index
Triggers: pg_sync_pg_pwd
```

PostgreSQL verfügt über ein UNIX-ähnliches Benutzersystem. Die Gruppenverwaltung dieses Systems erfolgt in `pg_group`:

```
buch=# \d pg_group
         Table "pg_group"
  Column  |   Type    | Modifiers
----------+-----------+-----------
 groname  | name      |
 grosysid | integer   |
 grolist  | integer[] |
Unique keys: pg_group_name_index,
             pg_group_sysid_index
```

Um einen Benutzer anzulegen, können Sie den Befehl `CREATE USER` verwenden. Die Syntax dieses Befehles ist relativ einfach und im nächsten Listing dargestellt:

```
buch=# \h CREATE USER
Command:     CREATE USER
Description: define a new database user account
Syntax:
CREATE USER username [ [ WITH ] option [ ... ] ]

where option can be:

          SYSID uid
          | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
          | CREATEDB | NOCREATEDB
          | CREATEUSER | NOCREATEUSER
          | IN GROUP groupname [, ...]
          | VALID UNTIL 'abstime'
```

Die Syntax des Kommandos ist einfach und extrem mächtig. Wir wollen also versuchen, einen Benutzer in der Datenbank anzulegen:

```
buch=# CREATE USER hans WITH PASSWORD 'wort' NOCREATEDB NOCREATEUSER;
CREATE USER
```

Sobald Sie den Benutzer anlegen, fügt PostgreSQL einen Datensatz zu `pg_shadow` hinzu, der sehr einfach abgefragt werden kann:

```
buch=# SELECT usename, usesuper, passwd FROM pg_shadow WHERE usename =
'hans';
 usename | usesuper | passwd
---------+----------+--------
 hans    | f        | wort
(1 row)
```

Nachdem Sie erfahren haben, wie ein Benutzer angelegt werden kann, werfen wir einen Blick auf die Gruppenverwaltung. Um eine Gruppe anzulegen, kann der Befehl `CREATE GROUP` verwendet werden, dessen Syntax im folgenden Listing beschrieben ist:

```
buch=# \h CREATE GROUP
Command:     CREATE GROUP
Description: define a new user group
Syntax:
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

      SYSID gid
      | USER  username [, ...]
```

Das folgende Beispiel zeigt, wie eine Gruppe angelegt werden kann:

```
buch=# CREATE GROUP cybertec WITH USER hans;
CREATE GROUP
```

Wir haben eine Gruppe namens `cybertec` angelegt und `hans` zu dieser Gruppe hinzugefügt.

Wenn Sie Benutzer und Gruppen einmal angelegt habenn, kann es leicht vorkommen, dass diese modifiziert werden. Auch für solche Zwecke bietet PostgreSQL die passenden Befehle und Werkzeuge. Eines dieser Werkzeuge ist der Befehl `ALTER USER`, der zum Modifizieren eines Benutzerkontos herangezogen werden kann. Alle wesentlichen Änderungen an einem User können mithilfe dieses Befehles leicht durchgeführt werden. Der Befehl bietet eine Reihe von Möglichkeiten:

```
buch=# \h ALTER USER
Command:     ALTER USER
Description: change a database user account
```

```
Syntax:
ALTER USER username [ [ WITH ] option [ ... ] ]

where option can be:

          [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
        | CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | VALID UNTIL 'abstime'
```

Im Prinzip kann man mit `ALTER USER` alles modifizieren, was beim Anlegen des Users definiert wurde. Um etwa ein Password zu ändern, können Sie `ALTER USER` verwenden. Im Folgenden ist zu erkennen, wie eine solche Operation durchgeführt werden kann. Nehmen wir an, wir wollen das Passwort von `hans` auf `juhu007` setzen:

```
buch=# ALTER USER hans WITH PASSWORD 'juhu007';
ALTER USER
```

Um noch andere Eigenschaften des Benutzerkontos zu verändern, müssen Sie nur die passenden Parameter setzen.

Dieselbe Operation, die wir soeben für Benutzer durchgeführt haben, kann auch auf Gruppen angewendet werden. Im Falle von Gruppen heißt der Befehl `ALTER GROUP` und hat folgende Syntax:

```
buch=# \h ALTER GROUP
Command:     ALTER GROUP
Description: add users to a group or remove users from a group
Syntax:
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Wie aus der Syntax hervorgeht, ist es möglich, Benutzer Gruppen zuzuordnen. Auch das Entfernen eines Benutzers aus einer Gruppe ist möglich und kann sehr leicht bewältigt werden, wie das nächste Listing beweist:

```
buch=# ALTER GROUP cybertec DROP USER hans;
ALTER GROUP
```

Sofern es zu keinem Fehler gekommen ist, hat PostgreSQL den Benutzer erfolgreich aus der Gruppe entfernt.

Um einen Benutzer komplett zu entfernen, bietet PostgreSQL den Befehl `DROP USER` an. Hier ist ein kurzer Überblick über die Syntax des Befehles:

```
buch=# \h DROP USER
Command:     DROP USER
Description: remove a database user account
Syntax:
DROP USER name
```

Das Löschen eines Benutzers ist denkbar einfach und kann mit einer simplen Befehlszeile sehr schnell bewerkstelligt werden. Im nächsten Beispiel können Sie sehen, wie der Benutzer hans aus der Datenbank entfernt wird:

```
buch=# DROP USER hans;
DROP USER
```

Um Gruppen zu löschen, können Sie DROP GROUP verwenden:

```
buch=# \h DROP GROUP
Command:     DROP GROUP
Description: remove a user group
Syntax:
DROP GROUP name
```

Die Syntax des Befehles ist wie die Syntax von DROP USER. Der einzige Unterschied ist, dass Sie den Namen der Gruppe statt des Namens des Benutzers angeben müssen. Versuchen wir also, die Gruppe Cybertec wieder vom System zu entfernen:

```
buch=# DROP GROUP cybertec;
DROP GROUP
```

4.3.2 Rechtevergabe

Nachdem wir uns eingehend mit der Verwaltung von Benutzern und Gruppen beschäftigt haben, wenden wir uns nun der Rechtevergabe zu. Im Prinzip sind zwei Befehle notwendig, um Rechte zuzuweisen beziehungsweise um einem Benutzer gewisse Rechte wegzunehmen. Wie im ANSI-Standard vorgeschrieben nennen sich diese beiden Befehle GRANT und REVOKE.

Um einem Benutzer Rechte zuzuweisen, wird der Befehl GRANT benötigt:

```
buch=# \h GRANT
Command:     GRANT
Description: define access privileges
Syntax:
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
TRIGGER } [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] objectname [, ...]
    TO { username | GROUP groupname | PUBLIC } [, ...]
```

Rechte werden in der Regel für Objekte definiert. Die Rechte für folgende Befehle können einem Benutzer zugewiesen werden: `SELECT`, `INSERT`, `UPDATE` und `DELETE`. Des Weiteren können noch Rechte auf Regeln, Referenzen und Trigger vergeben werden. Wenn Sie abfragen, welche Rechte auf welche Objekte definiert sind, müssen Sie sich mit einigen Kürzeln vertraut machen, die von PostgreSQL verwendet werden:

`r` `SELECT` (Lesen)

`w` `UPDATE` (Schreiben)

`a` `APPEND` (Anhängen)

`d` `DELETE` (Löschen)

`R` `RULE` (nicht ANSI-Erweiterung für PostgreSQL)

`x` `REFERENCES`

`t` `TRIGGER` (ein Feature von ANSI SQL 99)

`arwdRxt` Alle Rechte

Bevor wir nun endgültig zu praktischen Beispielen kommen, wollen wir uns noch die Syntax von `REVOKE` ansehen. Wie wir bereits erwähnt haben, dient `REVOKE` zum Entfernen von Rechten:

```
1 buch=# \h REVOKE
2 Command:     REVOKE
3 Description: remove access privileges
4 Syntax:
5 REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES |
6 TRIGGER } [,...] | ALL [ PRIVILEGES ] }
7     ON [ TABLE ] object [, ...]
8     FROM { username | GROUP groupname | PUBLIC } [, ...]
```

Die Syntax von `REVOKE` entspricht der von `GRANT` und ist relativ leicht zu verstehen.

Wenden wir uns nun einigen praktischen Beispielen zu. Bisher haben wir alle Operationen in der Datenbank als User `postgres` durchgeführt. `postgres` ist in der Regel (abhängig von der Installation) der Name des Superusers. Legen wir also eine neue Tabelle und einen neuen Benutzer an:

```
1 buch=# CREATE TABLE db (name text);
2 CREATE
3 buch=# CREATE USER hans WITH NOCREATEDB NOCREATEUSER PASSWORD 'xyz';
4 CREATE USER
```

Nachdem Sie den Benutzer angelegt haben, können Sie sich als User `hans` zur Datenbank verbinden:

```
[postgres@duron postgres_db]$ psql -U hans buch
```

Wenn Sie jetzt versuchen, die Tabelle db abzufragen, werden Sie Schiffbruch erleiden, weil Sie keine Berechtigung haben, die Tabelle zu lesen:

```
buch=> SELECT * FROM db;
ERROR:  db: Permission denied.
```

Sie dürfen die Tabelle nicht lesen, obwohl lokale Verbindungen standardmäßig auf trust gesetzt sind — das hat nichts mit Benutzerrechten, sondern lediglich mit Authentifizierung zu tun.

Wenn Sie also die Tabelle lesen und bearbeiten wollen, muss der User namens postgres die entsprechenden Rechte verteilen. Zu diesem Zwecke müssen Sie sich wieder als postgres einloggen und die Befehle als postgres ausführen:

```
buch=# GRANT SELECT ON TABLE db TO hans;
GRANT
buch=# GRANT INSERT ON TABLE db TO hans;
GRANT
buch=# GRANT UPDATE ON TABLE db TO hans;
GRANT
```

Nach dieser Operation können Sie wieder auf den neuen User namens hans wechseln und sehen, was passiert:

```
buch=> SELECT * FROM db;
 name
------
(0 rows)
```

Jetzt haben Sie die Rechte, diese Operationen durchzuführen.

Wichtig zu bemerken ist, dass sich ein Benutzer selbst keine Rechte zuordnen kann, was jedoch logisch ist, weil sich sonst das gesamte Konzept selbst ad absurdum führen würde:

```
buch=> GRANT DELETE ON TABLE db TO hans;
ERROR:  permission denied
```

Um herauszufinden, welche Rechte für eine Tabelle definiert sind, können Sie den Befehl \z verwenden:

```
buch=> \z db
 Access privileges for database "buch"
 Table |       Access privileges
-------+-------------------------------
 db    | {=,postgres=arwdRxt,hans=arw}
(1 row)
```

Aus der Tabelle geht hervor, dass der User `postgres` alle Rechte hat, der Benutzer `hans` jedoch mit einem eingeschränkten Spektrum an Befehlen auskommen muss.

Wenn Sie dem Benutzer auch diese Rechte wieder aberkennen wollen, müssen Sie sich als `postgres` am System anmelden und `REVOKE` verwenden:

```
1 buch=# REVOKE ALL ON db FROM hans;
2 REVOKE
```

Wenn Sie jetzt die Liste der Rechte für die Tabelle `db` abfragen, werden Sie sehen, dass nur mehr der Benutzer namens `postgres` Rechte hat:

```
1 buch=# \z db
2 Access privileges for database "buch"
3  Table |  Access privileges
4 -------+----------------------
5  db    | {=,postgres=arwdRxt}
6 (1 row)
```

Das Definieren von Rechten mittels `GRANT` und `REVOKE` geht relativ einfach vonstatten und bietet grundlegenden Schutz für Ihre Daten. Es empfiehlt sich daher, ein restriktives Benutzersystem zu entwickeln.

4.3.3 Superuser

Beim Arbeiten mit Benutzerrechten sind einige Dinge von großer Bedeutung, die an dieser Stelle nicht unerwähnt bleiben dürfen. Wenn Sie einen Benutzer anlegen, der seinerseits wieder das Recht hat, neue Benutzer anzulegen, ist dieser Benutzer wieder ein Superuser. Da ein Superuser alle Rechte hat, ist es unmöglich, einem derartigem Benutzer irgendwelche Recht abzuerkennen. Mit anderen Worten: Wenn ein Benutzer neue Benutzer anlegen darf, ist das Arbeiten mit `GRANT` und `REVOKE` absolut sinnlos, weil einem solchen Benutzer keine Rechte aberkannt werden können.

4.4 Tuning

Nachdem Sie eine Applikation zum Laufen gebracht haben, werden Sie in vielen Fällen versuchen, die Laufzeit Ihrer Programme signifikant zu senken. Da die von der Datenbank beanspruchte Systemzeit oft einen Großteil der gesamten Rechenzeit in Anspruch nimmt, ist es nur nahe liegend, den Versuch zu unternehmen, die Geschwindigkeit der Datenbank zu erhöhen. PostgreSQL bietet eine Reihe von Schrauben und Rädchen, an denen Sie drehen können, um auch noch den letzten Funken Speed aus Ihrem Rechner zu kitzeln. Einige wenige von diesen Rädchen werden wir uns in diesem Abschnitt ansehen. Für umfangreichere Informationen

zum Thema Tuning empfehlen wir unser »PostgreSQL Developers Handbook« (ISBN 0-672-32382-6) sowie »PHP And PostgreSQL: Advanced Web Development« (ISBN 0-672-32260-9) von SAMS.

4.4.1 Indices

Die größten Geschwindigkeitsgewinne können wohl durch das Definieren von Indices erreicht werden. Nehmen wir an, Sie haben eine Tabelle mit mehreren Millionen Datensätzen und müssen einen speziellen Datensatz suchen. Die Datenbank muss die gesamte Tabelle von vorne bis hinten durchlesen, um den Datensatz zu finden. Im Falle von Joins kann die Situation noch schlimmer sein: Bei großen Datenmengen muss die zu joinende Tabelle für jeden Datensatz in der ersten Tabelle einmal gelesen werden. Insgesamt ergibt das einen extrem großen Aufwand; die Zahl der notwendigen Schritte, um zum Ergebnis zu gelangen, steigt dabei im Quadrat zur Datenmenge. Im Falle von 1.000 Datensätzen in jeder Tabelle wären das immerhin schon 1.000.000 Operationen, bei höheren Mengen natürlich dementsprechend mehr. Dieser Aufwand ist in einem System nicht vertretbar und kann leicht vermieden werden.

Wenn Sie eine Spalte indizieren, wird intern eine so genannte Baumstruktur aufgebaut. Mithilfe eines Baumes ist es möglich, einen einzelnen Datensatz mit logarithmischem Aufwand statt mit linearem zu finden. Wenn Sie die Zahl der Datensätze vertausendfachen, steigt die benötigte Rechenzeit bei linearem Aufwand um den Faktor 1.000. Im Falle von logarithmischem Aufwand steigt dieser höchstens um den Faktor 10 (theoretisch). Das ist signifikant weniger. Wichtig zu bemerken ist, dass das errechnete Werte sind, die sehr stark von der Implementierung der Baumstruktur abhängen und wesentlich geringer sind — wichtig ist nur, dass Sie ein Bild und einen Eindruck bekommen, was intern passiert.

Mit diesem Wissen im Hinterkopf beginnen viele Datenbankentwickler, auf jede Spalte einen Index zu setzen. Bitte beachten Sie, dass ein Index nur dann etwas bringt, wenn nach speziellen Werten in dieser Spalte gesucht wird. Außerdem sollten Sie bedenken, dass ein Index bei `INSERT`-, `UPDATE`- und `DELETE`-Operationen signifikanten Overhead produzieren und es daher auch zu Verlangsamungen kommen kann. Diese Dinge müssen beachtet werden, wenn Sie planen, mit Indices zu arbeiten.

Nach diesem Überblick sehen wir uns an, wie Sie Indices praktisch anwenden können. Der Befehl `CREATE INDEX` dient zum Anlegen eines Indexes und hat folgende Syntax:

```
1 buch=# \h CREATE INDEX
2 Command:     CREATE INDEX
3 Description: define a new index
```

```
Syntax:
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( column [ ops_name ] [, ...] )
    [ WHERE predicate ]
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_method ] ( func_name( column [, ... ]) [ ops_name ] )
    [ WHERE predicate ]
```

Wie Sie dem Listing entnehmen können, ist die Syntax des Befehles denkbar einfach und kann leicht gemerkt werden. Sehen wir uns trotzdem an, wie ein Index auf eine Spalte definiert werden kann:

```
buch=# CREATE TABLE baum (id int4, name text);
CREATE
buch=# CREATE INDEX idx_baum_id ON baum (id);
CREATE
```

In diesem Fall haben wir den Index auf die erste Spalte definiert. Sofern nun größere Datenmengen eingefügt werden, ist es der Datenbank möglich, den Index zu benutzen. Zu bedenken ist, dass die Datenbank den Index benutzen kann, aber keinesfalls benutzen muss — das ist ein wichtiger Punkt, der niemals vergessen werden darf. Die Datenbank entscheidet, ob der Index in Verwendung ist oder nicht.

Laut ANSI SQL-Standard ist es auch möglich, einen Index über mehrere Spalten zu definieren; das nächste Listing zeigt, wie das bewerkstelligt werden kann:

```
buch=# CREATE INDEX idx_baum_id_name ON baum (id, name);
CREATE
```

In diesem Fall ist der Index nur dann sinnvoll, wenn Sie nach beiden Spalten suchen; suchen Sie nur nach der ersten oder der zweiten Spalte ist der Index wertlos.

In vielen Fällen kann es sinnvoll sein, einen Index zu definieren, bei denen jeder Wert nur einmal vorkommen kann. Das ist sinnvoll, um sicherzustellen, dass keine Werte doppelt vorkommen, oder schlicht und einfach, um die Geschwindigkeit der Datenbank zu steigern. Ein unique Index kann wie folgt definiert werden:

```
buch=# CREATE UNIQUE INDEX idx_baum_name ON baum (name);
CREATE
```

Um einen Index wieder zu löschen, können Sie den Befehl `DROP INDEX` heranziehen:

```
buch=# \h DROP INDEX
Command:     DROP INDEX
Description: remove an index
```

```
Syntax:
DROP INDEX index_name [, ...]
```

Die Syntax des Befehles ist denkbar einfach. Das folgende Beispiel zeigt, wie ein Index entfernt werden kann:

```
buch=# DROP INDEX idx_baum_name;
DROP
```

4.4.2 VACUUM

Ein wesentlicher Punkt für jeden Anwender, die sich intensiver mit PostgreSQL beschäftigt, ist der Befehl `VACUUM`. Bei vielen kleineren und größeren Problemen ist `VACUUM` die einzige und einfachste Lösung. Im Prinzip ist `VACUUM` der elektronische Staubsauger, der bei Bedarf auch gleich noch Statistiken erstellt.

Es kann oft passieren, dass sich in der Datenbank Datenschrott anhäuft. Man denke nur an eine Datenbank, auf der unzählige konkurrierende Transaktionen um Systemzeit kämpfen. Viele Operationen legen vielleicht temporäre Tabellen an und werden vom Benutzer währenddessen gestoppt. Es gibt viele Gründe, die zu Datenschrott führen können. Dieser Schrott ist für den Benutzer zwar nicht sichtbar, aber dennoch kann zu viel Abfall die Performance Ihrer Applikation beeinflussen. Um alle diese Probleme zu beheben, kann `VACUUM` verwendet werden, dessen Syntax im nächsten Listing kurz beschrieben ist:

```
buch=# \h VACUUM
Command:     VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [,
...] ) ] ]
```

Es gibt zahlreiche Stufen von `VACUUM`. Ein `VACUUM` ohne Parameter eliminiert den Großteil des Abfalles und sorgt für Ordnung. Wenn Sie versuchen wollen, den von der Datenbank verwendeten Speicherplatz wirklich zu minimieren, müssen Sie `VACUUM FULL` verwenden. Um während des Aufräumens für den Optimizer wichtige Statistiken zu erstellen, können Sie zusätzlich noch das Schlüsselwort `ANALYZE` verwenden. Wenn Sie verhindern wollen, dass Ihre Datenbank einen bestimmten Index einfach nicht verwenden will, kann `VACUUM ANALYZE` oft Wunder wirken.

Hier sehen Sie, was passiert, wenn wir `VACUUM` laufen lassen:

```
buch=# VACUUM FULL ANALYZE;
VACUUM
```

Nach wenigen Sekunden sollte das Spektakel vorbei sein. Während `VACUUM` aktiv ist, können andere Operationen ohne Probleme durchgeführt werden.

4.4.3 Tunen des Optimizers

Der Optimizer ist ein Kernstück jeder Datenbank. Die Aufgabe dieser Komponente ist es, den besten Weg durch eine Abfrage zu finden. Dabei wird ein so genannter Execution-Plan erstellt, der für die Durchführung der Abfrage von großer Bedeutung ist. Je besser der ausgearbeitete Execution-Plan ist, desto effizienter wird die eigentliche Abfrage der Daten im Anschluss sein. Aus diesem Grund muss sehr viel Wert auf das Analysieren solcher Pläne gelegt werden, um eine Datenbank wirklich schnell zu machen. Der wichtigste Befehl im Umgang mit dem Optimizer ist der Befehl `EXPLAIN`:

```
1 buch=# \h EXPLAIN
2 Command:     EXPLAIN
3 Description: show the Execution-Plan of a statement
4 Syntax:
5 EXPLAIN [ ANALYZE ] [ VERBOSE ] query
```

`EXPLAIN` dient zum Ausgeben des Execution-Plans einer Abfrage. So sieht ein einfacher Execution-Plan aus:

```
1 buch=# EXPLAIN SELECT * FROM baum;
2 NOTICE:  QUERY PLAN:
3
4 Seq Scan on baum  (cost=0.00..0.00 rows=1 width=36)
5
6 EXPLAIN
```

Sinnvollerweise liest PostgreSQL die gesamte Tabelle von vorne bis hinten. Das ist notwendig, da ohnehin alle Werte ausgegeben werden.

Um noch mehr Informationen über eine Abfrage zu erzeugen, kann zusätzlich zu `EXPLAIN` das Keyword `VERBOSE` verwendet werden:

```
1 buch=# EXPLAIN VERBOSE SELECT * FROM baum;
2 NOTICE:  QUERY DUMP:
3
4 { SEQSCAN :startup_cost 0.00 :total_cost 0.00 :rows 1 :width 36
5 :qptargetlist ({ TARGETENTRY :resdom { RESDOM :resno 1 :restype 23
6 :restypmod -1 :resname id :reskey 0 :reskeyop 0 :ressortgroupref 0
7 :resjunk false } :expr { VAR :varno 1 :varattno 1 :vartype 23
8 :vartypmod -1  :varlevelsup 0 :varnoold 1 :varoattno 1}} { TARGETENTRY
9 :resdom { RESDOM :resno 2 :restype 25 :restypmod -1 :resname name
10 :reskey 0 :reskeyop 0 :ressortgroupref 0 :resjunk false } :expr { VAR
```

```
:varno 1 :varattno 2 :vartype 25 :vartypmod -1  :varlevelsup 0
:varnoold 1 :varoattno 2}}) :qpqual <> :lefttree <> :righttree <>
:extprm () :locprm () :initplan <> :nprm 0
:scanrelid 1 }
NOTICE:  QUERY PLAN:

Seq Scan on baum  (cost=0.00..0.00 rows=1 width=36)

EXPLAIN
```

Mithilfe von VERBOSE erfahren Sie alles, was es über eine Query zu wissen gibt und vermutlich noch ein wenig mehr. Wenn Sie nicht wirklich mit PostgreSQL vertraut sind, macht es jedoch wenig Sinn, sich dieses Listing genau anzusehen.

Mit Execution-Plan können Sie sehr schnell die Schwachstellen einer Abfrage finden. Sehen wir uns ein Beispiel, einer Abfrage an. Zu diesem Beispiel benötigen wir einige Datensätze, die die durch ein kleines Perl-Programm sehr leicht generiert werden können:

```
#!/usr/bin/perl

# erstes File
open(FILE, "> data1.sql") or die "cannot open data.sql\n";
print FILE "CREATE TABLE data1 (id int4); \n";
print FILE "COPY data1 FROM stdin;\n";

for     ($i = 0; $i < 100000; $i++)
{
        print FILE "$i\n";
}

print FILE "\\.\n";
close FILE;

# zweites File
open(FILE, "> data2.sql") or die "cannot open data.sql\n";
print FILE "CREATE TABLE data2 (id int4); \n";
print FILE "COPY data2 FROM stdin;\n";

for     ($i = 0; $i < 100000; $i = $i + 10)
{
        print FILE "$i\n";
}

print FILE "\\.\n";
close FILE;
```

Das Script generiert zwei Tabellen. Die erste Tabelle enthält 100.000 Datensätze, die zweite Tabelle 10.000.

Versuchen wir, das Programm in der UNIX-Shell zu starten:

```
[postgres@duron code]$ time ./gen.pl

real    0m0.907s
user    0m0.870s
sys     0m0.000s
```

Da die Daten sehr einfach strukturiert sind, wird das Programm in weniger als einer Sekunde ausgeführt (AMD Duron 750; 384 MB Ram).

Im nächsten Schritt können die Daten in die Datenbank importiert werden:

```
[postgres@duron code]$ time psql buch < data1.sql
CREATE

real    0m1.910s
user    0m0.140s
sys     0m0.030s
[postgres@duron code]$ time psql buch < data2.sql
CREATE

real    0m0.288s
user    0m0.020s
sys     0m0.010s
```

Dabei geht auch der Import sehr schnell vor sich.

Im Folgenden sehen wir uns einige Abfragen und deren Execution-Pläne an. Auf diese Weise werden Sie sehr schnell erkennen, wie der Optimizer beeinflusst werden kann und welche Möglichkeiten zur Disposition stehen:

```
[postgres@duron code]$ time psql buch -c "SELECT * FROM data1 WHERE id
> 100 ORDER BY id LIMIT 3"
 id
-----
 101
 102
 103
(3 rows)


real    0m2.462s
user    0m0.010s
sys     0m0.030s
```

Bei dieser Abfrage werden die Daten sortiert und die ersten drei Datensätze ausgegeben, die größer als 100 sind. Intern muss PostgreSQL die Daten erst sortieren und dann die entsprechenden Datensätze auswählen, wie das folgende Listing beweist:

```
buch=# EXPLAIN SELECT * FROM data1 WHERE id > 100 ORDER BY id LIMIT 3;
NOTICE:  QUERY PLAN:

Limit  (cost=36.47..36.47 rows=3 width=4)
  ->  Sort  (cost=36.47..36.47 rows=333 width=4)
        ->  Seq Scan on data1  (cost=0.00..22.50 rows=333 width=4)

EXPLAIN
```

Ein Execution-Plan ist immer von rechts nach links zu lesen. Am Anfang wird die Tabelle komplett gelesen. Im Execution-Plan ist das leicht an »Seq Scan on data1« zu erkennen. Nach diesem sequential Scan werden die Daten sortiert. Schließlich werden die entsprechenden Datensätze ausgewählt. Hierbei dauert die Abfrage mehr als zwei Sekunden. Der Grund dafür ist, dass die Daten erst sortiert werden müssen. Wie Sie aus diesem Abschnitt bereits wissen, wird eine von ihnen definierte Menge an Daten im Speicher sortiert. In der Standardinstallation sind das 512 kb pro Sortiervorgang. Da die Datenmenge in unserem Fall nicht im Speicher sortiert werden kann, muss PostgreSQL auf die Festplatte ausweichen, was länger dauert als das Sortieren im Speicher.

Um das Problem zu lösen, können Sie auf zwei Arten vorgehen: Wenn Sie die Größe der Sortbuffer global erhöhen, wird diese spezielle Abfrage schneller laufen. Wenn sehr viele Abfragen zugleich laufen, kann das jedoch der falsche Weg sein, da zu viel Speicher für Sortbuffer verwendet wird. Die zweite Möglichkeit ist, den Sortbuffer nur für diese spezielle Abfrage zu ändern und das kann wie folgt gemacht werden:

```
[postgres@duron code]$ time psql buch -c "SET sort_mem TO 100000;
SELECT * FROM
data1 WHERE id > 100 ORDER BY id LIMIT 3"
 id
-----
 101
 102
 103
(3 rows)


real    0m1.076s
user    0m0.010s
sys     0m0.020s
```

In diesem Fall haben wir zwei Abfragen auf einmal abgesetzt. Der erste SQL-Befehl hat den Sortbuffer auf 100 MB erhöht. Anschließend haben wir die Abfrage losgeschickt. Da die Speichereinstellungen in diesem Fall nur lokal und temporär wirksam sind, müssen wir uns keine Sorgen machen, eine globale Einstellung modifiziert zu haben.

Deutlich erkennbar läuft die Abfrage signifikant schneller.

Im Weiteren wollen wir uns mit einer etwas komplexeren Abfrage beschäftigen. Ziel ist es alle Datensätze zu zählen, die in beiden Tabellen vorkommen; die Abfrage kann wie folgt aussehen:

```
SELECT COUNT(data2.id)
        FROM data1, data2
        WHERE data1.id=data2.id
```

Sehen wir uns den Execution-Plan der Abfrage an:

```
buch=# EXPLAIN SELECT COUNT(data2.id) FROM data1, data2 WHERE
data1.id=data2.id;
NOTICE:  QUERY PLAN:

Aggregate  (cost=219.66..219.66 rows=1 width=8)
  ->  Merge Join  (cost=139.66..207.16 rows=5000 width=8)
        ->  Sort  (cost=69.83..69.83 rows=1000 width=4)
              ->  Seq Scan on data1  (cost=0.00..20.00 rows=1000
                  width=4)
        ->  Sort  (cost=69.83..69.83 rows=1000 width=4)
              ->  Seq Scan on data2 (cost=0.00..20.00 rows=1000
                  width=4)

EXPLAIN
```

Das Listing ist schon wesentlich komplexer als die Execution-Pläne, die wir bisher gesehen haben. Am Beginn der Verarbeitung stehen wieder sequential Scans. Da wir keine Indices definiert haben, ist es der Datenbank nicht möglich, einen sequential Scan zu vermeiden. In diesem Falle ist das sinnvoll, da es sonst de facto unmöglich ist, meßbare Erfolge zu präsentieren.

In diesem Beispiel werden die sortierten Datenbestände zusammengemischt und anschließend aggregiert (gezählt). Sehen wir uns also an, wie lange die Abfrage benötigt, um das richtige Ergebnis zu berechnen:

```
[postgres@duron code]$ time psql buch -c "SELECT COUNT(data2.id) FROM
data1, data2 WHERE data1.id=data2.id"
 count
-------
 10000
(1 row)


real    0m2.422s
user    0m0.010s
sys     0m0.010s
```

Die Abfrage dauert etwas über zwei Sekunden. Um diese Abfrage ein wenig zu optimieren, können wir versuchen, Sortierungen auszuschalten. Bitte bedenken Sie, dass das nicht heißt, dass PostgreSQL nie mehr sortieren wird: Sortierungen werden nur weitgehend vermieden:

```
[postgres@duron code]$ time psql buch -c "SET enable_sort TO off;
SELECT COUNT(data2.id) FROM data1, data2 WHERE data1.id=data2.id"
 count
-------
 10000
(1 row)


real    0m2.021s
user    0m0.020s
sys     0m0.010s
```

Die Abfrage ist signifikant schneller als die vorherige Version. Das nächste Listing enthält den Execution-Plan der Abfrage:

```
[postgres@duron code]$ time psql buch -c "SET enable_sort TO off;
EXPLAIN SELECT COUNT(data2.id) FROM data1, data2 WHERE
data1.id=data2.id"
NOTICE:  QUERY PLAN:

Aggregate  (cost=372.50..372.50 rows=1 width=8)
  ->  Hash Join  (cost=22.50..360.00 rows=5000 width=8)
        ->  Seq Scan on data1  (cost=0.00..20.00 rows=1000 width=4)
        ->  Hash  (cost=20.00..20.00 rows=1000 width=4)
              ->  Seq Scan on data2 (cost=0.00..20.00 rows=1000 width=4)

EXPLAIN

real    0m0.055s
user    0m0.010s
sys     0m0.020s
```

Hierbei wird ein Hash generiert und dieser mit der zweiten Tabelle gejoint, bevor die Aggregierung durchgeführt wird.

Alle von PostgreSQL generierten Execution-Pläne basieren auf statistischer Information — wenn Sie diese regelmäßig erzeugen, werden Sie sehen, dass PostgreSQL wesentlich bessere Pläne baut und Abfragen schneller ausgeführt werden. Sollte auch ein `VACUUM` nicht zu besseren Ergebnissen führen, ist es sehr oft sinnvoll, dem Optimizer Hinweise zu geben, wie wir das in diesem Abschnitt gemacht haben. Es gibt keine Patentrezepte, die immer zu besseren Ergebnissen führen und es liegt daher an Ihnen, die besten Parametersettings zu finden. Durch das gezielte Modifizieren von Einstellungen können teilweise enorme Geschwindigkeitsgewinne erreicht werden. Ich kann Sie nur ermutigen, selbst aktiv zu werden und verschiedenste Einstellungen zu testen.

PostgreSQL verfügt über einen hoch entwickelten Optimizer. Wie bei allen Datenbanken ist dieser Optimizer nicht perfekt und mit einigen Handgriffen lassen sich solche Fehltritte leicht beheben. In diesem Abschnitt haben wir bewusst schlechte Settings verwendet, um Performancegewinne durch Tuning zu erreichen; in der Regel werden sehr gute Execution-Pläne generiert, die keiner weiteren Nachbearbeitung mehr bedürfen.

Kapitel 5

PL/pgSQL und Arbeiten mit Rules

Embedded Languages (eingebettete Sprachen) sind aus vielen Projekten nicht mehr wegzudenken. Wenn es darum geht, schnell und einfach kleine Erweiterungen für die Datenbank zu implementieren, ist es nicht immer sinnvoll, alle Berechnungen von der Applikation durchführen zu lassen. In vielen Fällen ist es wesentlich effizienter, einfacher und auch übersichtlicher, einfache Funktionen zu implementieren, die direkt in SQL eingesetzt werden können. Genau an Personen mit solchen Bedürfnissen richtet sich PL/pgSQL.

Wenn Sie bereits mit Oracle vertraut sind, wird Ihnen PL/SQL sicherlich ein Begriff sein. PL/pgSQL ist das abgespeckte Pendant von PL/SQL und kann in Kombination mit PostgreSQL eingesetzt werden. Wie so viele Komponenten von PostgreSQL ist auch PL/pgSQL als Plugin realisiert und in einer leeren Standarddatenbank nicht enthalten. Um PL/pgSQL zu Ihrer Datenbank hinzuzufügen, können Sie das Programm `createdb` verwenden:

```
1 [postgres@duron postgres_db]$ createlang plpgsql buch
```

Sofern es zu keinen Problemen gekommen ist, können Sie PL/pgSQL jetzt verwenden. Um wirklich sicherzugehen, dass die Sprache in der Datenbank ist, können Sie `createlang` erneut verwenden:

```
1 [postgres@duron html]$ createlang -l buch
2 Procedural languages
3   Name   | Trusted?
4 ---------+----------
5  plpgsql | t
6 (1 row)
```

Das Ergebnis des Befehles bestätigt das Vorhandensein von PL/pgSQL in der Datenbank.

Viele fragen sich jetzt vielleicht, warum PL/pgSQL erst zu einer Datenbank hinzugefügt werden muss. Die Antwort auf diese Frage liegt in der Philosophie von PostgreSQL begründet: Die Datenbank ist so modular wie nur irgendwie möglich. Viele Komponenten wie auch PL/pgSQL sind schlicht und einfach ersetzbar. Wenn Sie mit verschiedenen Versionen von PL/pgSQL arbeiten wollen, so kann das auch ohne Probleme bewerkstelligt werden — Sie müssen nur die entsprechende Version von PL/pgSQL in die passende Datenbank einfügen und schon gehts los. Die Modularität von PostgreSQL erlaubt einfache Erweiterbarkeit und garantiert die Übersichtlichkeit und Flexibilität des Gesamtsystems.

5.1 Einführung

Nach diesem kurzen Überblick wollen wir wieder in die Welt von PL/pgSQL zurückkehren. Wie PL/SQL von Oracle ist auch PL/pgSQL eine blockorientierte

Sprache. Das bedeutet, dass sehr wenige Klammern benötigt werden und Funktionen in der Regel sehr leicht lesbar sind. Das erlaubt schnelles und effizientes Entwickeln.

5.1.1 Einfache Funktionen

Bevor Sie beginnen, sich vor lauter Theorie zu langweilen, sehen wir uns ein einfaches Beispiel an:

```
1 CREATE FUNCTION summe (int4, int4) RETURNS int4 AS '
2         BEGIN
3                 RETURN $1 + $2;
4         END;
5 ' LANGUAGE 'plpgsql';
```

Die Funktion tut nichts anderes, als die Summe von zwei Werten zu berechnen. Wie in einem Bourne Shell-Programm werden auch hier die Werte, die an die Funktion weitergegeben werden, mit »$« gekennzeichnet.

Wichtig zu beachten ist, dass die gesamte Funktion unter einfachen Anführungszeichen übergeben wird. Die Funktion wird quasi als ein String an PostgreSQL übergeben. Das ist ein wichtiger Punkt, auf den wir noch zu sprechen kommen werden.

Da Sie nun also wissen, wie eine einfache Funktion implementiert werden kann, können wir diese testen. Zuvor müssen wir die Funktion jedoch erst in die Datenbank einfügen, was sehr leicht bewerkstelligt werden kann:

```
1 [postgres@duron code]$ psql buch < pl.sql
2 CREATE
```

Nach diesem Schritt können wir die Funktion ausführen:

```
1 buch=# SELECT summe(14, 42);
2  summe
3 ———————
4     56
5 (1 row)
```

Wie erwartet wird die Summe richtig berechnet. Was passiert, wenn die Funktion falsch aufgerufen wird?

```
1 buch=# SELECT summe(14, 'a string');
2 ERROR:  pg_atoi: error in "a string": can't parse "a string"
```

Der zweite Parameter kann nicht als Integer interpretiert werden und die Funktion bricht daher ab.

Um eine Funktion aus der Datenbank zu entfernen, können Sie den Befehl `DROP FUNCTION` verwenden. Hier ist die Syntax des Befehles:

```
buch=# \h DROP FUNCTION
Command:     DROP FUNCTION
Description: remove a user-defined function
Syntax:
DROP FUNCTION name ( [ type [, ...] ] )
```

Dabei müssen Sie einfach nur den Namen der Funktion und eine korrekte Liste der Parameter angeben, um die Funktion wieder aus der Datenbank zu entfernen:

```
buch=# DROP FUNCTION summe(int4, int4);
DROP
```

Versuchen wir die Funktion noch einmal aufzurufen:

```
buch=# SELECT summe(14, 42);
ERROR:  Function 'summe(int4, int4)' does not exist
        Unable to identify a function that satisfies the given
        argument types
        You may need to add explicit typecasts
```

Es kommt zu einem Fehler, der unscheinbar aussieht, aber als gutes Anschauungsbeispiel für die Funktionsweise von PostgreSQL dient. PostgreSQL sucht nach einer Funktion namens `summe(int4, int4)`. Sollte es Funktionen im System geben, die zwar ebenfalls zwei Parameter, aber eben nicht die richtigen Datentypen akzeptieren, wird es zu einem Problem kommen. Wie viele andere objektorientierte Systeme auch unterstützt PostgreSQL so genannte Überladungen, was heißt, dass es mehrere Funktionen mit demselben Namen, aber unterschiedlichen Parametern geben kann. Sehen wir uns das anhand eines kleinen Beispieles an:

```
CREATE OR REPLACE FUNCTION summe (int4, int4) RETURNS int4 AS '
        BEGIN
                RETURN $1 + $2;
        END;
' LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION summe (text, text) RETURNS text AS '
        BEGIN
                RETURN $1 || $2;
        END;
' LANGUAGE 'plpgsql';
```

Beide Funktionen haben den gleichen Namen und akzeptieren die gleichen Parameter. Da die Datentypen unterschiedlich sind, können die beiden Funktionen gut unterschieden werden. Hier sehen wir, was passiert, wenn wir die Funktionen aufrufen:

```
buch=# SELECT summe(32, 17);
 summe
-------
    49
(1 row)

buch=# SELECT summe('hello ', 'world');
    summe
-------------
 hello world
(1 row)
```

Die Datenbank sorgt dafür, dass die richtige Funktion automatisch gefunden wird.

An diesem Beispiel gibt es auch noch eine zweite Besonderheit: Statt `CREATE` haben wir diesmal `CREATE OR REPLACE` verwendet. Auf diese Weise können wir sicherstellen, dass die Funktion automatisch ersetzt wird.

Jetzt sehen wir uns eine leicht modifizierte Version unserer ersten Funktion an:

```
CREATE OR REPLACE FUNCTION summe (int4, int4) RETURNS int4 AS '
        DECLARE
                i       ALIAS FOR $1;
                j       ALIAS FOR $2;
                summe   int4;
        BEGIN
                summe := i + j;
                RETURN summe;
        END;
' LANGUAGE 'plpgsql';
```

Viele von Ihnen werden die `DECLARE`-Anweisung bereits von PL/SQL her kennen. Auch in PL/pgSQL-Funktionen kann `DECLARE` verwendet werden, um Variablen zu deklarieren. In unserem Fall definieren wir zwei Aliase und die Variable `summe`, die wir verwenden werden, um das Ergebnis zu speichern.

Nach dem Berechnen der Summe wird das Ergebnis der Variablen zugewiesen. Dabei ist die verwendete Syntax genau zu beachten. Das `:=` erinnert sehr stark an Pascal-Zeiten und wird Vielen noch bestens vertraut sein.

5.1.2 Kontrollstrukturen

Nachdem wir uns mit einfachen Funktionen beschäftigt haben, ist es Zeit, einen Blick auf einfache Kontrollstrukturen zu werfen.

Nehmen wir an, wir wollen eine Funktion schreiben, die überprüft, ob eine Zahl gerade ist oder nicht:

```
CREATE OR REPLACE FUNCTION even (int4) RETURNS bool AS '
        DECLARE
                i       ALIAS FOR $1;
                tmp     int4;
        BEGIN
                tmp := i % 2;
                IF      tmp = 0 THEN
                        RETURN true;
                ELSE
                        RETURN false;
                END IF;
        END;
' LANGUAGE 'plpgsql';
```

Die Funktion akzeptiert genau einen Parameter. In der Funktion wird die ihr übergebene Zahl modulo 2 berechnet. Auf diese Weise kann der Rest bestimmt werden, der bleibt, wenn die Zahl durch zwei dividiert wird. Wenn der Rest einer solchen Division gleich null ist, handelt es sich um eine gerade Zahl und `true` wird zurückgegeben. Andernfalls ist das Resultat der Funktion `false`.

Die nächsten beiden Beispiele zeigen die neue Funktion bei der Arbeit:

```
buch=# SELECT even(345);
 even
------
 f
(1 row)

buch=# SELECT even(346);
 even
------
 t
(1 row)
```

`IF/ELSE`-Konstruktionen sind nicht die einzigen Kontrollstrukturen, die von PL/pgSQL zur Verfügung gestellt werden. `FOR`-Schleifen sind feste Bestandteile fast jeder modernen Programmiersprache. Auch PL/pgSQL unterstützt `FOR`-Schleifen. Betrachten wir ein einfaches Beispiel:

```
CREATE OR REPLACE FUNCTION fakultaet (int4) RETURNS int4 AS '
        DECLARE
                i       ALIAS FOR $1;
                tmp     int4;
                result  int4;
        BEGIN
                result := 1;
                FOR tmp IN 1 .. i LOOP
                        result := result * tmp;
                END LOOP;
                RETURN result;
        END;
' LANGUAGE 'plpgsql';
```

Es zeigt eine einfache Funktion, die eine Zahl faktoriert. Die zu faktorierende Zahl wird übergeben und die Schleife sorgt dafür, dass die notwendigen Berechnungen durchgeführt werden. Das Token »`1 .. i`« bedeutet, dass die Schleife beim Wert 1 beginnt und mit i endet. Für jede Zahl wird genau eine Multiplikation durchgeführt. Am Ende wird das Ergebnis zurückgegeben. Hier zeigen wir, wie die Funktion aufgerufen werden kann:

```
buch=# SELECT fakultaet(8);
 fakultaet
-----------
    40320
(1 row)
```

Bedenken Sie, dass die Funktion keinerlei Sicherheitsabfragen enthält und in dieser Form nicht verwendet werden sollte. Sie werden aber sehen, wie die Funktion im Laufe des Kapitels noch verfeinert wird.

Eine andere Lösung für das Problem wäre die Verwendung einer `WHILE`-Schleife. Wir zeigen, wie `WHILE` funktioniert:

```
CREATE OR REPLACE FUNCTION fakultaet (int4) RETURNS int4 AS '
        DECLARE
                i       ALIAS FOR $1;
                tmp     int4;
                result  int4;
        BEGIN
                result  := 1;
                tmp     := 1;
                WHILE   tmp <= i LOOP
                        result := result * tmp;
                        tmp := tmp + 1;
                END LOOP;
                RETURN result;
        END;
' LANGUAGE 'plpgsql';
```

In diesem Beispiel ist die WHILE-Schleife etwas komplexer als die FOR-Schleife im letzten Beispiel. Eine WHILE-Schleife läuft, bis die Schleifenbedingung nicht mehr erfüllt ist. In unserem Fall ist das, wenn der Wert von tmp den Wert von i übersteigt. Sie dürfen auf keinen Fall vergessen, in der Schleife den Wert von tmp um 1 zu inkrementieren, da die Funktion sonst in eine Endlosschleife läuft, die nur mehr unsanft beendet werden kann.

Zusätzlich zu den bereits gezeigten Möglichkeiten unterstützt PL/pgSQL noch eine dritte Art von Schleife. Mithilfe von LOOP und EXIT können Sie ähnlich wie mit WHILE arbeiten. Hier haben wir also die dritte Version der Funktion fakultaet:

```
1  CREATE OR REPLACE FUNCTION fakultaet (int4) RETURNS int4 AS '
2          DECLARE
3                  i       ALIAS FOR $1;
4                  tmp     int4;
5                  result  int4;
6          BEGIN
7                  result := 1;
8                  tmp := 1;
9
10                 LOOP
11                         IF tmp > i THEN
12                                 EXIT;
13                         ELSE
14                                 result := result * tmp;
15                                 tmp := tmp + 1;
16                         END IF;
17                 END LOOP;
18                 RETURN result;
19         END;
20 ' LANGUAGE 'plpgsql';
```

Die Schleife wird so lange ausgeführt, bis sie mit EXIT beendet wird. EXIT sorgt dafür, dass das Programm auf der Schleife springt und mit dem nächsten Befehl fortfährt, der in unserem Fall das Ergebnis der Funktion zurückgibt.

Wie aus diesem Beispiel hervorgeht, bietet PL/pgSQL eine Reihe von Möglichkeiten, die für effizientes Programmieren von großer Bedeutung sind.

5.1.3 Arbeiten mit Abfragen

Nachdem Sie gelernt haben, wie einfache Verzweigungen und Schleifen implementiert werden können, wenden wir uns nun Abfragen innerhalb von PL/pgSQL zu. Ziel des nächsten Beispieles ist es, eine einfache Abfrage zu schreiben und deren Komponenten ein wenig zu formatieren:

```
CREATE OR REPLACE FUNCTION format () RETURNS text AS '
        DECLARE
                tmp     RECORD;
                result  text;
        BEGIN
                SELECT INTO tmp 1+1 AS a, 2+2 AS b;
                RETURN ''a = '' || tmp.a || ''; b = '' || tmp.b;
        END;
' LANGUAGE 'plpgsql';
```

Die Funktion macht nichts anderes als zwei Summen zu berechnen und diese mit ein wenig Text auszugeben. Sehen wir uns an, wie die Funktion implementiert worden ist: Am Beginn wird eine Variable vom Datentyp `RECORD` angelegt. Ein `RECORD` entspricht einer Zeile und kann mehrere Felder enthalten. Anschließend wird das Ergebnis der Abfrage in die Variable selektiert. Zu beachten ist, dass wir jedem Feld einen Namen zuordnen. Das wäre nicht unbedingt notwendig, aber es erhöht die Lesbarkeit ungemein. Danach wird das Ergebnis der Funktion zurückgegeben. Die Zeile, die die Rückgabe des Ergebnisses erledigt, ist es wert, näher betrachtet und analysiert zu werden. Da Strings unter einfachen Anführungszeichen stehen müssen, werden zwei Anführungszeichen benötigt. Bedenken Sie, dass die gesamte Funktion bereits unter einfachen Anführungszeichen an PostgreSQL übergeben wird — zwei einfachere Anführungszeichen sind daher notwendig, um das Zeichen zu maskieren. Des Weiteren werden die Werte im `RECORD` ausgegeben. Die Syntax ist relativ einfach, da nur der Punkt und der Name des Feldes an den Variablennamen angehängt werden müssen.

Hier steht was passiert, wenn wir die Funktion aufrufen:

```
buch=# SELECT format();
    format
---------------
 a = 2; b = 4
(1 row)
```

Das Ergebnis der Funktion ist nicht weiter überraschen:

In vielen Fällen kann es sinnvoll sein, eine Query ausführen, ohne sich das Ergebnis anzusehen. Zu diesem Zwecke stellt PL/pgSQL die Funktion `PERFORM` zur Verfügung, die im nächsten Beispiel gezeigt wird:

```
CREATE OR REPLACE FUNCTION format () RETURNS text AS '
        DECLARE
                result  text;
        BEGIN
                PERFORM ''SELECT 1+1'';
                RETURN ''t'';
```

```
        END;
' LANGUAGE 'plpgsql';
```

Der Code des aufzuführenden Statements kann sehr einfach unter doppelten, einfachen Anführungszeichen übergeben werden.

Um dynamisches SQL ausführen zu können, stellt PL/pgSQL die Funktion `EXECUTE` zur Verfügung:

```
CREATE OR REPLACE FUNCTION summe (int4, int4) RETURNS bool AS '
        BEGIN
                EXECUTE ''SELECT '' || $1 || '' + '' || $2;
                RETURN ''t'';
        END;
' LANGUAGE 'plpgsql';
```

In diesem Fall haben wir die Summe von zwei Integer-Variablen berechnet, die an die Funktion übergeben werden. `EXECUTE` wird üblicherweise verwendet, um `UPDATE`- oder `DELETE`-Operationen durchzuführen, bei denen die zu löschenden oder zu modifizierenden Werte erst berechnet werden müssen.

Wenn Sie eine Abfrage schreiben, wollen Sie in der Regel mit dem Ergebnis weiterrechnen. Auch dafür kann PL/pgSQL einfach und effizient verwendet werden. Das nächste Listing enthält ein einfaches Beispiel, das die Summe von Messwerten in einem bestimmten Zeitintervall berechnet:

```
CREATE TABLE messwert (
        zeit    time,
        wert    int4
);

COPY messwert FROM stdin;
21:17:03        19
21:18:12        43
21:19:09        32
21:20:17        49
21:21:29        48
\.

CREATE OR REPLACE FUNCTION summe (time, time) RETURNS int4 AS '
        DECLARE
                result  int4;
                s       RECORD;
        BEGIN
                result := 0;
                FOR  s IN SELECT * FROM messwert
                        WHERE zeit > $1 AND zeit < $2 LOOP
```

```
                        result := result + s.wert;
                END LOOP;
                RETURN result;
        END;
' LANGUAGE 'plpgsql';
```

Nach dem Anlegen einer Tabelle und nach dem Einfügen einiger Werte, geht es ans Eingemachte. Zwei Variablen werden vereinbart und eine Abfrage abgesetzt, die alle Werte in einem Zeitbereich selektiert. Jeder Datensatz im Ergebnis wird von der Schleife abgearbeitet und der entsprechende Wert zum Ergebnis addiert. Nach dem Abarbeiten aller Datensätze terminiert die Funktion und das Ergebnis wird zurückgegeben. Hier ist zu sehen, was passiert, wenn wir die Funktion ausführen:

```
buch=# SELECT summe('20:00:00'::time, '22:00:00'::time);
 summe
--------
    191
(1 row)
```

Das korrekte Ergebnis erscheint blitzschnell.

5.1.4 Fehlerabfragen

Bisher haben wir uns mit Funktionen aller Art beschäftigt, aber es hat sich bisher nie die Notwendigkeit ergeben, Fehler abzufragen. Dieser Teilabschnitt widmet sich gänzlich der Fehlerbehandlung. Hier ist ein Beispiel:

```
CREATE OR REPLACE FUNCTION divide (numeric, numeric) RETURNS numeric AS '
        BEGIN
                RETURN $1 / $2;
        END;
' LANGUAGE 'plpgsql';
```

Die Funktion sieht auf den ersten Blick korrekt aus und funktioniert in der Regel auch einwandfrei:

```
buch=# SELECT divide(32, 17);
     divide
----------------
 1.8823529412
(1 row)
```

Aber dennoch ist die Funktion nicht ganz fehlerfrei …

```
buch=# SELECT divide(32, 0);
NOTICE:  Error occurred while executing PL/pgSQL function divide
NOTICE:  line 2 at return
ERROR:  division by zero on numeric
```

Die Funktion beendet sich, wenn eine Division durch 0 auftritt, was nicht besonders angenehm ist. Vielmehr wollen wir das Verhalten der Funktion selber beeinflussen und einen schöneren Fehlercode ausgeben:

```
CREATE OR REPLACE FUNCTION divide (numeric, numeric) RETURNS numeric
AS '
        BEGIN
                IF      $2 = 0 THEN
                        RAISE EXCEPTION ''Fehler: ORA-9234324234234'';
                END IF;
                RETURN $1 / $2;
        END;
' LANGUAGE 'plpgsql';
```

Mithilfe von `RAISE EXCEPTION` kann ein beliebiger Fehler ausgegeben werden, der zum Abbruch der Funktion führt. Im Beispiel haben wir uns für eine vertraut aussehende Fehlermeldung entschieden:

```
buch=# SELECT divide(32, 0);
ERROR:  Fehler: ORA-9234324234234
```

Wenn Sie eine Meldung ausgeben wollen, die zwar auf einen Fehler hindeutet aber kein Fehler ist, können Sie `RAISE NOTICE` verwenden:

```
CREATE OR REPLACE FUNCTION divide (numeric, numeric) RETURNS numeric AS '
        BEGIN
                IF      $2 = 0 THEN
                        RAISE EXCEPTION ''Fehler: ORA-9234324234234'';
                END IF;

                IF      $1 > 1000 THEN
                        RAISE NOTICE ''% erscheint groß.'', $1;
                END IF;

                RETURN $1 / $2;
        END;
' LANGUAGE 'plpgsql';
```

Hier wird eine Meldung ausgegeben, falls der erste Parameter größer als 1.000 ist. Bei der Meldung handelt es sich um einen Hinweis, aber um keinen echten Fehler. Es ist sehr leicht möglich, Variablen in den Ausgabestring einzufügen. Mithilfe eines Prozentzeichens kann eine Variable an einer beliebigen Stelle eingefügt werden, wie das nächste Beispiel zeigt:

```
buch=# SELECT divide(3200, 17);
NOTICE:  3200 erscheint groß.
     divide
─────────────────
 188.2352941176
(1 row)
```

Die Meldung wird auf Standard Error ausgegeben und ist daher nicht in den Nutzdaten zu finden.

5.1.5 Cursor

Cursor sind ein komfortables Feature von PL/pgSQL. Mithilfe von Cursorn wird es möglich, nicht das gesamte Resultat einer Abfrage im Speicher halten zu müssen, sondern immer nur einige wenige Datensätze. Das hilft Ihnen, Speicher zu sparen und die Effizienz zu steigern. In diesem Abschnitt werden wir zeigen, wie mit Cursorn gearbeitet werden kann. Beginnen wir mit einem einfachen Beispiel:

```
CREATE OR REPLACE FUNCTION summe () RETURNS int4 AS '
        DECLARE
                mycursor        refcursor;
                tmp             RECORD;
                result          int4;
        BEGIN
                result := 0;
                OPEN mycursor FOR SELECT * FROM messwert;
                LOOP
                        FETCH mycursor INTO tmp;
                        EXIT WHEN NOT FOUND;
                        RAISE NOTICE ''value - %'', tmp.wert;
                        result := result + tmp.wert;
                END LOOP;
                CLOSE mycursor;
                RETURN result;
        END;
' LANGUAGE 'plpgsql';
```

Ziel ist es, alle Werte in einer Tabelle zu summieren. Das kann sehr leicht mit bereits in SQL eingebauten Befehlen bewerkstelligt werden, aber wir haben dieses Beispiel aus Gründen der Einfachheit gewählt.

Am Beginn wird ein Cursor vereinbart. Des Weiteren benötigen wir eine Variable vom Datentyp `RECORD` und eine Integer-Variable, der wir das Ergebnis zuweisen werden. Vor dem Ausführen der Schleife wird der Cursor geöffnet und an den SQL-Befehl gebunden. Die Schleife wird so lange ausgeführt, bis keine Datensätze mehr gefunden werden. Bei jedem Schleifendurchlauf wird der aktuelle Wert

zur Zwischensumme addiert. Nach dem Schließen des Cursors wird das Ergebnis zurück gegeben.

Nach dem Einfügen der Funktion in die Datenbank können Sie diese leicht testen:

```
buch=# SELECT summe();
NOTICE:  value - 19
NOTICE:  value - 43
NOTICE:  value - 32
NOTICE:  value - 49
NOTICE:  value - 48
 summe
-------
    191
(1 row)
```

Für jeden Datensatz wird eine Nachricht ausgegeben. Am Ende wird die Summe angezeigt.

5.1.6 Effizienzsteigerung

In vielen Fällen kann es sinnvoll sein, Cachingfunktionen von PostgreSQL zu nutzen. Zu diesem Zwecke stellt die Datenbank zwei einfache Möglichkeiten zur Verfügung, die beim Anlegen einer Funktion verwendet werden können.

Das folgende Beispiel zeigt beide von PostgreSQL zur Verfügung gestellten Möglichkeiten:

```
CREATE OR REPLACE FUNCTION summe (int4, int4) RETURNS int4 AS '
        BEGIN
                RETURN $1 + $2;
        END;
' LANGUAGE 'plpgsql' WITH (isstrict, iscachable);
```

`iscachable` sorgt dafür, dass die Ergebnisse einer Funktion gecacht werden und gleiche Inputparameter zu gleichen Ergebnissen führen. `isstrict` sorgt dafür, dass `NULL` zurückgegeben wird, sofern ein Inputparameter `NULL` ist — beide Argumente sind optional.

5.2 Trigger

Wenn Sie sich schon immer gewünscht haben, dass gewisse Dinge einfach automatisch passieren, dann sind Trigger wohl genau das Richtige für Sie. Da eine Operation nicht von Anfang an automatisch abläuft, sind einige Vorarbeiten zu leisten, die wir in diesem Abschnitt erläutern werden.

Heutzutage unterstützt jede moderne Datenbank diverse Möglichkeiten, Trigger zu implementieren und effizient zu benutzen. PostgreSQL darf im Reigen dieser Datenbanken natürlich nicht fehlen und bietet ausgereifte und zuverlässige Mechanismen und Möglichkeiten, die beim Arbeiten mit Triggern von Vorteil sind. Bevor wir lernen, wie Trigger implementiert werden können, sehen wir uns an, was Trigger sind und wozu sie verwendet werden können.

Ein Trigger kann als Selbstauslöser gesehen werden. Ist ein Trigger auf ein Objekt definiert, sorgt PostgreSQL automatisch dafür, dass eine bestimmte Aktion automatisch ausgeführt ist. Wenn Sie etwa einen Trigger für `INSERT`-Operationen in eine bestimmte Tabelle definieren, wird der Trigger jedesmal aufgerufen, wenn ein `INSERT` durchgeführt wird. Das ist sehr praktisch, weil man sich als Applikationsentwickler um gewisse Sachen nicht mehr kümmern muss, da sie ja ohnehin von der Datenbank erledigt werden.

5.2.1 Definieren von Triggern

In diesem Abschnitt werden Sie lernen, wie Trigger definiert werden können. Der wichtigste Befehl beim Anlegen eines Triggers ist das `CREATE TRIGGER`-Kommando. Hier ist ein Überblick über die Syntax:

```
buch=# \h CREATE TRIGGER
Command:     CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

Aus dem Listing geht hervor, dass ein Trigger für vor oder nach einem Ereignis definiert werden kann. Das bedeutet, dass Sie das Ergebnis eines Ereignisses noch beeinflussen können, wenn Sie einen `BEFORE` Trigger setzen (wir sehen nachher ein Beispiel dazu). In der Syntaxübersicht ist erkennbar, dass ein Trigger für eine Zeile oder ein Statement ausgeführt werden kann. Die Syntax für `STATEMENT` Level-Trigger wird zwar schon akzeptiert, aber diese Art von Trigger ist noch nicht implementiert. In der aktuell vorliegenden Version gibt es nur Trigger, die zeilenweise arbeiten.

Jedem Trigger ist eine Funktion zuzuordnen. Dabei muss beachtet werden, dass Funktionen, die für Trigger implementiert werden, den Rückgabewert `OPAQUE` haben müssen (`OPAQUE` bedeutet ungültig). Das ist sehr wichtig, da eine Funktion sonst nicht verwendet werden kann.

Das nächste Beispiel zeigt einen einfachen Trigger. Ziel ist es, dass alle `INSERT`s in der Tabelle `messwert` protokolliert werden. Zu diesem Zwecke legen wir eine neue Tabelle und eine Funktion an:

```
CREATE TABLE logging (zeitpunkt timestamp, code text);

CREATE OR REPLACE FUNCTION logentry() RETURNS opaque AS '
        BEGIN
                INSERT INTO logging VALUES (now(), ''messwert'');
                RETURN NEW;
        END;
' LANGUAGE 'plpgsql';
```

Die Funktion `logentry()` können wir nun als Basis für den Trigger verwenden, der im folgenden Listing zu sehen ist:

```
CREATE TRIGGER trig_messwert AFTER INSERT
        ON messwert FOR EACH ROW EXECUTE PROCEDURE logentry();
```

Der Trigger wird nach jeder `INSERT`-Operation automatisch ausgeführt:

```
buch=# INSERT INTO messwert VALUES ('21:22:49', 47);
INSERT 3050589 1
```

Der Befehl hat eine Zeile in `messwert` eingefügt und eine Zeile Logging-Informationen produziert:

```
buch=# SELECT * FROM messwert WHERE wert = 47;
   zeit   | wert
----------+------
 21:22:49 |   47
(1 row)

buch=# SELECT * FROM logging;
            zeitpunkt            |   code
---------------------------------+----------
 2002-04-13 09:38:55.351684+02 | messwert
(1 row)
```

Das Protokollieren von Änderungen in Ihrer Datenbank ist in vielen Fällen eine wichtige Aufgabe und sollte bei allen wichtigen IT-Projekten ins Auge gefasst werden.

Nachdem Sie wissen, wie Trigger angelegt werden können, wird es Zeit zu erfahren, wie Trigger wieder gelöscht werden können. Zum Löschen eines Triggers kann der Befehl `DROP TRIGGER` verwendet werden:

```
buch=# \h DROP TRIGGER
Command:     DROP TRIGGER
Description: remove a trigger
Syntax:
DROP TRIGGER name ON table
```

Versuchen wir also, unseren Trigger zu löschen:

```
1 buch=# DROP TRIGGER trig_messwert ON messwert;
2 DROP
```

Sofern kein Fehler aufgetreten ist, haben wir den Trigger sauber aus der Datenbank entfernt.

Wenn Sie einen Trigger löschen, wird die Funktion, auf der der Trigger basiert, natürlich nicht gelöscht:

```
1 buch=# SELECT logentry();
2  logentry
3 ──────────
4
5 (1 row)
```

Auch hier existiert die Funktion noch.

5.2.2 Vordefinierte Variablen

Beim Arbeiten mit Triggern und PL/pgSQL gibt es eine ganze Reihe vordefinierter Variablen, die das Leben ungemein erleichtern. Beim praktischen Arbeiten mit Triggern werden Sie sehr schnell sehen, wie oft Sie auf diese Variablen zurückgreifen können und welche Möglichkeiten sich dadurch bieten.

Das folgende Listing enthält eine Übersicht über alle in der vorliegenden Version definierten Variablen:

`NEW`: Entspricht einer Variable vom Datentyp `RECORD`, die die neue Zeile in der Datenbank enthält. Diese Variable ist nur im Falle von `INSERT`- und `UPDATE`-Operationen definiert.

`OLD`: Entspricht einer Variable vom Datentyp `RECORD`, die die alte Zeile in der Datenbank enthält. Diese Variable ist nur im Falle von `UPDATE`- und `DELETE`-Operationen definiert.

`TG_NAME`: Enthält den Namen des aktuellen Triggers.

`TG_WHEN`: Enthält `BEFORE` oder `AFTER`.

`TG_LEVEL`: Enthält `ROW` oder `STATEMENT`. Derzeit sind nur `ROW` Level-Trigger implementiert.

`TG_OP`: Enthält den Befehl, für den der Trigger ausgeführt wird (`INSERT`, `UPDATE` oder `DELETE`).

`TG_RELID`: Enthält die Object ID der Tabelle, die den Trigger gestartet hat.

TG_RELNAME: Enthält den Namen der Tabelle, die den Trigger gestartet hat.

TG_NARGS: Enthält die Zahl der Argumente.

TG_ARGV: Enthält die Liste der Argumente. Die Indizierung des Arrays beginnt mit 0.

Betrachten wir ein Beispiel, das alle vordefinierten Variablen als NOTICE ausgibt:

```
1  CREATE OR REPLACE FUNCTION logentry() RETURNS opaque AS '
2          DECLARE
3                  i       int4;
4          BEGIN
5                  RAISE NOTICE ''NEW: % %'', NEW.zeit, NEW.wert;
6                  RAISE NOTICE ''OLD: % %'', OLD.zeit, OLD.wert;
7                  RAISE NOTICE ''TG_NAME: %'', TG_NAME;
8                  RAISE NOTICE ''TG_WHEN: %'', TG_WHEN;
9                  RAISE NOTICE ''TG_LEVEL: %'', TG_LEVEL;
10                 RAISE NOTICE ''TG_OP: %'', TG_OP;
11                 RAISE NOTICE ''TG_RELID: %'', TG_RELID;
12                 RAISE NOTICE ''TG_RELNAME: %'', TG_RELNAME;
13                 RAISE NOTICE ''TG_NARGS: %'', TG_NARGS;
14
15                 FOR i IN 0 .. TG_NARGS LOOP
16                         RAISE NOTICE ''TG_ARGV[i]: %'', TG_ARGV[i];
17                 END LOOP;
18                 RETURN NEW;
19         END;
20 ' LANGUAGE 'plpgsql';
21
22 CREATE TRIGGER trig_messwert AFTER INSERT
23         ON messwert FOR EACH ROW EXECUTE PROCEDURE logentry();
```

Wenn wir einen Datensatz in die Tabelle messwert einfügen, werden alle vordefinierten Variablen ausgegeben:

```
1  buch=# INSERT INTO messwert VALUES ('21:22:49', 47);
2  NOTICE:  NEW: 21:22:49 47
3  NOTICE:  OLD: <NULL> <NULL>
4  NOTICE:  TG_NAME: trig_messwert
5  NOTICE:  TG_WHEN: AFTER
6  NOTICE:  TG_LEVEL: ROW
7  NOTICE:  TG_OP: INSERT
8  NOTICE:  TG_RELID: 3042424
9  NOTICE:  TG_RELNAME: messwert
10 NOTICE:  TG_NARGS: 0
11 NOTICE:  TG_ARGV[i]: <OUT_OF_RANGE>
12 INSERT 3050632 1
```

Wie in der Übersicht bereits angedeutet, sind gewisse Variablen bei bestimmten Operationen leer. Im vorliegenden Beispiel ist das bei der Variable `OLD` so, die im Falle von `INSERT` nicht definiert ist. Da wir keine Argumente übergeben haben, ist auch die Variable `TG_ARGV` leer.

Vielfach fällt den Variablen `NEW` und `OLD` besondere Bedeutung zu. Wenn Sie die Eingabewerte einer Operation verändern möchten, bevor sie tatsächlich in die Datenbank eingefügt werden, sind `NEW` und `OLD` von besonderer Bedeutung. Sehen wir uns das nächste Beispiel an (bitte löschen Sie vor dem Testen alle bestehenden Trigger):

```
CREATE OR REPLACE FUNCTION modify() RETURNS opaque AS '
        BEGIN
                NEW.wert = round(NEW.wert, -1);
                RETURN NEW;
        END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trig_messwert BEFORE INSERT
        ON messwert FOR EACH ROW EXECUTE PROCEDURE modify();
```

Die Funktion `modify()` rundet den Wert in der zweiten Spalte der Tabelle auf Zehner genau und weist das Ergebnis der Variablen `NEW` zu. Diesmal wollen wir, dass der Trigger vor dem Einfügen in die Tabelle ausgeführt wird. Da wir die Eingabewerte vor dem Einfügen mittels Trigger modifizieren, werden die Werte auch gerundet in die Datenbank eingefügt, wie dieses Beispiel zeigt:

```
buch=# INSERT INTO messwert VALUES ('21:23:12', '132');
INSERT 3050638 1
buch=# SELECT * FROM messwert WHERE wert > 100;
   zeit   | wert
----------+------
 21:23:12 |  130
(1 row)
```

Aus 132 ist 130 geworden.

5.3 Rules

Oft erfüllen Trigger alleine nicht alle Anforderungen. In solchen Fällen kann es sinnvoll sein, so genannte Rules (Regeln) zu verwenden. Bei Triggern werden zusätzliche Operationen durchgeführt. Im Gegensatz dazu erlauben es Rules, Operationen statt anderer Operationen durchzuführen. Mithilfe von Rules wird es möglich, Operationen einfach umzudefinieren. Deshalb sind Rules ein mächtiges

Instrument, das jedoch mit Vorsicht eingesetzt werden sollte. Wir empfehlen, Rules nur dann zu verwenden, wenn Sie sich über deren Konsequenzen im Klaren sind.

Der Befehl zum Anlegen einer Regel heißt `CREATE RULE` und ist im folgenden Listing dargestellt:

```
buch=# \h CREATE RULE
Command:     CREATE RULE
Description: define a new rewrite rule
Syntax:
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] action

where action can be:

NOTHING
|
query
|
( query ; query ... )
|
[ query ; query ... ]
```

Nehmen wir an, Sie wollen den `INSERT`-Befehl so umdefinieren, dass er eine `SELECT`-Operation durchführt. Die Lösung für das Problem könnte etwa so aussehen (bitte entfernen Sie zuvor alle Trigger von der Tabelle):

```
CREATE RULE rule_insert AS ON INSERT TO messwert
        DO INSTEAD (
        SELECT * FROM messwert
);
```

Wenn Sie jetzt eine `INSERT`-Operation ausführen, werden gar seltsame Dinge passieren:

```
buch=# INSERT INTO messwert VALUES ('21:24:09', 99);
   zeit   | wert
----------+------
 21:17:03 |   19
 21:18:12 |   43
 21:19:09 |   32
 21:20:17 |   49
 21:21:29 |   48
 21:22:49 |   47
 21:23:12 |  130
(7 rows)
```

Wenn PostgreSQL die Abfrage ausführt, wird sie intern auf ein `SELECT`-Statement umgeschrieben, genauso wie es von der Regel vorgeschrieben ist. Wir müssen nicht erwähnen, welche Macht Regeln dem Benutzer in die Hände legen und dass man damit auch sehr viel Unfug treiben kann.

Um eine Regel wieder zu löschen, können Sie den Befehl `DROP RULE` verwenden:

```
buch=# \h DROP RULE
Command:     DROP RULE
Description: remove a rewrite rule
Syntax:
DROP RULE name [, ...]
```

Im Falle unserer Regel würde der Befehl wie folgt aussehen:

```
buch=# DROP RULE rule_insert;
DROP
```

Wenn Sie eine Regel definieren wollen, die bestimmte Statements temporär abschaltet, so ist auch das ohne Probleme möglich, wie es am folgenden Beispiel demonstriert ist:

```
CREATE RULE rule_insert AS ON INSERT TO messwert
        DO INSTEAD NOTHING;
```

Mithilfe von `NOTHING` können Sie PostgreSQL mitteilen, dass keine Operation durchgeführt werden soll. Das nächste Beispiel beweist, wie die Regel arbeitet:

```
buch=# INSERT INTO messwert VALUES ('21:24:09', 99);
buch=# SELECT * FROM messwert WHERE wert=99;
 zeit | wert
------+------
(0 rows)
```

Die Datenbank hat keinen Wert in die Tabelle eingefügt.

Oft kann ein derartiges Verhalten erwünscht sein. Nehmen wir etwa an, Sie wollen in Ihre Applikation so etwas wie einen Testmodus einbauen — mithilfe von Rules ist das relativ einfach möglich.

5.4 Mathematische Funktionen

PL/pgSQL wird zur Implementierung zusätzlicher Features für PostgreSQL verwendet. In vielen Fällen ist es daher notwendig und sinnvoll, einfache mathematische Funktionen zu implementieren. In diesem Abschnitt werden wir Ihnen einige einfache Beispiele zeigen, die Sie inspirieren sollen, selbst aktiv zu werden und Funktionen zu schreiben.

5.4.1 Hyperbolische Funktionen

Beginnen wir mit einem Set einfacher hyperbolischer Winkelfunktionen. Vielen werden Sinus, Cosinus und Tangens noch aus der Schule bekannte Begriffe sein. Alle diese Funktionen sind am Kreis definiert und finden in verschiedensten Bereichen Anwendung. Zusätzlich zu Kreisfunktionen gibt es auch noch so genannte hyperbolische Funktionen, die allesamt über die Hyperbel definiert sind. Wie bei Kreisen gibt es auch hier Sinus, Cosinus und Tangens. Bei hyperbolischen Funktionen heißen diese hier jedoch Sinus hyperbolicus, Cosinus hyperbolicus und Tangens hyperbolicus. Sehen wir uns also an, wie diese Funktionen mithilfe von PL/pgSQL implementiert werden können:

```
CREATE OR REPLACE FUNCTION sinh(numeric) RETURNS numeric AS
'
        BEGIN
                RETURN (exp($1) - exp(-$1))/2;
        END;
' LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION cosh(numeric) RETURNS numeric AS
'
        BEGIN
                RETURN (exp($1) + exp(-$1))/2;
        END;
' LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION tanh(numeric) RETURNS numeric AS
'
        BEGIN
                RETURN (sinh($1) / cosh($1));
        END;
' LANGUAGE 'plpgsql';
```

Da die hier implementierten Funktionen allesamt mithilfe der Eulerschen Zahl berechnet werden können, sind die Funktionen relativ einfach. Die Quintessenz der gezeigten Beispiele liegt in der Behandlung der Eulerschen Zahl. Hier sehen Sie, wie diese extrahiert werden kann:

```
buch=# SELECT exp(1.0);
       exp
------------------
 2.71828182845905
(1 row)
```

Mithilfe der Funktion exp kann die Eulersche Zahl potenziert werden. Wenn wir die Zahl hoch 1 rechnen, kommt logischerweise die Zahl selbst als Ergebnis heraus.

Bei dieser Berechnung ist besonderer Wert auf den richtigen Datentyp zu legen. Wenn Sie dieselbe Berechnung mit Integer-Werten durchführen, kommt es zu einem Fehler, wie im folgenden Listing:

```
buch=# SELECT exp(1);
ERROR:  Function 'exp(int4)' does not exist
        Unable to identify a function that satisfies the given
        argument types
        You may need to add explicit typecasts
ERROR:  Function 'exp(int4)' does not exist
        Unable to identify a function that satisfies
        the given argument types
        You may need to add explicit typecasts
```

Die Umkehrfunktion zu `exp` nennt sich `ln` und dient zur Berechnung des natürlichen Logarithmus. Der natürliche Logarithmus der Eulerschen Zahl ist 1 (aufgrund der fehlenden Nachkommagenauigkeit in diesem Falle fast 1):

```
buch=# SELECT ln(2.71828);
         ln
--------------------
 0.999999327347282
(1 row)
```

Nach diesem kleinen Exkurs in die Welt der eingebauten Funktionen können wir unsere PL/pgSQL-Funktionen ausprobieren:

```
buch=# SELECT sinh(3.1415);
      sinh
----------------
 11.5476653707
(1 row)

buch=# SELECT cosh(3.1415);
      cosh
----------------
 11.5908832931
(1 row)

buch=# SELECT tanh(3.1415);
           tanh
-------------------------
 0.99627138663370159686
(1 row)
```

Fraglos funktioniert alles einwandfrei und zuverlässig.

5.4.2 Rekursionen mit PL/pgSQL

Viele Probleme in der Mathematik und der Technik lassen sich durch so genannte Rekursionen lösen. Eine rekursive Funktion ist eine, die sich selbst aufruft. In vielen Fällen werden Rekursionen für Probleme eingesetzt, die mithilfe von »Divide et Impere« Algorithmen gelöst werden. »Divide et Impere« (Teile und herrsche) bedeutet, dass ein komplexes Problem auf einfachere Teilprobleme zurückgeführt wird. Als Beispiel könnte man die Berechnung einer Summe heranziehen: Das Summieren von zehn Zahlen ist eine komplexe Ausgabe; wenn es Ihnen aber gelingt, das Problem auf das Summieren von zwei Zahlen zurückzuführen, haben Sie das Problem geteilt und de facto gelöst. Die Berechnung der Summe aus zehn Zahlen ist einfach das mehrmalige, rekursive Berechnen der Summe aus zwei Zahlen.

In diesem Abschnitt werden wir uns einfachen, rekursiven Funktionen zuwenden, die solche Probleme mithilfe von PostgreSQL lösen.

Das klassische Beispiel für Rekursionen sind Fibonacci-Zahlen. Mithilfe von Fibonacci-Zahlen lässt sich die Vermehrung von Hasen beschreiben, was jedoch an dieser Stelle nicht das Thema ist.

Hier finden Sie die Funktion:

```
1  CREATE OR REPLACE FUNCTION fib (int4) RETURNS int4 AS '
2          DECLARE
3                  n       ALIAS FOR $1;
4          BEGIN
5                  IF      (n = 1) OR (n = 2) THEN
6                          RETURN 1;
7                  ELSE
8                          RETURN fib(n - 1) + fib(n - 2);
9                  END IF;
10         END;
11 ' LANGUAGE 'plpgsql';
```

Die Berechnung der Zahlenfolge geht sehr einfach vonstatten. Die Fibonacci-Zahl von n ist die Summe aus den Fibonacci Zahlen von $(n-1)$ und $(n-2)$. In unserem Beispiel von $n = 1$ und $n = 2$ ist die Fibonacci Zahl 1.

Die Funktion in der Box zeigt, wie eine Funktion zur Berechnung einer Fibonacci Zahl implementiert werden kann. Sofern n größer als 2 ist, ruft sich die Funktion selbst auf. Das macht sie solange, bis das Ergebnis des Teilproblems gelöst werden kann. Schließlich wird das Ergebnis schrittweise zusammengebaut und ausgegeben.

Das nächste Listing zeigt, wie die Funktion aufgerufen werden kann:

```
test=# SELECT fib(1), fib(2), fib(3), fib(4), fib(5), fib(6);
 fib | fib | fib | fib | fib | fib
-----+-----+-----+-----+-----+-----
   1 |   1 |   2 |   3 |   5 |   8
(1 row)
```

Stellen Sie sicher, dass die Funktion nicht mit zu großen Zahlen aufgerufen wird, da die Berechnung sonst durchaus eine gewisse Zeit in Anspruch nehmen kann.

5.4.3 Iterationen mit PL/pgSQL

Jedes Problem, das mittels Rekursion gelöst werden kann, ist prinzipiell auch mithilfe so genannter Iterationen lösbar. In der Regel sind iterative Funktionen ein wenig schneller und benötigen weniger Speicher. Im Gegensatz dazu sind rekursive Funktionen kürzer und in vielen Fällen leichter zu debuggen.

In diesem Beispiel wird gezeigt, wie das Problem der Fibonacci-Zahlen mithilfe einer Iteration gelöst werden kann:

```
CREATE OR REPLACE FUNCTION fib_it (int4) RETURNS int4 AS '
        DECLARE
                pos1    int4;
                pos2    int4;
                cum     int4;
                i       int4;
                n       ALIAS FOR $1;
        BEGIN
                pos1 := 1;
                pos2 := 0;

                IF      (n = 1) OR (n = 2) THEN
                        RETURN 1;
                ELSE
                        cum := pos1 + pos2;
                        FOR i IN 3 .. n LOOP
                                pos2 := pos1;
                                pos1 := cum;
                        END LOOP;
                        RETURN cum;
                END IF;
        END;
' LANGUAGE 'plpgsql';
```

Wie versprochen ist die Funktion ein wenig länger, aber dafür auch ein wenig effizienter. Prüfen Sie, ob das korrekte Ergebnis berechnet wird:

```
buch=# SELECT fib(1), fib(2), fib(3), fib(4), fib(5), fib(6);
 fib | fib | fib | fib | fib | fib
-----+-----+-----+-----+-----+-----
   1 |   1 |   2 |   3 |   5 |   8
(1 row)
```

Die Ergebnisse der beiden Funktionen sind identisch. Es ist an Ihnen zu entscheiden, welche der beiden Varianten Sie bevorzugen und welcher der beiden Ansätze für Ihre Problemstellung am besten passt.

Rekursionen und Interationen gehören beinahe zum Grundrüstzeug eines jeden Programmieres und Sie sollten mit den wesentlichen und grundlegenden Ansätzen vertraut sein.

Kapitel 6

C/C++

Obwohl es eine Vielzahl von Programmiersprachen gibt, sind C und C++ in vielen Bereichen nach wie vor nahezu alleine auf weiter Flur. Die Gründe dafür liegen auf der Hand: C und C++ gehören zu den effizientesten und schnellsten Programmiersprachen, was die Laufzeit der Programme betrifft, und sie haben auch noch andere Vorteile. Durch den ANSI-Standard sind C-Programme relativ leicht portierbar und auf nahezu allen Systemen gibt es heutzutage einen C-Compiler.

Da PostgreSQL selbst in C geschrieben ist, verfügt es über eine mächtige C-Schnittstelle, die wir in diesem Abschnitt etwas genauer besprechen werden.

6.1 Die C-Schnittstelle

Die C-Schnittstelle von PostgreSQL ist die standardmäßig verwendete Schnittstelle. Viele Schnittstellen anderer Programmiersprachen verwenden direkt die C-Schnittstelle, um die eigenen Funktionalitäten darauf aufzubauen. Aus diesem Grund ist die sie von besonderer Wichtigkeit.

6.1.1 Datenbankverbindungen

Wie bei allen Systemen muss zuallererst eine Verbindung zur Datenbank hergestellt werden.

Im Falle von C kann die Verbindung zu einer Datenbank sehr leicht mithilfe der Funktion `PQconnectdb` hergestellt werden. Das folgende Beispiel zeigt, wie das vor sich geht:

```
1  #include <stdio.h>
2  #include "/usr/local/postgresql/include/libpq-fe.h"
3
4  int main(int argc, char **argv)
5  {
6          PGconn  *dbhandle;
7          dbhandle = PQconnectdb("dbname=buch user=postgres");
8          if      (PQstatus(dbhandle) == CONNECTION_BAD)
9          {
10                 char *message = PQerrorMessage(dbhandle);
11                 printf("keine Verbindung: %s \n", message);
12                 exit(1);
13         }
14         printf("Verbindung ok\n");
15         PQfinish(dbhandle);
16         return 0;
17 }
```

Am Beginn des Programmes binden wir einige Header-Files ein. Diese Header-Files enthalten alle notwendigen Prototypen. Im Hauptprogramm legen wir eine

Variable vom Typ `PGconn` an, die den Datenbankhandle enthalten wird. Anschließend verbinden wir uns zur Datenbank. Dabei übergeben wir den Connectstring an die Funktion. Dieser Connectstring enthält alle Parameter der Verbindung. Die folgenden Parameter werden akzeptiert:

`dbname`: Name der Datenbank

`host`: IP-Adresse oder Name des Datenbankservers

`port`: TCP-Port, auf den PostgreSQL hört

`tty`: TTY

`user`: Benutzername

`password`: Passwort

`options`: Optionen

Nach dem Versuch, die Verbindung zu erstellen, prüfen wir, ob der Datenbankhandle korrekt ist. Sofern das nicht der Fall ist, fragen wir den Fehler ab, der aufgetreten ist. Den Text des Fehlers können wir sehr leicht mit `printf` ausgeben.

Am Ende der Anwendung terminieren wir die Datenbankverbindung mithilfe des Befehles `PQfinish(dbhandle)`.

Um das Programm von zuvor zu kompilieren, verwenden Sie am besten ein Makefile. Makefiles haben viele Vorteile, die vor allem bei größeren Projekten zum Tragen kommen. An dieser Stelle seien Makefiles aus Gründen der Vollständigkeit erwähnt:

```
1 INC=    -I/usr/local/postgresql/include
2 LIB=    -lpq
3
4 prog    :       main.c
5         gcc $(LIB) $(INC) main.c -o $@
```

In diesem Beispiel verwenden wir den GNU C-Compiler — Sie können aber auch jeden anderen beliebigen C-Compiler benutzen.

Versuchen wir nun die Anwendung zu kompilieren:

```
1 [hs@duron c]$ make
2 gcc -lpq -I/usr/local/postgresql/include main.c -o prog
```

Sofern es zu keinen Problemen gekommen ist, sollten Sie die Anwendung nun ausführen können:

```
1 [hs@duron c]$ ./prog
2 Verbindung ok
```

Hier hat es keinen Fehler gegeben. Betrachten wir, wie das Ergebnis einer fehlerhaften Verbindung aussehen würde:

```
[hs@duron c]$ ./prog
keine Verbindung: FATAL 1:  Database "nixda" does not exist in the
system catalog.
```

Wir haben versucht, das Programm mit einem falschen Datenbanknamen zu starten, das hat zu einem Fehler geführt, den wir aber korrekt abgefangen haben.

Wenn Sie erst einmal eine Verbindung zur Datenbank aufgebaut haben, können Sie sehr leicht einige Verbindungsparameter abfragen. Das folgende Beispiel zeigt, wie das funktioniert:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        /* Verbinden zur Datenbank */
        PGconn  *dbhandle;
        dbhandle = PQconnectdb("dbname=buch user=postgres
                        host=localhost");
        if      (PQstatus(dbhandle) == CONNECTION_BAD)
        {
                printf("Fehler in der Verbindung\n");
                exit(1);
        }

        /* Abfrage zur Verbindung */
        printf("Status: %s\n", PQstatus(dbhandle));
        printf("Name der Datenbank: %s\n", PQdb(dbhandle));
        printf("Benutzer: %s\n", PQuser(dbhandle));
        printf("Passwort: %s\n", PQpass(dbhandle));
        printf("Host: %s\n", PQhost(dbhandle));
        printf("Port: %s\n", PQport(dbhandle));
        printf("TTY: %s\n", PQtty(dbhandle));
        printf("Optionen: %s\n", PQoptions(dbhandle));

        PQfinish(dbhandle);
        return 0;
}
```

Am Beginn des Programms wird wieder eine Verbindung zur Datenbank hergestellt. Anschließend analysieren wir den Datenbankhandle und geben alle Informationen mithilfe von `printf` aus. Die Namen der benutzten Funktionen sind weitgehend selbsterklärend.

Versuchen wir das Programm zu kompilieren und auszuführen:

```
[hs@duron c]$ make; ./prog
gcc -lpq -I/usr/local/postgresql/include main.c -o prog
Status: (null)
Name der Datenbank: buch
Benutzer: postgres
Passwort:
Host: localhost
Port: 5432
TTY:
Optionen:
```

Die Informationen werden ausgegeben.

6.1.2 Daten modifizieren

Nach diesem Überblick über Datenbankverbindungen lernen wir in diesem Abschnitt, wie Daten mithilfe von C-Programmen modifiziert werden können. Wir werden den folgenden Datenbestand verwenden:

```
CREATE TABLE t_produkt (
        id      int4,
        edvnr   int4,
        name    text,
        price   numeric(9,2)
);

COPY t_produkt FROM stdin USING DELIMITERS ';';
1;435353;Hamburger;2.69
2;234234;Cheeseburger;2.99
3;234999;Gurkensalat;2.49
\.
```

Jetzt wollen wir uns einem Beispiel widmen, das zeigt, wie Daten gelöscht werden können. Das Löschen von Daten ist eine wichtige Funktion und kann sehr einfach durchgeführt werden, da keine größeren Datenmengen abgefragt werden müssen. Werfen wir also einen Blick auf den Code:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        PGconn          *dbhandle;
        PGresult        *result;
        char            abfrage[1024];
```

```
9
10          dbhandle = PQconnectdb("dbname=buch user=postgres
11                                  host=localhost");
12          if      (PQstatus(dbhandle) == CONNECTION_BAD)
13          {
14                  printf("Fehler in der Verbindung\n");
15                  exit(1);
16          }
17
18          sprintf(abfrage, "DELETE FROM t_produkt WHERE price < '2.50'");
19          result = PQexec(dbhandle, abfrage);
20          if      (result != NULL || PQresultStatus(result) != PGRES_COMMAND_OK)
21          {
22                  printf("%s", PQerrorMessage(dbhandle));
23          }
24
25          PQfinish(dbhandle);
26          return 0;
27  }
```

Am Beginn des Programms legen wir einige Variablen an, die wir an späterer Stelle noch benötigen werden. Nach dem Herstellen einer Verbindung zur Datenbank weisen wir den auszuführenden Befehl auf Abfrage zu. Mithilfe der Funktion `PQexec` kann der Befehl dann ausgeführt werden. Um nicht auf böse Überraschungen zu treffen, fangen wir noch den Fehler ab und beenden das Programm. Wichtig zu bemerken ist auch der Unterschied zwischen `PGRES_COMMAND_OK` und `PGRES_TUPLES_OK`: Erstere Konstante prüft, ob ein Kommando ausgeführt worden ist und ob es auch keine Daten zurückgegeben hat. `PGRES_TUPLES_OK` wird dann retourniert, wenn Tuples zurückgegeben werden.

Nachdem Sie das Programm ausgeführt haben, wird das Ergebnis wie folgt aussehen:

```
1  buch=# SELECT * FROM t_produkt;
2   id | edvnr  |     name      | price
3  ----+--------+---------------+-------
4    1 | 435353 | Hamburger     |  2.69
5    2 | 234234 | Cheeseburger  |  2.99
6  (2 rows)
```

6.1.3 Einfache Abfragen

Nachdem Sie gelernt haben, wie Daten bequem modifiziert werden können, wenden wir uns nun einigen Datenbankabfragen zu. Ziel dieses Kapitels ist es, zu erkennen, welche Möglichkeiten sich bieten, Daten aus der Datenbank abzufragen.

Um Ihnen eine theoretische Übersicht zu ersparen, beginnen wir am besten gleich mit einem Beispiel, das alle Daten in der vorliegenden Tabelle abfragt und am Bildschirm ausgibt:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        PGconn          *dbhandle;
        PGresult        *result;
        char            abfrage[1024];
        int             zeilen;
        int             spalten;
        int             i;

        /* Verbinden zur Datenbank */
        dbhandle = PQconnectdb("dbname=buch user=postgres
                        host=localhost");
        if      (PQstatus(dbhandle) == CONNECTION_BAD)
        {
                printf("Fehler in der Verbindung\n");
                exit(1);
        }

        /* Durchführen der Abfrage */
        sprintf(abfrage, "SELECT edvnr, name, price FROM t_produkt");
        result = PQexec(dbhandle, abfrage);
        if      (!result || PQresultStatus(result) != PGRES_TUPLES_OK)
        {
                printf("%s", PQerrorMessage(dbhandle));
                exit(1);
        }

        /* Ausgeben des Ergebnisses */
        zeilen = PQntuples(result);
        spalten = PQnfields(result);
        printf("%i Zeilen; %i Spalten\n\n", zeilen, spalten);

        for     (i = 0; i < zeilen; i++)
        {
                /* Abarbeiten der Spalten */
                printf("EdvNr.: %s\n", PQgetvalue(result, i, 0));
                printf("Name: %s\n", PQgetvalue(result, i, 1));
                printf("Preis: %s\n\n", PQgetvalue(result, i, 2));
        }

        PQfinish(dbhandle);
        return 0;
}
```

Wie gewohnt werden am Beginn des Programms Variablen angelegt und eine Verbindung zur Datenbank wird hergestellt — dieses Prozedere sollte Ihnen schon bekannt sein.

Des Weiteren führen wir eine Abfrage aus. Hier prüfen wir auf `PGRES_TUPLES_OK`. Sofern diese Bedingung erfüllt ist, stehen Tuples zur Verfügung, die abgefragt werden können. Um einige Informationen über das Ergebnis zu haben, fragen wir die Anzahl der Spalten und die Anzahl der Zeilen ab. Zu diesem Zwecke stellt PostgreSQL die Funktionen `PQntuples` und `PQnfields` zur Verfügung. Danach gehen wir das Ergebnis durch. Der Einfachheit halber fragen wir alle Ergebnisse als String ab; es steht Ihnen aber auch frei, die EDV-Nummer nach dem Abfragen direkt auf Integer zu konvertieren.

Wenn wir das Programm ausführen, erhalten wir das folgende Ergebnis:

```
[hs@duron c]$ ./prog
2 Zeilen; 3 Spalten

EdvNr.: 435353
Name: Hamburger
Preis: 5.69

EdvNr.: 234234
Name: Cheeseburger
Preis: 5.99
```

Im Beispiel haben wir gesehen, dass auf `PGRES_TUPLES_OK` abgefragt wird. Das Backend stellt noch einige andere Konstanten zur Verfügung, um den Zustand eines Ergebnisses zu erhalten. Werfen wir einen Blick auf die nächste Übersicht:

`PGRES_EMPTY_QUERY`: Eine leere Abfrage ist ausgeführt worden.

`PGRES_COMMAND_OK`: Das Statement ist korrekt ausgeführt worden.

`PGRES_TUPLES_OK`: Das Statement ist korrekt ausgeführt worden.

`PGRES_COPY_OUT`: Der `COPY OUT`-Datentransfer ist gestartet worden.

`PGRES_COPY_IN`: Der `COPY IN`-Datentransfer ist gestartet worden.

`PGRES_BAD_RESPONSE`: Die Antwort vom Server konnte nicht verstanden werden.

`PGRES_NONFATAL_ERROR`: Ein non-fatal Error ist aufgetreten.

`PGRES_FATAL_ERROR`: Die Abfrage konnte nicht ausgeführt werden (schwerwiegender Fehler).

6.1.4 Metadaten

Bei der Abfrage von Ergebnissen werden eine Reihe von Metadaten erzeugt, die für Ihr Programm von großem Interesse sein können. In diesem Abschnitt wollen wir genauer auf diese Informationen eingehen und sehen, wie Metadaten verarbeitet werden können.

Wie immer beginnen wir mit einem Beispiel:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        PGconn          *dbhandle;
        PGresult        *result;
        char            abfrage[1024];
        int             zeilen, spalten, i;

        /* Verbinden zur Datenbank */
        dbhandle = PQconnectdb("dbname=buch user=postgres
                        host=localhost");
        if      (PQstatus(dbhandle) == CONNECTION_BAD)
        {
                printf("Fehler in der Verbindung\n");
                exit(1);
        }

        /* Durchführen der Abfrage */
        sprintf(abfrage, "SELECT edvnr, name, price FROM t_produkt");
        result = PQexec(dbhandle, abfrage);
        if      (!result || PQresultStatus(result) != PGRES_TUPLES_OK)
        {
                printf("%s", PQerrorMessage(dbhandle));
                exit(1);
        }

        /* Ausgeben der Metadaten */
        spalten = PQnfields(result);
        for     (i = 0; i < spalten; i++)
        {
                printf("Name von Spalte %i: %s\n",
                                i, PQfname(result, i));
                printf("Feldgröße: %d\n", PQfsize(result, i));
                printf("Datentyp: %d\n", PQftype(result, i));
        }

        PQfinish(dbhandle);
        return 0;
}
```

Das Script versucht, den Namen der Spalte, die Größe des Feldes und den Datentyp abzufragen. Aus dem Code können Sie leicht ersehen, welcher Wert wo abgefragt wird. Betrachten wir nun den Output des Programms:

```
[hs@duron c]$ make
gcc -lpq -I/usr/local/postgresql/include main.c -o prog
[hs@duron c]$ ./prog
Name von Spalte 0: edvnr
Feldgröße: 4
Datentyp: 23
Name von Spalte 1: name
Feldgröße: -1
Datentyp: 25
Name von Spalte 2: price
Feldgröße: -1
Datentyp: 1700
```

Wichtig zu bemerken ist, dass ein Datentyp immer als Integer-Variable ausgegeben wird. Intern sind die Namen eines Datentyps nur Kosmetik — ein Datentyp wird immer über seine eindeutige Nummer referenziert (siehe Systemtabellen). Das ist wichtig und wird oft vergessen.

Die Länge eines Feldes ist nicht immer definiert. Im Falle von Textfeldern ist die Länge nicht konstant. Aus diesem Grund gibt die Datenbank −1 zurück. Bei allen Datentypen, die eine definierte Länge haben, kann diese jedoch sehr schnell abgefragt werden.

6.1.5 Binäre Cursor

Speziell beim Umgang mit großen Datenmengen haben binäre Cursor große Vorteile. Intern werden alle Daten binär gespeichert, bei der Ausgabe sind die Daten also zu konvertieren. Wenn ein Datenbestand direkt nach der Abfrage weiterverarbeitet wird, kann man sich diesen Schritt ersparen und die Daten gleich binär verwenden.

Das folgende Beispiel zeigt, wie das gemacht werden kann:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        PGconn          *dbhandle;
        PGresult        *result;
        char            abfrage[1024];
        int             *erg;
```

```
10
11          /* Verbinden zur Datenbank */
12          dbhandle = PQconnectdb("dbname=buch user=postgres
13                          host=localhost");
14          if      (PQstatus(dbhandle) == CONNECTION_BAD)
15          {
16                  printf("Fehler in der Verbindung\n");
17                  exit(1);
18          }
19
20          /* Durchführen der Abfrage */
21          sprintf(abfrage, "BEGIN");
22          result = PQexec(dbhandle, abfrage);
23          if      (!result || PQresultStatus(result) != PGRES_COMMAND_OK)
24          {
25                  printf("Fehler: %s\n", PQerrorMessage(dbhandle));
26                  exit(1);
27          }
28
29          sprintf(abfrage, "DECLARE tmp BINARY CURSOR FOR
30                  SELECT edvnr, name, price FROM t_produkt");
31          result = PQexec(dbhandle, abfrage);
32          if      (!result || PQresultStatus(result) != PGRES_COMMAND_OK)
33          {
34                  printf("Fehler 2: %s\n", PQerrorMessage(dbhandle));
35                  exit(1);
36          }
37
38          sprintf(abfrage, "FETCH %d IN tmp");
39          result = PQexec(dbhandle, abfrage);
40
41          erg = (int * ) PQgetvalue(result, 0, 0);
42          printf("Ergebnis: %d\n", *erg);
43
44          sprintf(abfrage, "CLOSE tmp; COMMIT");
45          result = PQexec(dbhandle, abfrage);
46
47          PQfinish(dbhandle);
48          return 0;
49  }
```

Nach dem Starten einer Transaktion legen wir einen binären Cursor an, dem wir den Namen `tmp` geben. Wir prüfen, ob das Anlegen des Cursors funktioniert hat und fragen die erste Zeile des Ergebnisses ab. Aus Gründen der Einfachheit und der Übersichtlichkeit fragen wir nur den ersten Satz ab und verzichten auf eine komplette Schleife. Nach der Durchführung des `FETCH`-Kommandos extrahieren wir einen Wert aus dem Resultat. Diesen Wert weisen wir einer Variable zu und

geben ihn in der darauf folgenden Zeile aus. Am Ende des Programms werden der Cursor geschlossen und die Transaktion beendet.

Folglich sind binäre Cursor sehr einfach zu implementieren und funktionieren sehr schnell. Speziell bei größeren Datenmengen können Sie auch mit spürbaren Geschwindigkeitsgewinnen rechnen.

Wenn wir das Programm starten, wird der folgende Output generiert:

```
[hs@duron c]$ make
gcc -lpq -I/usr/local/postgresql/include main.c -o prog
[hs@duron c]$ ./prog
Ergebnis: 435353
```

Wie nicht anders zu erwarten war, wird der erste Integer-Wert im Ergebnis ausgegeben.

6.1.6 BLOBs

PostgreSQLs BLOB (Binary Large Object)-Interface ist eine komfortable Schnittstelle zur Verwaltung von binären Objekten in der Datenbank. Der Vorteil von BLOBs ist, dass Sie ein de facto unbegrenzt großes File in einer PostgreSQL-Datenbank speichern können. Die zur Bearbeitung von BLOBs verwendeten Schnittstellen sind den C-Funktionen zur Verarbeitung von Dateien ähnlich und können leicht erlernt werden. Im Prinzip stellt ein BLOB alles zur Verfügung, was das Filesystem auch zur Verfügung stellt. Eine BLOB kann wie eine Datei mittels `read`- und `write`-Operationen modifiziert werden. PostgreSQL verfügt hier de facto über eine Schnittstelle zur Verwaltung von Files in der Datenbank; die Macht von BLOBs orientiert sich an Filesystemen.

In diesem Abschnitt werden wir uns näher mit C und BLOBs beschäftigen. Beginnen wir wie immer mit einem Beispiel, das die grundlegenden Funktionsweisen erklären soll:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"

int main(int argc, char **argv)
{
        PGconn          *dbhandle;
        PGresult        *result;
        Oid             objectid;
        int             ostatus;

        /* Verbinden zur Datenbank */
        dbhandle = PQconnectdb("dbname=buch user=postgres
```

```
13                          host=localhost");
14          if      (PQstatus(dbhandle) == CONNECTION_BAD)
15          {
16                  printf("Fehler in der Verbindung\n");
17                  exit(1);
18          }
19
20          /* Erzeugen eines BLOBs */
21          result = PQexec(dbhandle, "BEGIN");
22          objectid = lo_import(dbhandle, "/etc/resolv.conf");
23          ostatus = lo_export(dbhandle, objectid, "/tmp/resolv.conf");
24          result = PQexec(dbhandle, "COMMIT");
25
26          printf("Object ID: %i\n", objectid);
27          printf("Status: %i\n", ostatus);
28
29          PQfinish(dbhandle);
30          return 0;
31  }
```

Im Abschnitt mit den Variablendeklarationen werden zwei neue Variablen angelegt. Die Variable `objectid` werden wir zum Speichern der Object IDs verwenden. Die Variable `ostatus` werden wir verwenden, um den Rückgabewert einer Funktion zu prüfen.

In diesem Beispiel geschieht Folgendes: Nachdem wir eine Verbindung zur Datenbank hergestellt haben, starten wir eine Transaktion. Dann importieren wir die Datei `/etc/resolv.conf` in die Datenbank. Beim Import erhalten wir die Object ID des neuen Objektes als Rückgabewert. Nach dem Import exportieren wir das File wieder. In diesem Fall müssen wir die Funktion `lo_export` verwenden, die genau drei Parameter akzeptiert. Der erste Parameter enthält den Datenbankhandle. Der zweite Parameter dient zur Weitergabe der Object ID, während der dritte Parameter das Ausgabefile definiert.

Nach dem Export schließen wir die Transaktion ab, geben einige Werte aus und beenden das Programm. Sehen wir uns also an, wie das Programm ausgeführt werden kann und was passiert:

```
1 [hs@duron c]$ make
2 gcc -lpq -I/usr/local/postgresql/include main.c -o prog
3 [hs@duron c]$ ./prog
4 Object ID: 405509
5 Status: 1
```

In der ersten Zeile wird die Object ID des neuen Objektes ausgegeben. Die zweite Zeile enthält den Status des Exports. Da es offensichtlich zu keinen Problemen gekommen ist, enthält die Datei `/tmp/resolv.conf` auch tatsächlich Daten:

```
[hs@duron c]$ cat /etc/resolv.conf
nameserver 195.34.133.10
nameserver 194.152.178.1
nameserver 195.34.133.11
```

Wie nicht anders zu erwarten war, ist die Datei gut gefüllt.

PostgreSQL stellt eine Reihe von Funktionen zum Umgang mit BLOBs zur Verfügung. Das folgende Listing enthält eine Zusammenstellung der wichtigsten Funktionalitäten der C-Schnittstelle für binäre Objekte:

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
int lo_close(PGconn *conn, int fd);
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
int lo_write(PGconn *conn, int fd, char *buf, size_t len);
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
Oid lo_creat(PGconn *conn, int mode);
int lo_tell(PGconn *conn, int fd);
int lo_unlink(PGconn *conn, Oid lobjId);
Oid lo_import(PGconn *conn, const char *filename);
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

Wenn Sie mit diesen Funktionen arbeiten wollen, ist zusätzlich noch die `libpq/libpq-fs.h` einzubeziehen.

6.2 Weitere Funktionen

Die C-Schnittstelle von PostgreSQL stellt ein Fülle von weiteren Funktionen zur Verfügung, die weit über das Gezeigte hinausgehen. Viele dieser Funktionen setzen bereits bestimmte Kenntnisse über die Funktionsweise des Backends voraus und würden den Rahmen dieses Buches sprengen.

Für weitere Informationen über die C-Schnittstelle empfehlen wir das exzellent dokumentierte Header File, dem Sie alle notwendigen Informationen leicht entnehmen können.

6.3 PgEasy

Da die Standard Schnittstelle von PostgreSQL durchaus eine gewisse Komplexität in sich birgt, wird von PostgreSQL noch eine zweite Schnittstelle zur Verfügung gestellt, die ein wenig einfacher zu handhaben ist und einen Gutteil der Komplexität versteckt. In diesem Abschnitt werden wird auf diese Schnittstelle eingehen und die wichtigsten Funktionen erörtern.

Beginnen wir wieder mit einem einfachen Beispiel:

```
#include <stdio.h>
#include "/usr/local/postgresql/include/libpq-fe.h"
#include "/usr/local/postgresql/include/libpgeasy.h"

int main(int argc, char **argv)
{
        PGconn              *dbhandle;
        PGresult            *result;
        char                id[256], name[256];


        dbhandle = connectdb("dbname=buch user=postgres");
        result = doquery("SELECT id, name FROM t_produkt");

        while   (fetch(id, name) != END_OF_TUPLES)
        {
                printf("Produkt: %s - %s\n", id, name);
        }

        return 0;
}
```

Nach dem Einschließen der verschiedenen Bibliotheken werden vier Variablen angelegt. Die ersten beiden Variablen sind Ihnen bereits bekannt. Die letzten beiden Variablen werden wir zum Zwischenspeichern des Ergebnisses verwenden.

Des Weiteren stellen wir eine Verbindung zur Datenbank her, was mit dem Befehl `connectdb` passieren kann. Im nächsten Schritt schicken wir eine Abfrage an die Datenbank. Das Durchführen der Abfrage ist denkbar einfach und wird zu keinen Problemen führen. Um das Ergebnis vollständig auszugeben, gehen wir die Daten mithilfe einer Schleife durch. Dabei wird jede Zeile mit `fetch` abgefragt. Hierbei wird `fetch` mit zwei Argumenten aufgerufen — ganz allgemein können es aber beliebig viele sein.

Die Schleife wird durchlaufen bis der gesamte Datenbestand abgearbeitet ist. Innerhalb der Schleife werden die Daten einfach von `printf` ausgegeben.

Wenn Sie das Programm kompilieren, dürfen Sie nicht vergessen, es mit der zusätzlichen Bibliothek zu linken. Hier sehen Sie, wie der Code kompiliert und ausgeführt werden kann:

```
[hs@duron c]$ make
gcc -lpq -lpgeasy -I/usr/local/postgresql/include main.c -o prog
[hs@duron c]$ ./prog
Produkt: 1 - Hamburger
Produkt: 2 - Cheeseburger
```

Im Ergebnis finden sich die beiden Spalten, die wir selektiert haben.

Die »easy« C-Schnittstelle zeichnet sich aus Gründen der Einfachheit durch keinen großen Funktionsreichtum aus: Es stehen nur wenige Funktionen zur Verfügung, die jedoch den Großteil der üblicherweise anfallenden Aufgaben leicht erledigen können. Das Header-File der Bibliothek ist derart einfach und leicht zu verstehen, dass es uns sinnvoll erscheint, es gleich als Ganzes einzubeziehen und von einer Referenz Abstand zu nehmen:

```
/*
 * pglib.h
 *
*/

PGresult   *doquery(char *query);
PGconn     *connectdb(char *options);
void            disconnectdb(void);
int                     fetch(void *param,...);
int                     fetchwithnulls(void *param,...);
void            on_error_continue(void);
void            on_error_stop(void);
PGresult   *get_result(void);
void            set_result(PGresult *newres);
void            unset_result(PGresult *oldres);
void            reset_fetch(void);

#define END_OF_TUPLES   (-1)
```

Wann immer es darum geht, einfache Programme zu schreiben, werden Sie mit der vereinfachten C-Schnittstelle zurecht kommen. Die Funktionen sind klar und einfach zu verwenden.

6.4 Die C++-Schnittstelle

Zusätzlich zur C-Schnittstelle steht dem Programmierer eine C++-Schnittstelle zur Verfügung, die auf das C Gegenstück aufsetzt und nur um objektorientierte Komponenten ergänzt worden ist. Das hat den Vorteil, dass Sie sich keine Sorgen um die Kompatibilität der beiden Schnittstellen machen müssen.

6.4.1 Ein einfaches Beispiel

In diesem Abschnitt werden wir auf die C++-Schnittstelle eingehen und sehen, welche Möglichkeiten sich hier bieten. Wie immer wollen wir mit einem einfachen Kapitel beginnen, damit Sie sich ebenso einfach an die Arbeiten machen können.

Das Ziel des folgenden Beispiels ist es, den in der Datenbank gespeicherten Datenbestand abzufragen und auszugeben:

```
#include <iostream.h>
#include "/usr/local/postgresql/include/libpq++.h"

// Sauberes Beenden
void ende(PgDatabase &dbhandle, char *msg)
{
        cout << msg << "\n";
        cout << dbhandle.ErrorMessage();
        dbhandle.~PgDatabase();
        exit(1);
}

// Hauptprogramm
int main(void)
{
        int             result, zeilen, spalten;
        int             i, j;

        // Erstellen der Verbindung zur Datenbank
        PgDatabase      dbhandle("dbname=buch user=postgres");
        if      (dbhandle.ConnectionBad())
                ende(dbhandle, "Verbindungsfehler");
        else
                cout << "Verbindung ok ...\n";

        // Abfragen der Daten
        result = dbhandle.Exec("BEGIN");
        if      (result != PGRES_COMMAND_OK)
                ende(dbhandle, "Transaktionsfehler");

        result = dbhandle.ExecCommandOk("DECLARE tmp CURSOR FOR
                SELECT id AS \"ID\", edvnr AS \"Edvnr\",
                        name AS \"Produktname\",
                        price AS \"Preis\" FROM t_produkt");
        if      (!result)
                ende(dbhandle, "Cursorfehler (declare)");

        /* ExecTuplesOk dient zum Abarbeiten von Tuples */
        /* Wenn es um Kommandos geht, die keine Tuples zurückgeben, */
        /*  kann ExecCommandOk verwendet werden */
        result = dbhandle.ExecTuplesOk("FETCH ALL IN tmp");
        if      (!result)
                ende(dbhandle, "Cursorfehler (fetch)");

        // Analysieren des Ergebnisses
        zeilen = dbhandle.Tuples();
        spalten = dbhandle.Fields();

```

```
49          // Ausgabe des Kopfes
50          cout << endl << "Spalten: " << spalten << endl;
51          cout << "Zeilen: " << zeilen << endl << endl;
52
53          for     (i = 0; i < spalten; i++)
54                  cout << dbhandle.FieldName(i) << "\t";
55          cout << endl;
56
57          // Ausgabe der Daten
58          for     (i = 0; i < zeilen; i++)
59          {
60                  for     (j = 0; j < spalten; j++)
61                  {
62                          cout << dbhandle.GetValue(i, j) << "\t";
63                  }
64                  cout << endl;
65          }
66
67          // Abschluss der Operation
68          result = dbhandle.Exec("CLOSE tmp");
69          result = dbhandle.Exec("COMMIT");
70          dbhandle.~PgDatabase();
71
72          return 0;
73  }
```

Das Programm ist bereits ein wenig länger, aber nicht sonderlich komplex. Sehen wir uns also den Code an.

Die ersten beiden Zeilen dienen wieder zum Einbinden der benötigen Module und Objekte. Im Falle der C++-Schnittstelle benötigen wir das Header-File `libpq++.h`. Die erste Funktion, die wir im Code finden können, dient zum sauberen Beenden des Programms. Die Funktion gibt Fehlermeldungen aus und ruft den Destruktor des PgDatabase-Objektes auf, der die Verbindung zur Datenbank beendet.

Anschließend haben wir das Hauptprogramm implementiert. Nach dem Festlegen einiger Variablen stellen wir eine Verbindung zur Datenbank her, was mithilfe des Konstruktors von PgDatabase passiert. Dabei müssen wir nur den Connect String übergeben.

Um herauszufinden, ob die Verbindung korrekt hergestellt worden ist, wenden wir die Methode `ConnectionBad` an. Sofern die Verbindung nicht korrekt ist, rufen wir die Funktion `ende` auf, die das Programm beendet.

Nachdem wir eine Verbindung zur Datenbank hergestellt und eine Transaktion gestartet haben, deklarieren wir einen Cursor, um Daten aus der Datenbank abzufragen. Sie müssen keinen Cursor definieren, Sie können die Daten auch analog

zur C-Schnittstelle abfragen. In diesem Beispiel haben wir uns für den Cursor entschlossen, um Ihnen auch diese Methode näher zu bringen.

Nach dem Anlegen des Cursors fragen wir die Anzahl der Zeilen und Spalten ab. Die auf diese Weise extrahierte Information werden wir im Folgenden zur Steuerung unserer Schleifen benötigen.

Bevor wir die Daten selbst ausgeben, geben wir die Spaltenköpfe aus. Zur Abfrage der Spaltenköpfe können wir die Methode `FieldName` verwenden. Die Daten können wir mit `GetValue` abfragen.

Am Ende schließen wir den Cursor und die Transaktion.

Um das Programm zu kompilieren, können wir wie bei C ein Makefile verwenden, das wir Ihnen aber an dieser Stelle ersparen. Das nächste Listing zeigt den Kompiliervorgang sowie einen Testlauf des Programmes:

```
[hs@duron c]$ make
g++ -lpq++  -I/usr/local/postgresql/include main.cpp -o prog
[hs@duron c]$ ./prog
Verbindung ok ...

Spalten: 4
Zeilen: 2

ID      Edvnr   Produktname     Preis
1       435353  Hamburger       5.69
2       234234  Cheeseburger    5.99
```

Dabei werden die Daten in einer Art Tabelle ausgegeben.

6.4.2 Ein Überblick über die C++-Schnittstelle

Die C++-Schnittstelle stellt mehrere Objekte mit zahlreichen Methoden zur Verfügung. In diesem Abschnitt wollen wir uns eine Übersicht über die wichtigsten Objekte und deren Methoden verschaffen.

PgConnection

Die folgenden Methoden werden bereitgestellt:

`PgConnection::PgConnection(const char *conninfo)`: Dient zum Herstellen einer Verbindung.

`int PgConnection::ConnectionBad()`: Prüft, ob eine Verbindung gültig ist.

`ConnStatusType PgConnection::Status()`: Gibt den Status einer Verbindung zurück.

`const char *PgConnection DBName()`: Gibt den Namen der Verbindung zurück.

`PGnotify* PgConnection::Notifies()`: Gibt die nächste Nachricht zurück.

`ExecStatusType PgConnection::Exec(const char *query)`: Schickt ein SQL-Statement an den Server.

`int PgConnection::ExecTuplesOk(const char *query)`: Schickt eine Abfrage an den Server.

`const char *PgConnection::ErrorMessage()`: Gibt eine Fehlermeldung zurück.

PgDatabase

Die folgenden Methoden werden bereitgestellt:

`PgDatabase(const char *conninfo)`: Stellt eine Verbindung zur Datenbank her.

`int PgDatabase::Tuples()`: Retourniert die Zeilen in einem Ergebnis.

`int PgDatabase::CmdTuples()`: Retourniert die von einem Statement betroffenen Zeilen.

`int PgDatabase::Fields()`: Gibt die Spalten in einem Ergebnis zurück.

`const char *PgDatabase::Fieldname(int field_num)`: Gibt den Namen eines Feldes zurück.

`int PgDatabase::FieldNum(const char * field_name)`: Gibt den Index einer Spalte zurück.

`Oid PgDatabase::FieldType(int field_num)`: Gibt den Datentyp eines Feldes zurück.

`Oid PgDatabase::FieldType(const char * field_name)`: Gibt den Datentyp eines Feldes zurück.

`short PgDatabase::FieldSize(int field_num)`: Gibt die Größe eines Feldes zurück.

`short PgDatabase::FieldSize(const char * field_name)`: Gibt die Größe eines Feldes zurück.

`const char *PgDatabase::GetValue(int tup_num, int field_num)`: Holt einen Wert aus dem Ergebnis.

`const char *PgDatabase::GetValue(int tup_num, const char * field_name)`: Holt einen Wert aus dem Ergebnis.

`int PgDatabase::GetLength(int tup_num, int field_num)`: Gibt die Länge eines Feldes zurück.

`int PgDatabase::GetLength(int tup_num, const char * field_name)`: Gibt die Länge eines Feldes zurück.

`void PgDatabase::DisplayTuples(FILE *out = 0, int fillAlign = 1, const char * fieldSep = \"|\", int printHeader = 1, int quiet = 0)`: Schreibt alle Tuples in einen Ausgabestream.

`void PgDatabase::PrintTuples(FILE *out = 0, int printAttName = 1, int terseOutput = 1, int width = 0)`: Schreibt alle Tuples in einen Ausgabestream.

`int PgDatabase::GetLine(char *string, int length)`: Liest einen String in einen Buffer.

`void PgDatabase::PutLIne(const char *string)`: Sendet einen String an das Backend.

`const char * *PgDatabase::OidStatus()`: Checkt den Status.

`int PgDatabase::EndCopy()`: Beendet einen `COPY`-Befehl.

Kapitel 7

ECPG — Der SQL-Preprocessor

In diesem Kapitel werden wir uns eingehend mit ECPG beschäftigen.

Die ersten Versionen von ECPG stammen von Linus Tolke, einem Schweden. Derzeit wird das Paket von Michael Meskes betreut, entwickelt und gewartet. Die Software wird direkt mit PostgreSQL installiert und stellt ein mächtiges Werkzeug für Entwickler aller Art dar.

Wir werden versuchen, die grundlegenden Konzepte von ECPG zu erklären und Lösungsansätze zu präsentieren. Sie werden sehr schnell erkennen, dass es sich um ein sinnvolles und einfach zu verwendendes Tool handelt, mit dem Sie komplexe Datenbankanwendungen aller Art schreiben und entwickeln können.

7.1 Was ist ECPG?

Die grundlegende Frage, die sich die meisten Leser stellen ist, was ECPG eigentlich macht und wozu man es verwenden kann.

Vereinfacht gesagt ist ECPG ein Tool, das C-Code generiert. Ein Eingabefile, das C- und ECPG-Code enthält, wird bearbeitet und auf C konvertiert — genauer gesagt wird nur der ECPG-Code im File konvertiert (logischerweise). Auf diese Weise können Sie sehr schnell Datenbankschnittstellen entwickeln, da Sie die C-Schnittstellen von PostgreSQL nicht direkt verwenden müssen. Mithilfe des Preprocessors können Sie SQL leicht in ein C-Programm einbinden. Es besteht etwa die Möglichkeit, direkt auf C-Variablen zuzugreifen und andere, derartige Operationen durchzuführen. Durch die Verbindung der einfachen Bedienbarkeit von ECPG mit der Macht der Programmiersprache C werden die Möglichkeiten nur durch die Fantasie des Anwenders begrenzt.

ECPG ist keine proprietäre Software. Die prinzipielle Funktionsweise von Embedded SQL ist in einem ANSI-Standard festgeschrieben, der von den Entwicklern so gut wie nur irgendwie möglich befolgt wird. Das hat den Vorteil, dass ihre Anwendungen einfach portiert werden können und gar keine oder nur geringe Anspassungskosten anfallen, wenn sie von oder nach PostgreSQL migrieren.

7.2 Grundlagen

Nach dieser Erläuterung des Preprocessors können wir uns bereits voll in die Arbeit stürzen. Bevor wir einen Blick auf das erste Beispiel werfen können, ist es notwendig, sich die prinzipielle Vorgehensweise bei der Erstellung eines Programmes anzusehen. Das Programm `ecpg` erzeugt aus einem Inputfile C-Code. Dieser C-Code kann kompiliert werden und steht somit als ausführbares Programm zur Verfügung.

Das nächste Beispiel zeigt, wie eine einfache Berechnung durchgeführt werden kann:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
int result;
EXEC SQL END DECLARE SECTION;

int     main()
{
        EXEC SQL CONNECT TO buch;
        EXEC SQL SELECT (1+1) INTO :result;

        printf("result: %i\n", result);
        return 0;
}
```

Das Programm beginnt wie ein normales C-Programm. Die Bibliothek `stdio.h` wird eingebunden, um eine sinnvolle Ausgabe auf dem Bildschirm zu ermöglichen.

Weiter ist es notwendig, Variablen zu vereinbaren. Alle Variablen, die in ECPG verwendet werden sollen, müssen in der `DECLARE`-Sektion definiert werden. Hier wird eine Integer-Variable als Integer-Variable angelegt.

Nach den Variablendefinitionen beginnt das Hauptprogramm. Wie üblich beginnt das C-Programm mit einer Main-Funktion. In dieser Funktion wird eine Verbindung zur Datenbank aufgebaut. Danach wird ein einfacher SQL-Befehl ausgeführt, der die Summe von 1 plus 1 berechnet. Das Ergebnis der Berechnung wird der zuvor vereinbarten Variablen zugewiesen. Sie können Sie mithilfe des vorangestellten Doppelpunktes auf `result` zugreifen. Sofern das Ergebnis des SQL-Befehles ein gültiger Integer-Wert ist, wird die Zuweisung wie gewünscht funktionieren.

Nach der Berechnung wird das Ergebnis auf dem Bildschirm ausgegeben und das C-Programm terminiert.

Nachdem wir das ECPG-Programm verfasst haben, können wir nun daran gehen, den C-Code zu generieren und dann zu kompilieren. Für diese Zwecke verwendet man sinnvollerweise Makefile, wie es im Listing zu sehen ist:

```
INC=    /usr/local/postgresql/include
LIB=    /usr/local/postgresql/lib

prog    :       main.pgc
        ecpg    -o tmp.c main.pgc
        gcc -g -I $(INC) -o prog tmp.c -L $(LIB) -lecpg -lpq
```

Wie vorhin bereits erwähnt, dient das Programm `ECPG` dazu, das C-Programm zu generieren. Im Weiteren wird dieses mit dem GNU C-Compiler (oder irgendeinem

anderen ANSI C-Compiler) kompiliert und mit den entsprechenden Bibliotheken gelinkt. Sofern Sie keinen Fehler gemacht haben, wird es zu keinen Problemen kommen und der Compiler ein funktionierendes Programm erzeugen.

Versuchen wir nun, das Programm auszuführen:

```
[hs@notebook ecpg]$ ./prog
result: 2
```

Dabei ist das Ergebnis 2. Um zu verdeutlichen, dass ECPG nichts anderes als eine Ergänzung zu C ist, können wir einen Blick auf das nächste Beispiel werfen. Es zeigt ein Tool, das zwei Zahlen, die via Standard-Input an das Programm geschickt werden, addieren kann:

```
#include <stdio.h>
#include <stdlib.h>

EXEC SQL BEGIN DECLARE SECTION;
int result;
int a, b;
EXEC SQL END DECLARE SECTION;

int     main(int argc, char **argv)
{
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        EXEC SQL CONNECT TO buch@localhost;
        EXEC SQL SELECT (:a + :b) INTO :result;;

        printf("result: %i\n", result);
        return 0;
}
```

Hier werden zwei Bibliotheken einbezogen. In der `DECLARE` section werden zwei zusätzliche Variablen aufgenommen, die wir verwenden können, um die von Standard-Input extrahierten Variablen zwischenzuspeichern. Das Hauptprogramm akzeptiert über die Konsole eingegebene Werte. Die ersten beiden Parameter werden auf Integer konvertiert und können direkt verwendet werden.

Bevor die eigentliche Berechnung durchgeführt wird, wird eine Verbindung zur Datenbank angelegt. Wie im vorangegangenen Beispiel verbinden wir uns zur Datenbank namens `buch`. In diesem Fall geben wir jedoch explizit an, dass die Datenbank am lokalen Server zu finden ist.

Nach dem Kompilieren des Programms mittels `make` können wir das Script testen:

```
[hs@notebook ecpg]$ ./prog 23 59
result: 82
```

Das Programm liefert das korrekte Ergebnis.

Um ECPG besser zu verstehen, ist es sinnvoll, sich den generierten C-Code genauer anzusehen. Das nächste Listing enthält den Inhalt von `tmp.c`:

```
/* Processed by ecpg (2.9.0) */
/* These three include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
#line 1 "main.pgc"
#include <stdio.h>
#include <stdlib.h>

/* exec sql begin declare section */



#line 5 "main.pgc"
   int  result   ;

#line 6 "main.pgc"
   int  a   ,  b   ;
/* exec sql end declare section */
#line 7 "main.pgc"


int     main(int argc, char **argv)
{
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        { ECPGconnect(__LINE__, "buch@localhost" , NULL,NULL ,
                NULL, 0); }
#line 14 "main.pgc"

        { ECPGdo(__LINE__, NULL, "select  ( ? + ? )     ",
        ECPGt_int,&(a),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L,
        ECPGt_int,&(b),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
```

```
39          ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);}
40 #line 15 "main.pgc"
41 ;
42
43          printf("result: %i\n", result);
44          return 0;
45 }
```

Am Beginn des Programmes kann man erkennen, dass eine Vielzahl von Bibliotheken eingebunden wird. Diese Bibliotheken sorgen für die reibungslose Interaktion des C-Programms mit der Datenbank. Weiter werden noch Header-Files, die wir in Folge zur Fehlerbehandlung benötigen werden, eingebunden.

Anschließend werden die notwendigen Variablen deklariert. Im Code ist klar gekennzeichnet, welche Variablen im ECPG gemeint sind. Das ist beim Debuggen sehr vorteilhaft und hilft Ihnen, das C-Programm leichter zu lesen.

Nach dem selbst geschriebenen C-Code können Sie sehen, was ECPG aus dem Embedded SQL gemacht hat. Im Prinzip wird das Embedded SQL »nur« durch C-Funktionen passenden Typs ersetzt.

Abschließend wird das Ergebnis mittels `printf` ausgegeben. Die von uns implementierten C-Teile werden einfach nur in das C-File durchkopiert und bleiben unverändert.

7.3 Fehlerbehandlung

Bisher haben wir gezeigt, wie einfache Operationen durchgeführt werden können. Was kann man aber tun, wenn Fehler auftreten? Was tun, wenn keine Verbindung zur Datenbank eingerichtet werden kann, und was passiert, wenn Queries nicht funktionieren? ECPG bietet ein umfangreiches System zur einfachen und effizienten Analyse von Fehlern und genau damit werden wir uns in diesem Abschnitt eingehend beschäftigen.

Um Fehlerroutinen von ECPG zu nutzen, benötigen Sie eine Bibliothek namens `sqlca`. Das folgende Listing gibt Ihnen einen Überblick über die zur Verfügung gestellten Mechanismen:

`-12 Out of memory in line %d:`
: Diese Meldung gibt an, dass dem System der Speicher ausgegangen ist.

`-200 Unsupported type %s on line %d:`
: In der Regel tritt diese Meldung auf, wenn die Versionen der einzelnen Bibliotheken nicht zusammenpassen. Stellen Sie sicher, dass die Versionen kompatibel sind. Der Grund für die Fehlermeldung ist, dass ECPG Code generiert hat, der von einer der Bibliotheken nicht verstanden wird.

`-201 Too many arguments line %d:`
Wenn die Datenbank mehr Felder zurückgibt, als von Ihrem Programm akzeptiert werden, tritt ein Fehler auf. In diesem Falle müssen Sie die Anzahl der selektierten und akzeptierten Spalten prüfen.

`-202 Too few arguments line %d:`
Die Meldung bedeutet, dass nicht genug Argumente im Funktionsaufruf übergeben werden.

`-203 Too many matches line %d:`
Ein `SELECT`-Statement hat mehr Zeilen zurückgegeben, als von Ihrem Programm akzeptiert werden kann. In der Regel tritt dieser Fehler dann auf, wenn Sie eine Zeile erwarten, die Datenbank jedoch ein Set von Datensätzen zurückgibt.

`-204 Not correctly formatted int type: %s line %d:`
Der von PostgreSQL zurückgegebene Wert kann nicht als Integer-Wert interpretiert werden.

`-205 Not correctly formatted unsigned type: %s line %d:`
Der von PostgreSQL zurückgegebene Wert kann nicht als unsigned Integer interpretiert werden.

`-206 Not correctly formatted floating point type: %s line %d:`
Der von PostgreSQL zurückgegebene Wert kann nicht als Float interpretiert werden.

`-207 Unable to convert %s to bool on line %d:`
Der retourierte Wert kann nicht als boolescher Wert gedeutet werden (also »t« oder »f«).

`-208 Empty query line %d:`
Der Rückgabewert von PostgreSQL ist `PGRES_EMTPY_QUERY`. Dieser Fehler tritt bei leeren Zeilen auf.

`-220 No such connection %s in line %d:`
Wenn Ihr Programm versucht, eine Connection zu verwenden, die es nicht gibt, wird dieser Fehler ausgegeben.

`-221 Not connected in line %d:`
Sofern Sie sich ordnungsgemäß zur Datenbank verbunden haben, tritt dieser Fehler dann auf, wenn die Verbindung zur Datenbank aus irgendwelchen Gründen nicht funktioniert. Die Verbindung ist vorhanden, aber nicht offen.

`-230 Invalid statement name %s in line %d:`
Die Abfrage, die Sie bearbeiten, ist nicht ordnungsgemäß vorbereitet worden.

`-400 Postgres error: %s line %d:`
Es ist ein Fehler im Backend aufgetreten.

`-401 Error in transaction processing line %d:`
Es ist ein Fehler in einer Transaktion aufgetreten. `Begin`, `Commit` und `Rollback` sind unmöglich. Dieser Fehler tritt eher selten auf.

`-402 connect: could not open database %s:`
Der Versuch, sich zur Datenbank zu verbinden, ist aus diversen Gründen gescheitert.

`100 Data not found line %d:`
Das Objekt, nach dem Sie suchen, existiert nicht. Dieser Fehler tritt beispielsweise auf, wenn Sie alle Datensätze abgearbeitet haben.

Nach dieser Übersicht zeigen wir einige Beispiele, in denen Fehler bearbeitet werden. Das nächste Beispiel zeigt, wie Fehler abgefangen werden können, die die Datenbankverbindung betreffen:

```
1  #include <stdio.h>
2
3  EXEC SQL BEGIN DECLARE SECTION;
4  int result;
5  EXEC SQL END DECLARE SECTION;
6
7  EXEC SQL INCLUDE sqlca;
8
9  int        main(int argc, char **argv)
10 {
11         EXEC SQL CONNECT TO unbekannt@irgendwo
12                 AS verbindung USER keinuser/keinpasswd;
13         if        (sqlca.sqlcode)
14         {
15                 printf("Fehler: %s\n", sqlca.sqlerrm.sqlerrmc);
16         }
17
18         EXEC SQL SELECT (1 + 1) INTO :result;;
19
20         printf("result: %i\n", result);
21         return 0;
22 }
```

Am Beginn des Programms ist es notwendig, eine Bibliothek aufzunehmen, die für das Fehlermanagement zuständig ist. Da es sich hier um eine ECPG-Bibliothek handelt, müssen wir Sie mittels `EXEC SQL INCLUDE` einbinden.

Im Anschluss daran versuchen wir uns zur Datenbank zu verbinden. Alle Parameter (Name der Datenbank, Host, User und Passwort) sind falsch und das Herstellen der Verbindung wird daher auf alle Fälle fehlschlagen.

Um den Fehler beim Aufbau der Verbindung abzufangen, müssen wir einen Test durchführen. Hierbei prüfen wir, ob `sqlca.sqlcode` einen Fehler enthält oder nicht. Sofern die `if`-Bedingung erfüllt ist, wird ein Fehler ausgegeben. Weiterhin wird die Summe von 1 und 1 berechnet und am Bildschirm ausgegeben.

Bevor wir uns weiteren Beispielen zuwenden, sehen wir uns die von ECPG zur Verfügung gestellte Struktur zur Behandlung an. Diese Struktur ist essentiell, weil sie es ermöglicht, einen aufgetretenen Fehler genau zu analysieren. Das nächste Listing zeigt, welche Elemente in der Struktur `sqlca` enthalten sind:

```
1  struct sqlca
2  {
3          char                sqlcaid[8];
4          long                sqlabc;
5          long                sqlcode;
6          struct
7          {
8                  int                          sqlerrml;
9                  char                 sqlerrmc[SQLERRMC_LEN];
10         }                          sqlerrm;
11         char                sqlerrp[8];
12         long                sqlerrd[6];
13         /* Element 0: empty                                        */
14         /* 1: OID of processed tuple if applicable                 */
15         /* 2: number of rows processed                             */
16         /* after an INSERT, UPDATE or                              */
17         /* DELETE statement                                        */
18         /* 3: empty                                                */
19         /* 4: empty                                                */
20         /* 5: empty                                                */
21         char                sqlwarn[8];
22         /* Element 0: set to 'W' if at least one other is 'W'      */
23         /* 1: if 'W' at least one character string                 */
24         /* value was truncated when it was                         */
25         /* stored into a host variable.                            */
26
27         /*
28          * 2: if 'W' a (hopefully) non-fatal notice occured
29          */        /* 3: empty */
30         /* 4: empty                                                 */
31         /* 5: empty                                                 */
32         /* 6: empty                                                 */
33         /* 7: empty                                                 */
34
```

```
35          char                sqlext[8];
36  };
37
38
39  #ifdef __cplusplus
40  }
41  #endif
42
43  #endif
```

Bei einem auftretenden Fehler ist der Wert von `sqlca.sqlcode` ungleich 0. Wenn der Wert kleiner als 0 ist, ist ein grober Fehler aufgetreten. Sofern der Fehler größer als 0 ist, handelt es sich nicht um einen schwerwiegenden Fehler. Gehen Sie noch einmal die vorher gezeigte Liste von Fehlern durch und prüfen Sie, welche Arten von Fehlern auftreten können.

Um mehr über den aktuellen Fehler herauszufinden, können Sie einen Blick auf den Inhalt von `sqlca.sqlerrm.sqlerrmc` werfen. Um die Stelle zu lokalisieren, an der der Fehler auftetreten ist, wird die Zeile im Programm von ECPG aufgelistet.

Das nächste Beispiel zeigt, wie ein Fehler in einem SQL-Statement abgefangen werden kann. Das Programm soll dazu dienen, einfache Divisionen durchzuführen:

```
1  #include <stdio.h>
2
3  EXEC SQL BEGIN DECLARE SECTION;
4  int a, b;
5  double result;
6  EXEC SQL END DECLARE SECTION;
7
8  int         main(int argc, char **argv)
9  {
10          a = atoi(argv[1]);
11          b = atoi(argv[2]);
12
13          EXEC SQL CONNECT TO buch@localhost AS verbindung USER postgres;
14          if          (sqlca.sqlcode)
15          {
16                  printf("Fehler: %s\n", sqlca.sqlerrm.sqlerrmc);
17          }
18
19          EXEC SQL SELECT (:a::float / :b) INTO :result;
20          if          (sqlca.sqlcode)
21          {
22                  printf("Fehlercode: %d\nFehlerstring: %s\n",
23                          sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
24          }
25
```

```
26          printf("result: %f\n", result);
27          return 0;
28  }
```

Am Beginn des Programms werden einige Variablen angelegt. Dann werden zwei Variablen von Standard-Input gelesen. Das ist notwendig, da `argc` und `argv` nicht innerhalb von ECPG verwendet werden können.

Nach dem Aufbauen der Verbindung zur Datenbank wird eine Division durchgeführt. Hierbei ist zu beachten, dass der Datenbank mitgeteilt werden muss, dass es sich hierbei um eine Gleitkomma-Operation handelt, wie das folgende Beispiel zeigt:

```
1  buch=# select 4 / 8;
2   ?column?
3  ----------
4           0
5  (1 row)
```

Das Ergebnis der Division ist 0. Der Grund dafür liegt in der ANSI-Spezifikation, die dieses Verhalten explizit fordert. Um das korrekte mathematische Ergebnis zu berechnen, müssen Sie die Berechnung als Gleitkommadivision kennzeichnen. Das kann auf mehrere Arten passieren. Zwei der Möglichkeiten sind im nächsten Listing dargestellt:

```
1  buch=# select 4 / 8.0;
2   ?column?
3  ----------
4         0.5
5  (1 row)
6
7  buch=# select 4::float / 8;
8   ?column?
9  ----------
10        0.5
11 (1 row)
```

Wenden wir uns nun also weiter unserem ECPG-Programm zu. Wenn die Berechnung daneben geht, ist ein Fehler auszugeben. Wie im vorangehenden ECPG-Programm passiert das mittels `printf`.

Sobald Sie das Programm mithilfe des Makefiles kompiliert haben, können Sie dessen Funktionsumfang testen:

```
1  [hs@duron ecpg]$ ./prog 8 16
2  result: 0.500000
```

Das Ergebnis der Berechnung ist korrekt, sofern Sie keine Division durch 0 durchführen, wie das im nächsten Listing gezeigt wird:

```
[hs@duron ecpg]$ ./prog 8 0
Fehlercode: -400
Fehlerstring: 'ERROR:  float8div: divide by zero error' in line 19.
result: 0.000000
```

Unsere Fehlerroutine kommt zum Einsatz und ein Error wird ausgegeben und zwar der Error mit der Nummer −400. Da der Errorcode kleiner als 0 ist, handelt es sich um einen schweren Fehler.

Bei komplexeren Anwendungen empfiehlt es sich, das Errorhandling eine eigene Subfunktionen auszugliedern, die das gesamte Errorhandling erledigen. Auch in diesem Fall leistet Ihnen der Preprocessor gute Dienst, wie das folgende Beispiel verdeutlichen soll:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
int a, b;
double result;
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO error();
EXEC SQL WHENEVER SQLWARNING DO warn();

void error()
{
        printf("Fehler: %s\n", sqlca.sqlerrm.sqlerrmc);
        exit(1);
}

void warn()
{
        printf("Warnung - problematischer Verarbeitungsschritt ...\n");
}

int        main(int argc, char **argv)
{
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        EXEC SQL CONNECT TO buch@localhost AS verbindung USER postgres;
        EXEC SQL SELECT (:a::float / :b) INTO :result;

        printf("result: %f\n", result);
        return 0;
}
```

Ziel dieses Beispieles ist es, alle Warnings und Fehler in nur zwei Routinen zusammenzufassen, um den Code zu verkürzen und etwas übersichtlicher zu gestalten. Zu diesen Zwecke stellt ECPG zwei Mechanismen zur Verfügung. Im Falle eines Fehlers wird in unserem Fall die Funktion `error` aufgerufen. Falls eine Warning auftreten sollte, haben wir die Funktion `warn` implementiert, die bei Bedarf automatisch aufgerufen wird. Der Rest des Codes entspricht dem vorherigen Beispiel. Wichtig ist nur, dass es nicht mehr notwendig ist, jede Fehlermeldung einzeln abzufragen.

Nach dem Kompilieren des Beispieles kann es wie gewohnt ausgeführt werden:

```
[hs@duron ecpg]$ ./prog 2 0
Fehler: 'ERROR:  float8div: divide by zero error' in line 28.
```

Fehlerbehandlung ist eine wichtige Komponente eines jeden Softwarepaketes. Wir empfehlen, ausreichend Checks einzubauen und sicherzustellen, dass es zu keinen unvorhersehbaren Problemen kommen kann. Derartige Sicherstellungen werden in der Regel durch das vorherige Berechnen der Weakest Preconditions und dergleichen sichergestellt.

7.4 Arbeiten mit mehreren Datenbankverbindungen

In vielen Fällen ist es notwendig, mit mehreren Datenbankverbindungen zu arbeiten. Auch für diesen Fall ist ECPG bestens gerüstet. Hier werden wir uns näher mit gleichzeitigen Verbindungen beschäftigen und erklären, wie solche Dinge gehandhabt werden können.

Legen wir nun noch eine zweite Datenbank an:

```
[hs@duron ecpg]$ createdb -U postgres cd
CREATE DATABASE
```

Ziel des folgenden Beispiels ist es, zwei Verbindungen zu zwei verschiedenen Datenbanken zu erstellen und zu zeigen, wie auf verschiedene Datenbanken zugegriffen werden kann:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
int result;
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO error();

void error()
{
```

```
11          printf("Fehler: %s\n", sqlca.sqlerrm.sqlerrmc);
12          exit(1);
13  }
14
15  int         main()
16  {
17          EXEC SQL CONNECT TO buch@localhost AS herkunft USER postgres;
18          EXEC SQL CONNECT TO cd@localhost AS ziel USER postgres;
19
20          EXEC SQL AT herkunft SELECT (1 + 1) INTO :result;
21          EXEC SQL AT ziel SELECT (1 + 1) INTO :result;
22          return 0;
23  }
```

Der Befehl zum Aufbauen einer Verbindung zur Datenbank sowie der Befehl zum Absetzen einer Abfrage müssen leicht modifiziert werden. Aus dem Programm ist ersichtlich, dass jede der beiden Datenbankverbindungen einen Namen hat, um sie eindeutig zu identifizieren. Bei nur einer Verbindung ist der Name nicht notwendig, da die Verbindungen nicht unterschieden werden müssen. Im Fall mehrerer, gleichzeitiger Verbindungen ist das sehr wohl notwendig und aus diesem Grund ist ein Name essentiell.

Der Rest des Beispieles enthält nichts Neues.

7.5 SQL-Abfragen

Bisher haben wir nur SQL-Befehle ausgeführt, die genau eine Zelle zurückgeben. In vielen Fällen ist es interessant zu sehen, wie eine ganze Reihe von Datensätzen effizient verarbeitet werden kann. Zu diesem Zwecke verwendet man so genannte Cursor. Cursor sind im Prinzip nichts anderes als Verweise auf das Ergebnis einer SQL-Abfrage, die es nach der Reihe abzuarbeiten gilt.

Hinweis: Es ist auch möglich, einen Datenbestand ohne Cursor abzufragen, indem man ein ganzes Array mit einem Aufruf füllt.

Das nächste Listing enthält ein einfaches Beispiel, das die prinzipielle Vorgehensweise beim Verwenden eines Cursors zeigt.

Zu diesem Zwecke schreiben wir ein File namens `cd.sql`, das eine Tabelle und mehrere Datensätze anlegt:

```
1  CREATE TABLE cd (
2          id int4,
3          name char(128)
4  );
5
```

```
COPY cd FROM stdin;
1       Falco – Out of the Dark
2       Falco – Junge Römer
3       Falco – Nachtflug
4       Falco – Einzelhaft
\.
```

Der folgende Befehl fügt die Daten als User `postgres` in die Datenbank ein:

```
[hs@duron ecpg]$ psql –U postgres buch < cd.sql
CREATE
```

Sofern keine Fehler aufgetreten sind, können wir nun beginnen, ein ECPG-Programm zu schreiben, das die Daten abfragt und am Bildschirm ausgibt:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
   char       query[1024];
   int        id;
   char       name[128];
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR DO error();

void error()
{
        printf("Fehler: %s\n", sqlca.sqlerrm.sqlerrmc);
        exit(1);
}

int        main()
{
        EXEC SQL CONNECT TO buch@localhost AS herkunft USER postgres;

        snprintf(query, sizeof query, "SELECT id, name FROM cd");
        EXEC SQL PREPARE MYQUERY FROM :query;
        EXEC SQL DECLARE MYCURSOR CURSOR FOR MYQUERY;

        EXEC SQL OPEN MYCURSOR;
        EXEC SQL WHENEVER NOT FOUND DO BREAK;

        while       (1)
        {
                EXEC SQL FETCH IN MYCURSOR INTO :id, :name;
                printf("%i – %s\n", id, name);
        }

```

```
34          EXEC SQL CLOSE MYCURSOR;
35          EXEC SQL DISCONNECT;
36
37          return 0;
38  }
```

Am Beginn des Programms werden einige Variablen deklariert. Die Variable `query` wird uns dazu dienen, das auszuführende SQL-Statement zu speichern. `id` und `name` werden die Daten der beiden Spalten in der Datenbank enthalten. Weiterhin wird die Fehlerroutine festgelegt. Bei jedem im Programm auftretenden ECPG-Fehler wird die Funktion `error` automatisch aufgerufen. Nach dem Verbinden zur Datenbank wird der Code der Abfrage in der Variable namens `query` gespeichert. Dafür wird die Funktion `snprintf` verwendet. Diese Funktion stellt sicher, dass kein zu langer String zugewiesen wird. Sofern Sie ein statisches SQL Statement ausführen wollen, können Sie auch gerne auf die Variable `query` verzichten.

Danach wird eine Abfrage vorbereitet. Bevor die Abfrage endgültig ausgeführt wird, muss noch ein Cursor deklariert und geöffnet werden.

Danach wird eine Schleife solange durchlaufen, bis keine Daten mehr extrahiert werden können. In unserem Beispiel werden die extrahierten Datensätze einfach am Bildschirm ausgegeben.

```
1  [hs@duron ecpg]$ ./prog
2  1 - Falco - Out of the Dark
3  2 - Falco - Junge Römer
4  3 - Falco - Nachtflug
5  4 - Falco - Einzelhaft
```

Das Ergebnis der Abfrage ist korrekt und es sind keine Probleme aufgetreten.

7.6 Arbeiten mit Transaktionen

Nachdem wir kennen gelernt haben, wie einfache Abfragen durchgeführt werden können, wenden wir uns dem Arbeiten mit Transaktionen zu. Transaktionen sind ein wesentlicher Bestandteil jedes hoch entwickelten Datenbanksystems und zur Implementierung intelligenter und kritischer Anwendungen essentiell.

Dieses Beispiel zeigt, wird Transaktionen gestartet und beendet werden können:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  EXEC SQL BEGIN DECLARE SECTION;
5  int result;
```

```
6  int a;
7  char b[128];
8  EXEC SQL END DECLARE SECTION;
9
10 int        main(int argc, char **argv)
11 {
12         a = atoi(argv[1]);
13         if        (argv[2])
14         {
15
16                 snprintf(b, sizeof b, argv[2]);
17                 printf("Data: %s\n", argv[2]);
18         }
19         else
20                 exit(1);
21
22         EXEC SQL CONNECT TO buch@localhost;
23         EXEC SQL BEGIN TRANSACTION;
24         EXEC SQL INSERT INTO cd (id, name) VALUES (:a, :b);
25
26         if        (argv[3])
27                 EXEC SQL COMMIT;
28         else
29                 EXEC SQL ROLLBACK;
30
31         EXEC SQL DISCONNECT;
32         return 0;
33 }
```

Das Programm akzeptiert drei Parameter. Der erste Parameter enthält die ID des einzufügenden Datensatzes. Aus Gründen der Einfachheit haben wir diesen Parameter nicht auf seine Existenz geprüft. Nach der Konvertierung der ID auf Integer wird der Wert der zweiten Spalte eingelesen. Diese Operation erfolgt mittels `snprintf`. Der eingelesene String wird auf dem Bildschirm ausgegeben. Sofern er nicht defininiert ist, beendet sich das Programm.

Im Folgenden werden eine Verbindung zur Datenbank geöffnet und eine Transaktion gestartet. Das erfolgt über einen normalen SQL-Befehl wie Sie ihn bereits aus vorangegangenen Kapiteln kennen.

Im nächsten Schritt wird ein `INSERT`-Statement ausgeführt. Das Ende der Transaktion ist vom dritten Parameter abhängig. Sofern ein beliebiger Wert als Argument übergeben wird, wird `COMMIT` ausgeführt — anderenfalls wird das `INSERT` rückgängig gemacht. Schließlich wird die Verbindung zur Datenbank getrennt und das Programm beendet.

Nach dem Kompilieren der Software kann das Programm bequem über die Kommandozeile aufgerufen werden:

```
[hs@notebook ecpg]$ ./prog 5 'Falco – Einzelhaft' commit
```

Dabei wird der Wert zur Datenbank hinzugefügt, da wir einen dritten Parameter angegeben haben. Wenn das commit fehlt, wird der `INSERT`-Befehl nie wirksam.

Ein wichtiger Punkt im Umgang mit Transaktionen ist `AutoCommit`. Sofern `AutoCommit` eingeschaltet ist, wird jede Transaktion automatisch korrekt abgeschlossen und die Änderungen werden wirksam — das ist die Standardeinstellung.

Wenn Sie sich entschließen `AutoCommit` abzuschalten, werden Sie sehr schnell erkennen, dass Sie sich selbst und explizit um das Beenden von Transaktionen kümmern müssen. Oracle verwendet standardmäßig diese Einstellung. ECPG stellt einfache Möglichkeiten zur Verfügung, `AutoCommit` zu aktivieren beziehungsweise zu deaktivieren.

Im nächsten Beispiel steht, wie `AutoCommit` ausgeschaltet werden kann:

```
#include <stdio.h>
#include <stdlib.h>

int        main(int argc, char **argv)
{
         EXEC SQL CONNECT TO buch@localhost AS verbindung;
         EXEC SQL SET AUTOCOMMIT TO OFF;
         EXEC SQL INSERT INTO cd (id) VALUES (99);
         EXEC SQL DISCONNECT;

         return 0;
}
```

Wenn Sie das vorangegangene Programm kompilieren und ausführen, werden Sie erkennen, dass das `INSERT`-Statement ohne Wirkung geblieben ist, da PostgreSQL ein implizites `ROLLBACK` durchgeführt hat:

```
buch=# SELECT * FROM cd WHERE id = 99;
 id | name
----+------
(0 rows)
```

Um den Datensatz tatsächlich sichtbar und verfügbat zu machen, ist es notwendig, `BEGIN` und `COMMIT` explizit aufzurufen, bei eingeschaltenem `AutoCommit` wäre das nicht nötig.

7.7 Bauen von Abfragetools

Nach diesem Überblick über ECPG werden wir einige Beispiele zeigen.

7.7.1 Abfragen von Detailinformationen

Bei vielen praktischen Anwendungen ist es notwendig, Informationen über die Daten zu extrahieren und genau damit werden wir uns im folgenden Beispiel beschäftigen. Wir haben dabei versucht, möglichst großen Wert auf die Einfachheit der Verarbeitung zu legen und sind daher bewusst nicht auf Details wie etwa die Arten von Timestamps eingegangen. Hier ist der Source-Code der Applikation:

```
#include <stdio.h>
#include <stdlib.h>

EXEC SQL INCLUDE sql3types;
EXEC SQL WHENEVER SQLERROR DO error();

void error()
{
       printf("#%ld:%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
       exit(1);
}

int         main(int argc, char **argv)
{
        EXEC SQL BEGIN DECLARE SECTION;
                  bool BOOLVAR;
                  char NAME[120];
                  char QUERY[1024];
                  char STRINGVAR[1024];
                  double DOUBLEVAR;
                  float FLOATVAR;
                  int COUNT;
                  int INDEX;
                  int INDICATOR;
                  int INTVAR;
                  int NULLABLE, RETURNED_OCTET_LENGTH;
                  int TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE;
        EXEC SQL END DECLARE SECTION;
        EXEC SQL ALLOCATE DESCRIPTOR MYDESC;

        if        (argv[1])
                snprintf(QUERY, sizeof QUERY, argv[1]);
        else
        {
                printf("Keine Query definiert ...\n");
```

```
36                exit(1);
37        }
38
39          EXEC SQL CONNECT TO buch@localhost:5432;
40
41          EXEC SQL PREPARE MYQUERY from :QUERY;
42          EXEC SQL DECLARE MYCURS CURSOR FOR MYQUERY;
43
44          EXEC SQL OPEN MYCURS;
45
46          while         (1)
47          {
48                EXEC SQL FETCH IN MYCURS INTO SQL DESCRIPTOR MYDESC;
49                EXEC SQL GET DESCRIPTOR MYDESC :COUNT = count;
50
51                printf("Zahl der Spalten: %d\n",COUNT);
52                printf("Detailinformation: \n");
53
54                for         (INDEX = 1; INDEX <= COUNT; ++INDEX)
55                {
56                        EXEC SQL GET DESCRIPTOR MYDESC VALUE :INDEX
57                                :TYPE = type,
58                                :LENGTH = length,
59                                :OCTET_LENGTH = octet_length,
60                                :PRECISION = precision,
61                                :SCALE = scale,
62                                :NULLABLE = nullable,
63                                :NAME = name,
64                                :RETURNED_OCTET_LENGTH =
65                                        returned_octet_length;
66                        printf("%s: ", NAME);
67
68                        switch (TYPE)
69                        {
70                                    case SQL3_BOOLEAN:
71                                         printf("bool ");
72                                         break;
73                                    case SQL3_NUMERIC:
74                                            printf("numeric(%d,%d) ",
75                                                 PRECISION,SCALE);
76                                         break;
77                                    case SQL3_DECIMAL:
78                                            printf("decimal(%d,%d) ",
79                                                 PRECISION,SCALE);
80                                         break;
81                                    case SQL3_INTEGER:
82                                            printf("integer ");
83                                            break;
```

```
84                                      case SQL3_SMALLINT:
85                                              printf("smallint ");
86                                      break;
87                                      case SQL3_FLOAT:
88                                              printf("float(%d,%d) ",
89                                                      PRECISION,SCALE);
90                                      break;
91                                      case SQL3_REAL:
92                                              printf("real ");
93                                      break;
94                                      case SQL3_DOUBLE_PRECISION:
95                                              printf("double precision ");
96                                      break;
97                                      case SQL3_DATE_TIME_TIMESTAMP:
98                                      printf("timestamp ");
99                                      break;
100                                     case SQL3_INTERVAL:
101                                             printf("interval ");
102                                     break;
103                                     case SQL3_CHARACTER:
104                                     if          (LENGTH > 0)
105                                             printf("char(%d) ",
106                                                     LENGTH);
107                                     else
108                                             printf("char(?) ");
109                                             break;
110                                     case SQL3_CHARACTER_VARYING:
111                                     if          (LENGTH > 0)
112                                             printf("varchar(%d) ",
113                                                     LENGTH);
114                                     else
115                                             printf("varchar() ");
116                                     break;
117                                     default:
118                                     if          (TYPE < 0)
119                                             printf("<OID %d> ",
120                                                     -TYPE);
121                                     else
122                                             printf("<SQL3 %d> ",
123                                                     TYPE);
124                                     break;
125                             }
126                             putchar('\n');
127                     }
128                     break;
129             }
130             putchar('\n');
131             EXEC SQL CLOSE MYCURS;
```

```
132         EXEC SQL DEALLOCATE DESCRIPTOR MYDESC;
133         return 0;
134 }
```

Am Beginn des Programms werden einige Bibliotheken aufgenommen. Eine davon nennt sich `SQL3TYPES` und enthält Informationen über die Datentypen, die wir im Folgenden benötigen werden.

Was Sie bereits in vorangegangenen Beispielen gelernt haben, werden wir auch diesmal wieder verwenden,nämlich eine Funktion zum Ausgeben von Fehlern. Auch diese Funktion nennt sich wieder `error`.

Nach der Fehlerbehandlungsroutine steigen wir gleich direkt ins Hauptprogramm ein. Dort werden eine Reihe von Variablen deklariert, die wir für die weitere Verarbeitung benötigen werden. Ganz besonders zu beachten ist dabei die Variable `QUERY`, die den SQL-Code der zu analysierenden Abfrage enthalten wird.

Um Informationen über eine Abfrage zu analysieren, benötigen wir einen so genannten Deskriptor. Dieser wird eingesetzt, um Informationen zu extrahieren, die wir anschließend am Bildschirm ausgeben werden.

Bevor wir jedoch eine Verbindung zur Datenbank herstellen, prüfen wir, ob eine Abfrage an das Programm übergeben wurde. Sofern die Bedingung erfüllt ist, werden die Daten von Standard-Input auf `QUERY` zugewiesen.

Des Weiteren wird ein Cursor vereinbart und die Abfrage vorbereitet. Das Ergebnis wird so lange durchgegangen, bis die Schleife terminiert wird. Nach dem Holen des ersten Datensatzes wird der DeScriptor gefüllt. Die Anzahl der Spalten wird ausgegeben und das Programm macht sich daran, jede einzelne Spalte zu analysieren. Auch in diesem Falle kommt wieder der DeScriptor zum Einsatz. Nach dem Ausgeben des Spaltennamens versucht das Programm den Datentyp der Spalte zu finden. Zu diesen Zweck gibt es ein `switch`-Statement. Sofern der korrekte Datentyp gefunden worden ist, wird eine Beschreibung des jeweiligen Datentyps ausgegeben. Danach wird ein Zeilenumbruch gemacht und das Programm terminiert.

Es ist also ohne Probleme möglich, detaillierte Informationen über das Ergebnis einer Abfrage zu generieren. Beim Bauen von Abfrage-Tools ist diese Information essentiell und Sie werden derartige Routinen immer wieder benötigen.

7.7.2 Abfragen von Daten und Spaltenköpfen

In diesem Abschnitt werden Sie lernen, wie ein prototypisches Programm zum Abfragen von Daten und zum Erzeugen von Tabellen mittels ECPG implementiert

werden kann. Ziel ist es, dem Programm ein `SELECT`-Statement übergeben zu können, welches dann ausgeführt und das Ergebnis in Form einer Tabelle dargestellt wird.

Das nächste Listing enthält ein Beispiel für ein derartiges Programm:

```
#include <stdio.h>
#include <stdlib.h>

EXEC SQL INCLUDE sql3types;
EXEC SQL WHENEVER SQLERROR DO error();

void error()
{
        printf("#%ld:%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
           exit(1);
}

int         main(int argc, char **argv)
{
        EXEC SQL BEGIN DECLARE SECTION;
                char NAME[120], STRINGVAR[1024];
                char QUERY[1024];
                int COUNT;
                int INDEX;
        EXEC SQL END DECLARE SECTION;
        EXEC SQL ALLOCATE DESCRIPTOR MYDESC;

        if        (argv[1])
                snprintf(QUERY, sizeof QUERY, argv[1]);
        else
        {
                printf("Keine Query definiert ...\n");
                exit(1);
        }

        EXEC SQL CONNECT TO buch@localhost:5432;

        EXEC SQL PREPARE MYQUERY from :QUERY;
        EXEC SQL DECLARE MYCURS CURSOR FOR MYQUERY;

        EXEC SQL OPEN MYCURS;

        EXEC SQL FETCH IN MYCURS INTO SQL DESCRIPTOR MYDESC;
        EXEC SQL GET DESCRIPTOR MYDESC :COUNT = count;

        printf("Zahl der Spalten: %d\n",COUNT);

```

```
43          for         (INDEX = 1; INDEX <= COUNT; ++INDEX)
44          {
45                  EXEC SQL GET DESCRIPTOR MYDESC VALUE :INDEX
46                          :NAME = name;
47                  printf("%s\t", NAME);
48          }
49          putchar('\n');
50
51          // Durchgehen der Daten
52          EXEC SQL WHENEVER NOT FOUND DO BREAK;
53          while         (1)
54          {
55                  for         (INDEX = 1; INDEX <= COUNT; ++INDEX)
56                  {
57                          EXEC SQL GET DESCRIPTOR MYDESC
58                                  value :INDEX :STRINGVAR = data;
59                          printf("\%s\t", STRINGVAR);
60                  }
61                  putchar('\n');
62                  EXEC SQL FETCH IN MYCURS INTO SQL DESCRIPTOR MYDESC;
63          }
64          EXEC SQL CLOSE MYCURS;
65          EXEC SQL DEALLOCATE DESCRIPTOR MYDESC;
66
67          return 0;
68  }
```

Nach Abarbeiten der Prozeduren, die Sie bereits kennen, wird eine Verbindung zur Datenbank aufgemacht und eine Abfrage ausgeführt. Nach dem Öffnen eines Cursors wird die Anzahl der Spalten in den DeScriptor ausgelesen. Die Anzahl der Spalten werden wir benötigen, um die Ausgabe des Tabellenkopfes und der Daten zu steuern.

Zuallererst wird der Tabellenkopf ausgegeben. Zu diesem Zwecke wird der Name der aktuellen Spalte mithilfe des DeScriptors ausgegeben. Nach der Ausgabe der Tabellentitel wird ein Zeilenumbruch dargestellt.

Nach dem Abarbeiten des Titels sind die Datenzeilen an der Reihe. Beim Abarbeiten der Datenzeilen wird der aktuelle Wert einer Spalte mittels Deksriptor ausgegeben. Zu beachten ist, dass die nächste Zeile am Ende der Schleife ausgelesen wird, da wir andernfalls die erste Zeile im Ergebnis verlieren würden.

Nach Abarbeiten der Schleife werden die üblichen Aufräumarbeiten durchgeführt und das Programm beendet.

Nach dem Kompilieren können Sie das Programm abrufen — das Ergebnis könnte so aussehen:

```
[hs@notebook ecpg]$ ./prog 'SELECT * FROM cd'
Zahl der Spalten: 2
id      name
1       Falco – Out of the Dark
2       Falco – Junge Römer
3       Falco – Nachtflug
4       Falco – Einzelhaft
5       Falco – Einzelhaft
```

Sollte im Ergebnis etwas erscheinen, das wie eine Leerzeile aussieht, so hängt das damit zusammen, dass wir die Länge des Felder `name` auf 128 Zeichen gesetzt haben. Dadurch kann es passieren, dass bei schmalen Ausgabefenstern die Leerzeichen am Schluss einen Zeichenumbruch verursachen. Das ist kein Bug in ihrem Programm, sondern natürliches Verhalten. Sie sollen dadurch erkennen, was beim Arbeiten mit Zeichenketten fixer Länge passieren kann.

Kapitel 8

Perl

Die Geschichte von Perl ist voll von außergewöhnlichen Erfolgen. Perl ist eine Programmiersprache wie keine andere und kann in allen Bereichen der IT eingesetzt werden.

Die Faszination von Perl kann am besten mit einem Zitat von Larry Wall, dem Vater von Perl, beschrieben werden: »It is the magic that counts«. In der Tat, Perl ist magisch und ein Stück Perl-Code versprüht etwas, was man von keiner anderen Programmiersprache kennt. Auf Larry Walls Website (`http://www.wall.org/~larry/pm.html`) gibt es einen interessanten Artikel, in dem Perl als erste postmoderne Computersprache bezeichnet wird. Auch Tim O'Reilly (O'Reilly-Bücher werden vielen von Ihnen ein Begriff sein) ist ein großer Fan von Perl, wie er in seinem Artikel »The Importance of Perl« (`http://perl.oreilly.com/news/importance_0498.html`) eindrucksvoll beweist. Perl ist etwas Besonderes und wird das wohl auch bleiben.

Um die Macht von PostgreSQL und Perl gemeinsam nutzen zu können, bieten sich eine Vielzahl von Möglichkeiten an, die wir in diesem Abschnitt zu erläutern versuchen. Sie werden sehen, welch gewaltige Möglichkeiten sich auftun, wenn sie einmal in die Welt von Perl und PostgreSQL eingetaucht sind. There is more than just one way to do it — tauchen wir also ein in die Welt von Sonderzeichen …

8.1 PL/Perl

PostgreSQL unterstützt eine Reihe eingebetteter Sprachen. Eine dieser Sprachen, die auch als embedded Language verwendet werden kann, ist Perl.

Um PL/Perl einer Ihrer Datenbanken hinzuzufügen, können Sie den Befehl `createlang` verwenden:

```
1 [postgres@duron pgsql]$ createlang plperl buch
```

Bitte bedenken Sie, dass das nur dann funktionieren wird, wenn Sie PostgreSQL mit `–with–perl` kompiliert haben. An dieser Stelle soll nicht verschwiegen werden, dass es auf manchen Systemen notwendig sein kann, Perl mit speziellen Flags zu kompilieren, um PL/Perl für PostgreSQL kompilieren zu können. Die meisten dieser Probleme haben sich mit PostgreSQL 7.2 erübrigt, dennoch wollen wir noch einmal ausdrücklich darauf hinweisen, dass auf manchen Systemen Vorarbeiten notwendig sind. Bei Problemen sollten Sie versuchen, `libperl` als Shared Library zu kompilieren.

8.1.1 Grundlegendes

PL/Perl wird als »trusted language« behandelt. Das heißt, dass gewisse Operationen von Perl nicht erlaubt sind, da es sonst zu Sicherheitslöchern kommen kann.

Denken wir etwa an den Perl Befehl `system`. Mithilfe von `system` kann ein Benutzer sehr viel Schaden anrichten. Aus diesem Grund sind solche Operationen gesperrt und dürfen daher nicht ausgeführt werden.

Wenn Sie PL/Perl explizit als »untrusted language« installieren wollen, so müssen Sie folgenden Befehl verwenden:

```
[postgres@duron html]$ createlang plperlu buch
```

In diesem Fall ist der Autor einer Funktion für die Sicherheit verantwortlich. `plperlu` erlaubt alle Operationen so auch das Arbeiten mit dem Dateisystem oder das Durchführen von System Calls. Wenn Sie PL/Perl untrusted verwenden, sollten Sie sicherstellen, dass niemand, dem Sie nicht absolut vertrauen, auf Ihre Datenbank zugreifen kann.

8.1.2 Einfache Funktionen

Beginnen wir also mit einer einfachen Funktion, die dazu dient, Texte auszugeben:

```
CREATE OR REPLACE FUNCTION hallo(text) RETURNS text AS '
return "hallo ".$_[0];
' LANGUAGE 'plperl';
```

Es wird der Funktion eine Variable übergeben, die von Perl direkt als Standard-Input verwendet werden kann. Der Rückgabewert der Funktion wird als Ergebnis verwendet.

Wenn wir die Funktion mit einem Namen aufrufen, erscheint ein String als Ergebnis:

```
buch=# SELECT hallo('Paul');
   hallo
------------
 hallo Paul
(1 row)
```

Einfache Perl-Funktionen können schnell und mit geringem Aufwand implementiert werden.

Ein wichtiger Punkt beim Arbeiten mit Funktionen ist die Behandlung von `NULL`-Werten. Wenn `NULL`-Werte an eine Perl-Funktion übergeben werden, werden diese innerhalb von Perl als `undef` behandelt. Wenn Sie auf die Variable zugreifen, wird diese jedoch als 0 interpretiert. Das ist wichtig zu wissen, da es immer wieder vorkommen kann, dass `NULL`-Werte übergeben werden, die korrekt zu verarbeiten sind. Sehen wir uns ein Beispiel an. Ziel der Funktion ist es zu prüfen, ob eine Zahl in einem bestimmten Intervall ist oder nicht. Sofern die Zahl im Intervall ist, soll sie zurückgegeben werden, anderenfalls soll `NULL` retourniert werden:

```
CREATE OR REPLACE FUNCTION check_interval(int4, int4, int4)
RETURNS int4 AS '
if      ($_[0] eq undef || $_[1] eq undef || $_[2] eq undef)
{
        return undef;
}

if      ($_[0] < $_[1] || $_[0] > $_[2])
{
        return undef;
}
return $_[0];

' LANGUAGE 'plperl';
```

Am Beginn der Funktion werden alle Parameter auf `NULL` geprüft. Sofern nur einer der Parameter `NULL` enthält, wird `NULL` zurückgegeben. Danach wird geprüft, ob die Zahl im gewünschten Bereich liegt und der entsprechende Wert zurückgegeben.

Versuchen wir, die Funktion zu testen:

```
buch=# SELECT check_interval(3, 5, 5);
 check_interval
----------------

(1 row)

buch=# SELECT check_interval(3, 0, 5);
 check_interval
----------------
              3
(1 row)
```

Die Funktion retourniert die gewünschten Resultate.

8.1.3 Tabellen als Inputparameter

In vielen Fällen kann es sinnvoll sein, mit variablen Parameterlisten zu arbeiten. Da PostgreSQL Funktionsüberladungen unterstützt, ist es prinzipiell möglich, für jede Situation eine Funktion zu definieren aber in vielen Fällen führt das nicht zum gewünschten Ergebnis. Mit PL/Perl ist es möglich, ganze Tabellen als Parameter an eine Funktion zu übergeben. Einer Funktion kann dadurch direkt auf die Spalten einer Tabelle zugreifen.

Das nächste Beispiel legt eine Tabelle an und fügt einige Werte in diese ein. Anschließend werden die Werte in der Tabelle mittels Funktion bearbeitet:

```
CREATE TABLE person (
        vorname text,
        nachname text,
        adresse text,
        ort text
);

COPY person FROM stdin USING DELIMITERS ';';
Ewald;Geschwinde;Hauptstraße 16;Mauerbach
Hans-Jürgen;Schönig;Vierhausgasse 19;Wien
\.

CREATE OR REPLACE FUNCTION anschrift(person) RETURNS text AS '
my ($person) = @_;
$ergebnis = $person->{''vorname''}." ".$person->{''nachname''}.
        " wohnhaft in " .$person->{''adresse''}." in ".
        $person->{''ort''};
return $ergebnis;
' LANGUAGE 'plperl';
```

Wir übergeben den Namen der Tabelle an die Funktion und erlauben es so, dass Perl auf die Felder einer Zeile zugreifen kann. Auf diese Weise ist es nicht notwendig, dass wir jede Spalte der Tabelle einzeln übergeben, was das Implementieren von Funktionen für Tabellen mit vielen Spalten ungleich einfacher macht. Bitte vergessen Sie auf keinen Fall die Klammern um `$person`, da das Programm sonst nicht richtig funktioniert.

Innerhalb der Funktion können wir auf die einzelnen Felder zugreifen, egal, um welchen Datentyp es sich handelt. Es ist unnötig zu erwähnen, welche Macht und Flexibilität Ihnen von Perl und PostgreSQL durch dieses Feature in die Hand gelegt werden, da es dadurch relativ leicht möglich wird, allgemeine Funktionen zu schreiben.

```
buch=# SELECT anschrift(person) FROM person;
                          anschrift
-------------------------------------------------------------
 Ewald Geschwinde wohnhaft in Hauptstraße 16 in Mauerbach
 Hans-Jürgen Schönig wohnhaft in Vierhausgasse 19 in Wien
(2 rows)
```

8.1.4 Datenbankzugriffe

Aktuell ist es noch nicht zuverlässig möglich, von PL/Perl aus auf die Interna der Datenbank zuzugreifen. Alex Pilosov arbeitet jedoch an einem Perl-Modul namens `DBD-PgSPI`, das es erlauben wird, das SPI-Interface über eine genormte DBI-Schnittstelle zu benutzen. Wir werden uns in diesem Kapitel noch eingehend mit dem DBI-Interface beschäftigen.

Eine Funktion, die für jede PL/Perl-Funktion von großer Bedeutung ist, nennt sich `elog` und dient zur Generierung von Logging-Informationen. Das nächste Beispiel zeigt, wie die Funktion verwendet werden kann:

```
CREATE OR REPLACE FUNCTION elog(text) RETURNS bool AS '
elog NOTICE, $_[0];
' LANGUAGE 'plperl';
```

Diesem Listing können Sie entnehmen, was passiert, wenn die Funktion aufgerufen wird:

```
buch=# SELECT elog('das ist eine Nachricht');
NOTICE:  das ist eine Nachricht
 elog
------

(1 row)
```

`elog` unterstützt die folgenden Logging-Nachrichten: `DEBUG`, `ERROR` und `NOTICE`.

8.1.5 Untrusted Perl

Bei sehr speziellen Anwendungen kann es vorkommen, dass Ihnen die trusted Version von Perl nicht mehr ausreicht. In solchen Fällen können Sie entweder auf C-Funktionen zurückgreifen oder untrusted Perl verwenden.

Das folgende Beispiel zeigt, was passiert, wenn wir versuchen, diese Funktion zu starten:

```
CREATE OR REPLACE FUNCTION systemcall(text) RETURNS bool AS '
system("$_[0]");
' LANGUAGE 'plperl';
```

PostgreSQL wird den System Call abfangen und eine Fehlermeldung ausgeben:

```
buch=# SELECT systemcall('ls -l');
ERROR:  creation of function failed: system trapped by operation mask
at (eval 2) line 2.
```

Um das Problem zu lösen, müssen Sie `plperlu` verwenden:

```
DROP FUNCTION systemcall(text);

CREATE OR REPLACE FUNCTION systemcall(text) RETURNS text AS '
$x = system("$_[0]");
return $x;
' LANGUAGE 'plperlu';
```

Da die von uns implementierte Funktion einen anderen Datentyp hat als die vorige Funktion mit demselben Namen, müssen wir die Funktion zuerst explizit löschen — das kann nicht mehr von `CREATE OR REPLACE` bewerkstelligt werden. Das Programm führt die Funktion aus, die wir via Standard-Input übergeben, und gibt den Rückgabewert der Shell an PostgreSQL zurück. Mit `plperlu` kann man alles machen, was man auch sonst mit Perl machen kann und es stehen Ihnen alle Türen offen.

8.2 Das Pg-Modul von PostgreSQL

Wenn Sie den Perl Support beim Kompilieren von PostgreSQL einschalten, wird ein Modul generiert, das eine weitgehend getreue Abbildung der C-Schnittstelle von PostgreSQL für Perl darstellt. Wenn Sie bereits mit der C-Schnittstelle vertraut sind, wird es für Sie kein Problem sein, die Perl-Schnittstelle zur erlernen. In diesem Abschnitt werden wir uns eingehend mit diesem Modul beschäftigen und kennen lernen, wie das Modul effizient eingesetzt werden kann. Dieses Modul ist ein Ansatz, PostgreSQL mit Perl zu kombinieren. Im nächsten Abschnitt werden wir uns mit dem DBI Layer vertraut machen und sehen, welche Vorzüge dieser zu bieten hat, aber beginnen wir einfach mit einem Überblick über das Pg-Modul.

8.2.1 Verbinden zur Datenbank

Am Beginn jeder Applikation ist es notwendig, eine Verbindung zur Datenbank herzustellen. Im Fall des Pg-Modules kann das leicht mit dem Befehl `connectdb` bewerkstelligt werden, wie das folgende Beispiel zeigt:

```
1  #!/usr/bin/perl
2
3  use Pg;
4
5  $handle = Pg::connectdb("dbname=buch user=postgres");
6  if      (!$handle)
7  {
8          print "Die Verbindung konnte nicht erstellt werden.\n";
9          exit 1;
10 }
11 else
12 {
13         print "Die Verbindung wurde erfolgreich erstellt.\n";
14 }
```

Sofern Sie PostgreSQL korrekt kompiliert haben und alle Verbindungsparameter stimmen, können Sie eine funktionierende Verbindung zu PostgreSQL aufbauen.

Das Pg-Modul unterstützt zwei Varianten von Befehlen. Das vorangegangene Beispiel basiert bereits auf den neuen Funktionen. Aus Kompatibilitätsgründen ist die alte Version der Funktionen jedoch noch verfügbar. In diesem Buch werden wir uns schwerpunktmäßig mit den neuen Funktionen beschäftigen.

Bevor wir uns an das Arbeiten mit Abfragen machen, betrachten wir, wie die Parameter einer Abfrage extrahiert werden können. Zu diesem Zwecke stellt das Pg-Modul einige einfache Funktionen zur Verfügung, die im nächsten Beispiel zusammengefasst sind:

```
#!/usr/bin/perl

use Pg;

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann Verbindung nicht erstellen\n";

print "Host: ". $handle->host()."\n";
print "User: ". $handle->user()."\n";
print "Pass: ". $handle->pass()."\n";
print "Port: ". $handle->port()."\n";
print "Tty: ". $handle->tty()."\n";
print "Options: ". $handle->options()."\n\n";
print "Status: ". $handle->status()."\n";

if      ($handle->status() eq PGRES_CONNECTION_OK)
{
        print "Der Rückgabewert entspricht PGRES_CONNECTION_OK\n";
}
```

Wenn Sie das Programm ausführen, werden Sie die entsprechenden Werte im Ergebnis erhalten:

```
[postgres@duron tmp]$ ./connect.pl
Host:
User: postgres
Pass:
Port: 5432
Tty:
Options:

Status: 0
Der Rückgabewert entspricht PGRES_CONNECTION_OK
```

Hierbei sind einige Felder leer, da wir diese auch nicht beim Aufbauen der Verbindung angegeben haben.

8.2.2 Einfügen und Abfragen von Daten

Das Abfragen und Einfügen von Daten ist die Kernaufgabe einer jeden Datenbankapplikation. Nachdem Sie eine Abfrage abgesetzt haben, ist es notwendig, den Status des Ergebnisses abzufragen. Um ein Statement auszuführen, können Sie den Befehl exec verwenden. Für die Analyse des Zustandes des Abfrage stellt das Pg-Modul einige Konstanten zur Verfügung, die bequem abgefragt werden können. Im nächsten Beispiel wird ein einfaches SQL-Statement ausgeführt und der Rückgabewert geprüft:

```
1  #!/usr/bin/perl
2
3  use Pg;
4
5  $handle = Pg::connectdb("dbname=buch user=postgres") or
6          die "kann keine Verbindung aufbauen\n";
7
8  $result = $handle->exec("SELECT 1+1") or
9          die "kann Ergebnis nicht berechnen\n";
10
11 $status = $result->resultStatus();
12 if ($status eq PGRES_COMMAND_OK) { print "PGRES_COMMAND_OK\n"; }
13 if ($status eq PGRES_TUPLES_OK) { print "PGRES_TUPLES_OK\n"; }
14 if ($status eq PGRES_COPY_OUT) { print "PGRES_COPY_OUT\n"; }
15 if ($status eq PGRES_COPY_IN) { print "PGRES_COPY_IN\n"; }
16 if ($status eq PGRES_BAD_RESPONSE) { print "PGRES_BAD_RESPONSE\n"; }
17 if ($status eq PGRES_NONFATAL_ERROR) { print "PGRES_NONFATAL_ERROR\n"; }
18 if ($status eq PGRES_FATAL_ERROR) { print "PGRES_FATAL_ERROR\n"; }
```

Hier haben wir eine Liste aller Konstanten aufgeführt, die von PostgreSQL zur Verfügung gestellt werden. Mithilfe der If-Abfragen können Sie herausfinden, welchen Wert der Status hat. Wenn Sie eine Abneigung gegen derartige If-Konstruktionen haben, können Sie auch case-Anweisungen verwenden.

Beim Ausführen des Beispiels werden Sie den folgenden Output erhalten:

```
1  [postgres@duron code]$ ./connect.pl
2  PGRES_TUPLES_OK
```

Sofern PGRES_TUPLES_OK zurückgegeben wird, hat PostgreSQL die Abfrage korrekt abgesetzt und Sie können die Daten der Abfrage bequem aus dem Ergebnis extrahieren.

Sie haben gelernt, wie das Ergebnis geprüft werden kann. Jetzt wenden wir uns einem Stück Code zu, das nicht nur den Rückgabewert prüft. Bevor wir uns jedoch den Code betrachten, legen wir eine Tabelle mit Daten an:

```
CREATE TABLE ort (
        name       text,
        einwohner  int4,
        staat      text
);

COPY ort FROM stdin USING DELIMITERS ';';
Wien;1634209;Österreich
Graz;245903;Österreich
Los Cancachos;43;Spanien
\.
```

Unterstehend ist ein Programm, das die Ergebnisse einer Abfrage genau analysiert:

```
#!/usr/bin/perl

use Pg;

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$result = $handle->exec("SELECT name, einwohner FROM ort") or
        die "kann Ergebnis nicht berechnen\n";

# prüfen, ob das Ergebnis korrekt ist
if      ($result->resultStatus() eq PGRES_TUPLES_OK)
{
        print "Felder: ".$result->nfields."\n";
        print "Tuples: ".$result->ntuples."\n";

        print "Feldnamen: ".$result->fname(0)
                ." - ".$result->fname(1)."\n";

        print "Feldnummer: ".$result->fnumber('name')
                ." - ".$result->fnumber('einwohner')."\n";

        print "Datentypen: ".$result->ftype(0)
                ." - ".$result->ftype(1)."\n";

        print "Größe: ".$result->fsize(0)
                ." - ".$result->fsize(1)."\n";
}
```

Aus dem Listing geht hervor, dass PostgreSQL eine ganze Menge an Funktionen bereitstellt, die zur Analyse des Ergebnisses herangezogen werden können. Das vorliegende Beispiel zeigt, wie die Anzahl der Felder, die Anzahl der Datensätze, die Feldnamen, die Feldnummern sowie die Datentypen und Größen abgefragt werden können.

Das folgende Listing zeigt, was passiert, wenn wir das Programm ausführen:

```
[hs@duron tmp]$ ./connect.pl
Felder: 2
Tuples: 3
Feldnamen: name – einwohner
Feldnummer: 0 – 1
Datentypen: 25 – 23
Größe: –1 – 4
```

Da das SQL-Statement genau zwei Spalten zurückgibt, ist die Anzahl der Felder logischerweise genau zwei. In der Tabelle befinden sich drei Datensätze. Die Namen der Felder sowie deren Indices können ebenfalls ohne Probleme extrahiert werden.

Ein wichtiger Punkt ist die Abfrage der Datentypen. Dabei werden Zahlen und keine Namen zurückgegeben. Intern verwendet PostgreSQL immer Zahlen, da der Name eines Datentyps relativ leicht geändert werden kann. Wenn Sie wissen wollen, was die einzelnen Zahlen bedeuten, müssen Sie in den internen Systemtabellen nachsehen, die wir jedoch an dieser Stelle nicht behandeln werden.

Der Speicherplatz, der von einem Datentyp belegt wird, kann bei Text nicht eindeutig bestimmt werden, da Textfelder von variabler Länge sind. Der Rückgabewert der Funktion ist daher -1. Die Länge der zweiten Spalte ist vier Bytes, da es sich hier um einen Integer-Wert handelt.

Nach diesem ausgiebigen Exkurs können wir uns nun endgültig einigen einfachen Abfragen widmen. Das folgende Beispiel zeigt, wie Daten abgefragt und ausgegeben werden können:

```
#!/usr/bin/perl

use Pg;
use strict;

my ($handle, $result, $i, $j);

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$result = $handle->exec("SELECT name, einwohner FROM ort") or
        die "kann Ergebnis nicht berechnen\n";

# prüfen, ob das Ergebnis korrekt ist
if      ($result->resultStatus() eq PGRES_TUPLES_OK)
{
        for     ($i = 0; $i < $result->nfields; $i++)
```

```
18          {
19                  for     ($j = 0; $j < $result->ntuples; $j++)
20                  {
21                          print "i: $i, j: $j - wert: "
22                                  .$result->getvalue($j, $i)."\n";
23                  }
24          }
25  }
```

Die Verbindung zur Datenbank wird wie gewohnt erstellt. Anschließend führen wir das `SELECT`-Statement aus. Nach dem Prüfen des Status werden die Daten zellenweise ausgegeben. Zu diesem Zwecke haben wir zwei Schleifen implementiert. Die Schleifen werden so lange aktiv sein bis alle Daten abgearbeitet sind. Das Ergebnis wird mittels `print` ausgegeben. Auf diese Weise erhalten wir eine Liste aller Werte inklusive deren Position in der Ergebnismatrix:

```
1 [postgres@duron tmp]$ ./connect.pl
2 i: 0, j: 0 - wert: Wien
3 i: 0, j: 1 - wert: Graz
4 i: 0, j: 2 - wert: Los Cancachos
5 i: 1, j: 0 - wert: 1634209
6 i: 1, j: 1 - wert: 245903
7 i: 1, j: 2 - wert: 43
```

Der Befehl `getvalue` erfüllt alle Wünsche; Sie müssen nur definieren, welchen Wert Sie gerade abfragen wollen und `getvalue` erledigt den Rest für Sie.

Wenn Sie `INSERT`-Operationen durchführen, ist es oft sinnvoll, die Object ID des neuen Datensatzes abzufragen. Viele Applikationen verwenden die OID als Identifier, weil sie eindeutig ist und daher leicht zur Identifikation eines Datensatzes herangezogen werden kann. Betrachten wir also, was man machen kann, um diese Information zu gewinnen. Ziel der nächsten Applikation ist es, einen Datensatz einzufügen und dessen OID auszugeben:

```
1  #!/usr/bin/perl
2
3  use Pg;
4  use strict;
5
6  my ($handle, $result);
7
8  $handle = Pg::connectdb("dbname=buch user=postgres") or
9          die "kann keine Verbindung aufbauen\n";
10
11 $result = $handle->exec("INSERT INTO ort VALUES "
12         ."('Leoben', '43254', 'Österreich')") or
13                 die "kann Operation nicht durchführen\n";
```

```
14
15 print "Status: ".$result->cmdStatus."\n";
16 print "OID: ".$result->oidStatus."\n";
```

Nach dem Ausführen des `INSERT`-Statements rufen wir zwei Funktionen auf. Die erste Funktion retourniert den aufgerufenen Befehl und die Zahl der modifizierten Datensätze. Die zweite Funktion gibt lediglich die Object ID des neuen Datensatzes zurück.

Der Output des Programmes könnte wie folgt aussehen:

```
1 [postgres@duron tmp]$ ./connect.pl
2 Status: INSERT 354849 1
3 OID: 354849
```

Wenn wir statt `INSERT` den Befehl `DELETE FROM ort` verwenden, sieht der Output so aus:

```
1 [postgres@duron tmp]$ ./connect.pl
2 Status: DELETE 4
3 OID:
```

Dabei werden zwar der Befehl und die Anzahl der gelöschten Datensätze ausgegeben, das Feld mit der OID bleibt allerdings leer, weil es ja keinen neuen Datensatz gibt.

8.2.3 Behandeln von NULL-Werten und Feldlängen

`NULL`-Werte sind eine ganze spezielle Sache und sollten auch so behandelt werden. Das Pg-Modul stellt eine Funktion zur Verfügung, die es ermöglicht zu prüfen, ob eine Zelle einen `NULL`-Wert enthält oder nicht. Diese Funktion ist immer dann von großer Bedeutung, wenn es darum geht, `NULL`-Werte getrennt zu behandeln und Ausnahmen abzufangen.

Eine weitere wichtige Funktion, der wir uns in diesem Abschnitt widmen werden, dient zum Abfragen der Länge eines Feldes. Es ist sinnvoll, diese Funktion im Zusammenhang mit `NULL`-Werten zu erwähnen, weil man auf diese Weise leicht erkennen kann, welche Länge ein `NULL`-Wert hat.

Das folgende Beispiel schickt eine Abfrage an die Datenbank und analysiert das Ergebnis:

```
1 #!/usr/bin/perl
2
3 use Pg;
4
```

```
$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$result = $handle->exec("SELECT NULL, 'ein text'");

print "Test auf NULL:\n";
print "Spalte 1: ".$result->getisnull(0, 0)."\n";
print "Spalte 2: ".$result->getisnull(0, 1)."\n\n";

print "Längenabfrage:\n";
print "Spalte 1: ".$result->getlength(0, 0)."\n";
print "Spalte 2: ".$result->getlength(0, 1)."\n";
```

Die Funktion `getisnull` gibt `true` zurück, wenn es sich um einen `NULL`-Wert handelt. Mithilfe der Funktion `getlength` kann die Länge eines Feldes abgefragt werden. Wenn wir das Programm ausführen, erhalten wir Folgendes:

```
[hs@duron tmp]$ ./connect.pl
Test auf NULL:
Spalte 1: 1
Spalte 2: 0

Längenabfrage:
Spalte 1: 0
Spalte 2: 8
```

Da die erste Spalte ein `NULL`-Wert ist, wird `true` zurückgegeben. Des Weiteren hat ein `NULL`-Wert die Länge 0. Das ist wichtig zu beachten und hilft Ihnen in vielen Fällen weiter.

8.2.4 Arbeiten mit `COPY`

Wenn Sie mit mehr als nur einigen wenigen Datensätzen arbeiten müssen, empfiehlt es sich sich aus Performancegründen, dass Sie mit `COPY` anstatt mit `INSERT` arbeiten. Das bringt meistens signifikante Geschwindigkeitsgewinne.

Das Pg-Modul stellt einige Funktionen zur Verfügung, die Ihnen helfen werden, `COPY` effizient zu verwenden. Werfen wir einen Blick auf den folgenden Datenbestand:

```
Wien;1634209;Österreich
Graz;245903;Österreich
Los Cancachos;43;Spanien
```

Sie kennen den Datenbestand bereits aus den letzten Beispielen. In diesem Abschnitt werden wir die Daten aber mittels Perl-Script in die Datenbank einfügen:

```
#!/usr/bin/perl

use Pg;
use strict;

my ($handle, $result);
open (DATA, "< data.sql") or die "kann Datei nicht öffnen.\n";

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$result = $handle->exec("COPY ort FROM stdin USING DELIMITERS ';'") or
                die "kann Operation nicht durchführen\n";

while   (<DATA>)
{
        $handle->putline($_);
}
close(DATA);
$handle->putline("\\.\n");
```

Die Daten im File `data.sql` werden zeilenweise abgearbeitet und mittels `putline` direkt an das Backend geschickt. Dieses verarbeitet die Daten und wartet, bis der `COPY`-Befehl terminiert wird. Das Terminieren von `COPY` erfolgt wie gewöhnt.

Nachdem Sie das Programm ausgeführt haben, werden Sie die Daten in der Tabelle finden:

```
buch=# SELECT * FROM ort;
     name      | einwohner |   staat
---------------+-----------+-----------
 Wien          |   1634209 | Österreich
 Graz          |    245903 | Österreich
 Los Cancachos |        43 | Spanien
(3 rows)
```

8.2.5 Tracing

Wie Sie bereits in anderen Kapiteln erfahren haben, ist es ohne größeren Aufwand möglich, eine Verbindung zur Datenbank zu tracen.

Tracing bedeutet, dass man sich die Kommunikation mit dem Backend ansieht, um genauere Informationen über die internen Abläufe zu bekommen. Wollen wir uns also ein einfaches Beispiel ansehen, das uns zeigt, wie man Informationen ohne großen Aufwand auf Standard Error schreiben kann:

```
#!/usr/bin/perl

use Pg;

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$handle->trace( STDERR );
$result = $handle->exec("SELECT 1+1");

$handle->untrace();
```

Um Tracing einzuschalten, kann der Befehl `trace` verwendet werden. Nach diesem Befehl wird die gesamte Information an `Standard Error` geschickt. Um dieses Verhalten zu beenden, kann `untrace` verwendet werden.

Das nächste Listing zeigt die von PostgreSQL generierten Informationen, wenn wir 1+1 rechnen:

```
[hs@duron code]$ ./connect.pl
To backend> Q
To backend> SELECT 1+1
From backend> P
From backend> "blank"
From backend> T
From backend (#2)> 1
From backend> "?column?"
From backend (#4)> 23
From backend (#2)> 4
From backend (#4)> -1
From backend> D
From backend (1)> ~@
From backend (#4)> 5
From backend (1)> 2
From backend> C
From backend> "SELECT"
From backend> Z
From backend> Z
```

Das Listing ist relativ lang, enthält jedoch einige recht interessante Informationen. Es ist etwa ersichtlich, welche Nachrichten vom Backend kommen beziehungsweise welche Informationen an das Backend geschickt werden. In diesem Beispiel führen wir eine Query (Q) aus. Die vom Backend zurückgegebenen Daten enthalten wesentlich mehr Informationen: Da das Ergebnis nicht nur aus den gelieferten Daten, sondern auch aus Zusatzinformation besteht, ist das Listing etwas länger. Um aufzuzeigen, welche Daten retourniert werden, wollen wir auf einige Zeilen genauer eingehen. Diese Informationen sind für ein grundlegendes Verständis der Frontend-Backendinformationen von großer Bedeutung.

Sehen wir uns die Zeile

```
From backend (\#4)> 23
```

nun etwas genauer an: Da der Rückgabewert der Berechnung integer ist, wird das von PostgreSQL mitgeteilt. Sie wissen, dass Datentypen intern als Zahlen repräsentiert sind — 23 entspricht einer Integer-Zahl. Dieses Beispiel zeigt sehr deutlich, dass viele Funktionen, die vom Pg-Modul zur Verfügung gestellt werden, direkt auf die vom Backend gelieferten Daten zurückgreifen können, was den gesamten Ablauf sehr effizient gestaltet.

8.2.6 Exception Handling

Fehlerbehandlung ist ein essentieller Teil einer jeden Applikation. Auch bei Perl und dem Pg-Modul ist das nicht anders. In diesem Abschnitt werden wir erfahren, welche grundlegenden Möglichkeiten das Pg-Modul zur Verfügung stellt und wie Fehler effizient behandelt werden können.

Beginnen wir mit einem einfachen Beispiel:

```
#!/usr/bin/perl

use Pg;

$handle = Pg::connectdb("dbname=buch user=postgres") or
        die "kann keine Verbindung aufbauen\n";

$result = $handle->exec("SELECT geht nicht ...");
print $result->getvalue(0, 0)."\n";
```

Das `SELECT`-Statement wird nicht funktionieren beziehungsweise keine sinnvollen Werte liefern. Daraus resultiert, dass auch `getvalue` nicht sinnvoll angewendet werden kann:

```
[hs@duron code]$ ./connect.pl
row number 0 is out of range 0..-1
```

Die Datenbank beschwert sich, dass die abgefragten Werte nicht im gültigen Bereich sind. Um den durch das `SELECT` Statement auftretenden Fehler abzufangen, können Sie wie folgt arbeiten:

```
#!/usr/bin/perl

use Pg;

$handle = Pg::connectdb("dbname=buch user=postgres") or
```

```
6         die "kann keine Verbindung aufbauen\n";
7
8 $result = $handle->exec("SELECT geht nicht ...");
9
10 if      ($result->cmdStatus() eq PG_TUPLES_OK)
11 {
12        print $result->getvalue(0, 0)."\n";
13 }
14 else
15 {
16        print "Ein Fehler ist aufgetreten\n";
17        print "Fehlercode: ".$handle->errorMessage();
18 }
```

Die `If`-Abfrage stellt sicher, dass der Block nur dann aufgeführt wird, wenn PostgreSQL sinnvolle Daten geliefert hat. Sofern das nicht der Fall ist, wird der `else`-Zweig abgerufen. Mithilfe der Funktion `errorMessage()` kann die letzte vom Backend gelieferte Fehlermeldung ausgegeben werden. Bei uns wird das Ergebnis so aussehen:

```
1 [hs@duron code]$ ./connect.pl
2 Ein Fehler ist aufgetreten
3 Fehlercode: ERROR:  parser: parse error at or near "nicht"
```

Da wir kein gültiges `SELECT`-Statement an den Server geschickt haben, liefert die Datenbank einen Fehler.

8.2.7 Fazit

Das Pg-Modul ist eine direkte Nachbildung der C-Schnittstelle. Das führt dazu, dass sich das Modul oft nicht ganz nach Perl anfühlt. Der Vorteil ist aber, dass Sie die Schnittstelle leicht erlernen können, wenn Sie bereits mit C gearbeitet haben.

8.3 DBI-Programmierung

Wenn Sie sich mit der hohen Schule der Datenbankprogrammierung unter Perl beschäftigen, werden Sie kaum am DBI-Modul von Tim Bunce vorbeikommen. Das DBI-Modul ist ein Abstraktionslayer, der eine genormte Schnittstelle für eine Vielzahl von Datenbanken bereitstellt. Ein vollständiges System besteht im Prinzip aus zwei Komponenten: Der DBI Layer ist die genormte Abstraktionsschicht, die für die Interaktion mit der Datenbank auf einen so genannten DBD-Treiber zurückgreift. Der DBD-Treiber enthält alle datenbankspezifischen Komponenten und kann nativ mit der Datenbank kommunizieren. Durch die zusätzliche Abstraktionsschicht ist es vergleichsweise einfach, Anwendungen zu schreiben, die

unabhängig von der darunter liegenden Datenbank sind. Ein Problem wird sich jedoch immer stellen: Da die verschiedenen SQL-Akzente nicht unter einen Hut zu bekommen sind, wird es kaum möglich sein, völlig unabhängige Applikationen zu schreiben, aber mithilfe von DBI werden Sie diesem Ziel einen gewaltigen Schritt näher rücken.

Dieser Abschnitt ist allen Fans von DBI gewidmet und soll die grundlegende Arbeitsweise mit diesem Layer erläutern und anhand von praktischen Beispielen demonstrieren.

8.3.1 Installation

Zuallererst müssen Sie das DBI-Modul downloaden und installieren. Um die neueste Version des Moduls zu downloaden, surfen Sie am besten zu `http://search.cpan.org`. Auf dieser Website können Sie dann bequem nach dem gewünschten Modul suchen.

Nach dem Download können Sie das Modul entpacken, was folgendermaßen funktioniert:

```
1 tar xvfz DBI-1.21.tar.gz
```

Um das Modul zu installieren, benötigen Sie die folgenden Befehle:

```
1 perl Makefile.PL
2 make && make test && make install
```

Sofern es zu keinen Problemen gekommen ist, können Sie das Modul jetzt verwenden.

Nach der Installation des DBI-Moduls können Sie sich dem DBD-Treiber für PostgreSQL zuwenden. Nach dem Download des Moduls von obiger Site können Sie dieses installieren. Im ersten Schritt müssen Sie das Modul wieder entpacken:

```
1 tar xvfz DBD-Pg-1.13.tar.gz
```

Jetzt können Sie versuchen, das Makefile zu erzeugen:

```
1 [root@duron DBD-Pg-1.13]# perl Makefile.PL
2 Configuring Pg
3 Remember to actually read the README file !
4 please set environment variables POSTGRES_INCLUDE and POSTGRES_LIB !
```

Wenn Sie den Source-Code von PostgreSQL anstatt der Binaries installiert haben, werden die Meldungen wie im letzten Listing ausgegeben. Sofern das bei Ihnen der Fall ist, müssen Sie diese Environment-Variablen zumindest temporär setzen, was so funktioniert:

```
[root@duron DBD-Pg-1.13]# export POSTGRES_INCLUDE=
/usr/local/postgresql/include/
[root@duron DBD-Pg-1.13]# export POSTGRES_LIB=/usr/local/postgresql/lib/
```

Sofern Sie die korrekten Pfade angegeben haben, können Sie das Makefile jetzt ohne Probleme generieren:

```
[root@duron DBD-Pg-1.13]# perl Makefile.PL
Configuring Pg
Remember to actually read the README file !
OS: linux
Using DBI 1.21 installed in
/usr/lib/perl5/site_perl/5.6.1/i386-linux/auto/DBI
Checking if your kit is complete...
Looks good
Writing Makefile for DBD::Pg
```

Nach diesem Schritt können Sie die Sourcen wie folgt kompilieren:

```
make && make test && make install
```

In der Regel kommt es dabei zu keinen Problemen und Sie können die Module nun verwenden.

8.3.2 Datenbankverbindungen

Wie bei allen datenbankgestützten Applikationen müssen Sie zuerst eine Verbindung zur Datenbank herstellen. Bei einem DBI-Modul kann das mit dem Befehl `connect` erfolgen. Dieser Befehl gilt für alle Datenbanken.

Das nächste Listing enthält ein Stück Code, das zeigt, wie eine Verbindung zu PostgreSQL hergestellt werden kann:

```
#!/usr/bin/perl

use DBI;
use strict;

my $string="dbi:Pg:dbname=buch;host=localhost;port=5432";
my $dbh=DBI->connect("$string","postgres","") or
        die "kann nicht zur Datenbank verbinden\n";

print "Die Verbindung konnte hergestellt werden ...\n";
$dbh->disconnect();
```

Nach dem Einbinden der benötigten Module definieren wir einen String, der einige Verbindungsparameter enthält. Dann rufen wir `connect` auf und setzen den String

ein. Der zweite Parameter der `connect`-Funktion enthält den Benutzer, unter dem wir uns anmelden wollen. Der dritte Parameter dient zur Übergabe eines Passwortes. In unserem Fall ist dieses Feld leer, da wir uns als Superuser einloggen.

Sofern die Verbindung zur Datenbank erstellt werden kann, wird der Output des Programmes wie folgt aussehen:

```
[hs@duron code]$ ./connect.pl
Die Verbindung konnte hergestellt werden ...
```

Sofern wir die Parameter, die zum Erstellen der Verbindung notwendig sind, nicht explizit definieren, werden die Werte aus Environment-Variablen verwendet. Die folgende Tabelle beschreibt eine Liste der verwendeten Variablen beziehungsweise der defaultmäßig verwendeten Werte:

Parameter	Environment-Variable	Defaultwert
`dbname`	PGDATABASE	Name des aktuellen Benutzers
`host`	PGHOST	localhost
`port`	PGPORT	5432
`options`	PGOPTIONS	
`tty`	PGTTY	
`username`	PGUSER	Name des aktuellen Benutzers
`password`	PGPASSWORD	

Das Herstellen einer Verbindung zur Datenbank geht wie gewohnt vor sich und Sie müssen sich keine Gedanken über die Interna machen, wenn Sie Ihre Perl-Applikationen entwickeln.

Oft kann es essentiell sein, herauszufinden, ob ein Datenbankhandle noch verwendbar ist oder nicht. Zu diesem Zwecke stellt der DBI Layer eine Funktion names `ping` zur Verfügung:

```
#!/usr/bin/perl -w

use DBI;

$string="dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh=DBI->connect("$string","postgres","") or
        die "kann nicht zur Datenbank verbinden\n";

$dbh->ping or die "Handle ungültig.\n";
$dbh->disconnect();
```

Sofern das Programm nicht mit einem Fehler abbricht, hat `ping` funktioniert. Intern tut `Ping` nichts anderes, als eine leere Abfrage an die Datenbank zu schicken, um zu sehen, ob die Datenbank auch korrekt antwortet. Bei unsicheren Verbindungen kann das wichtig sein und das Leben eines Applikationsentwicklers enorm erleichtern.

8.3.3 Abfragen

Nach diesem Überblick über Datenbankverbindungen wenden wir uns dem Abarbeiten von SQL-Statements zu. In diesem Abschnitt werden Sie schnell erkennen, wir mächtig der DBI Layer mittlerweile geworden ist und welche Fülle von Möglichkeiten zur Verfügung gestellt werden.

Beginnen wir mit einem einfachen Beispiel, das zeigt, wie eine einfache Berechnung durchgeführt werden kann:

```
1  #!/usr/bin/perl
2
3  use DBI;
4  use strict;
5
6  my $string="dbi:Pg:dbname=buch;host=localhost;port=5432";
7  my $dbh=DBI->connect("$string","postgres","") or
8          die "kann nicht zur Datenbank verbinden\n";
9
10 my $sql = "SELECT 1+1 AS summe";
11 my $sth=$dbh->prepare($sql) or
12         die "kann Abfrage nicht vorbereiten\n";
13 $sth->execute() or
14         die "kann Abfrage nicht durchführen\n";
15 my @row = $sth->fetchrow_array();
16 print "Ergebnis: ".$row[0]."\n";
17 $sth->finish;
18
19 $dbh->disconnect();
```

Nachdem wir eine Verbindung zur Datenbank erstellt haben, definieren wir eine Variable, die das SQL-Statement enthält. Diese Variable wird dann an die Funktion `prepare` übergeben. Der Rückgabewert von `prepare` ist ein so genannter Statement Handle. Dieser Handle kann letztendlich zum Ausführen der eigentlichen Abfrage verwendet werden. Dieses zweistufige Verfahren hat einige Vorteile, auf die wir in später folgenden Beispielen noch eingehen werden.

Wenden wir uns noch einmal der Abfrage der Daten zu. In diesem Beispiel haben wir die Funktion `fetchrow_array` verwendet. Wie der Name schon erahnen lässt, gibt die Funktion ein Array zurück. Auf diese Weise können die Daten leicht

abgefragt und weiterverarbeitet werden. Meistens ist es jedoch sinnvoll, bei der Datenabfrage von Arrays Abstand zu nehmen. In solchen Fällen sind andere Datenstrukturen, die von Perl zur Verfügung gestellt werden, sinnvoller. Bevor wir einen Blick auf das nächste Beispiel werfen, wollen wir uns den Datenbestand in Erinnerung rufen, das wird Ihnen sinnlose Blätterarbeit ersparen:

```
CREATE TABLE ort (
        name text, einwohner int4, staat text
);

COPY ort FROM stdin USING DELIMITERS ';';
Wien;1634209;Österreich
Graz;245903;Österreich
Los Cancachos;43;Spanien
\.
```

Das nächste Beispiel zeigt, wie ein `SELECT`-Statement ausgeführt werden kann und wie Daten mithilfe einer Datenstruktur abgerufen werden können:

```
#!/usr/bin/perl -w

use DBI;
use strict;

my $string="dbi:Pg:dbname=buch;host=localhost;port=5432";
my $dbh=DBI->connect("$string","postgres","") or
        die "kann nicht zur Datenbank verbinden\n";

my $sql = "SELECT name, einwohner, staat FROM ort";
my $sth=$dbh->prepare($sql) or die "kann Abfrage nicht vorbereiten\n";
$sth->execute() or die "kann Abfrage nicht durchführen\n";

my $row_ref;
while   ($row_ref = $sth->fetchrow_arrayref)
{
        print "$row_ref->[0] - $row_ref->[1] - $row_ref->[2]\n";
}

$sth->finish;
$dbh->disconnect();
```

Hierbei unterscheiden sich die beiden letzten Beispiele nur durch die intern verwendete Datenstruktur und die Syntax.

Viele arbeiten lieber mit Hashes als mit Arrays. Auch dafür bietet das DBI-Modul die passenden Funktionen. Sehen wir uns den Block der Datenextraktion in einer Variante an, die auf Hashes beruht:

```
my $hash_ref;
while   ($hash_ref = $sth->fetchrow_hashref)
{
        print "$hash_ref->{name} - $hash_ref->{einwohner} - "
                ."$hash_ref->{staat}\n";
}
```

Um auf einen Hash zugreifen zu können, benötigen wir den Namen des Feldes. Das hat viele Vorteile, weil man auf diese Weise relativ unabhängig von der darunter liegenden Datenstruktur agieren kann und Namensänderungen nicht zu schleichenden Bugs, sondern zu Fehlern führen, was als Vorteil bezeichnet werden kann. Es können sich keine verborgenen Fehler einschleichen, die nur mehr schwer zu beheben sind.

Wenn wir das Programm ausführen, werden die Daten wie erwartet ausgegeben:

```
Wien - 1634209 - Österreich
Graz - 245903 - Österreich
Los Cancachos - 43 - Spanien
```

In vielen Fällen kann es sinnvoll sein, den gesamten Datenbestand auf einmal abzufragen. Zu diesem Zwecke kann die Funktion `fetchall_arrayref` verwendet werden. Das nächste Listing enthält den relevanten Block:

```
my $table = $sth->fetchall_arrayref or die "Fehler ...\n";
my($i, $j);
for $i ( 0 .. $#{$table} )
{
        for $j ( 0 .. $#{$table->[$i]} )
        {
                print "$table->[$i][$j] \t ";
        }
        print "\n";
}
```

Die Tabelle wird mit nur einem Befehl abgefragt. Weiterhin werden die Daten zellenweise ausgegeben. Dafür werden zwei Schleifen benötigt, die jeweils die einzelnen Zeilen beziehungsweise Spalten durchlaufen. Dem Rautezeichen kommt große Bedeutung zu, da es verwendet wird, um herauszufinden, wie viele Zeilen und Spalten im Ergebnis zu finden sind.

Dieser Abschnitt zeigt in aller Deutlichkeit, welche Möglichkeiten das DBI-Modul bereitstellt und wie Sie Abfragen durchführen können.

8.3.4 Exception Handling

Wie das Pg-Modul stellt auch das DBI-Modul einige effiziente Möglichkeiten zur Fehlerbehandlung zur Verfügung.

Werfen wir einen Blick auf ein Beispiel:

```
1  #!/usr/bin/perl -w
2
3  use DBI;
4
5  $string="dbi:Pg:dbname=buch;host=localhost;port=5432";
6  $dbh=DBI->connect("$string","postgres","") or
7          die "kann nicht zur Datenbank verbinden\n";
8
9  $sql = "SELECT not there";
10 $sth=$dbh->prepare($sql);
11 $sth->execute() or die "Fehler: ".$dbh->errstr."\n";
12
13 print "wird nicht mehr ausgefuehrt ...\n";
14
15 $sth->finish;
16 $dbh->disconnect();
```

Da das SQL-Statement scheitern muss, ist es notwendig, einen Fehler auszugeben. Hier geschieht das mithilfe der Funktion `errstr`, die den letzten Fehler enthält, der aufgetreten ist.

Wenn wir das Programm ausführen, können wir beobachten, wie sich Perl verhält:

```
1  [hs@duron tmp]$ ./connect.pl
2  DBD::Pg::st execute failed: ERROR:  Attribute 'there' not found at
3  ./connect.pl line 11.
4  Fehler: ERROR:  Attribute 'there' not found
```

Ein Fehler wird ausgegeben und das Programm bricht ab.

Um den Status eines Handles abzufragen, sieht der DBI Layer eine Funktion namens `state` vor. Alle die mit dieser Funktion bereits vertraut sind, müssen bei PostgreSQL verzichten, da es diese Funktion in der derzeit aktuellen Version leider noch nicht gibt.

8.3.5 Durchführen von Modifikationen

Bisher haben wir nur Abfragen losgeschickt. Abfragen liefern in der Regel Daten zurück, die es weiterzuverarbeiten gilt. Meistens ist es jedoch notwendig, Operationen durchzuführen, die keine Daten liefern. Dabei kann die Funktion `do` verwendet werden, wie das im folgenden Beispiel gezeigt wird:

```
#!/usr/bin/perl

use DBI;

$string="dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh=DBI->connect("$string","postgres","") or
        die "kann nicht zur Datenbank verbinden\n";

$sql = "CREATE TEMPORARY TABLE math (code text, ergebnis int4)";
$dbh->do($sql) or die "kann Tabelle nicht anlegen\n";

$sql = "INSERT INTO math VALUES ('1+1 = ', 2)";
$dbh->do($sql) or die "kann Datensatz nicht anlegen\n";

$dbh->disconnect();
```

Der Befehl `CREATE TABLE` liefert keine Daten und `prepare/execute` ist daher fehl am Platz — mithilfe von `do` kann die Abfrage schnell und einfach ausgeführt werden.

Wie das Anlegen von Tabellen ist auch das Anlegen eines Datensatzes eine Operation, die keine nennenswerten Nutzdaten liefert. Auch hier kommt `do` zum Einsatz. Dasselbe gilt für `UPDATE`-Queries.

8.3.6 Bind-Variablen

Bind-Variablen sind ein weiteres, mächtiges Feature, von dem viele DBI-Programmierer gerne Gebrauch machen. Mithilfe von Bind-Variablen ist es möglich, eine Vielzahl gleicher Operationen effizient auszuführen.

Werfen wir einen Blick auf das folgende Beispiel, das die grundlegenden Konzepte erläutert:

```
#!/usr/bin/perl

use DBI;

$string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh = DBI->connect("$string","postgres","") or
        die "kann nicht zur Datenbank verbinden\n";

$dbh->do("CREATE TABLE testtab (eins int4, zwei int4)") or
        die "kann Tabelle nicht anlegen\n";

$sth = $dbh->prepare("INSERT INTO testtab VALUES (?, ?)");

$sth->execute(4, 1) or die "kann nicht einfügen\n";
```

```
15 $sth->execute(5, 7) or die "kann nicht einfügen\n";
16 $sth->execute(3, 2) or die "kann nicht einfügen\n";
17
18 $dbh->disconnect();
```

Nach der Herstellung einer Verbindung zur Datenbank wird eine neue Tabelle angelegt, die zwei Felder speichern soll. Anschließend wird ein SQL-Statement vorbereitet, das zwei Parameter akzeptiert. Nach der Vorbereitung dieses Statements kann es beliebig oft aufgerufen und parametrisiert werden. Speziell beim Einfügen vieler Datensätze, deren Inhalt zur Laufzeit generiert wird, hat das viele Vorteile, die auch die Lesbarkeit Ihrer Applikationen verbessern.

Mithilfe eines einfachen Shell-Befehls können Sie prüfen, ob die Daten in der Tabelle vorhanden sind:

```
1 [hs@duron tmp]$ psql -U postgres -d buch -c "SELECT * FROM testtab"
2  eins | zwei
3 ------+------
4     4 |    1
5     5 |    7
6     3 |    2
7 (3 rows)
```

8.3.7 COPY

Der `COPY`-Befehl ist einer der wichtigsten Befehle beim Umgang mit größeren Datenmengen. Nicht nur, dass er hilft, sehr viel Overhead zu sparen, er ermöglicht es auch auf einfache Weise, Daten in größeren Blöcken einzufügen.

In diesem Abschnitt werden wir erfahren, wir man den `COPY`-Befehl in Kombination mit dem DBI-Modul verwenden kann und welche Fallen auftreten.

Wie wir in diesem Buch bereits gelernt haben, besteht ein `COPY`-Befehl aus einem SQL-Statement und den nachfolgenden Daten. Der Datenbestand selbst wird zeilenweise an das Backend geschickt. Um das Ende zu markieren, werden ein Backslash und ein Punkt benötigt. Dieselbe Vorgehensweise können wir nun bei der Anwendung des DBI-Moduls anwenden. Zuerst führen wir den `COPY`-Befehl aus und senden die Daten anschließend an das Backend. Im Fall von `DBD::Pg` sieht das dann wie folgt aus:

```
1 #!/usr/bin/perl -w
2
3 use DBI;
4
5 $string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
6 $dbh = DBI->connect("$string","postgres","") or
```

```
        die "kann nicht zur Datenbank verbinden\n";

$dbh->do("COPY testtab FROM stdin;") or
        die "kann COPY nicht starten\n";

$dbh->func("10  12\n", 'putline') or die "kann nicht einfügen\n";
$dbh->func("11  13\n", 'putline') or die "kann nicht einfügen\n";
$dbh->func("12  14\n", 'putline') or die "kann nicht einfügen\n";
$dbh->func("\\.\n", 'putline') or die "kann nicht einfügen\n";

$dbh->disconnect();
```

Mithilfe der Funktion `func` ist es möglich, die Daten an den Server zu schicken. Wichtig dabei ist, dass wir der Funktion den Namen der intern aufzurufenden Funktion übergeben müssen. In unserem Fall nennt sich diese `putline`.

Nachdem wir das Programm ausgeführt haben, können wir uns wieder das Ergebnis anzeigen lassen:

```
[hs@duron tmp]$ psql -U postgres -d buch -c "SELECT * FROM testtab
WHERE eins >= 10"
 eins | zwei
------+------
   10 |   12
   11 |   13
   12 |   14
(3 rows)
```

Alle Werte sind in die Datenbank eingefügt worden und stehen zur Verfügung.

8.3.8 DBI-Parameter

Das DBI-Modul ist eine überaus mächtige Softwarekomponente, die eine breite Palette an Einstellungsmöglichkeiten anbietet. Dieser Abschnitt beschäftigt sich mit Parametern, die zur Konfiguration eines Programms herangezogen werden können.

Das erste Beispiel zeigt, wie der Umgang mit Transaktionen modifiziert werden kann:

```
#!/usr/bin/perl

use DBI;

$string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh = DBI->connect("$string","postgres", "",
        { RaiseError => 1, AutoCommit => 0 })
```

```
8                   or die "kann nicht zur Datenbank verbinden\n";
9
10 $dbh->disconnect();
```

Sie können die für die aktuelle Datenbankverbindung verwendeten Parameter leicht setzen, indem Sie der `Connect`-Funktion weitere Parameter mitgeben, wie wir das in unserem Beispiel gemacht haben. Der Parameter `RaiseError` redefiniert die Behandlungsweise von Fehlern. Wenn Sie `AutoCommit` auf 0 setzen, teilen Sie Perl mit, dass Transaktionen händisch beendet werden müssen, um die Änderungen wirksam werden zu lassen.

Die nächste Übersicht enthält alle vom `DBD::Pg`-Treiber zur Verfügung gestellten Parameter. Bedenken Sie, dass manche dieser Parameter nicht modifiziert, sondern nur gesetzt werden lönnen:

`Warn (boolean, inherited)`: Schaltet Warnungen ein beziehungsweise aus. Defaultmäßig ist dieses Flag ausgeschaltet.

`Active (boolean, read-only)`: Dieses Flag wird in Applikationen nicht verwendet und ist derzeit noch etwas schwammig definiert. Im Prinzip beschreibt es, wie Handles zu behandeln sind. Änderungen an diesem Flag sind nicht zu empfehlen.

`Kids (integer, read-only)`: Enthält die Zahl der Datenbankhandles, die einem Treiber zugeordnet sind beziehungsweise die Anzahl der Statement Handles, die einem Datenbankhandle zugeordnet sind

`CachedKids (hash ref)`: Gibt entweder eine Referenz auf einen Hash der Datenbankhandles beziehungsweise eine Referenz auf einen Hash der Statementhandles zurück

`InactiveDestroy (boolean)`: Sorgt dafür, dass unbenutzte oder aus dem Sichtbarkeitsbereich verschwundene Datenbankhandles geschlossen werden

`PrintError (boolean, inherited)`: Dieser Parameter sorgt dafür, dass jeder Fehler zusätzlich mittels warn ausgegeben wird

`RaiseError (boolean, inherited)`: Sorgt dafür, dass Fehler statt Fehlercodes ausgegeben werden.

`ChopBlanks (boolean, inherited)`: Definiert, ob unnötige Leerzeichen bei Character-Feldern abgeschnitten werden oder nicht

`LongReadLen (integer, inherited)`: Definiert die maximale Länge eines Wertes, der aus einem Feld ausgelesen wird. Sofern der Wert auf 0 gesetzt ist, werden Daten beliebiger Länge gelesen.

`LongTruncOk (boolean, inherited)`: Dieses Flag hilft dem User zu beeinflussen, ob das Abfragen eines zu langen Wertes zu einem Fehler oder zum Abschneiden des Inhaltes des betroffenen Feldes führt.

`Taint (boolean, inherited)`: Dieses Flag ist nur wirksam, wenn Perl im Taint-Modus läuft.

`private_*`: Es gibt die Möglichkeit, eigene Module als Attribute zu einem Handle abzuspeichern. Diese werden üblicherweise mit `private` gekennzeichnet.

Es bietet der DBI Layer eine Fülle von Möglichkeiten, die es erlauben, die Funktionsweise des Moduls genau zu definieren und unter die Lupe zu nehmen.

8.3.9 Binärobjekte

Das DBD-Modul für PostgreSQL bietet einfache Möglichkeiten, mit binären Objekten umzugehen. In diesem Abschnitt werden wir einige Beispiele betrachten, die uns zeigen, wie mit dem Modul umzugehen ist.

Um ein binäres Objekt in der Datenbank anzulegen, können Sie auf die Funktion `func` zurückgreifen, die wir bereits in einem anderen Kontext kennen gelernt haben. Beim Anlegen des Objektes ist es wichtig, dass Sie einen Modus definieren, der bestimmt, wie das Objekt behandelt werden soll. Nehmen wir also ein Beispiel, das zeigt, wie ein Objekt in der Datenbank angelegt und dessen Object ID ausgegeben werden kann:

```
1  #!/usr/bin/perl
2
3  use DBI;
4
5  $string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
6  $dbh = DBI->connect("$string","postgres", "")
7                  or die "kann nicht zur Datenbank verbinden\n";
8
9  $mode = $dbh->{pg_INV_WRITE};
10 $objectid = $dbh->func($mode, 'lo_creat') or
11         die "kann BLOB nicht anlegen";
12
13 print "Object ID: $objectid\n";
14
15 $dbh->disconnect();
```

Wir verwenden den Befehl `func` in Kombination mit `lo_creat`. Als Rückgabewert erhalten wir die Object ID, die für die weitere Verarbeitung des Objektes von großer Bedeutung ist.

Wenn wir das Programm ausführen, wird die ID des Objektes ausgegeben.

```
[hs@duron tmp]$ ./connect.pl
Object ID: 405343
```

Des Weiteren werden wir versuchen, das soeben angelegte Objekt zu öffnen und Daten einzufügen beziehungsweise auszulesen. Zu diesem Zwecke können wir die ID des neuen Objektes verwenden:

```
#!/usr/bin/perl

use DBI;

$string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh = DBI->connect("$string","postgres", "",
                { RaiseError => 1} )
                or die "kann nicht zur Datenbank verbinden\n";

# Öffnen des Objektes
$oid = 405343;
$mode = $dbh->{pg_INV_READ} or
        die "kann Modus nicht definieren\n";
$dbh->begin_work or
        die "kann Transaktion nicht starten\n";
$descriptor = $dbh->func($oid, $mode, 'lo_open');

# Anlegen eines Textes und Schreiben
$buf = 'Hello World ...';
$len = length $buf;

$nbytes = $dbh->func($descriptor, $buf, $len, 'lo_write') or
        die "kann nicht in das Objekt schreiben ($!)\n";
print "$nbytes geschrieben ...\n\n";

# Positionieren und Auslesen der Daten
$offset = 0;
$loc = $dbh->func($descriptor, $offset, 0, 'lo_lseek');
$nbytes = $dbh->func($descriptor, $buffer, $len, 'lo_read');

print "Buffer: $buffer\n";
print "Länge des Textes: $nbytes\n";

$dbh->func($descriptor, 'lo_close');

$dbh->commit or die "kann Transaktion nicht beenden\n";
$dbh->disconnect();
```

Nach dem Festlegen des Modus starten wir eine Transaktion, um sicherzustellen, dass alle Operationen innerhalb eines Blockes abgearbeitet werden. Danach öffnen wir das Objekt und definieren zwei Variablen. Die Variable `$buf` enthält den Text,

den wir in das Objekt einfügen wollen. Die Variable `$len` enthält die Länge der zu schreibenden Daten. Hierbei haben wir die Berechnung der Länge aus dem Funktionsaufruf herausgelöst, um die Lesbarkeit des Programmes zu erhöhen. In einer praxisbezogenen Anwendung kann diese Berechnung auch ohne Bedenken direkt im Funktionsaufruf durchgeführt werden.

Wichtig ist `pg_INV_READ` — es sorgt dafür, dass das Objekt im Lesenmodus geöffnet wird.

Nach der Variablendefinition werden die Daten in das Objekt geschrieben. Der Rückgabewert enthält die Länge der geschriebenen Daten. Nach dem Schreiben setzen wir den Filepointer auf die erste Position im Objekt und lesen die Daten aus. Die ausgelesenen Daten werden wieder in einen Buffer geschrieben, den wir am Bildschirm darstellen.

Abschließend wird die Transaktion beendet und die Verbindung zur Datenbank geschlossen. Was passiert, wenn wir das Programm ausführen?

```
[hs@duron tmp]$ ./connect.pl
15 geschrieben ...

Buffer: Hello World ...
Länge des Textes: 15
```

Der Text wird korrekt ausgelesen.

Um ein Objekt wieder aus der Datenbank zu entfernen, können Sie den folgenden Befehl anwenden:

```
$descriptor = $dbh->func($descriptor, 'lo_unlink');
```

8.4 DBI-Proxies

Wenn Sie in der Situation sind, Welten miteinander verbinden zu müssen, werden Sie mit herkömmlichen Methoden schnell an Grenzen stoßen. Aus diesem Grund haben sich kluge Köpfe so genannte DBI-Proxies ausgedacht, die in der Lage sind, als Datenbank-Proxies die verschiedenen Welten zu verbinden und über eine einheitliche Schnittstelle zugänglich zu machen. Der Proxy sorgt daher dafür, dass Sie auf Module zugreifen können, die nicht auf Ihrem Rechner installiert sind. Alle Anfragen werden an den Proxy weitergeleitet, der dann im Gegensatz zum Client weiß, wie eine spezielle Datenquelle (etwa eine PostgreSQL-Datenbank) anzusprechen ist. Auf dem Client muss daher kein DBD-Modul für PostgreSQL installiert sein.

Ein DBI-Proxy schafft eine Zwischenschicht, die es erlaubt, individuell Rechte zu vergeben und den Zugang zu den Datenbanken »hinter« dem Proxy zu definieren.

Für die Installation des Proxies sind einige Module notwendig. Welche Module Sie genau benötigen, hängt vom System ab, das Sie verwenden. Von großer Bedeutung ist das Modul `Thread.pm`, das in der Praxis jedoch hin und wieder zu Problemen führen kann. Sollte es bei Ihnen zu Problemen beim Betrieb des DBI-Proxies kommen, empfehlen wir `fork` anstelle von `threads` als Modus zu verwenden.

Nachdem Sie das DBI-Modul installiert haben, gibt es auf Ihrem Rechner ein Programm namens `DBI Proxy`, das folgende Möglichkeiten bietet:

```
[hs@duron hs]$ dbiproxy --help
Usage: /usr/bin/dbiproxy <options>

Possible options are:

  --nocatchint            Try to catch interrupts when calling system
                         functions like bind(), recv()), ...
  --childs <num>          Set number of preforked childs, implies
                                mode=single.
  --chroot <dir>          Change rootdir to given after binding to port.
  --compression <type>           Set compression type to off (default)
                                or gzip.
  --configfile <file>     Read options from config file <file>.
  --debug                 Turn debugging mode on
  --facility <facility>   Syslog facility; defaults to 'daemon'
  --group <gid>           Change gid to given group after binding
                                to port.
  --help                  Print this help message
  --localaddr <ip>        IP number to bind to; defaults to INADDR_ANY
  --localpath <path>      UNIX socket domain path to bind to
  --localport <port>      Port number to bind to
  --logfile <file>        Force logging to <file>
  --loop-child            Create a child process for loops
  --loop-timeout <secs>   Looping mode, <secs> seconds per loop
  --maxmessage <size>           Set max message size to <size>
                                (Default 65535).
  --mode <mode>           Operation mode (threads, fork or single)
  --pidfile <file>        Use <file> as PID file
  --proto <protocol>        transport layer protocol: tcp (default)
                                or unix
  --user <user>           Change uid to given user after binding to port.
  --version               Print version number and exit

DBI::ProxyServer 0.2005, Copyright (C) 1998, Jochen Wiedmann
```

Deutlich zu erkennen, stellt der DBI-Proxy ein mächtiges Werkzeug zur Verfügung.

Kern eines DBI-Proxies ist eine Konfigurationsdatei, die sich in der Regel `dbiproxy.conf` nennt. Das nächste Listing zeigt eine Beispielkonfiguration:

```
{
    'localport' => 1234,
    'pidfile' => '/tmp/dbiproxy.pid',
    'logfile' => 1,
    'debug' => 1,
    'mode' => 'fork',
    'timeout' => 60,
    'clients' => [
    {
        'mask' => '^localhost$',
        'accept' => 1 },
    {
        'mask' => '.*cybertec.*',
        'accept' => 1
    }
    ]
}
```

Nachdem Sie den DBI-Proxy konfiguriert haben, können Sie ihn bequem starten:

```
[root@duron tmp]# dbiproxy --configfile=dbiproxy.conf --logfile
/tmp/dbi.log
Thu May  9 13:12:07 2002 debug, Server starting in operation mode fork
Thu May  9 13:12:07 2002 notice, Server starting
Thu May  9 13:12:07 2002 debug, Writing PID to /tmp/dbiproxy.pid
```

Um zu prüfen, ob der Proxy auch tatsächlich funktioniert, können Sie `nmap` verwenden. In unserem Beispiel hört der Proxy auf Port 1234, den wir mit `nmap` gezielt ansteuern können:

```
[hs@duron hs]$ nmap localhost -p 1234

Starting nmap V. 2.54BETA30 ( www.insecure.org/nmap/ )
Interesting ports on duron (127.0.0.1):
Port       State       Service
1234/tcp   open        hotline

Nmap run completed -- 1 IP address (1 host up) scanned in 0 seconds
```

Sofern der Port offen ist, können wir nun versuchen, via DBI-Proxy auf PostgreSQL zuzugreifen. Das geht am einfachsten mit einer kleinen Perl-Applikation:

```
#!/usr/bin/perl

use DBI;

$string = "dbi:Pg:dbname=buch;host=localhost;port=5432";
$dbh = DBI->connect("dbi:Proxy:hostname=localhost;port=1234;dsn=$string",
        'postgres', '', { RaiseError => 1, PrintError => 1 } ) or
                die $DBI::errstr;

$sth = $dbh->prepare( 'SELECT 1+1' );
$sth->execute() or die "kann Abfrage nicht durchführen\n";
@row = $sth->fetchrow_array();
print "Ergebnis: ".$row[0]."\n";
$sth->finish;

$dbh->disconnect();
```

Die Interaktion mit der Datenbank erfolgt wie gewohnt. Der einzige Unterschied ist, dass wir uns diesmal zum Datenbank-Proxy und nicht direkt zur Datenbank verbinden.

Wenn wir das Programm ausführen, wird das Ergebnis der Berechnung gezeigt:

```
[hs@duron hs]$ ./connect.pl
Ergebnis: 2
```

Speziell in Bereichen, in denen Sie es mit vielen Datenbanken zu tun haben, ist der DBI-Proxy eine einfache Möglichkeit, Schnittstellen zu schaffen. Durch die vielfältigen Features ergeben sich unzählige Möglichkeiten, die im Umgang mit vielen verschiedenen Rechnern große Vorteile bringen und zur klareren Strukturierung einer IT-Umgebung beitragen.

Kapitel 9

PHP

PHP ist eine der mächtigsten Scriptsprachen, die je für die Webentwicklung implementiert worden sind. Der Vorteil von PHP liegt in der Kombination aus Einfachheit und Effizienz. Es ist möglich, relativ komplexe Anwendungen mit geringem Aufwand zu realisieren. Im Gegensatz zu Perl ist PHP eine Sprache des Internets und nur für dieses geschaffen. Das hat zur Folge, dass PHP nur schwer für eine Anwendung außerhalb des Webs verwendet werden kann. Weiters stellt PHP so gut wie keine Lowlevel-Schnittstellen zur Verfügung, da das für eine Webanwendung auch nicht notwendig ist.

In diesem Abschnitt werden Sie erfahren, wie mit PHP gearbeitet werden kann und Sie werden lernen, wie PHP an eine PostgreSQL-Datenbank angebunden werden kann.

9.1 Grundfunktionen

PHP stellt eine Reihe von Grundoperationen zur Verfügung, auf die beim Arbeiten mit Datenbanken nicht verzichtet werden kann. Hier werden wir versuchen, Ihnen diese Funktionen näher zu bringen und zu zeigen, wie man mit Funktionen von PHP effizient arbeiten kann.

9.1.1 Verbinden zur Datenbank

Die erste und wichtigste Operation im Umgang mit PHP und PostgreSQL ist das Erstellen einer Datenbankverbindung. Für alle Operationen, die Sie durchführen wollen, benötigen Sie einen Connex zur Datenbank. Ziel dieser Ausführungen ist, dass Sie lernen, wie Verbindungen erstellt und wieder getrennt werden können.

Um sich zur Datenbank zu verbinden, können Sie den Befehl `pg_connect` verwenden. Bevor wir die Möglichkeiten dieses Befehls ausführlich beschreiben, zeigen wir ein entsprechendes Beispiel:

```
<?php
        $data = "user=postgres dbname=buch";

        $database = pg_connect ($data);
        if      (!$database)
        {
                die ("ein Fehler ist aufgetreten");
        }
        else
        {
                echo "die Verbindung wurde hergestellt";
                pg_close($database);
        }
?>
```

Zuerst legen wir eine Variable an, die die Parameter der Datenbankverbindung enthält. Danach versuchen wir, uns zu PostgreSQL zu verbinden. Der Rückgabewert der Funktion ist ein so genannter Datenbankhandle. Dieser Handle identifiziert eine Verbindung eindeutig und dient zur Kommunikation mit der Datenbank.

Anschließend prüfen wir, ob der von `pg_connect` zurückgegebene Datenbankhandle gültig ist oder nicht. Sollte ein Fehler aufgetreten sein, wird das Programm beendet. Sofern der Handle korrekt ist, wird ein Text ausgegeben und die Verbindung ordnungsgemäß mit `pg_close` geschlossen.

Sofern alles korrekt funktioniert, wird folgende Meldung ausgegeben:

```
1 die Verbindung wurde hergestellt
```

Sie haben gesehen, wie eine Verbindung prinzipiell geöffnet werden kann. Jetzt ist es notwendig, sich ein wenig eingehender mit den Möglichkeiten von `pg_connect` zu befassen.

`host`: Definiert den Namen oder die IP-Adresse des Rechners, auf dem die Datenbank liegt. Wenn diese Option nicht angegeben ist, wird der lokale Rechner verwendet und via UNIX-Sockets angesprochen.

`port`: Definiert den Port, auf den PostgreSQL hört

`user`: Definiert den Namen des Users, den Sie für die Authentifizierung verwenden wollen

`password`: Definiert das Passwort des Benutzers, unter dem Sie sich anmelden

`tty`: Definiert den `tty`

`dbname`: Definiert den Namen der Datenbank, die Sie ansprechen wollen

`options`: Optionen für den Verbindungsaufbau

Zusätzlich zur soeben besprochenen Syntax unterstützt PHP noch eine zweite, aber nicht mehr oft verwendete Syntax, die uns kurz das nächste Listing zeigt:

```
1 $database = pg_connect ("host", "port", "options", "tty", "dbname")
```

Dabei werden die Parameter der Reihe nach an die Funktion übergeben. Diese Schreibweise wird nicht sehr häufig verwendet und sollte daher ganz vermieden werden.

Manchmal kann es vorkommen, dass Sie mehrere Datenbankverbindungen geöffnet haben. In der Regel startet PostgreSQL für jede zur Datenbank erstellte Verbindung einen Prozess. Wenn Sie versuchen, von PHP aus mehrere Verbindungen zur selben Datenbank zu öffnen, wird jedoch etwas Seltsames passieren, wie Sie im folgenden Beispiel erkennen werden:

```
<?php
        $data = "user=postgres dbname=buch";

        for     ($i = 0; $i < 100; $i++)
        {
                $database = pg_connect ($data) or
                        die ("ein Fehler ist aufgetreten");
        }

        sleep(20);
?>
```

Sie werden davon ausgehen, dass PostgreSQL 100 Verbindungen zur Datenbank öffnet und diese erst am Ende des Programmes schließt — weit gefehlt. Wenn Sie sich ansehen, was passiert, während das Programm ausgeführt wird, werden Sie sich wundern:

```
[hs@duron php]$ ps ax | grep postg | grep buch
27361 ?        S      0:00 postgres: postgres buch [local] idle
```

PHP hat nur eine Verbindung geöffnet, da die Verbindungsparameter allesamt gleich sind. Dieses Verhalten der Sprache kann Vorteile, aber auch massive Nachteile haben. Aus diesem Grund sei noch einmal nachdrücklich darauf hingewiesen.

9.1.2 Abfragen von Verbindungsparametern

Wenn Sie sich einmal erfolgreich zu einer Datenbank verbunden haben, kann es interessant sein, die Parameter einer Verbindung abzufragen. Das kann beim Arbeiten mit mehreren Verbindungen manchmal von Bedeutung sein.

Das nächste Beispiel zeigt, wie eine Verbindung zur Datenbank aufgebaut und die Paramter abgefragt werden:

```
<?php
        $data = "user=postgres dbname=buch";

        $database = pg_connect ($data) or
                        die ("ein Fehler ist aufgetreten");

        print "<br>Host: ".pg_host($database);
        print "<br>Datenbank: ".pg_dbname($database);
        print "<br>Port: ".pg_port($database);
        print "<br>Tty: ".pg_tty($database);
        print "<br>Optionen: ".pg_options($database);
?>
```

Am Ende des Programmes werden die entsprechenden Parameter abgefragt und auch gleich am Bildschirm ausgegeben.

Wenn Sie diese Funktionen verwenden, ist absolute Achtsamkeit gefragt, da es sehr leicht zu Problemen kommen kann. Wenn Sie das obige Beispiel mit PHP 4.06 ausführen, werden Sie sich wundern, warum das Ausführen des Beispiels mittels Webbrowser zu keinem Ergebnis führt.

Der Grund für dieses Verhalten kann im `error_log` von Apache nachgelesen werden:

```
[Fri Apr 19 23:30:41 2002] [notice] child pid 27461 exit signal
Segmentation fault (11)
```

Auch Open-Source-Entwicklern kann einmal ein Bug passieren. In diesem Beispiel haben wir versucht, die Verbindung zur Datenbank mittels UNIX-Sockets herzustellen (das erkennt man daran, dass wir keinen Host angegeben haben, zu dem wir eine TCP/IP-Verbindung aufbauen wollen). Im Falle von UNIX-Sockets kollabieren bei manchen PHP-Versionen die Funktionen `pg_host` und `pg_port`.

Sofern wir die Verbindung via TCP/IP erstellen, liefert das Programm korrekte Daten, wie das nächste Listing zeigt:

```
Host: 62.116.21.99
Datenbank: buch
Port: 5432
Tty:
Optionen:
```

In diesem Beispiel haben wir die Verbindung zur Datenbank nicht explizit terminiert. Das ist auch nicht notwendig, da PHP eine Verbindung automatisch schließt, sobald das Programm am Ende angelangt ist.

Nachdem Sie sich zu einer Datenbank verbunden haben, kann es sinnvoll sein, den Status einer Verbindung abzufragen. Das folgende Listing zeigt zwei Möglichkeiten, das zu bewerkstelligen:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
        {
                print "Ein Fehler ist aufgetreten<br>\n";
        }

        print pg_connection_status($dbh);
?>
```

Die If-Abfrage prüft, ob der Datenbank Handle gültig ist — mithilfe von pg_connection_status ist es möglich, den Zustand eines Datenbank Handles auszugeben. Die Funktion kennt die Rückgabewerte 0 (PGSQL_CONNECTION_OK) und 1 (PGSQL_CONNECTION_BAD).

9.1.3 Daten abfragen

Nachdem Sie nun wissen, wie Verbindungen zur Datenbank angelegt werden können, wenden wir uns wieder SQL-Abfragen zu. Ziel eines jeden Programmes, das mit einer Datenbank interagiert, ist es, Daten aus der Datenbank abzufragen. PHP stellt einige leicht verwendbare Funktionen zur Verfügung, mit denen das bewerkstelligt werden kann.

Bevor wir uns diese Funktionen im Detail ansehen, legen wir einfach eine Tabelle an und fügen einige Daten ein:

```
CREATE TABLE t_text (
        id              int4,
        code            text,
        sprache         text,
        typ             text
);

COPY t_text FROM stdin USING DELIMITERS ';';
1;Tischlampe;deutsch;kurzbeschreibung
2;Tischlampe grün, 60 Watt; deutsch;detailbeschreibung
3;Rasenmäher;deutsch;kurzbeschreibung
4;Tisch;deutsch;kurzbeschreibung
5;Table;englisch;kurzbeschreibung
\.
```

Ziel des nächsten Programms ist es, die Daten einer Tabelle in einer Liste auszugeben:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
        {
                print "Ein Fehler ist aufgetreten<br>\n";
        }

        // Abfragen der Daten
        $sql= "SELECT id, code, sprache, typ FROM t_text
                WHERE id < 4";
        $result = pg_query ($dbh, $sql);
```

```
13
14          if      (!$result)
15          {
16                  print "Fehler in ($sql).\n";
17                  exit;
18          }
19
20          // Ausgabe der Daten
21          $zeilen = pg_num_rows($result);
22          for     ($i = 0; $i < $zeilen; $i++)
23          {
24                  $z = pg_fetch_row($result, $i);
25                  print "<br>Zeile $i:<br>\n";
26                  for     ($j = 0; $j < count($z); $j++)
27                  {
28                          print "Spalte $j: $z[$j] <br>";
29                  }
30          }
31  ?>
```

Um eine Abfrage durchzuführen, können Sie den Befehl `pg_query` verwenden (bei älteren Versionen von PHP bitte `pg_exec` aufrufen). Diese Funktion gibt einen Handle auf das Ergebnis zurück, der anschließend abgearbeitet werden kann. Zuerst berechnen wir die Zahl der Zeilen im Ergebnis. Des Weiteren gehen wir die Zeilen der Reihe nach durch und geben die Spalten eine nach der anderen aus. Die Werte selbst werden einfach aus der Zeile ausgelesen.

Wenn Sie das Programm ausführen, wird das Ergebnis wie folgt aussehen:

```
1  Zeile 0:
2  Spalte 0: 1
3  Spalte 1: Tischlampe
4  Spalte 2: deutsch
5  Spalte 3: kurzbeschreibung
6
7  Zeile 1:
8  Spalte 0: 2
9  Spalte 1: Tischlampe grün, 60 Watt
10 Spalte 2: deutsch
11 Spalte 3: detailbeschreibung
12
13 Zeile 2:
14 Spalte 0: 3
15 Spalte 1: Rasenmäher
16 Spalte 2: deutsch
17 Spalte 3: kurzbeschreibung
```

In diesem Beispiel haben wir die Funktion `pg_fetch_row` verwendet, die genau eine Zeile enthält. Die jeweilige Spalte haben wir mithilfe eines Indexes abgefragt. Diese Vorgehensweise hat einige Nachteile, die an dieser Stelle nicht unerwähnt bleiben dürfen. Wenn Sie beispielsweise eine zusätzliche Spalte abfragen, verschieben sich die Indices der Spalten und es kann zu Problemen kommen. Das wiegt in PHP besonders schwer, weil PHP de facto keine Datentypen unterstützt und ein Fehler daher schwerer zu erkennen ist (schließlich gibt es keinen Compiler, der sich beschweren könnte).

In vielen Fällen ist es sinnvoller, den Datenbestand direkt als Objekt oder als Array auszulesen. Diese Vorgehensweise hat den Vorteil, dass Sie eine Spalte über den Namen und nicht über den Index ansprechen. Zur Abfrage der Daten als Array können Sie die Funktion `pg_fetch_array` verwenden, wenn Sie lieber ein Objekt abfragen wollen, können Sie `pg_fetch_object` verwenden. Alle Funktionen dienen zur Abfrage von genau einer Zeile, geben aber die Daten in unterschiedlicher Form zurück.

9.1.4 Metadaten

Wie bei nahezu allen anderen Programmiersprachen auch ist es in PHP möglich, Metadaten über eine Abfrage zu extrahieren. In diesem Abschnitt werden wir erfahren, wie mit Metadaten gearbeitet werden kann und wie Informationen einfach und schnell extrahiert werden können.

Um die Vorgehensweise gleich praktisch zu erläutern, finden Sie hier ein Beispiel:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
        {
                print "Ein Fehler ist aufgetreten<br>\n";
        }

        // Abfragen der Daten
        $sql= "SELECT id, code, sprache, typ FROM t_text
                WHERE id < 4";
        $result = pg_query ($dbh, $sql);
        if      (!$result)
        {
                print "Fehler in ($sql).\n";
                exit;
        }

        // Ausgabe der Feldnamen
        for     ($i = 0; $i < pg_num_fields($result); $i++)
```

```
21          {
22                  $fieldname = pg_field_name($result, $i);
23                  print "<br>Feld $i: $fieldname<br>\n";
24                  print "Typ: ".pg_field_type($result, $i)."<br>\n";
25                  print "Länge 1: ".pg_field_size($result, $i)."<br>\n";
26                  print "Länge 2: ".pg_field_prtlen($result, 0, $fieldname)
27                          ."<br>\n";
28
29
30          }
31  ?>
```

Nachdem wir die Abfrage durchgeführt haben, können wir daran gehen, alle relevanten Metadaten zu extrahieren. Die erste wichtige Information extrahieren wir bereits in der `for`-Schleife. Die Funktion `pg_num_fields` berechnet die Zahl der Spalten, die von der Abfrage zurückgegeben werden. Da wir Informationen über jede einzelne Spalte generieren wollen, benötigen wir diese Aussage. Innerhalb der Schleife berechnen wir zuerst den Feldnamen der aktuellen Spalte. Diesen geben wir auch aus. Im nächsten Schritt sehen wir uns den Datentyp des Feldes an. Die beiden letzten Befehle berechnen jeweils die Länge von Feldern. `pg_field_size` retourniert die Länge einer Spalte allgemein. Die Funktion `pg_field_prtlen` gibt die Länge eines Feldes in einer speziellen Zeile zurück.

Hier sehen Sie, welchen Output das Script liefert:

```
1  Feld 0: id
2  Typ: int4
3  Länge 1: 4
4  Länge 2: 1
5
6  Feld 1: code
7  Typ: text
8  Länge 1: -1
9  Länge 2: 10
10
11 Feld 2: sprache
12 Typ: text
13 Länge 1: -1
14 Länge 2: 7
15
16 Feld 3: typ
17 Typ: text
18 Länge 1: -1
19 Länge 2: 16
```

Es ist interessant zu bemerken, dass PHP nicht die Id eines Datentyps, sondern dessen Namen zurückgibt. Das ist auf den ersten Blick ein wenig ungewohnt, da

üblicherweise nur die ID retourniert wird. Über den Sinn dieser Vorgehensweise kann man streiten, fest steht jedoch, dass PHP hier aus der Reihe tanzt.

9.1.5 COPY

Nach dem Überblick über die wichtigsten Funktionen von PHP wollen wir uns nun dem COPY-Befehl zuwenden. Immer, wenn es darum geht, größere Datenmengen in eine Tabelle einzufügen, kann der COPY-Befehl aus Gründen der Performance von großer Bedeutung sein.

Wie in den vorangegangenen Abschnitten, müssen Sie beim Einfügen von Daten mithilfe von COPY drei Abschnitte an die Datenbank schicken. Der erste Teil teilt PostgreSQL mit, dass es sich um einen COPY Befehl handelt. Im zweiten Abschnitt können Sie die Daten an die Datenbank übermitteln. Der dritte Teil schließt die Operation ab. Das folgende Beispiel zeigt, wie Daten eingefügt werden können:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        $result = pg_query($dbh, "COPY t_text FROM stdin
                USING DELIMITERS ';'");
        pg_put_line($dbh, "10;Hamburger;deutsch;produktname\n");
        pg_put_line($dbh, "11;Zwiebelringe;deutsch;produktname\n");
        pg_put_line($dbh, "12;Currysauce;deutsch;produktname\n");
        pg_put_line($dbh, "\\.\n");
        pg_end_copy();

        print "Es hat funktioniert :)\n";
?>
```

In unserem Beispiel haben wir drei Datensätze an die Datenbank geschickt. Das folgende Listing zeigt, dass diese Datensätze auch wirklich eingefügt worden sind:

```
buch=# SELECT * FROM t_text WHERE id >= 10;
 id |     code     | sprache |     typ
----+--------------+---------+-------------
 10 | Hamburger    | deutsch | produktname
 11 | Zwiebelringe | deutsch | produktname
 12 | Currysauce   | deutsch | produktname
(3 rows)
```

Wichtig ist, dass alle Operationen innerhalb einer Transaktion bewerkstelligt werden — speziell bei größeren Datenmengen ist das sehr wichtig und erleichtert die Arbeit.

9.1.6 Tracing

Um das Backend zu belauschen, können Sie es tracen. Dadurch können Sie sich den Transfer der Daten ansehen und verstehen, wie verschiedene Operationen vor sich gehen. Das ist vor allem beim Debuggen sehr wichtig, setzt aber grundlegende Kenntnisse der internen Kommunikation voraus.

Sehen wir uns anhand eines einfachen Beispieles an, wie man PHP mitteilen kann, dass es eine Verbindung tracen soll:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        pg_trace("/tmp/pg_trace.txt", w, $dbh);
        $result = pg_query($dbh, "SELECT 1+1");
        pg_untrace($dbh);

        print "Es hat funktioniert :)\n";
?>
```

Mithilfe von `pg_trace` können Sie das Tracing einschalten. Hierbei akzeptiert die Funktion drei Parameter. Der erste Parameter enthält das Logfile, in das der Output geschrieben werden soll. Der zweite Parameter definiert den Modus und der dritte Parameter enthält den Datenbankhandle.

Um das Tracing wieder zu deaktivieren, stellt PHP die Funktion `pg_untrace` zur Verfügung.

Wenn Sie das obige Programm ausführen, wird das Logfile die folgenden Zeilen enthalten:

```
To backend> Q
To backend> SELECT 1+1
From backend> P
From backend> "blank"
From backend> T
From backend (#2)> 1
From backend> "?column?"
From backend (#4)> 23
From backend (#2)> 4
From backend (#4)> -1
From backend> D
From backend (1)> ~@
From backend (#4)> 5
From backend (1)> 2
```

```
From backend> C
From backend> "SELECT"
From backend> Z
From backend> Z
```

Dem Listing können Sie entnehmen, welche Zeilen an das Backend geschickt beziehungsweise welche Zeilen vom Backend zurückgegeben worden sind.

9.2 Arbeiten mit Binärobjekten

In diesem Abschnitt werden Sie lernen, welche Funktionen zur Verarbeitung von binären Objekten von PHP zur Verfügung gestellt werden.

Wie mit anderen Programmiersprachen ist es auch mit PHP möglich, einfache SQL-Funktionen zu verwenden. In den meisten Fällen ist es aber übersichtlicher und komfortabler die Schnittstellen von PHP anstatt der normalen SQL-Funktionen zu verwenden. Wir zeigen Ihnen, welche Funktionalitäten PHP zur Verfügung stellt.

Beginnen wir wie gewohnt mit einem einfachen Beispiel. Der folgende Code zeigt, wie eine Datei in die Datenbank importiert werden kann:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        pg_query($dbh, "BEGIN");
        $oid = pg_lo_import($dbh, "/tmp/test.cs");

        if      ($oid)
                echo "Oid: $oid<br>\n";
        else
                echo "Fehler: ".pg_last_error($dbh);

        pg_query($dbh, "COMMIT");
?>
```

Nachdem wir eine Verbindung zur Datenbank hergestellt haben, starten wir eine Transaktion. Das ist notwendig, da PHP das File sonst nicht importieren kann. Anschließend importieren wir die Datei `test.cs` im `/tmp`-Verzeichnis. Abschließend prüfen wir, ob der Import funktioniert hat und beenden die Transaktion.

Der Output des Scripts enthält eine Zeile mit der OID des neuen Objektes:

```
Oid: 405562
```

Das folgende Programm zeigt, wie das Objekt wieder exportiert werden kann. Jetzt wollen wir den Inhalt der Datei auflisten:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        pg_query($dbh, "BEGIN");
        $bool = pg_lo_export($dbh, 405562, "/tmp/result.txt");

        if      ($bool == TRUE)
        {
                echo "Der Export hat funktioniert:<br><br>\n";
                $fcontents = file ('/tmp/result.txt');
                while (list ($line_num, $line) = each ($fcontents))
                {
                        echo htmlspecialchars ($line), "<br>\n";
                }
        }
        else
                echo "Fehler ...<br>\n";
        pg_query($dbh, "COMMIT");
?>
```

Wieder arbeiten wir die gesamte Operation in einer Transaktion ab. Beim Export übergeben wir die Object ID des binären Objektes und den Namen der Ausgabedatei an die Funktion. Als Rückgabewert erhalten wir eine boolesche Variable. Sofern diese `TRUE` ist, können wir den Inhalt der Datei bequem ausgeben. Andernfalls geben wir `Fehler ...<br>` aus.

Sofern es zu keinen Problemen kommt, könnte der Output so aussehen:

```
Der Export hat funktioniert:

using System;

class WelcomeCSS
{
public static void Main(string[] args)
{
Console.WriteLine("Welcome to www.postgresql.at");
}
}
```

Hier sehen Sie ein Programm, das Sie z. B. mithilfe von `Mono` leicht ausführen können.

Im Beispiel haben wir die Daten mithilfe einer Zwischendatei ausgegeben. Wenn Sie die Daten direkt ausgeben wollen, ist das aber nicht notwendig, Sie können auch direkt auf den BLOB zugreifen, wie das im folgenden Listing gezeigt wird:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        pg_query($dbh, "BEGIN");
        $handle = pg_lo_open($dbh, 405562, "r");
        pg_lo_read_all ($handle);
        pg_query($dbh, "COMMIT");

?>
```

Jetzt wird das Objekt erst einmal geöffnet. Der Object Handle wird dann an `pg_lo_read_all` übergeben. Diese Funktion gibt den Inhalt des gesamten Objektes direkt am Bildschirm aus.

Wenn Sie nicht gleich alles ausgeben wollen, müssen Sie ein wenig elementarer vorgehen:

```
<?php
        // Verbinden zur Datenbank
        $dbh = pg_connect ("dbname=buch user=postgres");
        if      (!$dbh)
                print "Ein Fehler ist aufgetreten<br>\n";

        pg_query($dbh, "BEGIN");

        // Öffnen und Lesen ...
        $handle = pg_lo_open($dbh, 405562, "r");
        $data = pg_lo_read ($handle, 10);
        echo "die ersten 10 Zeichen: $data<br>\n";

        // aktuelle Position ...
        echo "aktuelle Position: ".pg_lo_tell ($handle)."<br>\n";

        // setzen der Position und Ausgeben der Daten ...
        $myint = pg_lo_seek($handle, -8);
        $data = pg_lo_read ($handle, 10);
        echo "Daten: $data<br>\n";

        pg_query($dbh, "COMMIT");
?>
```

Nach dem Öffnen des Objektes lesen wir zehn Bytes. Das führt dazu, dass der Filepointer um zehn Bytes weitergesetzt wird. Mithilfe des Befehles `pg_lo_tell` können Sie feststellen, wo der Verweis auf das Objekt gerade steht. Dann setzen wir den Zeiger um acht Zeichen zurück und geben erneut zehn Zeichen aus, bevor wir das Programm beenden — das führt zum folgenden Resultat:

```
1 die ersten 10 Zeichen: using Syst
2 aktuelle Position: 10
3 Daten: ing System
```

Es wird zweimal ein Auszug ausgegeben.

Wenn Sie in ein Objekt schreiben wollen, so ist auch das einfach möglich:

```
1 int pg_lo_write ( resource large_object, string data)
```

Mithilfe von `pg_lo_write` können Sie einen String in das Objekt schreiben.

9.3 Persistente Datenbankverbindungen

PHP unterstützt persistente Datenbankverbindungen. Um persistente Verbindungen wird sehr viel Trubel gemacht. In diesem Abschnitt wollen wir das Themengebiet genauer beleuchten und uns ansehen, wie mit persistenten Verbindungen umgegangen werden kann, um einen optimalen Nutzen zu erzeugen.

9.3.1 Allgemeines

Bevor Sie erfahren, was persistente Verbindungen genau sind, wollen wir erklären, was persistente Verbindungen nicht sind; das ist extrem wichtig, weil hier sehr viel Unwissen herrscht:

Wenn Sie mit persistenten Verbindungen arbeiten, heißt das nicht, dass Sie auf verschiedenen Screens dieselbe Verbindung verwenden: Es ist nicht so, dass jeder User »seine« Verbindung bekommt, die er erst hergibt, wenn er die Seite verlässt. Das wäre technisch nur schwer möglich.

In der Praxis funktionieren persistente Verbindungen so: Wenn eine Verbindung angefordert wird, sieht PHP nach, ob es in einem Pool offener Verbindungen bereits eine passende (gleicher Benutzer, gleicher Host, gleiche Datenbank) Verbindung gibt. Sofern es kein korrektes Backend gibt, wird eine neue Verbindung erstellt, andernfalls wird eine bereits offene Verbindung zugeteilt. Das hat den Vorteil, dass sehr viel Authentifizierungsoverhead gespart werden kann und die Datenbank daher spürbar schneller arbeitet. Wenn eine Verbindung geschlossen wird, wird sie nicht wirklich beendet, sondern kommt in den Pool der unbenutzten Verbindungen und kann bei Bedarf wieder zugeteilt werden.

Was hier in der Theorie sehr einfach klingt, kann in der Praxis zu einigen Problemen führen, die nur schwer zu finden sind, da a priori ja nicht feststeht, welche offene Verbindung einem gerade zugeteilt wird. Sofern es Verbindungen gibt, die noch offene Transaktionen beinhalten, kann es leicht zu teuflischen Problemen kommen.

9.3.2 Befehle

Persistente Verbindungen sind nahezu wie andere Verbindungen auch zu handhaben. Der einzige Unterschied ist der Befehl zum Herstellen des Connexes. Sehen wir uns ein Beispiel an:

```
1  <?php
2          // Verbinden zur Datenbank
3          $dbh = pg_pconnect ("dbname=buch user=postgres");
4          if      (!$dbh)
5          {
6                  print "Ein Fehler ist aufgetreten<br>\n";
7                  exit(1);
8          }
9          else
10         {
11                 print "alles ok ...<br>\n";
12         }
13         pg_close($dbh);
14 ?>
```

Der einzige Unterschied ist, dass wir `pg_pconnect` anstatt `pg_connect` verwenden, den Rest erledigt PHP intern.

Kapitel 10

Python

Python hat sich in den letzten Jahren zu einem wichtigen Bestandteil vieler Systeme entwickelt. Als Gegenpol zu Perl hat Python zahlreiche Fans gefunden und stellt eine solide Plattform für fast jede nur denkbare Anwendung dar. Im Gegensatz zu Perl stellt Python eine relativ einfache, schlichte Syntax zur Verfügung. Das macht die Lösungsansätze in einem Programm durchsichtiger und einfacher zu verstehen als die kryptische Syntax mancher Perl-Programme. Egal für welche Sprache Sie sich auch immer entscheiden mögen, in diesem Kapitel werden wir uns einzig und allein mit Python und all seinen Facetten beschäftigen.

Wie viele anderen Sprachen auch steht Python in zwei Versionen zur Verfügung. Die embedded Version PL/Python kann zur Implementierung von Ergänzungen zur Datenbank herangezogen werden. Python selbst kann aber auch als Scriptsprache verwendet werden, um herkömmliche Applikationen zu schreiben. Wir werden uns mit beiden Möglichkeiten beschäftigen.

10.1 PL/Python

Python eignet sich wunderbar als eingebettete Sprache. Durch die große Effizienz der Sprache und aufgrund der klaren Strukturierung von Python-Programmen kann Python leicht und vor allem sehr schnell eingesetzt werden.

Mit PostgreSQL 7.2 hat Python als eingebettete Sprache Einzug in die Standard-Distribution gefunden und erfreut sich großer Beliebtheit.

Um die Sprache zur Datenbank hinzuzufügen, können Sie wie folgt vorgehen:

```
1 [hs@duron python]$ createlang plpython buch
```

Diesen Vorgang werden Sie schon von anderen Sprachen kennen; auch die Installation von Python tanzt hier nicht aus der Reihe. Sofern Sie beim Kompiliervorgang Python eingeschaltet haben, sollte es hier also zu keinen Problemen kommen.

10.1.1 Einfache Beispiele

In diesem Abschnitt wollen wir einige einfache Beispiele benutzen, um zu erläutern, wie Python zu verwenden ist.

Werfen wir also einen Blick auf das erste Beipiel, das zeigt, wie ein Prozentsatz berechnet werden kann:

```
1 CREATE OR REPLACE FUNCTION percent (float, float) RETURNS float AS '
2         return args[0] / args[1] * 100.0
3 ' LANGUAGE 'plpython';
```

Was passiert, wenn wir die Funktion aufrufen?

```
buch=# SELECT percent(22.22, 65.65);
    percent
────────────────
 33.8461538462
(1 row)
```

Python berechnet die Funktion und gibt das Ergebnis als `float`-Variable aus.

Ein wichtiger Punkt ist wie immer die Ausgabe von Fehlermeldungen respektive von Warnungen. Auch dafür bietet PL/Python eine einfache Schnittstelle. Das folgende Programm tut nichts anderes, als verschiedenste Fehler und Warnungen auszugeben:

```
CREATE OR REPLACE FUNCTION message () RETURNS text AS '
plpy.debug("this is debugging information")
plpy.notice("this is a notice")
plpy.Fatal("this is a fatal error")
plpy.Error("this is an error")

return "perfect"
' LANGUAGE 'plpython';
```

Mithilfe des `plpy`-Objektes können wir bequem auf jede Meldung zugreifen. Sie müssen einfach nur den Text an die Methode übergeben und Python erledigt den Rest für Sie. Was passiert, wenn die Funktion ausgeführt wird?

```
buch=# SELECT message();
NOTICE:  ('this is a notice',)
 message
─────────
 perfect
(1 row)
```

10.1.2 Trigger und Datenbankschnittstellen

Python kann sehr einfach für die Implementierung von Triggern herangezogen werden. Wie bei vielen anderen Sprachen ist es auch Python möglich, im Falle von Triggern spezielle Variablen und Funktionen zu nutzen, die Ihnen das Leben mit dem Trigger einfacher machen.

Im folgenden Beispiel definieren wir zwei Tabellen. Ziel ist es, im Falle eines `INSERT`-Statements einen Trigger zu starten, der alle vordefinierten Variablen ausgibt. Zu diesem Zwecke benötigen wir eine Funktion sowie den dazugehörigen Trigger:

```
CREATE TABLE messdaten (
        id              int4,
        code            text,
        messwert        numeric(9,3)
);

CREATE TABLE logging (
        id              serial,
        tstamp          timestamp DEFAULT now()
);

COPY messdaten FROM stdin USING DELIMITERS ';';
1;Ventil 23c;17.23
2;Ventil 21c;22.34
\.

CREATE OR REPLACE FUNCTION python_trig_insert() RETURNS opaque AS '
plpy.notice("NEW:", TD["new"]);
plpy.notice("OLD:", TD["old"]);
plpy.notice("Ereignis:", TD["event"]);
plpy.notice("Zeitpunkt:", TD["when"]);
plpy.notice("Level:", TD["level"]);
plpy.notice("Name:", TD["relid"]);
plpy.notice("Argumente:", TD["args"]);
' LANGUAGE 'plpython';

CREATE TRIGGER trig_messwert_insert BEFORE INSERT ON messdaten
        FOR EACH ROW EXECUTE PROCEDURE python_trig_insert();
```

Nachdem wir die Tabelle angelegt und Daten eingefügt haben, wenden wir uns der Funktion zu. Wieder verwenden wir `plpy`, um die entsprechenden Informationen auszugeben. Alle vordefinierten Variablen sind in TD zu finden, dabei ist es einfach, auf die Datenquelle zuzugreifen.

Abschließend definieren wir den Trigger. Dabei ist zu bemerken, dass der Trigger vor dem Einfügen aktiviert werden soll.

Im Falle von `INSERT` könnte der Output wie folgt aussehen:

```
buch=# INSERT INTO messdaten VALUES (1, 'Ventil 21c', '22.10');
NOTICE:  ('NEW:', {'code': 'Ventil 21c', 'messwert': 22.1, 'id': 1})
NOTICE:  ('OLD:', None)
NOTICE:  ('Ereignis:', 'INSERT')
NOTICE:  ('Zeitpunkt:', 'BEFORE')
NOTICE:  ('Level:', 'ROW')
NOTICE:  ('Name:', '454708')
NOTICE:  ('Argumente:', None)
INSERT 454722 1
```

Jetzt wollen wir versuchen, die Funktion auszubauen. Ziel ist es, automatisch einen Eintrag in der Logging-Tabelle zu erzeugen. Zu diesem Zwecke müssen wir mithilfe der `Python`-Funktion direkt auf die Datenbank zugreifen, um den Datensatz einzufügen:

```
CREATE OR REPLACE FUNCTION python_trig_insert() RETURNS opaque AS '
xstr = "INSERT INTO logging (tstamp) VALUES (now()) "
plpy.execute(xstr)
' LANGUAGE 'plpython';
```

Wir setzen ein `INSERT`-Statement zusammen und senden es an den Server. Hierbei wird das Statement ausgeführt:

```
buch=# INSERT INTO messdaten VALUES (1, 'Ventil 21c', '22.10');
INSERT 455111 1
```

Nun wird der Eintrag auch in die Logging-Tabelle eingefügt:

```
buch=# SELECT * FROM logging;
 id |            tstamp
----+-------------------------------
  1 | 2002-06-14 13:06:01.568739+02
(1 row)
```

Sofern es notwendig ist, Daten nicht nur einzufügen, sondern direkt mit PL/Python abzufragen, ist auch das möglich. Sie müssen einfach den Rückgabewert von `plpy.execute` in einer Variable abfangen und können diese dann wie folgt verarbeiten:

```
rv = plpy.execute(query)
inhalt = rv[i]["feldname"]
```

Die Variable inhalt enthält jetzt das entsprechende Datenfeld.

10.2 Python als Scriptsprache

Python kann nicht nur als eingebettete Sprache, sondern auch ganz normal als Scriptsprache verwendet werden. Die PostgreSQL-Schnittstelle stellt eine Reihe von Möglichkeiten bereit, mit denen es möglich wird, Python direkt mit PostgreSQL zu verbinden. Sie werden schnell mit Python arbeiten können.

10.2.1 Datenbankverbindungen

Wie gewohnt beginnen wir mit der Herstellung einer Verbindung zur Datenbank. Das kann mithilfe des Pg-Moduls bewerkstelligt werden:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(dbname='buch', user='hs')
        print "Die Verbindung konnte hergestellt werden :)"
except:
        print "Ein Fehler ist aufgetreten :("
```

Am Beginn des Programms importieren wir das Modul. Danach stellen wir die Verbindung zur Datenbank her. Alle Parameter der Verbindung werden als Parameter übergeben, intern wird der korrekte Connect-String zusammengestellt.

Ein Script in Aktion würde etwa so aussehen:

```
[hs@duron python]$ ./connect.py
Die Verbindung konnte hergestellt werden :)
```

Die Methode `connect` unterstützt die folgenden Parameter:

```
connect([dbname], [host], [port], [opt], [tty], [user], [passwd])
```

Python bietet auch die Möglichkeit, Defaultwerte für eine Verbindung zu setzen. Hier ist ein Beispiel, wo das extensiv praktiziert wird:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        pg.set_defbase('buch')
        pg.set_defhost('localhost')
        pg.set_defport(5432)
        pg.set_defopt('-i')
        pg.set_deftty('tty3')

        conn = pg.connect(user='hs')
        print "Die Verbindung konnte hergestellt werden :)"
except:
        print "Ein Fehler ist aufgetreten :("
```

Dabei stellen wir eine Reihe von Parametern bereit. Wenn wir das Programm ausführen, wird es erfolgreich sein, obwohl wir im Connect-String selbst keinen Datenbanknamen mehr angeben:

```
[hs@duron python]$ ./connect.py
Die Verbindung konnte hergestellt werden :)
```

Alle defaultmäßig gesetzten Parameter können auch abgefragt werden; dafür gibt es eine Reihe von Methoden, deren Namen mit den Namen der im letzten Programm vorgestellten Methoden korrespondieren. Ersetzen Sie einfach das `set` durch `get` und schon haben Sie die Funktion gefunden (aus `set_defbase` wird also `get_getbase`).

Mithilfe der get-Funktionen können Sie die Default-Parameter abfragen.

10.2.2 Daten abfragen

Eines der wichtigsten Objekte beim Umgang mit Python ist das pgobject-Objekt. Es deckt den Großteil der für die tägliche Arbeit benötigten Funktionen ab und arbeitet weitgehend problemlos.

Eine der wichtigsten Funktionen des Objektes ist das Abfragen von Daten. Daten können mithilfe der `query`-Methode extrahiert werden. Werfen wir einen Blick auf ein einfaches Beispiel:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(user='hs',dbname='buch')
        res = conn.query("SELECT id, code, messwert FROM messdaten")
        print res

except:
        print "Ein Fehler ist aufgetreten :("
```

Wir weisen das Ergebnis der Abfrage `res` zu — die Daten werden einfach 1:1 ausgegeben, wie das aus dem nächsten Listing hervorgeht:

```
[hs@duron python]$ ./connect.py
id|code      |messwert
--+----------+--------
 1|Ventil 23c|  17.230
 2|Ventil 21c|  22.340
 1|Ventil 21c|  22.100
(3 rows)
```

Wenn Sie Daten als Dictionary abfragen wollen, so kann das auch einfach bewerkstelligt werden. Im folgenden Beispiel arbeiten wir die Daten zeilenweise ab und geben die Zeilen des Dictionaries entsprechend aus:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(user='hs',dbname='buch')
        code = "SELECT id, code, messwert FROM messdaten"
        for result in conn.query(code).dictresult():
                print result

except:
        print "Ein Fehler ist aufgetreten :("
```

In diesem Fall ist der Output nicht sonderlich elegant, aber die Python-Programmierer unter Ihnen werden schnell erkennen, wie ein Dictionary zu verarbeiten ist:

```
[hs@duron python]$ ./connect.py
{'code': 'Ventil 23c', 'messwert': 17.23, 'id': 1}
{'code': 'Ventil 21c', 'messwert': 22.34, 'id': 2}
{'code': 'Ventil 21c', 'messwert': 22.1, 'id': 1}
```

Sehen wir uns trotzdem eine umgänglichere Version der Daten an:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(user='hs',dbname='buch')
        code = "SELECT id, code, messwert FROM messdaten"
        for result in conn.query(code).dictresult():
                print "Ventil: ", result["code"], ", Messwert: ", \
                        result["messwert"]

except:
        print "Ein Fehler ist aufgetreten :("
```

Dabei ist es überhaupt kein Problem, auf die einzelnen Felder der Datenstruktur zuzugreifen.

Hierbei sieht der Output eine Spur freundlicher aus:

```
[hs@duron python]$ ./connect.py
Ventil:  Ventil 23c , Messwert:  17.23
Ventil:  Ventil 21c , Messwert:  22.34
Ventil:  Ventil 21c , Messwert:  22.1
```

Wenn Sie kein Fan von Dictionaries sind, können Sie das Ergebnis wie folgt extrahieren: Das nächste Listing enthält eine weitere Möglichkeit, die zum selben Ergebnis führt:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(user='hs',dbname='buch')
        code = "SELECT id, code, messwert FROM messdaten"
        for result in conn.query(code).getresult():
                print "Ventil: ", result[1], ", Messwert: ", \
                        result[2]

except:
        print "Ein Fehler ist aufgetreten :("
```

Hierbei verwenden wir die Methode `getresult`, um auf das Ergebnis zuzugreifen.

In vielen Fällen kann es sinnvoll sein, Metadaten eines Ergebnisses zu benutzen. Im folgenden Beispiel versuchen wir einige Daten abzufragen und am Bildschirm auszugeben:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.connect(user='hs',dbname='buch')
        code = "SELECT id, code, messwert FROM messdaten"
        result = conn.query(code)
        print "Tuples: ", result.ntuples()
        print "Spalten: ", result.listfields()
        print "Name von Spalte 3: ", result.fieldname(2)
        print "Nummer der Spalte code: ", result.fieldnum("code")

```

```
16 except:
17         print "Ein Fehler ist aufgetreten :("
```

Nach dem Herstellen der Verbindung zur Datenbank wird eine Abfrage ausgeführt. Dann geben wir die Zahl der gefundenen Datensätze aus und listen die Namen der Felder im Ergebnis auf. Mithilfe der Methode `fieldname` versuchen wir, den Namen der dritten Spalte abzufragen; bedenken Sie, dass auch Python die Spalten mit 0 zu indizieren beginnt. Die Methode `fieldnum` sucht den Index einer Spalte.

Bei uns sieht der Output des Programmes wie folgt aus:

```
1 [hs@duron python]$ ./connect.py
2 Tuples:  4
3 Spalten:  ('id', 'code', 'messwert')
4 Name von Spalte 3:  messwert
5 Nummer der Spalte code:  1
```

Die Information wird wie erwartet ausgegeben.

10.2.3 COPY

Immer wenn es um größere Datenmengen geht, ist der `COPY`-Befehl die erste Wahl, um die Performance Ihres Systemes zu erhöhen.

Zum Abarbeiten eines Datenbestandes werden folgenden Methoden zur Verfügung gestellt:

`putline(line)`: Sendet eine Zeile an das Backend

`endcopy()`: Beendet einen `COPY`-Befehl

Beide Funktionen funktionieren wie bei PHP und anderen Sprachen auch.

10.2.4 Die DB Wrapper-Klasse

Die DB Wrapper-Klasse stellt einige Funktionen zur Verfügung, die das Leben mit der Datenbank spürbar erleichtern.

Wenn Sie beispielsweise eine Liste der am System verfügbaren Datenbanken auslesen wollen, so können Sie das mit der Methode `get_databases` leicht bewerkstelligen:

```
1 #!/usr/bin/python
2
3 # Import des Modules
4 import pg
```

```
5
6 # Herstellen der Verbindung zur Datenbank
7 try:
8         conn = pg.DB(user='hs',dbname='buch')
9         print conn.get_databases()
10
11 except:
12         print "Ein Fehler ist aufgetreten :("
```

In unserem Fall gibt es fünf Datenbanken am System:

```
1 [hs@duron python]$ ./connect.py
2 ['test', 'buch', 'template1', 'template0', 'pearson']
```

Eine Verbindung zu irgendeiner Datenbank ist hier zwingend notwendig, da intern Systemtabellen ausgelesen werden. Man kann hier auch `template1` verwenden.

Wenn Sie abfragen wollen, ob eine bestimmte Spalte einen Primary Key hat, so können Sie auch dafür die DB Wrapper-Klasse verwenden:

```
1 #!/usr/bin/python
2
3 # Import des Modules
4 import pg
5
6 # Herstellen der Verbindung zur Datenbank
7 try:
8         conn = pg.DB(user='hs',dbname='buch')
9         try:
10                print conn.pkey('messdaten')
11        except:
12                print "kein Primary Key definiert"
13
14 except:
15        print "Ein Fehler ist aufgetreten :("
```

Wichtig ist, dass wir den Fehler abfangen, der zurückgegeben wird, wenn es keinen Primärschlüssel gibt — in unserem Fall wird der `except`-Zweig aufgerufen und der Fehler sauber abgefangen:

```
1 [hs@duron python]$ ./connect.py
2 kein Primary Key definiert
```

Hier existiert kein Schlüssel.

Nachdem wir herausgefunden haben, ob eine Tabelle Schlüssel hat oder nicht, wollen wir prüfen, ob es in der Datenbank noch weitere Tabellen gibt beziehungsweise welche Datenbankstruktur unsere Tabelle aufweist:

```
#!/usr/bin/python

# Import des Modules
import pg

# Herstellen der Verbindung zur Datenbank
try:
        conn = pg.DB(user='hs',dbname='buch')
        try:
                print "Tabellen: ", conn.get_tables()
                print "Messdaten: ", conn.get_attnames('messdaten')
        except:
                print "Fehler ..."

except:
        print "Ein Fehler ist aufgetreten :("
```

Mit der Methode `get_tables` fragen wir die Liste der Tabellen ab. `get_attnames` liefert anschließend die Liste der Spalten in der Tabelle:

```
[hs@duron python]$ ./connect.py
Tabellen:  ['messdaten']
Messdaten:  {'messwert': 'text', 'id': 'int', 'code': 'text', 'oid':
'int'}
```

Dabei gibt es nur diese eine Tabelle. Etwas interessanter ist die zweite Zeile: Hier werden nicht nur die Felder sondern auch noch Zusatzinformationen ausgegeben, die von großem Interesse sein können. Speziell der Datentyp ist für viele Applikationen von Bedeutung und kann mithilfe der DB Wrapper-Klasse bestimmt werden.

Die DB Wrapper-Klasse stellt noch weitere Funktionen zur Verfügung, die jedoch aus unserer Sicht allesamt als eher unwichtig eingestuft werden können.

Kapitel 11

Eine wissenschaftliche Anwendung: EFEU

Im wissenschaftlich-technischen Umfeld stellen sich häufig Aufgaben, die über die üblichen, einfachen Anforderungen hinausgehen. Jede wissenschaftliche Disziplin stellt eigene Anforderungen, die es bestmöglich zu bewältigen gilt.

Eine dieser wissenschaftlichen Disziplinen beschäftigt sich mit Arbeitsmarktbeobachtung und der Analyse von Individualdaten. In genau diesem Umfeld ist das EFEU-Paket entstanden und heute noch fester Bestandteil wissenschaftlich arbeitender Institutionen in Österreich.

Bei der Analyse von Individualdaten geht es darum, vergleichsweise große Datenmengen (zig Millionen Datensätze) effizient zu analysieren und in Form von Tabellen darzustellen und genau damit werden wir uns in diesem Abschnitt beschäftigen.

11.1 Über EFEU

Das EFEU-Paket wurde fast komplett von Erich Frühstück entwickelt und gepflegt. Wie bereits erwähnt wird das Paket im wissenschaftlichen Bereich eingesetzt, um Analysen über den österreichischen Arbeitsmarkt zu erstellen. Um diesen Anforderungen gerecht zu werden, verfügt EFEU über ausgereifte Funktionen zur Verarbeitung von Datumsformaten sowie zur Durchführung von mathematischen Berechnungen.

11.1.1 Lizenz und Verfügbarkeit

Alle Teile, die nicht speziell für die Arbeitsmarktbeobachtung entwickelt worden sind, werden von Erich Frühstück frei zur Verfügung gestellt. Alle Bibliotheken sind unter der LGPL verfügbar, die anderen Komponenten stehen unter der GPL.

Das EFEU-Paket kann von `http://efeu.cybertec.at` heruntergeladen werden.

11.1.2 Aufbau

EFEU ist ein modulares System, das leicht erweiterbar und leicht zu warten ist. EFEU kann in die folgenden Komponenten unterteilt werden:

`mkmf`: `mkmf` dient zur Generierung von Makefiles. So genannte Imakefiles werden interpretiert und in Makefiles verwandelt. In früheren Versionen von EFEU sind diese Modifikationen mithilfe des C-Präprozessors durchgeführt worden. Da die Funktionsweise eines Präprozessors nicht im ANSI-Standard genormt ist, wird diese Vorgehensweise aber nicht mehr verfolgt. Das Konzept von `mkmf` ist sehr stark an die Makefilegenerierung angelehnt, die von X11 bekannt ist.

`dir2make`: EFEU erlaubt es, Makefiles direkt aus Sourcebäumen zu generieren. Dieses Tool macht es sehr einfach, Funktionen zu EFEU hinzuzufügen und nahtlos in das System zu integrieren.

`esh`: Der `esh`-Interpreter bildet den Kern des gesamten Paketes. Die Syntax des Interpreters ist C/C++ ähnlich und leicht zu verstehen beziehungsweise leicht zu erlernen. Mithilfe des Interpreters ist es ohne Probleme möglich, effizient mit mehrdimensionalen Datenstrukturen zu arbeiten.

`mksource`: Vielfach ist es sinnvoll, C-Sourcen generieren zu lassen. Speziell wenn es darum geht, neue Klassifikationen für verschiedenste Datenbestände in das System einzubinden, kann es sehr sinnvoll sein, den Code nicht direkt schreiben zu müssen, sondern generieren zu lassen.

`eis`: Sobald neue Klassifikationen in das System eingebunden sind, sind diese auch automatisch in `eis` verfügbar. `eis` ist das EFEU-Informationssystem und in zwei Versionen verfügbar. `eis-cgi` stellt die Information in Form von HTML zur Verfügung. `eis` selbst ist ein Curses-basiertes Programm.

Das gesamte System basiert auf einer Reihe von C-Bibliotheken, die für Dinge wie das Speichermanagement, das Arbeiten mit Strings und dergleichen verwendet werden können. Diese Bibliotheken sind leicht zu verwenden und können einfach für die Entwicklung eigener Software herangezogen werden.

11.2 Installation

Eigentlich ist es fast überflüssig, der Installation von EFEU einen eigenen Abschnitt zu widmen, da der gesamte Prozess nur ein einziger Befehl ist.

Nach dem Entpacken des Archives können Sie EFEU mithilfe von `make` installieren:

```
1 [hs@notebook efeu]$ make all
```

EFEU sucht sich alle Komponenten selbst. Wenn die Header-Files von PostgreSQL nicht gefunden werden, wird das Modul einfach nicht mitkompiliert. Sie sollten also darauf achten, dass PostgreSQL in einem »vernünftigen« Verzeichnis installiert ist, damit es zu keinen Problemen kommen kann.

Zusätzlich zur Kompilierung sollten Sie noch das `bin`-Verzeichnis in den Pfad übernehmen.

Bei der Installation von EFEU werden die Dokumentationen zum Paket automatisch mit generiert. Alle Dokumentationen sind als HTML oder Postscript verfügbar.

11.3 esh als Interpretersprache

Der esh-Interpreter spielt eine zentrale Rolle im gesamten EFEU-Paket. Mithilfe von esh können Sie komplexe Operationen realisieren und effiziente Scripts schreiben. In diesem Abschnitt zeigen wir, welche Möglichkeiten esh birgt. Dieses Wissen werden Sie benötigen, um in den nächsten Abschnitten mit PostgreSQL arbeiten zu können. Dieser Abschnitt soll eine kurze Einführung geben.

esh erlaubt es, interaktiv Berechnungen durchzuführen. Um den Interpreter zu starten, müssen Sie einfach esh eingeben. Hier folgt ein kurzes Beispiel an:

```
esh: 3+4*2
11
```

In diesem Beispiel ist die EFEU-Shell auch als Taschenrechner sehr brauchbar.

esh unterstützt typgebundene Funktionen. Jeder Variablen kann ein Datentyp zugeordnet werden. Auf diese Weise können Sie genau festlegen, wie eine Operation zu erfolgen hat. Das nächste Beispiel zeigt, wie mit double gearbeitet werden kann:

```
esh: double i
0.00
esh: i=34.2
34.20
```

Auch Typumwandlungen können problemlos durchgeführt werden:

```
esh: int(i)
34
```

esh kann nicht nur interaktiv eingesetzt werden. In den meisten Fällen wird esh zur Implementierung von Scripts verwendet. Das folgende Beispiel zeigt ein einfaches Programm:

```
#!/usr/local/efeu/bin/esh

#if     (int(argv[1]) > int(argv[2]))
        printf("argv[1] ist größer als argv[2]\n");
#else
        printf("argv[2] ist größer als argv[1]\n");
#endif
```

Das Programm tut nichts anderes als zu prüfen, welcher Parameter größer ist. Anhand dieser Anwendung können Sie erkennen, wie ein esh-Programm aussieht und welche Syntax verwendet wird. Wie bereits erwähnt, haben esh-Programme große Ähnlichkeit mit C/C++- und Shell-Programmen.

11.4 Datenbankinteraktion

Die Synthesis-Erweiterungen für EFEU bieten eine Reihe von Möglichkeiten zur Interaktion mit EFEU, spezifischen Datenbankformaten, die großteils auf Bitfeldern beruhen. Speziell für unser »PostgreSQL Developers Handbook« hat Erich Frühstück eine Reihe von Erweiterungen für PostgreSQL implementiert, die unter der GPL frei zur Verfügung stehen. Wir beschäftigen uns hier genauer mit den PostgreSQL-Erweiterungen für EFEU. Sie werden erkennen, welche Möglichkeiten sich bieten und wie EFEU eingesetzt werden kann.

Um mit der Datenbank zu arbeiten und um zu sehen, wie Daten abgefragt werden können, müssen Sie zuerst eine Tabelle mit einigen Datensätzen anlegen, die etwa so aussehen könnten:

```
CREATE TABLE person (
        name            text,
        geschlecht      char(1),
        staat           text
);

COPY person FROM stdin USING DELIMITERS ';';
Karl;m;Österreich
Paul;m;Österreich
Johann;m;Schweiz
Carla;f;Österreich
Jola;f;Schweiz
\.
```

Der Datenbestand enthält Informationen über das Geschlecht und die Staatsbürgerschaft von Personen. Betrachten wir also, wie Daten aus der Datenbank abgefragt werden können:

```
loadlib("PG");

//      verbinden zur datenbank
PG pg = PG("dbname=buch user=hs");
pg.query(string !
SELECT geschlecht, staat, COUNT(*)
        FROM person
        GROUP BY geschlecht, staat
!);

//      ausgeben der daten
PrintListDelim = "\t";
mdmat md = pg.mdmat(int, "COUNT", "geschlecht, staat")

iostd << "\n# geschlecht als spalten\n";
```

```
16 md.mark("geschlecht").print(iostd, "title");
17
18 iostd << "\n# staat als spalten\n";
19 md.mark("stat").print(iostd, "title");
20 free(md);
21 close(pg);
```

Am Beginn des Programmes binden wir die PG-Bibliothek ein, die für die Interaktion mit PostgreSQL notwendig ist. Im Folgenden erstellen wir eine Verbindung zur Datenbank. Der Datenbankhandle ist vom Datentyp `PG` und kann über eigene Funktionen angesprochen werden.

Nach dem Verbinden zur Datenbank wird eine Abfrage abgesetzt. Nach dem Ausführen der Abfrage werden die Daten in zwei verschiedenen Formaten ausgegeben. Bevor die eigentliche Ausgabe erfolgt, wird das Ergebnis in einen Datenwürfel konvertiert. Wir werden uns im nächsten Schritt noch genauer mit Datenwürfeln beschäftigen.

```
1 [hs@notebook tmp]$ esh Script.esh
2 MD: name="COUNT by geschlecht, staat", type=int, dim=2, size=16
3
4 # geschlecht als spalten
5 COUNT by geschlecht, staat
6 2       f       m
7 Schweiz 1       1
8 Österreich      1       2
9
10 # staat als spalten
11 COUNT by geschlecht, staat
12 1       .
13 f.Schweiz       1
14 f.Österreich    1
15 m.Schweiz       1
16 m.Österreich    2
```

11.5 Mehrdimensionale Datenmatrizen

Mehrdimensionale Datenmatrizen bilden den Core vieler esh-Anwendungen. Wir wollen uns hier näher mit diesen Datenstrukturen beschäftigen und beobachten, welche Möglichkeiten der Datenverarbeitung sich bieten.

Bevor wir die mehrdimensionalen Strukturen behandeln, wollen wir die Frage stellen, wie Datenmatrizen funktionieren. Eine Datenmatrix besteht aus einer Reihe von Achsen, die wiederum eine Reihe von Elementen enthalten. Achsen werden miteinander verkreuzt und der dadurch beschriebene Datenwürfel wird als Datenmatrix bezeichnet. Mit anderen Worten: Ein Datenwürfel enthält Werte für alle

Kombinationen von Elementen. Nehmen wir an, eine Datenmatrix enthält zwei Achsen mit je zwei Werten (etwa die Achse Geschlecht, die die Elemente »Mann« und »Frau« enthält, und die Achse »Staat«, die eine Liste von Staaten enthält). Die Matrix enthält nun Werte für Männer und Frauen in den jeweiligen Staaten. Kommt eine dritte Achse wie etwa das Einkommen dazu, enthält die Matrix Informationen über alle Einkommensklassen von Männern und Frauen nach Staaten gegliedert.

Nach dieser zugegeben etwas theoretischen Übersicht über Datenmatrizen wollen wir uns einem praktischen Beispiel zuwenden:

```
1  loadlib("PG");
2
3  PG pg = PG("dbname=buch user=hs");
4
5  pg.query(string !
6  SELECT geschlecht, staat, COUNT(*)
7          FROM person
8          GROUP BY geschlecht, staat
9  !);
10
11 mdmat md = pg.mdmat(int, "COUNT", "geschlecht, staat");
12 md.save("ergebnis.dat");
13
14 free(md);
15 close(pg);
```

Nach dem Anlegen eines Objektes vom Typ `PG` führen wir eine Abfrage aus. Diese Abfrage zählt und gruppiert die Elemente in der Datenbank. Danach wird die eigentliche Datenmatrix angelegt. Der Datentyp der Werte in der Datenmatrix wird auf Integer festgesetzt. Der zweite Parameter gibt an, welche Spalte die eigentlichen Werte enthält. Der letzte Parameter listet die Achsen auf, die als Klassifikationen zu verwenden sind. In unserem Beispiel haben wir die Werte nach Geschlecht und Staatsbürgerschaft gruppiert und aus diesem Grund ist es nur nahe liegend, diese Achsen zu verwenden. Das Ergebnis der Abfrage wird in `ergebnis.dat` gespeichert.

Dieses Beispiel versucht zu beschreiben, was unter einem Würfel und einer Achse zu verstehen ist. Eine Achse enthält Elemente gleichen Typs. Hier haben wir beispielsweise verschiedene Geschlechter auf einer Achse zusammengefasst. Der Würfel wird durch die Kombination der jeweiligen Achsen beschrieben Das Ergebnis der Abfrage wird in `ergebnis.dat` gespeichert.

Um auf eine Datenmatrix zuzugreifen, können Sie den Befehl `mdprint` verwenden:

```
[hs@notebook tmp]$ mdprint ergebnis.dat
##MDMAT   2.0
##Titel   COUNT by geschlecht, staat
##Type    int
##Locale  C
##Zeilen  geschlecht
##Spalten staat
2       Schweiz Österreich
f       1       1
m       1       2
```

Wir haben hier die gesamte Datenmatrix ausgegeben. Nach dem beschreibenden Header folgen die Daten. Wie Sie erkennen können, wird die Liste der Staaten in der x-Achse angeführt, was nicht dem Ergebnis der Abfrage entspricht. Wenn wir die Liste der Geschlechter als x-Achse deuten wollen, so können wir das dem System mithilfe des -x-Flags mitteilen:

```
[hs@notebook tmp]$ mdprint -x geschlecht ergebnis.dat
##MDMAT   2.0
##Titel   COUNT by geschlecht, staat
##Type    int
##Locale  C
##Zeilen  staat
##Spalten geschlecht
2       f       m
Schweiz 1       1
Österreich      1       2
```

Jetzt haben wir das Ergebnis in transponierter Form ausgegeben. Das kann sinnvoll sein und die Möglichkeit, Daten schnell zu transponieren, kann ein bedeutendes Feature sein.

Wenn bestimmte Werte auf einer Achse aggregieren wollen, ist das leicht zu bewerkstelligen. Mithilfe eines normalen Doppelpunktes können Sie anzeigen, dass jetzt ein neuer, durch Summierung entstandener Wert folgt. Das folgende Beispiel zeigt, wie die Elemente der Achse `geschlecht` summiert werden können:

```
[hs@notebook tmp]$ mdprint ergebnis.dat -b geschlecht=:GES[m,f]
2       Schweiz Österreich
GES     2       3
```

Das `-b`-Flag sorgt dafür, dass der Header unterdrückt wird. Weiter werden die Elemente der Achse `geschlecht` entsprechend summiert.

Um von EFEU erzeugte Daten bequem importieren zu können, ist es möglich, Daten als `cvs`-Dateien abzuspeichern. Zu diesem Zwecke ist die Option `-csv` an den Befehl anzuhängen:

```
[hs@notebook tmp]$ mdprint ergebnis.dat -b geschlecht=:GES[m,f] --csv
"2";"Schweiz";"Österreich"
"GES";2;3
```

`mdprint` dient zum Ausgeben von Informationen in einer Datenmatrix. Der Befehl mdfile kann dazu verwendet werden, um Informationen über die Datenmatrix zu extrahieren. Betrachten wir, was `mdfile` über unsere Datenmatrix zu sagen hat:

11.6 Textgenerierung mit Efeudoc

Das Generieren von Texten und Berichten ist eng mit der Verwendung von Datenmatrizen verflochten. Um mit Daten angereicherte Berichte zu erzeugen, ist es sinnvoll, die vorgenerierten Daten aus der Datenmatrix direkt in ein Dokument zu übernehmen. Mithilfe von LaTeX lassen sich daher auf diesem Wege hübsche Berichte generieren, die durch das Aufrufen eines Scriptes leicht aktualisiert werden können.

In diesem Abschnitt werden wir erörtern, wie PostScriptdateien auf Basis von Efeudoc erstellt werden können. Ziel des Abschnittes ist es, einen Einblick in die Domäne von multidimensionalen Datenwürfeln und Efeudoc zu geben.

Im folgenden Beispiel werden wir uns mit einer Datenmatrix beschäftigen, die Informationen über Standardbeschäftigte in Österreich enthält. Die Datenmatrix kann einfach mithilfe eines `esh`-Programms erzeugt werden und sollte kein Problem darstellen, da wir das im letzten Beispiel bereits gezeigt haben. Werfen wir einen kurzen Blick auf die Daten:

```
[hs@duron tmp]$ mdprint tab.dat -b
3       GES     MANN    FRAU
1996    2841998.91      1514941.26      1327057.65
1997    2831466.85      1488255.34      1343211.51
1998    2943072.33      1495309.32      1447763.01
1999    2972969.32      1503380.00      1469589.32
2000    3062792.90      1566618.58      1496174.32
```

Hier enthält die Matrix Informationen über Beschäftige nach Jahren und Geschlecht. Ziel ist es nun, die Daten in der Datenmatrix als Basis für einen LaTeX-Bericht heranzuziehen:

```
\title  Tabellenbeispiel

/*      Makrodefinition
*/

\mktab={{
str arg = ParseLine(cin);
```

```
8  str name = strcut(arg, "%s");   /* Filename von Selektion abspalten */
9  mdmat md = mdload(name, arg);
10
11 fprintf(cout, "\\tab |l|%s\n", "r|" * dim(md.label("jahr")));
12 fprintf(cout, "[-]| %s\n", cat(" | ", md.label("jahr")));
13
14 /*      Tabellendefinition (nächster Absatz) laden
15         und als IO-Struktur abarbeiten
16 */
17
18 IO def = ParsePar(cin);
19
20 while (getline(def, name, arg))
21 {
22         cout << "[-]" + arg;
23
24         for (x in md.data("grp=" + name))
25                 fprintf(cout, " | %.0f", x);
26
27         cout << "\n";
28 }
29
30 cout << "[-]\n\n";
31 }}
32
33 /*      Text mit Tabellen
34 */
35
36 Vortext
37
38 \mktab  tab.dat
39 GES     Insgesamt
40 MANN    Männer
41 FRAU    Frauen
42
43 Zwischentext
44
45 \mktab  tab.dat jahr=#-3:
46 GES     Insgesamt
47 MANN    Männer
48 FRAU    Frauen
49
50 Nachtext
```

Am Beginn des Programms sind Makrodefinitionen zu finden. Diese Makros sind notwendig, um die Daten in der Datenmatrix korrekt zu verarbeiten. Folglich wird die Datenmatrix mithilfe von `mdload` geladen. Ist die Matrix einmal geladen, kann sie sehr schnell weiterverarbeitet werden.

Im nächsten Schritt wird der Tabellenkopf ausgegeben. Zu diesem Zweck wird `fprintf` verwendet. Wenn Sie mit C vertraut sind, wird Ihnen ein derartiger Befehl nicht unbekannt sein. Anschließend wird eine IO Struktur angelegt. IO-Strukturen sind ein wichtiger Bestandteil von EFEU, der Ihnen ein mächtiges Werkzeug in die Hände gibt. Nach dem Anlegen der IO-Struktur wird die Datenmatrix abgearbeitet. Die Tabelle wird zeilenweise aufgebaut und esh sucht sich dabei die relevanten Elemente der Datenmatrix.

Nach der Makrodefinition geht es an die Generierung der Tabellen. Zu diesem Zwecke wird die Funktion mehrmals aufgerufen. Vor und nach den Funktionsaufrufen wird jeweils ein Text ausgegeben. Das wird Ihnen helfen, zu erkennen, welche Bereiche von welchen Makroaufrufen erzeugt werden.

Am Beginn wird der gesamte Datenbestand ausgegeben. Die zweite Tabelle enthält nur mehr Informationen über die letzten drei Jahre.

Um das Dokument zu generieren, müssen Sie das Programm `efeudoc` wie folgt aufrufen:

```
1 efeudoc test.doc -p -o test.ps
```

Das Flag `-p` sorgt dafür, dass PostScript-Code generiert wird. `-o` definiert das Outputfile. In unserem Falle heißt dieses File `test.ps`.

Und so sieht das Resultat aus:

1

Vortext

	1996	1997	1998	1999	2000
Insgesamt	2841999	2831467	2943072	2972969	3062793
Männer	1514941	1488255	1495309	1503380	1566619
Frauen	1327058	1343212	1447763	1469589	1496174

Zwischentext

	1998	1999	2000
Insgesamt	2943072	2972969	3062793
Männer	1495309	1503380	1566619
Frauen	1447763	1469589	1496174

Nachtext

11.7 Fazit

EFEU ist ein mächtiges, stabiles Tool, das für spezielle Anwendungen große Vorteile bringt. Vor allem, wenn es um Anwendungen im Bereich der Massendatenverarbeitung geht, kann EFEU seine Trümpfe voll ausspielen. Durch die großzügige Unterstützung mehrdimensionaler Datenstrukturen bietet sich EFEU hervorragend für eine Vielzahl von Anwendungen an.

Kapitel 12

Tcl

Seit John Ousterhout im Jahre 1980 seine erste Version von Tcl vorgestellt hat, hat sich viel verändert und Tcl ist zu einer der bekanntesten Sprachen überhaupt geworden. Speziell in Kombination mit Tk hat sich Tcl stets gut behauptet und wird immer noch mit großem Erfolg eingesetzt. Durch die große Flexibilität der Sprache gibt es viele Fans von Tcl weltweit. Tcl reiht sich auch in die Reihen der von PostgreSQL unterstützten Sprachen ein. Die Möglichkeiten, die sich dabei ergeben, wollen wir in diesem Kapitel erläutern.

12.1 PL/Tcl

Meistens wird Tcl als eingebettete Sprache verwendet. Da Tcl bereits als eine solche entwickelt worden ist, stellt es kein großes Problem dar, PostgreSQL mit Tcl zu verbinden. Im Prinzip stehen zwei Möglichkeiten zur Verfügung. Wenn wir Tcl als Trusted Sprache verwenden, stehen uns keine System Calls zur Verfügung — der Untrusted Interpreter erlaubt hingegen jede Art von Operation. Wir werden hier auf beide Möglichkeiten eingehen.

12.1.1 Trusted PL/Tcl

Um die »vertrauenswürdige« Version von Tcl zu unserer Datenbank hinzuzufügen, gehen wir wie folgt vor:

```
1 [hs@duron tcl]$ createlang pltcl buch
```

Sofern die Datenbank keinen Fehler ausgibt, ist es zu keinen Problemen gekommen und wir können Tcl jetzt verwenden.

Einfache Beispiele

Beginnen wir damit, wie wir eine einfache Funktion schreiben können:

```
1 CREATE OR REPLACE FUNCTION summe(int4, int4) RETURNS int4 AS '
2         return [expr $1 + $2]
3 ' LANGUAGE 'pltcl';
```

Also berechnen wir nur die Summe von zwei Zahlen, was sehr einfach möglich ist. Wichtig dabei ist, dass das Ergebnis auch wieder als Integer-Wert zurückgegeben wird.

Wenn wir die Funktion testen, werden wir gar Wundersames erleben:

```
1 buch=# SELECT summe(22, 33);
2  summe
3 ────────
4     55
5 (1 row)
```

Tcl berechnet die Summe korrekt und gibt das Ergebnis in der gewünschten Form zurück.

NULL-Werte

Nach diesem zugegeben etwas trivialen Beispiel wollen wir uns nun einigen weiteren Möglichkeiten zuwenden. Beginnen wir mit der Behandlung von NULL-Werten:

```
buch=# SELECT 22 + NULL;
 ?column?
----------

(1 row)
```

Sofern einer der beiden Werte, die wir summieren wollen, NULL ist, wird auch NULL als Ergebnis ausgegeben. Dieses Verhalten ist im ANSI-Standard festgelegt. Wenn Sie NULL-Werte jedoch als 0 Werte betrachten wollen, können Sie beispielsweise mit einer Tcl-Funktion arbeiten, die wie folgt aussehen könnte:

```
CREATE OR REPLACE FUNCTION summe(int4, int4) RETURNS int4 AS '
        if      {[argisnull 1] || [argisnull 2]} {
                return 0
        }
        return [expr $1 + $2]
' LANGUAGE 'pltcl';
```

Wir prüfen, ob einer der beiden Eingabewerte ein NULL-Wert ist. Ist das der Fall, geben wir als Ergebnis 0 statt NULL aus. Ist keiner der Eingabewerte NULL, berechnen wir das Ergebnis wie gehabt.

Versuchen wir, die Funktion zu testen:

```
buch=# SELECT summe(NULL, 22);
 summe
-------
     0
(1 row)
```

Hier wird 0 statt NULL zurückgegeben.

Für die SQL-Puristen unter Ihnen ist auch eine Lösung mit CASE/WHEN denkbar, die etwa so aussehen könnte (prototypisch):

```
buch=# SELECT CASE WHEN (22 + NULL IS NULL) THEN 0 ELSE 22 END;
 case
------
    0
(1 row)
```

Im Praxiseinsatz sollten die jeweiligen Zahlen durch Spalten ersetzt werden.

Tabellen als Inputparameter

Wenn Sie Tcl-Funktionen schreiben wollen, die eine große Anzahl von Spalten einer Tabelle akzeptieren sollen, können Sie auf zwei Arten vorgehen. Eine Möglichkeit ist, die Spaltennamen explizit aufzuzählen. Das ist bei sehr großen Tabellen eine denkbar ungünstige Variante. In solchen Fällen ist es effizienter, einer Tcl-Funktion den Namen einer Tabelle zu übergeben. Legen wir also eine Tabelle an und fügen einige Datensätze ein:

```
CREATE TABLE produkt (
        id      int4,
        name    text,
        preis   numeric(9,2)
);

COPY produkt FROM stdin USING DELIMITERS ';';
1;Perlenkette;4632.20
2;Diamant;932.99
3;Golduhr;60320
\.
```

Ziel ist es, eine kleine Funktion zu schreiben, die herausfindet, ob ein Produkt mehr oder weniger als eine bestimmte Summe kostet:

```
CREATE OR REPLACE FUNCTION pruefen(int4, produkt) RETURNS text AS '

        if      {$1 < $2(preis)} {
                return "mehr"
        }

        return "weniger"
' LANGUAGE 'pltcl';
```

In diesem Beispiel übergeben wir der Funktion zwei Parameter. Der erste enthält den Wert, den wir prüfen wollen. Der zweite Parameter ist die an die Funktion übergebene Tabelle. Innerhalb der Funktion können wir auf beide Komponenten zugreifen. Beim zweiten Parameter können wir in Klammern angeben, auf welches Feld wir zugreifen wollen. Abhängig vom Ergebnis des If-Statements geben wir den gewünschten Text aus.

Um die Funktion aufzurufen, gehen wir wie folgt vor:

```
buch=# SELECT name || ' kostet ' || pruefen(1000, produkt)
FROM produkt;
           ?column?
-----------------------------
 Perlenkette kostet mehr
 Diamant kostet weniger
 Golduhr kostet mehr
(3 rows)
```

Wir prüfen alle Produkte, die mehr oder weniger als 1.000 Geldeinheiten kosten.

Datenbankinteraktion mithilfe von SPI

In sehr vielen Fällen kann es notwendig sein, innerhalb einer `Tcl`-Funktion direkt auf die Datenbank zuzugreifen. Da es wenig Sinn hätte, eine Verbindung zur Datenbank herzustellen (Sie sind ja schon »in« der Datenbank), bietet PostgreSQL eine Möglichkeit, direkt auf einige Interna zuzugreifen. Das geschieht mithilfe des so genannten SPI-Interfaces. SPI ist eine C-Schnittstelle, die es erlaubt, direkt auf den Executor von PostgreSQL zuzugreifen. Tcl stellt einige Funktionen zur Verfügung, die auf das SPI aufsetzen. Wir werden uns hier einige Beispiele ansehen. Beginnen wir mit einer einfachen Anwendung:

```
CREATE OR REPLACE FUNCTION ausgabe() RETURNS text AS '
        spi_exec -array MYDATA "SELECT max(preis) AS r FROM produkt"
        return "der maximale Preis: $MYDATA(r)"
' LANGUAGE 'pltcl';
```

Wir führen eine Abfrage aus, die genau einen Wert liefert. Das Ergebnis wird auf `MYDATA` zugewiesen und kann bequem ausgegeben werden.

Sehen wir uns also an, was passiert, wenn wir die Funktion aufrufen:

```
buch=# SELECT ausgabe();
            ausgabe
-------------------------------
 der maximale Preis: 60320.00
(1 row)
```

Die Funktion gibt einen einfachen Text aus.

Nach diesem Beispiel betrachten wir, welche Möglichkeiten die SPI-Schnittstelle zur Verfügung stellt; das nächste Listing enthält eine Referenz:

```
spi_exec ?-count n? ?-array name? query ?loop-body?
spi_prepare query typelist
spi_execp ?-count n? ?-array name? ?-nulls string? queryid
        ?value-list? ?loop-body?
```

```
5 spi_lastoid
6 quote string
7 elog level msg
```

Sehen wir uns die Funktionen jetzt im Detail an:

`spi_exec`	Führt ein SQL-Statement aus. `n` definiert die maximale Anzahl der vom Statement betroffenen Datensätze. `name` definiert das Array für das Ergebnis des Statements. `query` enthält den Code der Abfrage und `loop-body` eine Schleife zum Ausführen von weiterem Code. In dieser Schleife können Sie etwa das Array ausgeben — die Schleife wird für jeden Datensatz aufgerufen.
`spi_prepare`	Bereitet eine Abfrage vor und speichert den Execution-Plan
`spi_execp`	Führt eine vorbereitete Abfrage aus
`spi_lastoid`	Gibt die letzte Object ID eines eingefügten Datensatzes aus
`quote`	Maskiert Zeichen
`elog`	Erzeugt Logging-Information

Funktionsüberladungen

Tcl selbst unterstützt keine überladenen Funktionen. Da PostgreSQL aber Überladungen als Core-Features unterstützt, haben sich die Tcl-Entwickler etwas Besonderes einfallen lassen. Intern enthalten alle Funktionen die Object ID der entsprechenden Zeile der Funktion in der Systemtabelle `pg_proc` — auf diese Weise ist es möglich, auch auf überladene Tcl-Funktionen zurückzugreifen.

12.1.2 Untrusted PL/Tcl

Wenn Sie in keiner sicherheitskritischen Umgebung sind, können Sie die untrusted Version von PL/Tcl verwenden. Um diese zur Datenbank hinzuzufügen, gehen Sie wie folgt vor:

```
1 [hs@duron hs]$ createlang pltclu buch
```

Sobald Sie Tcl zu Ihrer Datenbank hinzugefügt haben, können Sie jedes beliebige Script in eine Funktion einbauen. Versuchen wir eine etwas eigenwillige Anwendung für PostgreSQL zu schreiben:

```
1 CREATE OR REPLACE FUNCTION Script(text) RETURNS bool AS '
2         exec $1
3         return "t"
4 ' LANGUAGE 'pltclu';
```

Sofern Sie `xhost localhost` ausgeführt haben, wird das obrige Script zu keinen Problemen führen und brav eine beliebige Anwendung starten:

```
buch=# SELECT skript('xclock');
```

In unserem Fall können wir sehr bequem das Programm `xclock` starten.

Diese Anwendung wird Ihnen fremd und sinnlos vorkommen aber sie zeigt sehr gut, welche Möglichkeiten sich auftun. In der Regel werden Sie die uneingeschränkte Version von Tcl verwenden, um Mails oder dergleichen zu verschicken — für die Kreativen unter Ihnen wird es aber auch möglich sein, mithilfe der Datenbank ein Programm zu kompilieren. Der Fantasie sind keine Grenzen gesetzt. Ich denke, die eben vorgestellten Möglichkeiten sind ein guter Grund, PostgreSQL einzusetzen.

12.2 Tcl-Scripts

Nach diesem Exkurs über PL/Tcl wenden wir uns nun Tcl als Interpetersprache zu. Welche Möglichkeiten bieten sich?

12.2.1 Datenbankverbindungen

Wie immer beginnen wir damit, eine Verbindung zur Datenbank herzustellen. Sehen wir uns das folgende Beispiel an:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect -conninfo "dbname=buch user=postgres"]
puts "Verbindung aufgebaut";
pg_disconnect $conn
```

Am Beginn des Programms stellen wir eine Verbindung zur Datenbank her. Nach dem Ausgeben einer Meldung trennen wir diese Verbindung wieder. Wenn Sie das Programm ausführen, sollte es zu keinen Problemen mehr kommen:

```
[hs@duron tcl]$ ./connect.tcl
Verbindung aufgebaut
```

Es gibt auch noch eine zweite Art, sich zur Datenbank zu verbinden:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect buch -host localhost]
puts "Verbindung aufgebaut";
pg_disconnect $conn
```

Dabei nutzen wir eine leicht andere Syntax, die im Folgenden erläutert wird:

```
pg_connect databaseName [-host hostName] [-port portNumber]
        [-tty pgtty]
```

Wenn Sie detaillierte Informationen über eine Verbindung abfragen wollen, können Sie `pg_conndefaults` verwenden, wie es im nächsten Listing gezeigt wird:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect buch -host localhost]
puts [pg_conndefaults]
pg_disconnect $conn
```

Wenn Sie das Programm ausführen, werden Sie gleich mit einer Liste von Informationen überschüttet:

```
[hs@duron tcl]$ ./connect.tcl
{authtype Database-Authtype D 20 {}} {service Database-Service {} 20
{}} {user Database-User {} 20 hs} {password Database-Password * 20 {}}
{dbname Database-Name {} 20 hs} {host Database-Host {} 40 {}}
{hostaddr Database-Host-IPv4-Address {} 15 {}} {port Database-Port {}
6 5432} {tty Backend-Debug-TTY D 40 {}} {options Backend-Debug-Options
D 40 {}}
```

12.2.2 Datenmodifikationen

Nach dieser Einführung in die Datenbankverbindungen sehen wir uns nun an, wie Daten modifiziert werden können. Tcl stellt einfache Möglichkeiten für derartige Operationen zur Verfügung.

Ziel ist es, alle Preise in der Tabelle `produkt` um 1 zu erhöhen:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect -conninfo "dbname=buch user=postgres"];
set res [pg_exec $conn "UPDATE produkt SET preis=preis + 1"];

pg_disconnect $conn
```

Das `UPDATE`-Kommando kann mithilfe von `pg_exec` an die Datenbank geschickt werden. Die Änderungen werden sofort wirksam:

```
buch=# SELECT * FROM produkt;
 id |    name     |  preis
----+-------------+---------
  1 | Perlenkette |  4633.20
```

```
  2 | Diamant     |   933.99
  3 | Golduhr     | 60321.00
(3 rows)
```

Wenn Sie den Inhalt der Tabelle mit den ursprünglichen Daten vergleichen, werden Sie sehen, dass sich die Daten genau um eine Geldeinheit unterscheiden.

12.2.3 Abfragen

Die Kernaufgabe einer Datenbank ist das Abspeichern von Daten. Die Kernaufgabe von SQL ist es unter anderem, Daten abzufragem. Die Aufgabe einer Programmiersprache ist es, die Daten in Kooperation mit der Datenbank abzufragen und in einer zu bestimmenden Form weiterzuverarbeiten. In diesem Abschnitt werden wir uns mit dem Abfragen von Daten beschäftigen. Da dieser Vorgang in Tcl denkbar einfach ist, beginnen wir mit einem entsprechenden Beispiel:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect -conninfo "dbname=buch user=postgres"];

pg_select $conn "SELECT * FROM produkt" array {
        puts [format "%d %s %s" $array(id) $array(name) $array(preis)]
}

pg_disconnect $conn
```

Mithilfe des Befehles `pg_select` führen wir eine Abfrage aus. Für jede Zeile des Ergebnisses führen wir die Schleife jeweils einmal aus. Innerhalb dieser Schleife geben wir die Werte formatiert aus. Auf diese Weise können wir ein Listing erzeugen, das etwa so aussieht:

```
[hs@duron tcl]$ ./connect.tcl
1 Perlenkette 4633.20
2 Diamant 933.99
3 Golduhr 60321.00
```

Wenn Sie das Ergebnis nicht einfach nur ausgeben wollen, müssen Sie etwas anders vorgehen. Das nächste Beispiel zeigt, wie Sie mit Metainformationen arbeiten können:

```
#!/usr/local/postgresql/bin/pgtclsh

set conn [pg_connect -conninfo "dbname=buch user=postgres"];
set result [pg_exec $conn "SELECT name FROM produkt"]

set ntuples [pg_result $result -numTuples]
```

```
for     {set i 0} {$i < $ntuples} {incr i} {
        puts "$i: [lindex [pg_result $result -getTuple $i] 0]"
}

pg_disconnect $conn
```

In diesem Fall geben wir eine Spalte inklusive Index aus:

```
[hs@duron tcl]$ ./connect.tcl
0: Perlenkette
1: Diamant
2: Golduhr
```

Für diese Operation haben wir die Funktion `pg_result` verwendet, um die Zahl der Zeilen zu berechnen. In diesem Beispiel benötigen wir nur die Zeilen, aber wenn wir mehr Informationen brauchen, können wir auch weitere Komponenten der Funktion `pg_result` verwenden. Im Folgenden sei eine Liste der wichtigsten Möglichkeiten zusammengefasst:

`-status`:
: Gibt den Status zurück.

`-error`:
: Enthält im Falle eines Fehlers eine Fehlermeldung.

`-conn`:
: Enthält die Verbindung, zu der das Ergebnis gehört.

`-oid`:
: Die OID eines neuen Datensatzes.

`-numTuples`:
: Die Anzahl der Datensätze im Ergebnis.

`-numAttrs`:
: Die Anzahl der Spalten.

`-assign arrayName`:
: Zuweisen des Ergebnisses auf ein Array.

`-getTuple tupleNumber`:
: Holt einen Datensatz.

`-tupleArray tupleNumber arrayName`:
: Enthält ein Feld von Tuples.

`-attributes`:
: Enthält eine Liste der Spalten.

`-lAttributes`:
Gibt eine Liste von Unterlisten zurück.

`-clear`:
Löscht das Ergebnis.

12.3 Trigger Procedures

Da Tcl eine mächtige, umfangreiche Sprache ist, wird Sie sehr oft zum Implementieren von Triggern herangezogen. Durch die große Flexibilität und die Macht der Sprache ist Tcl ein wichtiger Bestandteil der Datenbank und kann für komplexe Aufgaben eingesetzt werden.

Bei Triggern ist wichtig zu bemerken, dass es innerhalb der Funktion bereits einige vordefinierte Variablen gibt. Das folgende Listing enthält eine Liste dieser Variablen:

`TG_name`	Enthält den Namen des aktuellen Triggers.
`TG_relid`	Object ID der Tabelle, für die der Trigger ausgeführt wird.
`TG_relatts`	Enthält alle relevanten Einträge in der Systemtabelle `pg_attribute`.
`TG_when`	Enthält `BEFORE` oder `AFTER`. Der Wert ist abhängig vom Ausführungszeitpunkt des Triggers.
`TG_level`	Enthält derzeit immer `ROW`, da es noch keine Statement Level Trigger gibt.
`TG_op`	Enthält `INSERT`, `UPDATE` oder `DELETE`.
`NEW`	Im Falle von `INSERT` oder `UPDATE` enthält `NEW` die neuen Werte der einzelnen Spalten.
`OLD`	Im Falle von `UPDATE` oder `DELETE` enthält `OLD` die alten Werte der einzelnen Spalten.

Werfen wir abschließend einen Blick auf einen Trigger, der alle Informationen sich selbst ausgibt:

```
1 CREATE OR REPLACE FUNCTION insert_trig () RETURNS opaque AS
2 '
3         elog "NOTICE" [concat "TG_name: " $TG_name]
4         elog "NOTICE" [concat "TG_relid: " $TG_relid]
5         elog "NOTICE" [concat "TG_relatts: " $TG_relatts]
6         elog "NOTICE" [concat "TG_when: " $TG_when]
7         elog "NOTICE" [concat "TG_level: " $TG_level]
8         elog "NOTICE" [concat "TG_op: " $TG_op]
```

```
9           elog "NOTICE" [concat "NEW: " [array get NEW]]
10
11          return [array get NEW]
12  ' LANGUAGE 'pltcl';
13
14  CREATE TRIGGER trig_insert_produkt BEFORE INSERT ON produkt
15          FOR EACH ROW EXECUTE PROCEDURE insert_trig();
```

Wir verwenden die `elog`-Funktion, um einen Hinweis auszugeben. Als Rückgabewert haben wir `opaque` angegeben und retournieren das gesamte Array namens `NEW` an PostgreSQL.

Der Trigger wird immer dann aufgerufen, wenn ein Datensatz in `produkt` eingefügt wird. Wenn wir den Trigger in die Datenbank eingefügt haben, können wir uns seine Auswirkungen ansehen:

```
1  buch=# INSERT INTO produkt(name, preis) VALUES ('Silberschmuck',
2  '4325.32');
3  NOTICE:  TG_name: trig_insert_produkt
4  NOTICE:  TG_relid: 405566
5  NOTICE:  TG_relatts: {} id name preis
6  NOTICE:  TG_when: BEFORE
7  NOTICE:  TG_level: ROW
8  NOTICE:  TG_op: INSERT
9  NOTICE:  NEW: name Silberschmuck preis 4325.32
10 INSERT 429944 1
```

Alle relevanten Informationen werden umgehend ausgegeben.

Kapitel 13

Java und PostgreSQL im Team

In den letzten Jahren erfreut sich Java immer größer werdender Beliebtheit. Zahlreiche Anwendungen sind mittlerweile verfügbar, die in vielen Bereichen gewisse Vorteile gegenüber anderen Sprachen haben.

In diesem Abschnitt wollen wir beschreiben, wie Sie PostgreSQL und Java im Team verwenden können. Wir werden versuchen, zu zeigen, welche Möglichkeiten es gibt und wie effizient mit Java gearbeitet werden kann.

13.1 Grundlagen

Im Prinzip gibt es zwei Möglichkeiten, PostgreSQL mit Java zu verbinden: ODBC ist eine standardisierte Schnittstelle, die vor allem bei Windows sehr verbreitet ist. Die zweite Möglichkeit ist, JDBC mit einem nativen Treiber zu verwenden. Dieser Methode ist eindeutig der Vorzug zu geben, weil ODBC-Schnittstellen eher als instabil gelten und im Vergleich zu einem nativen Treiber zweite Wahl sind.

Um PostgreSQL in Kombination mit Java zu verwenden, müssen Sie Java mitkompilieren. Das kann mithilfe des Flags `–with–java` beim Aufruf von `configure` leicht bewerkstelligt werden. Sofern Sie `Ant` korrekt auf Ihrem System installiert haben, wird die Unterstützung für Java in der Regel korrekt funktionieren.

Um zu sehen, ob Ihre Installation von PostgreSQL Java unterstützt, suchen Sie am besten nach der Datei `postgresql.jar`:

```
[hs@duron hs]$ locate postgresql.jar
/usr/local/postgresql/share/java/postgresql.jar
/usr/src/postgresql–7.2.1/src/interfaces/jdbc/jars/postgresql.jar
```

Auf unserem System ist Java installiert und wir können loslegen.

13.1.1 Verbinden zur Datenbank

In diesem Abschnitt werden Sie lernen, wie Sie eine Verbindung zu PostgreSQL aufbauen können.

Bevor wir uns an die Arbeit machen, müssen wir noch das Archiv mit den PostgreSQL-Objekten kopieren. Auf diese Weise können wir direkt auf die Datenbank zugreifen:

```
cp /usr/local/postgresql/share/java/postgresql.jar
/usr/local/j2sdk1.3.1/lib/ext/
```

Nachdem wir nun alles vorbereitet haben, können wir uns daran machen, ein Programm zu schreiben, das sich zur Datenbank verbindet und den Fehler bei Bedarf abfängt. Als erstes File implementieren wir ein Makefile, das wir zum Kompilieren der Sourcen und zum Starten der Binaries verwenden:

```
JAVAC=  /usr/local/j2sdk1.3.1/bin/javac
JAVA=   /usr/local/j2sdk1.3.1/jre/bin/java

IN=     connect.java
OUT=    connect

dummy   :       $(IN)
        $(JAVAC) $(IN)
        $(JAVA) $(OUT)
```

Aus Gründen der Einfachheit sind der Befehl zum Kompilieren des Codes und der Befehl zum Starten der Anwendung im selben Block.

Sehen wir uns nun die Datei `connect.java` an:

```
import java.sql.*;

class connect
{
   static void main (String[] args)
   {
      try
      {
         // Verbinden zur Datenbank
         String str = "jdbc:postgresql:buch";
         Class.forName("org.postgresql.Driver");
         Connection dbhandle = DriverManager.getConnection(str,
            "postgres", "");
         System.out.println("Verbindung aufrecht :)");
         dbhandle.close();
      }
      catch (java.lang.Exception ex)
      {
         System.out.println("Kann nicht verbinden:" + ex);
      };
   }
}
```

Der Code der Anwendung ist einfach und leicht zu verstehen. In die ersten Zeile werden einige Objekte einbezogen, die wir für die weitere Verarbeitung benötigen.

Im nächsten Schritt wird ein Objekt definiert. Hier wollen wir dieses Objekt `connect` nennen, da es zum Erstellen einer Verbindung zur Datenbank dient. Innerhalb des `try`-Blockes versuchen wir uns zu PostgreSQL zu verbinden. Zu diesem Zwecke legen wir einen Connect String an, der Java sagt, wie wir uns verbinden wollen. Wir wollen uns über den PostgreSQL-Driver für JDBC zur Datenbank `buch` verbinden.

Nach diesen Anweisungen wird die eigentliche Verbindung zur Datenbank erstellt. Die Methode `getConnection` sorgt dafür, dass eine Verbindung erstellt wird. Als Parameter übergeben wir den Connect String, den Benutzer und das Passwort. In unserem Falle kann das Passwort leer bleiben, da der Superuser keines benötigt (alle lokalen Verbindungen sind auf `trust` gesetzt — siehe `pg_hba.conf`).

Nachdem wir eine kurze Meldung ausgegeben haben, wird die Verbindung zur Datenbank auch schon wieder geschlossen.

Sofern beim Herstellen der Verbindung ein Fehler aufgetreten ist, so wird dieser von `catch` behandelt. Bei uns werden eine Meldung sowie der Grund des Fehles ausgegeben.

13.1.2 Einfache Statements ausführen

Nach diesem ersten Einstieg in Java können wir uns bereits einem etwas komplexeren Beispiel zuwenden. Ziel des Beispieles ist es, Ihnen zu zeigen, wie ein einfaches SQL-Statement ausgeführt werden kann. Auf diese Weise sehen Sie, wie das Ganze prinzipiell vor sich geht. Werfen wir also einen Blick auf den Code:

```
1  import java.sql.*;
2
3  class connect
4  {
5     static Connection dbhandle;
6
7     static void main (String[] args)
8     {
9        try
10       {
11          String str = "jdbc:postgresql:buch";
12          Class.forName("org.postgresql.Driver");
13          dbhandle = DriverManager.getConnection(str, "postgres", "");
14
15          // fuehrt ein SQL Statement aus
16          System.out.println("Versuche: " + args[0]);
17          query(args[0]);
18          dbhandle.close();
19       }
20       catch (java.lang.Exception ex)
21       {
22          System.out.println("Kann nicht verbinden:" + ex);
23       }
24    };
25
26    public static boolean query (String input)
```

```
27     {
28        try
29        {
30           Statement statement = dbhandle.createStatement();
31           ResultSet result = statement.executeQuery(input);
32        }
33        catch (java.lang.Exception ex)
34        {
35           System.out.println("Kann " + input +
36              "nicht ausfuehren:" + ex);
37           return false;
38        };
39
40        System.out.println("Das Statement hat funktioniert");
41        return true;
42     }
43
44  }
```

Hierbei geht es um zwei Funktionen, den Konstruktur und die Methode `query`, die als Eingabeparameter eine Abfrage erwartet. Diese Abfrage wird via Standard Input an das Programm übergeben. Nach dem Anlegen der Verbindung im Konstruktur wird die auszuführende Query auf dem Bildschirm dargestellt. Danach rufen wir die `query`-Methode auf. Die Funktionsweise dieser Methode ist von großer Bedeutung und es ist daher wichtig, dass Sie die grundlegenden Prinzipien verstehen.

Zuerst wird ein Statement angelegt. Diese Hülle wird im Folgenden noch benötigt. Das Ausführen der eigentlichen Abfrage erfolgt durch das Aufrufen der Methode `executeQuery`. Dieser Aufruf erzeugt ein Objekt der Klasse `ResultSet`.

Nach dem Kompilieren des Codes können wir versuchen, das Programm auszuführen:

```
1 [hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
2 "SELECT 1+1"
3 Versuche: SELECT 1+1
4 Das Statement hat funktioniert
```

Wenn Sie alles richtig gemacht haben, wird die Abfrage korrekt ausgeführt. Sofern Sie versuchen, eine inkorrekte Abfrage auszuführen, wird der Fehler korrekt abgefangen und Java eine Fehlermeldung ausgeben:

```
1 [hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect " sdfdsf "
2 Versuche:  sdfdsf
3 Kann  sdfdsf nicht ausfuehren:java.sql.SQLException: ERROR:  parser:
4 parse error at or near "sdfdsf"
```

13.1.3 Abfragen von Daten

Bisher haben wir uns mit dem Ausführen einfacher SQL-Befehle beschäftigt, ohne jedoch das Ergebnis einer solchen Abfrage abzufangen und am Bildschirm auszugeben. In diesem Abschnitt wollen wir uns eingehend mit der Datenabfrage und deren Möglichkeiten beschäftigen.

Ein einfaches Beispiel

Wie die meisten anderen Programmiersprachen auch bietet Java einfache Möglichkeiten, Daten von einer PostgreSQL-Datenbank abzufragen. Um Sie nicht unnötig mit Theorie zu quälen, wollen wir anhand eines Beispieles zeigen, wie Daten abgefragt werden können:

```
1  import java.sql.*;
2
3  class connect
4  {
5     static String input = "SELECT 1+1 UNION SELECT 2+2";
6
7     static void main (String[] args)
8     {
9        try
10       {
11          String str = "jdbc:postgresql:buch";
12          Class.forName("org.postgresql.Driver");
13          Connection dbhandle = DriverManager.getConnection(str,
14             "postgres", "");
15
16          // Ausführen der Abfrage
17          Statement statement = dbhandle.createStatement();
18          ResultSet result = statement.executeQuery(input);
19
20          // Ausgeben des Ergebnisses
21          while   ( result.next() )
22          {
23             System.out.println("Ergebnis: " + result.getInt(1) );
24          }
25
26          dbhandle.close();
27       }
28       catch (java.lang.Exception ex)
29       {
30          System.out.println("Fehler im Programm\n" + ex );
31       }
32    };
33 }
```

Nach dem Einbinden der SQL-Bibliotheken legen wir einen String an, der einen SQL-Befehl enthält. Dieser Befehl gibt genau zwei Werte zurück, die wir abfragen wollen. In der Hauptbibliothek stellen wir eine Verbindung zur Datenbank her und führen den SQL-Code aus, wie wir das bereits im letzten Beispiel gemacht haben.

Nachdem wir das Statement an den Server geschickt haben, geht es nun darum, das Ergebnis effizient abzufragen. Aus diesem Grund haben wir eine Schleife implementiert, die das Ergebnis zeilenweise verarbeiten soll. Mithilfe der Methode `next` kann der Zeiger auf die Daten jeweils um eine Zeile weitergesetzt werden. Solange Daten gefunden werden, werden diese ganz einfach auf dem Bildschirm ausgegeben.

Wichtig ist, dass die Methode `next` nur zum Weitersetzen der aktuellen Position dient — die Abfrage der einzelnen Felder ist eine eigene Angelegenheit. Jetzt enthält das Ergebnis unserer Abfrage genau eine Spalte und zwei Zeilen. Um den Wert der ersten Spalte abzufragen, verwenden wir die Funktion getInt. Da wir den Wert der ersten Spalte abfragen wollen, rufen wir die Funktion mit dem Parameter 1 auf. Bitte behalten Sie im Kopf, dass die Indizierung nicht mit 0, sondern mit 1 beginnt; das ist ein wichtiger Punkt, der oft übersehen oder überlesen wird. In unserem Beispiel fragen wir einen Integerwert ab, aus diesem Grund müssen wir die Funktion `getInt` verwenden. Wollten wir einen Timestamp oder dergleichen abfragen, müssten wir auf eine andere Funktion zurückgreifen.

Nach der Ausgabe des Ergebnisses schließen wir die Verbindung zur Datenbank.

Modifikationen

Wenn Sie Daten in einer Datenbank modifizieren, kann es interessant sein zu wissen, wie viele Datensätze von einer Änderung betroffen sind. Das ist ganz besonders dann wichtig, wenn von dieser Information die weitere Vorgehensweise Ihrer Software abhängt. Wie viele andere Programmiersprachen bietet Java auch dafür eine gute und einfache Lösung.

Das nächste Listing enthält einen kleinen Datenbestand, den wir zum Modifizieren der Daten verwenden werden:

```
1  CREATE TABLE einkommen (
2          name            text,
3          einkommen       numeric(9,2)
4  );
5
6  COPY einkommen FROM stdin USING DELIMITERS ';';
7  Hugo;34000
8  Paul;60000
9  Berni;21000
10 \.
```

Sofern Sie die Daten in eine PostgreSQL-Datenbank eingefügt haben, können Sie beginnen, ein Programm zu schreiben, das den Datenbestand modifiziert. Damit Sie die gesamte Lösung des Problemes gleich auf einen Blick sehen, haben wir uns entschlossen, das gesamte Programm aufzulisten:

```
import java.sql.*;

class connect
{
   static void main (String[] args)
   {
      try
      {
         String str = "jdbc:postgresql:buch";
         Class.forName("org.postgresql.Driver");
         Connection dbhandle = DriverManager.getConnection(str,
            "postgres", "");

         // Ausführen der Abfrage
         Statement statement = dbhandle.createStatement();
         String input;
         input = "UPDATE einkommen SET einkommen = einkommen + 1000";
         int zeilen = statement.executeUpdate(input);
         System.out.println("Betroffene Personen: " + zeilen );
         dbhandle.close();
      }
      catch (java.lang.Exception ex)
      {
         System.out.println("Fehler im Programm\n" + ex );
      }
   };
}
```

Dabei verwenden wir die Methode `executeUpdate`, um die Modifikation durchzuführen. Der Rückgabewert der Methode ist vom Typ Integer und enthält die Zahl der von `UPDATE` modifizierten Zeilen. Dieser Wert wird anschließend ausgegeben.

Nach dem Kompilieren der Software, können Sie das Programm ausführen:

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
Betroffene Personen: 3
```

Da wir allen Personen in der Datenbank eine Gehaltserhöhung verpasst haben, sind drei Personen von der Änderung betroffen; das Abfragen der Tabelle liefert den Beweis:

```
buch=# SELECT * FROM einkommen;
 name  | einkommen
-------+-----------
 Hugo  |  35000.00
 Paul  |  61000.00
 Berni |  22000.00
(3 rows)
```

Alle Werte sind um EUR 1.000 höher als zuvor.

Datentypabhängige Ergebnisabfrage

Wie wir bereits erwähnt haben, ist die Abfrage der einzelnen Felder sehr stark vom jeweiligen Datentyp abhängig. Im Prinzip kann jedes Feld auch als Text abgefragt werden, aber sinnvollerweise benutzt man auch auf Applikationsebene einen zur Datenbank äquivalenten Datentyp.

Java unterstützt eine Reihe von Funktionen, um den korrekten Datentyp direkt abzufragen. Die folgende Aufstellung enthält eine Liste aller relevanten Informationen:

- `String getString(int columnIndex)`
- `boolean getBoolean(int columnIndex)`
- `byte getByte(int columnIndex)`
- `short getShort(int columnIndex)`
- `int getInt(int columnIndex)`
- `long getLong(int columnIndex)`
- `float getFloat(int columnIndex)`
- `double getDouble(int columnIndex)`
- `java.math.BigDecimal getBigDecimal(int columnIndex)`
- `byte[] getBytes(int columnIndex)`
- `java.sql.Date getDate(int columnIndex)`
- `java.sql.Time getTime(int columnIndex)`
- `java.sql.Timestamp getTimestamp(int columnIndex)`
- `java.io.Inputstream getAsciiStream(int columnIndex)`
- `java.io.Inputstream getUnicodeStream(int columnIndex)`

- `java.io.Inputstream getBinaryStream(int columnIndex)`
- `Object getObject(int columnIndex)`

Anstatt des Spaltenindexes können Sie auch den Namen der Spalte als String übergeben.

Wenn Sie mit geometrischen Datentypen oder mit Datentypen arbeiten, die in obiger Liste nicht aufgeführt sind, können Sie die Daten entweder als Object oder als String extrahieren. Wenn Sie die entsprechenden Konstruktoren selbst implementieren, ist es ohne Probleme möglich, auch solche Datentypen direkt anzusprechen.

Metainformationen

Wenn Sie eine Abfrage durchführen, entsteht automatisch jede Menge Metainformationen, die es gilt auszuwerten, um die Funktionsweise Ihrer Programme zu steuern. In vielen Fällen benötigen Sie diese Informationen auch, um Tabellenköpfe oder dergleichen aufzubereiten.

In diesem Abschnitt wollen wir uns genauer mit Metainformationen beschäftigen. Sie werden schnell sehen, wie einfach es ist, alle notwendigen Daten aus dem Ergebnis einer Abfrage zu extrahieren.

Beginnen wir mit einem Beispiel, das Ihnen zeigt, wie die wichtigsten Informationen extrahiert werden können:

```
1  import java.sql.*;
2
3  class connect
4  {
5     static void main (String[] args)
6     {
7        try
8        {
9           String fname;
10          String str = "jdbc:postgresql:buch";
11
12          Class.forName("org.postgresql.Driver");
13          Connection dbhandle = DriverManager.getConnection(str,
14             "postgres", "");
15
16          // Ausführen der Abfrage
17          Statement statement = dbhandle.createStatement();
18          String input = "SELECT name, einkommen FROM einkommen";
19          ResultSet result = statement.executeQuery(input);
20
21          // Abfragen der MetaDaten
```

```
22          ResultSetMetaData md =  result.getMetaData();
23
24          // Zahl der Spalten
25          System.out.println("Spalten: " + md.getColumnCount() );
26
27          // Zahl der Spalten
28          for    (int i = 1; i <= md.getColumnCount(); i++)
29          {
30             System.out.println(i + ": Währung -  " + md.isCurrency(i) );
31             fname = md.getColumnLabel(i);
32             System.out.println(i + ": Name - " + fname );
33             System.out.println(i + ": Type - " +
34                 md.getColumnTypeName(i) );
35             System.out.println(i + ": Idx - " +
36                 result.findColumn(fname) );
37             System.out.println();
38          }
39
40          dbhandle.close();
41       }
42       catch (java.lang.Exception ex)
43       {
44          System.out.println("Fehler im Programm\n" + ex );
45       }
46    };
47 }
```

Nachdem wir eine Verbindung zur Datenbank hergestellt und eine Variable namens `fname` angelegt haben, senden wir eine Abfrage an die Datenbank. Danach legen wir ein Objekt vom Typ `ResultSetMetaData` an. Dieses Objekt wird zurückgegeben, wenn wir die Methode getMetaData auf das Objekt `ResultSet` anwenden.

Die erste Information, die wir aus den Metadaten extrahieren, ist die Zahl der Spalten im Ergebnis. Es handelt sich hierbei um einen Integer-Wert, der aber gleich mittels `println` ausgegeben wird.

Danach gehen wir jede Spalte mithilfe einer einfachen Schleife durch. Zu jeder Spalte versuchen wir ein wenig Information zu extrahieren und diese auf dem Bildschirm auszugeben. In der ersten Zeile prüfen wir nach, ob ein bestimmtes Feld eine Währung ist oder nicht. Nach dieser Berechnung extrahieren wir den Feldnamen der aktuellen Spalte und speichern ihn in einer Variable. Weiter wird noch der Datentyp des Feldes abgefragt. Abschließend geben wir den Index zum gerade erst berechneten Feldnamen aus.

Sofern Sie keinen Fehler gemacht haben, können Sie das Programm nun kompilieren und ausführen:

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
Spalten: 2
1: Währung -  false
1: Name - name
1: Type - text
1: Idx - 1

2: Währung -  false
2: Name - einkommen
2: Type - numeric
2: Idx - 2
```

Das Ergebnis ist nicht weiter überraschend. Alle Werte werden in der korrekten Reihenfolge ausgegeben und enthalten keine unerwarteten Dinge.

Metadaten enthalten wichtige Informationen, die Ihnen viel Ärger ersparen können. Wie Sie erkennen ist die Abfrage dieser Daten trivial und bedarf nur weniger Zeilen Code.

13.2 Exception Handling

Wie in allen Programmiersprachen ist auch in Java das Abfragen von Fehlern von großer Bedeutung. Egal welche Art von Operationen Sie durchführen, Fehlerabfrage und Fehlerlokalisierung werden Sie immer begleiten. In diesem Abschnitt wollen wir uns mit den Möglichkeiten von JDBC auseinander setzen.

Das wichtigste Objekt bei der Bearbeitung von Fehlern ist das Objekt `SQLException`. Immer, wenn es zu einem SQL-Fehler kommt, können Sie einen auftretenden Fehler mithilfe dieses Objektes analysieren und ausgeben.

Ein weiteres Objekt, das sehr häufig eingesetzt wird, ist vom Typ `ClassNotFoundException` und tritt dann auf, wenn eine benötigte Klasse nicht vorhanden ist. Diese Klasse ist wichtig, da Sie nicht sicher sein können, dass die für PostgreSQL benötigten Objekte und Funktionen auch tatsächlich vorhanden sind. Um mit Warnungen zu arbeiten, können Sie auf `SQLWarning` zurückgreifen.

13.2.1 Fehler abfangen

Das folgende Beispiel enthält Code, mit dem verschiedenste Fehler abgefangen werden können:

```
import java.sql.*;

class connect
{
```

```
5     static Connection dbhandle;
6
7     static String fname;
8     static String str = "jdbc:postgresql:buch";
9     static Statement statement;
10
11    static void main (String[] args)
12    {
13       try
14       {
15          Class.forName("org.postgresql.Driver");
16       }
17       catch (ClassNotFoundException notfound)
18       {
19          System.out.println("Klasse nicht gefunden");
20       }
21
22       try
23       {
24          dbhandle = DriverManager.getConnection(str, "postgres", "");
25       }
26       catch (SQLException connfehler )
27       {
28          System.out.println("Fehler in DB Verbindung\n" + connfehler );
29       }
30
31       // Starten einer Abfrage ...
32       query("SELECT fehler");
33
34    }
35
36    static void query (String input)
37    {
38       try
39       {
40          // Ausführen der Abfrage
41          statement = dbhandle.createStatement();
42          ResultSet result = statement.executeQuery(input);
43       }
44       catch (SQLException two)
45       {
46           System.out.println("Fehler in SQL Statement:\n" + two );
47       }
48    }
49 }
```

Zuerst fragen wir ab, ob die benötigten Klassen vorhanden sind. Anschließend versuchen wir einen eventuell auftretenden SQL-Fehler zu behandeln. Abschließend wird eine Abfrage ausgeführt, die ebenfalls wieder einen SQL-Fehler abfragt.

Wenn wir das Programm starten, obwohl die Datenbank nicht läuft, werden wir einen Fehler erhalten, wie er im nächsten Listing zu sehen ist:

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
Fehler in DB Verbindung
Connection refused. Check that the hostname and port is correct, and
that the postmaster is running
with the -i flag, which enables TCP/IP networking.
Exception in thread "main" java.lang.NullPointerException
        at connect.query(connect.java:41)
        at connect.main(connect.java:32)
```

In diesem Beispiel haben wir den Fehler sauber abgefangen.

Wenn die Datenbank aktiv ist, wird die Fehlermeldung wie folgt aussehen:

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
Fehler in SQL Statement:
java.sql.SQLException: ERROR:  Attribute 'fehler' not found
```

Wieder haben wir den Fehler sauber abgefangen und Java sagt uns auch, welcher Fehler aufgetreten ist.

Wenn Sie Fehler aus dem `SQLException`-Objekt extrahieren wollen, können Sie auch so vorgehen:

```
catch(SQLException ex) i
{
   System.err.println("SQLException: " + ex.getMessage());
}
```

13.2.2 Warnungen abfragen

Zusätzlich zu Ausnahmen kennt die JDBC-Schnittstelle auch Warnungen, die in vielen Anwendungen mit großem Erfolg zum Einsatz kommen. Um Ihnen das Arbeiten mit Warnungen näher zu bringen, bringen wir das folgende Beispiel:

```
import java.sql.*;

class connect
{
   static Connection dbhandle;

```

```
7     static String fname;
8     static String str = "jdbc:postgresql:buch";
9     static Statement statement;
10
11    static void main (String[] args)
12    {
13       try
14       {
15          Class.forName("org.postgresql.Driver");
16          dbhandle = DriverManager.getConnection(str, "postgres", "");
17          warnungen (dbhandle.getWarnings ());
18       }
19       catch (Exception fehler)
20       {
21          System.out.println("Fehler");
22       }
23    }
24
25    private static boolean warnungen (SQLWarning warnung)
26    {
27       boolean flag = false;
28       if (warnung != null)
29       {
30          flag = true;
31          while (warnung != null)
32          {
33             System.out.println ("SQLState: " + warnung.getSQLState ());
34             System.out.println ("Message: " + warnung.getMessage ());
35             System.out.println ("Vendor: " + warnung.getErrorCode ());
36             System.out.println ("");
37             warnung = warnung.getNextWarning ();
38          }
39       }
40       return flag;
41    }
42 };
```

Die Methode `getWarnings` wird für das `Connection`-Objekt aufgrufen und das Ergebnis der Methode wird an `warnungen` übergeben. Unsere Methode gibt alle Warnungen auf Standard Output aus und gibt `true` zurück.

Aus dem Beispiel geht hervor, wie einfach es ist, mit Warnungen zu arbeiten.

13.2.3 Stack Tracing

Bei einem Fehler ist es auch möglich, gleich den ganzen Stack ausgeben zu lassen. Zu diesem Zwecke kann eine Methode namens `printStackTrace` angewendet werden, die im folgenden Beispiel vorgeführt wird:

```
1  import java.sql.*;
2
3  class connect
4  {
5     static Connection dbhandle;
6
7     static String fname;
8     static String str = "jdbc:postgresql:buch";
9     static Statement statement;
10
11    static void main (String[] args)
12    {
13       try
14       {
15          Class.forName("org.postgresql.Driver");
16          dbhandle = DriverManager.getConnection(str, "postgres2", "");
17       }
18       catch (Exception fehler)
19       {
20          System.out.println("Fehler!!!\n");
21          fehler.printStackTrace();
22       }
23    }
24 };
```

Wieder wird der Fehler sauber abgefangen. Dabei generiert die Fehlerroutine aber wesentlich mehr Code. In unserem Fall tritt der Fehler auf, weil es den User `postgres2` in der Datenbank nicht gibt:

```
1  [hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
2  Fehler!!!
3
4  Something unusual has occured to cause the driver to fail. Please
5  report this exception: Exception:
6  java.sql.SQLException: FATAL 1:  user "postgres2" does not exist
7
8  Stack Trace:
9
10 java.sql.SQLException: FATAL 1:  user "postgres2" does not exist
11
12         at org.postgresql.Connection.openConnection(Connection.java:274)
13         at org.postgresql.Driver.connect(Driver.java:149)
```

```
        at java.sql.DriverManager.getConnection(DriverManager.java:517)
        at java.sql.DriverManager.getConnection(DriverManager.java:177)
        at connect.main(connect.java:16)
End of Stack Trace

        at org.postgresql.Driver.connect(Driver.java:166)
        at java.sql.DriverManager.getConnection(DriverManager.java:517)
        at java.sql.DriverManager.getConnection(DriverManager.java:177)
        at connect.main(connect.java:16)
```

Im Listing können Sie sehen, dass Java gleich eine ganze Menge an Infos ausgibt, die für die weitere Analyse eines Fehlers von Interesse sein können. Bei schwer auffindbaren Fehlern ist es auf alle Fälle ratsam, sich den Stack näher anzusehen.

13.3 Vorbereitete Abfragen

Wenn es darum geht, mehr als nur einige wenige Datensätze einzufügen, kann es aus Performancegründen sinnvoll sein, auf `INSERT`-Statements zu verzichten, da diese doch beträchtlichen Overhead erzeugen.

Wie in allen anderen Programmiersprachen ist es auch in Java ohne Probleme möglich, auf effizientere Methoden zurückzugreifen.

Da der `COPY`-Befehl wie in anderen Programmiersprachen verwendet werden kann, werden wir uns in diesem Abschnitt mit vorbereiteten Abfragen beschäftigen.

Wenn Sie immer wiederkehrende Befehle ausführen wollen, die sich nur durch Ihre Parameter unterscheiden, kann es sinnvoll sein, derartige Statements bereits vorbereitet im Speicher zu halten. Das hat den Vorteil, dass die Last am Server sinkt und die Verarbeitung effizienter vor sich geht.

Hier haben wir ein Beispiel, in dem es darum geht, Daten in eine Datenbank einzufügen:

```
import java.sql.*;

class connect
{
   static Connection dbhandle;
   static String str = "jdbc:postgresql:buch";
   static Statement statement;
   static PreparedStatement pstat = null;

   static void main (String[] args)
   {
      try
```

```
13        {
14           Class.forName("org.postgresql.Driver");
15           dbhandle = DriverManager.getConnection(str, "postgres", "");
16           statement = dbhandle.createStatement();
17           statement.executeUpdate(
18              "CREATE TABLE data (id int4, data text)");
19
20           pstat = dbhandle.prepareStatement(
21                  "INSERT INTO data VALUES (?, ?)");
22
23           // erster Datensatz
24           pstat.setInt(1, 17);
25           pstat.setString(2, "Bohnen");
26           pstat.executeUpdate();
27
28           // zweiter Datensatz
29           pstat.setInt(1, 18);
30           pstat.setString(2, "Erbsen");
31           pstat.executeUpdate();
32        }
33        catch (Exception fehler)
34        {
35           System.out.println("Fehler!!!\n");
36           fehler.printStackTrace();
37        }
38     }
39  };
```

Am Beginn des Programms werden einige Variablen angelegt, die wir später benötigen werden. Besonders wichtig an diesen Beispiel ist die Variable vom Typ `PreparedStatement`. Diese Instanz werden wir benötigen, um die Hülle der Abfrage zu speichern.

Nachdem wir eine Verbindung zur Datenbank hergestellt haben, legen wir eine Tabelle an, um die einzufügenden Daten zu speichern. Da wir zum Einfügen der Daten mehr als nur einmal `INSERT` aufrufen werden, ist es sinnvoll, die Abfrage vorzubereiten, was wir auch machen. Die Fragezeichen in der vorzubereitenden Abfrage sind Platzhalter, die entsprechenden Werte werden anschließend eingesetzt. Für das Einsetzen der korrekten Daten verwenden wir die Befehle `setInt` und `setString`. Um die Abfrage nun durchzuführen, werden wir die Methode `executeUpdate` anwenden.

Jetzt führen wir denselben Typ von Abfrage mit jeweils unterschiedlichen Werten aus — das `INSERT`-Statement selbst ist jedoch nur einmal im Code enthalten.

Sehen wir uns an, was passiert, wenn wir das Programm testen:

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
[hs@duron java]$ psql -c "SELECT * FROM data" buch
 id |  data
----+--------
 17 | Bohnen
 18 | Erbsen
(2 rows)
```

Die Daten werden korrekt in die Datenbank eingefügt und stehen somit für die weitere Nutzung zur Verfügung.

13.4 Transaktionen

Bei allen professionellen IT-Projekten im Datenbankbereich spielen Transaktionen eine wichtige Rolle. Wir haben in diesem Buch bereits ausführlich behandelt, wie Transaktionen sinnvoll eingesetzt werden können. An diesem Abschnitt wollen wir uns mit JDBC und Transaktionen auseinander setzen.

13.4.1 `AutoCommit` & Co

PostgreSQL kann in zwei verschiedenen Modi angesprochen werden. Standardmäßig ist der Modus `AutoCommit` eingeschaltet, was bedeutet, dass jedes Statement in einer eigenen Transaktion ausgeführt wird, sofern das Transaktionsverhalten nicht explizit beeinflusst wird. Wenn dieses Verhalten nicht Ihren Anforderungen entspricht, können Sie es jederzeit umstellen.

Im Beispiel zeigen wir, wie das gemacht werden kann:

```
import java.sql.*;

class connect
{
   static Connection dbhandle;
   static String str = "jdbc:postgresql:buch";
   static Statement statement = null;

   static void main (String[] args)
   {
      try
      {
         Class.forName("org.postgresql.Driver");
         dbhandle = DriverManager.getConnection(str, "postgres", "");
         dbhandle.setAutoCommit(false);

         statement = dbhandle.createStatement();
```

```
18            statement.executeUpdate(
19               "INSERT INTO data VALUES ('21', 'Oliven')");
20         }
21         catch (Exception fehler)
22         {
23            System.out.println("Fehler!!!\n");
24            fehler.printStackTrace();
25         }
26      }
27   };
```

Nach dem Erstellen der Verbindung zur Datenbank setzen wir `AutoCommit` auf `False`. Auf diese Weise werden Transaktionen nur mehr dann sichtbar, wenn Sie auch mit einem `Commit` abgeschlossen werden. Im obigen Programm ist zu sehen, dass Commit nicht explizit aufgerufen wird, aus diesem Grund wird das `INSERT` auch nie wirksam, weil PostgreSQL intern ein Rollback durchführt.

Bitte leeren Sie die Tabellen `data`, bevor Sie das Programm ausführen.

Nachdem Sie das Programm ausgeführt haben, werden Sie sehen, dass die Tabelle immer noch leer ist:

```
1 buch=# SELECT * FROM data;
2  id | data
3 ----+------
4 (0 rows)
```

Um die Transaktion abzuschließen, müssen Sie `Commit` explizit aufrufen:

```
1 dbhandle.commit();
```

Um ein Rollback durchzuführen, können Sie wie folgt arbeiten:

```
1 dbhandle.rollback();
```

13.4.2 Transaction Isolation Levels

Wie Sie in diesem Buch bereits an verschiedensten Stellen gesehen haben, schreibt der ANSI-Standard vier Transaction Isolation Levels vor. Derzeit unterstützt PostgreSQL zwei der vier vorgeschriebenen Levels vollständig. Oft kann es sinnvoll sein, das Level zur Laufzeit zu ändern beziehungsweise herauszufinden, welches Level gerade aktiv ist.

Das nächste Beispiel zeigt ein Programm, das das aktuelle Transaction Isolation Level ausliest und von `Read Committed` auf `Serializable` umstellt:

```
import java.sql.*;

class connect
{
   static Connection dbhandle;
   static String str = "jdbc:postgresql:buch";

   static void main (String[] args)
   {
      try
      {
         Class.forName("org.postgresql.Driver");
         dbhandle = DriverManager.getConnection(str, "postgres", "");

         // Auslesen des aktuellen Levels ...
         int level = dbhandle.getTransactionIsolation();
         System.out.println("aktuelles Level: " + level);

         // Serializable einschalten ...
         dbhandle.setTransactionIsolation(8);
         level = dbhandle.getTransactionIsolation();
         System.out.println("aktuelles Level: " + level);
      }
      catch (Exception fehler)
      {
         System.out.println("Fehler!!!\n");
         fehler.printStackTrace();
      }
   }
};
```

Im Prinzip benötigen Sie nur zwei Methoden, um alle Operationen durchführen zu können. `getTransactionIsolation()` liefert das aktuelle Level und `setTransactionIsolation(8)` setzt das Level auf 8. Das Isolierungslevel 2 bedeutet `Read Committed`, während 8 für `Serializable` steht.

Was passiert, wenn wir das Programm ausführen?

```
[hs@duron java]$ /usr/local/j2sdk1.3.1/jre/bin/java connect
aktuelles Level: 2
aktuelles Level: 8
```

Wie erwartet werden zwei Zeilen ausgegeben.

13.5 Binäre Objekte

Die Java-Schnittstelle für PostgreSQL bietet die Möglichkeit, direkt auf binäre Objekte zuzugreifen. Der Zugriff auf Objekte innerhalb der Datenbank geht sehr schnell vonstatten und funktioniert weitgehend zuverlässig.

PostgreSQL stellt zwei Schnittstellen zur Verarbeitung von BLOBs zur Verfügung (`bytea` und OID). Für beide Varianten gibt es eine Reihe von JDBC-Funktionen, auf die wir an dieser Stelle aber nicht eingehen wollen, da man auch problemlos über die normalen SQL-Funktionen gehen kann. Die JDBC-Funktionen haben keine signifikanten Vorteile, sondern bergen eher Risiken.

Kapitel 14

Delphi/Kylix

Borland Delphi hat sich in vielen Bereichen fest als objektorientierter Nachfolger von Turbo Pascal etabliert. Für viele Einsatzgebiete ist Delphi eine gelungene Alternative zu Sprachen wie C/C++. Vor allem für spezielle Datenbankanwendungen kann Delphi schnell für die Entwicklung von Windows-Anwendungen herangezogen werden. Die Delphi Enterprise Edition wird bereits mit einer Vielzahl von Datenbankschnittstellen geliefert, die für alle gängigen Datenbanken ausgelegt sind.

Vor kurzem hat Borland auch Kylix veröffentlicht, das es Benutzern von UNIX ermöglichen soll, Delphi Anwendungen zu entwickeln. In diesem Abschnitt wollen wir uns jedoch nur mit Windows beschäftigen.

Wir werden uns mit PostgreSQL und Delphi als Team beschäftigen. Sie werden sehen, welche Möglichkeiten sich bieten, und wir werden anhand eines Beispieles erörtern, wie einfache Programme implementiert werden können.

14.1 Theoretisches

PostgreSQL kann auf verschiedene Weisen angesprochen werden. In vielen Fällen wird die ODBC-Schnittstelle verwendet, die jedoch bei größeren Datenmengen vielfach zu Problemen führen kann.

Üblicherweise wird die Zoes Library verwendet, die von `www.zeoslib.org` gratis heruntergeladen werden kann.

Wenn Sie sich nativ zu PostgreSQL verbinden wollen, empfehlen wir den so genannten `pgExpress`-Treiber von Vita Voom Software (`www.vitavoom.com`). Dieser Treiber stellt eine gute Plattform zur Verfügung, auf der Sie zügig aufbauen können. Im Vergleich zur Zoes Library kann der `pgExpress`-Treiber aber wesentlich leichter installiert werden.

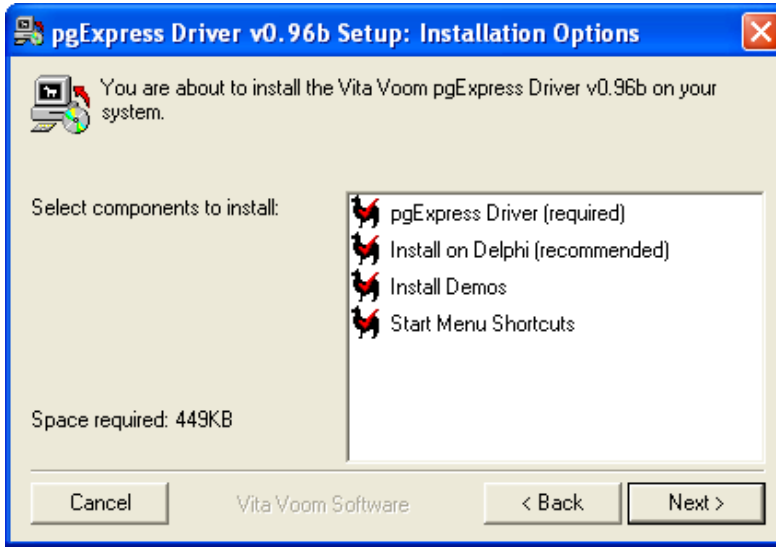
In diesem Kapitel werden wir uns ausschließlich mit dem `pgExpress`-Treiber beschäftigen.

14.2 pgExpress

Das System ist einfach zu verwenden.

14.2.1 Installation

Nach dem Download des Treibers können Sie das Paket bequem per Windows Installer installieren. In der Regel kommt es dabei zu keinen Problemen.



Sofern ein `ini`-File nicht gefunden werden kann, müssen in der Datei `dbxdrivers.ini` einige Zeilen hinzugefügt werden:

```
[Installed Drivers]
DB2=1
Interbase=1
MYSQL=1
Oracle=1
PostgreSQL=1
```

Fügen Sie einfach bei Bedarf die letzte Zeile des obigen Listings hinzu. Weiter müssen Sie noch den folgenden Teil ergänzen:

```
[PostgreSQL]
GetDriverFunc=getSQLDriverPOSTGRESQL
LibraryName=dbexppge.dll
VendorLib=LIBPQ.DLL
```

In weiterer Folge ist noch die Datei `dbxconnections.ini` zu editieren:

```
[PGEConnection]
BlobSize=32
HostName=host
Database=database_name
DriverName=PostgreSQL
Password=temp123
TransIsolation=ReadCommited
User_Name=steve
```

Dieser Teil definiert bereits die Standard-Verbindung zur Datenbank. Wenn Sie diese Einstellungen ändern wollen, können Sie das mithilfe der Entwicklungsumgebung tun.

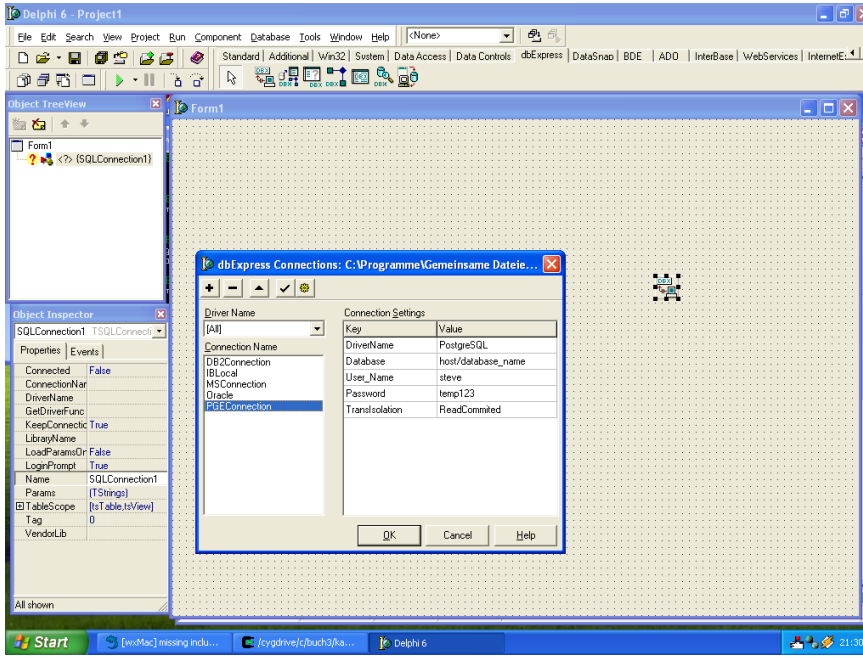
Bedenken Sie, diese Änderungen müssen Sie nur durchführen, wenn bei der Installation das `ini`-File nicht gefunden wird. Wenn Sie über eine nicht englische Windows-Version verfügen, kann dieser Fehler auftreten. Im Zuge der Erstellung des Manuskriptes zu diesem Buch hat der Autor des Treibers diesen Fehler aber bereits behoben — Vorsicht ist also bei älteren Treibern geboten.

14.2.2 Beispiele

Nach der Installation des Treibers folgt ein Beispiel, das zeigt, wie einfache Applikationen implementiert werden können. Wir werden uns dabei aber nicht in die Details des Treibers verbeißen. Ziel des Beispieles ist es, ein interaktives Abfrage-Tool zu bauen.

Nach dem Anlegen eines Projektes ziehen wir ein Objekt vom Typ `SQLConnection` im Menü `dbExpress` in das Form. Dadurch wird die Datenbankverbindung für dieses Form bereitgestellt Verfügung. Nach dem Hereinziehen müssen Sie lediglich die Parameter der Verbindung modifizieren.

Das Bild enthält eine Beschreibung des Vorganges:



Anschließend können Sie ein Objekt vom Typ `SQLClientDataSet` in das Form ziehen. In den Eigenschaften zu diesem Objekt müssen Sie dann die korrekte Datenbankverbindung auswählen. In unserem Falle heißt diese Verbindung `SQLConnection1`.

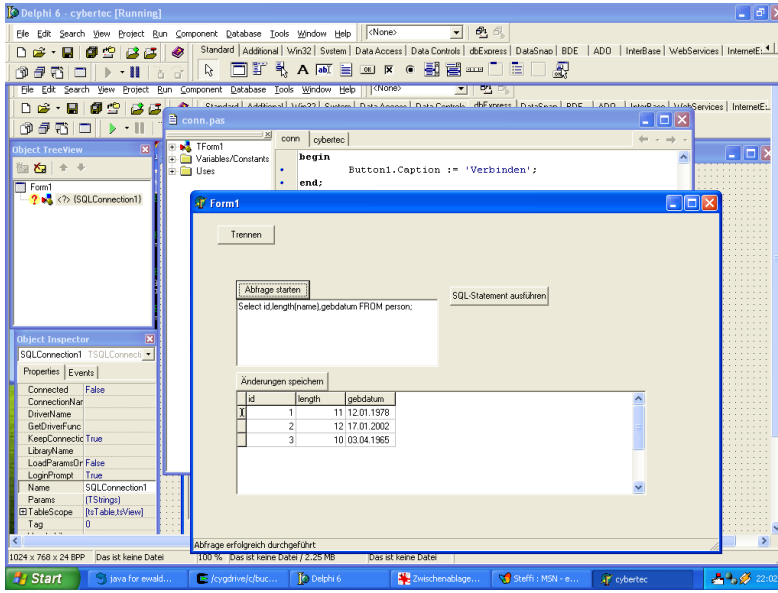
Bevor wir uns die Daten ansehen können, müssen wir noch ein Objekt vom Typ `DataSource` zum Form hinzufügen, welches im Menüpunkt `DataAccess` zu finden ist. Diesem Objekt müssen Sie in den Eigenschaften wiederum das `SQLClientDataSet` zuweisen.

Hier zeigen wir Ihnen, mit welchem Datenbestand Sie in diesem Abschnitt testen:

```
1  CREATE TABLE person (
2          id int4,
3          name text,
4          gebdatum date
5  );
6
7  COPY person FROM stdin USING DELIMITERS ';';
8  1;Hansi Huber;1978-01-12
9  2;Pauli Gruber;2002-01-17
10 3;Rudi Bauer;1965-04-03
11 \.
```

Die Tabelle `person` enthält drei Datensätze.

Im folgenden Screenshot ist fertige Applikation abgebildet:



Am oberen Ende des Forms gibt es einen Button mit dem Namen `Button1`, der für die Herstellung der Verbindung zur Datenbank verantwortlich ist. Abhängig vom Zustand der Datenbankverbindung enthält der Button den entsprechenden Text. Gleich unter dem `Button1` befindet sich ein Button mit dem Namen `Button2`, der zum Ausführen von `SELECT`-Statements verwendet werden kann. Mithilfe dieses Buttons wird eine Abfrage an den Server geschickt und das Grid am unteren Ende der Anwendung aktualisiert.

Unter dem Eingabefeld (Name: `Memo1`) befindet sich ein Button (`Button3`), mit dem Sie Änderungen im Grid direkt wirksam werden lassen können. Das Grid selbst enthält Daten, die von einem SQL Statement zurückgegeben werden. Wenn Sie SQL-Statements ausführen wollen, die keine Abfragen sind (also `INSERT`, `UPDATE`, `DELETE` und dergleichen), müssen Sie den `Button4` verwenden (»SQL-Statement ausführen«).

Sehen wir uns nun den Code an, der für die Applikation maßgebend ist:

```
{ Button Verbinden Verbindet und Trennt die SQLConnection }
procedure TForm1.Button1Click(Sender: TObject);
begin
    SQLConnection1.Connected := not SQLConnection1.Connected;
    if SQLConnection1.Connected then
    begin
        StatusBar1.Panels[0].Text := 'Verbunden';
    end
    else
        StatusBar1.Panels[0].Text := 'Nicht verbunden';
end;

{ Die Buttonbeschriftung wird verändert falls die Connection
  abgebrochen wird }
procedure TForm1.SQLConnection1AfterConnect(Sender: TObject);
begin
        Button1.Caption := 'Trennen';
end;

{ Button wird umgeschrieben nach dem connect }
procedure TForm1.SQLConnection1AfterDisconnect(Sender: TObject);
begin
        Button1.Caption := 'Verbinden';
end;

{ SQLClientDataSet wird geschlossen, der Text aus dem Memo Feld genommen
  und die Query mit Commandtype ctQuery gegen den Server gefahren
  Dataset wird mit Daten geöffnet und editierbar gemacht }

procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
        SQLClientDataset1.Close;
        SQLClientDataset1.CommandText := Memo1.Lines.Text;
        SQLClientDataset1.CommandType := ctQuery;
        SQLClientDataset1.Open;
        SQLClientDataset1.Edit;
        StatusBar1.Panels[0].Text := 'Abfrage erfolgreich durchgeführt';
end;

{ Hier werden Updates gemacht, Button Änderungen speichern }
procedure TForm1.Button3Click(Sender: TObject);
var
Fehler : Integer;
begin
        Fehler := SQLClientDataset1.ApplyUpdates(-1);
        if Fehler > 0 then
           ShowMessage(Format('Fehler: %d ', [Fehler]))
end;

{ Hier wird direkt das Insert durchgeführt. Damit geht's
  direkt auf die DB. INSERT,Update,DELETE kann durchgeführt werden. }
procedure TForm1.Button4Click(Sender: TObject);
begin
        sqlconnection1.Executedirect(Memo1.Lines.Text);
        sqlclientdataset1.Refresh;
end;
```

Die erste Funktion definiert die Funktionsweise von `Button1`. Abhängig vom Zustand der Verbindung wird der entsprechende Text angezeigt. Wie Sie sehen ist auch das Aufbauen der Verbindung zur Datenbank sehr effizient und einfach zu bewerkstelligen. Die nächsten beiden Funktionen dienen ebenfalls zur Behandlung des Buttons.

Etwas spektakulärer ist der `Button2`. Das `SQLClientDataSet` wird geschlossen der Text wird aus dem `Memo`-Feld genommen. Weiter wird die Abfrage mithilfe von `Commandtype ctQuery` an den Server geschickt. Das Dataset wird geöffnet und das Feld wird auf `editierbar` gesetzt.

Der `Button3` dient zum Durchführen von Updates. Um direkt einen Befehl an die Datenbank zu schicken, kann der `Button4` verwendet werden. Mit Hilfe der Methode `Executedirect` wird der Code direkt an das Backend weitergeleitet.

Das Beispiel zeigt, wie einfache Applikationen mithilfe von Delphi geschrieben werden können. Wir ermutigen Sie selbst aktiv zu werden und Delphi einzusetzen, weil es nur wenige Programmiersprachen gibt, bei denen Sie mit geringem Zeitaufwand Windows-Anwendungen schreiben können.

Kapitel 15

PostgreSQL-Interna

Im letzten Kapitel dieses Buches wollen wir uns die Interna von PostgreSQL ansehen. Bei vielen praktischen Aufgaben kann es höchst sinnvoll sein, zu wissen, was in der Datenbank so alles vor sich geht. Wir wollen versuchen, Ihnen näher zu bringen, was Sie für das praktische Arbeiten benötigen und was Ihnen hilft, die Effizienz zu steigern. Sie werden schnell erkennen, welche Möglichkeiten sich ergeben und wie Sie PostgreSQL noch besser einsetzen können.

15.1 Das Abarbeiten von Statements

Das Abarbeiten von SQL-Statements ist alles andere als trivial. Es gibt zahlreiche Bereiche innerhalb der Datenbank, die durchlaufen werden müssen, um ein Ergebnis in der gewünschten Form zu erzeugen. In diesem Abschnitt sehen wir uns die wichtigsten Schritte an.

15.1.1 Der Parser

Sofern bereits eine Verbindung zur Datenbank besteht, können SQL-Statements abgearbeitet werden. Am Beginn der Verarbeitungskette steht ein so genannter Parser. Ein Parser liest ein Statement und analysiert dessen Syntax. Dabei versucht der Parser einen so genannten Parsierungsbaum aufzubauen. In einem derartigen Baum sind alle Komponenten einer Abfrage als Knoten realisiert. Sofern das zu prüfende Statement nicht korrekt ist, gibt der Parser einen Fehler aus, der angibt, wo und warum der Fehler aufgetreten ist. Dabei wird die Verarbeitung eines Befehles beendet. Sollte es beim Aufbau des Parsierungsbaumes zu keinen Problemen gekommen sein, wird der Baum für die weitere Verarbeitung herangezogen.

15.1.2 Das Rewrite System von PostgreSQL

Nach dem Aufbau des Parsierungsbaumes wird eine Abfrage neu formuliert. Das bedeutet, dass die Abfrage so aufbereitet wird, damit Sie besser weiterverarbeitet werden kann. An dieser Stelle spielen Rules eine große Rolle, da Sie das Verhalten eines Statements beeinflussen. Wie Sie in diesem Buch bereits gelernt haben, ist es möglich, mit Hilfe von Rules Befehle einfach umzudefinieren. Intern passiert dieses Umdefinieren durch das Rewrite System.

Das Rewrite System wird maximal zehnmal durchlaufen — sofern dieses Limit überschritten ist, bricht die Datenbank die Verarbeitung ab, da es sich dann mit allergrößter Wahrscheinlichkeit um eine Endlosschleife handelt (etwa hervorgerufen durch falsch definierte Rules).

15.1.3 Der Optimizer

Nach dem Rewrite ist der Optimizer am Zug. Ziel des Optimizers ist es, die Abfrage bestmöglich zu planen und einen Weg zu finden, den SQL-Code so schnell wie möglich auszuführen. Bei der Optimierung der Abfragen gibt es mehrere Möglichkeiten.

Extensive Searching

Wenn eine einfache Abfrage ausgeführt werden soll, prüft PostgreSQL alle möglichen Arten, die Abfrage auszuführen und sucht aus den verschiedenen Execution Plans den aus, der mit großer Wahrscheinlichkeit die kürzeste Ausführungszeit aufweist. Bei sehr einfachen Abfragen ist es nicht besonders aufwändig, alle Möglichkeiten zu prüfen und die Optimierung der Abfrage stellt kein allzu großes Problem dar.

GEQO — Genetische Algorithmen

Wenn die Anzahl der Wege durch die Abfrage so groß ist, dass ein vollständiges Prüfen der Möglichkeiten keinen Sinn mehr macht, verwendet PostgreSQL GEQO (Genetic Query Optimizer). GEQO basiert auf einem so genannten genetischen Algorithmus, der im Folgenden ansatzweise erklärt wird:

Man nehme eine zufällig ausgewählte Menge von möglichen Wegen durch die Abfrage. Diese Abfragen werden dann geprüft und deren vermutete Ausführungszeiten werden verglichen. Die besten dieser zufällig ausgewählten Wege werden dann herangezogen und die Vererbung beginnt. Jeder potenzielle Weg durch die Abfrage hat gewisse Eigenschaften (beispielsweise »Sequential Scan on A« oder »Hash Join«), die an weitere, neue Execution Plans weitervererbt werden. Diese neuen, leicht modifizierten Pläne werden wieder geprüft und die besten Execution Plans bekommen die Chance, Ihre Eigenschaften weiterzuvererben und in leicht modifizierter Version weiterzuleben.

Auf diese Weise werden nicht alle Möglichkeiten geprüft. Das Ergebnis der Optimierung muss nicht optimal sein, es ist jedoch immer besser als ein zufällig ausgewählter Execution-Plan. Bedenken Sie: Der genetische Algorithmus wird nur dann angewendet, wenn eine vollständige Prüfung aller möglichen Wege aufgrund der vielen Möglichkeiten in endlicher Zeit beziehungsweise in einer vernünftigen Zeitspanne nicht mehr bewältigt werden kann.

15.1.4 Der Executor

Nach dem Aufbau der Execution Plans wird die eigentliche Abfrage durchgeführt — das passiert mithilfe des Executors. Auf der Ebene des Executors setzt

die SPI-Schnittstelle an, die es direkt erlaubt, mit diesem Teil der Datenbank zu interagieren. Bei Tcl haben wir gesehen, welche Möglichkeiten sich hier bieten und wie mächtig diese Schnittstelle ist.

15.2 Das Frontend/Backend-Protokoll

Für die Kommunikation zwischen einer Client-Software und dem Server stellt PostgreSQL ein Protokoll zur Verfügung. Dieses Protokoll dient einzig und alleine der Übertragung von Informationen und verpackt die zu übertragenden Daten in client- und serverseitig lesbare Pakete.

Seit PostgreSQL 6.4 gibt es das Protokoll in Version 2.0 und genau damit wollen wir uns in diesem Abschnitt beschäftigen.

15.2.1 Der Verbindungsaufbau

Um eine Anfrage an die Datenbank zu stellen, versucht der Client zuerst eine Verbindung zu PostgreSQL herzustellen. Das passiert durch das Senden eines Startpaketes, das alle relevanten Verbindungsdaten enthält.

Dann teilt der Server dem Client mit, ob die Anfrage akzeptiert worden ist oder nicht. Das erste Byte in einem Datenstrom definiert, um welchen Datenstrom es sich handelt. Das gilt für alle Pakete außer dem Paket zum Herstellen einer Verbindung zur Datenbank. Dieser Unterschied hat historische Gründe und ist an dieser Stelle irrelevant.

15.2.2 Das Protokoll im Detail

Eine Verbindung unterscheidet vier Zustände: `start-up`, `query`, `function call` und `termination`. Diese vier Zustände entscheiden über das weitere Verhalten der Datenbank. Sehen wir uns also an, welche Zustände welche Elemente enthalten können.

Startup

PostgreSQL kann auf die folgenden Arten antworten:

`ErrorResponse`: Die Verbindung wird danach sofort geschlossen.

`AuthenticationOk`: Erfolgreiche Authentifizierung.

`AuthenticationKerberosV4`: Löst eine Kerberos IV-Authentifizierung aus.

`AuthenticationKerberosV5`: Löst eine Kerberos V-Authentifizierung aus. Ist diese erfolgreich, wird wieder `AuthenticationOk` zurückgegeben.

`AuthenticationCleartextPassword`: Ein Passwort wird gefordert — bei Erfolg wird `AuthenticationOk` zurückgegeben.

`AuthenticationCryptPassword`: Ein Passwort wird gefordert — bei Erfolg wird `AuthenticationOk` zurückgegeben.

`AuthenticationMD5Password`: Ein Passwort wird gefordert — bei Erfolg wird `AuthenticationOk` zurückgegeben.

`AuthenticationSCMCredential`: Wird nur von UNIX-Sockets unterstützt.

Wenn die Verbindung zur Datenbank in Ordnung ist, werden weitere Daten geliefert:

`BackendKeyData`: Enthält einen Schlüssel, um eine Abfrage abbrechen zu können.

`ReadyForQuery`: Eine Abfrage kann jetzt ausgeführt werden.

`ErrorResponse`: Ein Fehler ist aufgetreten.

`NoticeResponse`: Eine Warnung ist ausgegeben worden.

Query

Wenn ein Client eine Abfrage ausführen will, sendet er diese an den Server. Der Server kann wie folgt antworten:

`CompletedResponse`: SQL-Kommando erfolgreich.

`CopyInResponse`: Daten können nun kopiert werden.

`CopyOutResponse`: Daten können nun von PostgreSQL kopiert werden.

`CursorResponse`: Markiert einen Cursor.

`RowDescription`: Definiert die folgende Datenreige — die Daten selbst werden durch `AsciiRow` oder `BinaryRow` gekennzeichnet.

`EmptyQueryResponse`: Eine leere Abfrage wurde verarbeitet.

`ErrorResponse`: Ein Fehler ist aufgetreten.

`ReadyForQuery`: Wird im Fall einer korrekten Abfrage geschickt.

`NoticeResponse`: Gibt eine Meldung (Warnung) aus.

Function Call

Wenn das Frontend die Meldung für einen Function Call an das Backend schickt, können die folgenden Werte retourniert werden:

`ErrorResponse`: Ein Fehler ist aufgetreten.

`FunctionResultResponse`: Der Aufruf war erfolgreich und ein Resultat steht zur Verfügung.

`FunctionVoidResponse`: Der Funktionsaufruf war erfolgreich, hat aber kein Ergebnis zurückgegeben.

`ReadyForQuery`: Der Funktionsaufruf ist fertig.

`NoticeResponse`: Ein Hinweis.

Abbrechen von Anfragen

Um eine Abfrage abzubrechen, muss ein neuer Backendprozess gestartet werden. Statt des Authentifizierungsprozesses gibt es hier jedoch ein anderes Prozedere. Bei jeder Abfrage gibt es ein geheimes Zertifikat, das hier zum Einsatz kommt und PostgreSQL mitteilt, welche Abfrage abzubrechen ist. Dabei ist zu bemerken, dass es aus Sicherheitsgründen keinen Rückgabewert gibt. Jeder Prozess kann eine Abfrage töten; der Sicherheitscheck erfolgt durch das geheime Zertifikat.

Termination

Wenn eine Verbindung geschlossen werden soll, ist das einfach dem Backend mitzuteilen. Dieses verabschiedet sich still und leise.

15.3 Systemtabellen

Jede hoch entwickelte Datenbank stellt so genannte Systemtabellen bereit. Solche Tabellen nehmen in der Datenbank einen ganz besonderen Stellenwert ein und sind nicht mit den Tabellen eines Benutzers vergleichbar; für Systemtabellen gelten eigene Gesetze und Restriktionen, die sich aus der Natur der Sache ergeben. Da Systemtabellen in den meisten Fällen nicht via SQL angesprochen sondern direkt von PostgreSQL modifiziert werden, ergeben sich einige Punkte, die unbedingt zu bedenken sind, wenn Sie sich näher mit Systemtabellen beschäftigen.

Systemtabellen sind nicht triggerbar: Da Systemtabellen, wie bereits erwähnt, nicht immer über die normalen Schnittstellen angesprochen, werden, ergeben sich daraus Risiken, die das Verwenden von Triggern unmöglich machen. Da Systemtabellen als Interna zu betrachten sind, sollte das aber kein gröberes Problem aufwerfen.

Eine vollständige Liste aller Systemtabellen kann leicht erzeugt werden:

```
buch=# \dS
               List of relations
            Name             |  Type   |  Owner
-----------------------------+---------+----------
 pg_aggregate                | table   | postgres
 pg_am                       | table   | postgres
 pg_amop                     | table   | postgres
 pg_amproc                   | table   | postgres
 pg_attrdef                  | table   | postgres
 pg_attribute                | table   | postgres
 pg_class                    | table   | postgres
 pg_database                 | table   | postgres
 pg_description              | table   | postgres
 pg_group                    | table   | postgres
 pg_index                    | table   | postgres
 pg_indexes                  | view    | postgres
 pg_inherits                 | table   | postgres
 pg_language                 | table   | postgres
 pg_largeobject              | table   | postgres
 pg_listener                 | table   | postgres
 pg_opclass                  | table   | postgres
 pg_operator                 | table   | postgres
 pg_proc                     | table   | postgres
 pg_relcheck                 | table   | postgres
 pg_rewrite                  | table   | postgres
 pg_rules                    | view    | postgres
 pg_shadow                   | table   | postgres
 pg_stat_activity            | view    | postgres
 pg_stat_all_indexes         | view    | postgres
 pg_stat_all_tables          | view    | postgres
 pg_stat_database            | view    | postgres
 pg_stat_sys_indexes         | view    | postgres
 pg_stat_sys_tables          | view    | postgres
 pg_stat_user_indexes        | view    | postgres
 pg_stat_user_tables         | view    | postgres
 pg_statio_all_indexes       | view    | postgres
 pg_statio_all_sequences     | view    | postgres
 pg_statio_all_tables        | view    | postgres
 pg_statio_sys_indexes       | view    | postgres
 pg_statio_sys_sequences     | view    | postgres
 pg_statio_sys_tables        | view    | postgres
 pg_statio_user_indexes      | view    | postgres
 pg_statio_user_sequences    | view    | postgres
 pg_statio_user_tables       | view    | postgres
 pg_statistic                | table   | postgres
 pg_stats                    | view    | postgres
 pg_tables                   | view    | postgres
 pg_trigger                  | table   | postgres
```

```
 pg_type                 | table   | postgres
 pg_user                 | view    | postgres
 pg_views                | view    | postgres
 pg_xactlock             | special | postgres
(48 rows)
```

Wie Sie sehen, ist die Liste schon recht stattlich. Weiter ist zu beachten, dass alle Tabellen mit pg_ beginnen — dieses Präfix kennzeichnet eine Systemtabelle und kann nur für solche verwendet werden, wie das nächste Listing zeigt:

```
buch=# CREATE TABLE pg_hans(id int4);
ERROR:  invalid relation name "pg_hans"; the 'pg_' name prefix is
reserved for system catalogs
```

Da dieses Präfix intern sehr oft abgefragt wird, kann es nicht verwendet werden.

Aus Sicherheitsgründen können Systemtabellen auch nicht gelöscht werden:

```
buch=# DROP TABLE pg_proc;
ERROR:  table "pg_proc" is a system table
```

Viele der Systemtabellen sind Views und basieren auf anderen Tabellen. Das dient zur Vereinfachung der Lesbarkeit: Da Systemtabellen ohnehin nur für den lesenden Zugriff gedacht sind, müssen Sie sich aber keine Gedanken machen, ob eine Relation nun eine View oder eine »echte« Tabelle ist.

Hier finden Sie alle Tabellen in einer Übersicht:

Spalte	Bedeutung
pg_database	Liste der Datenbanken
pg_class	Liste der Tabellen
pg_attribute	Spalteninformation
pg_index	Liste der Indices
pg_proc	Liste der vorhandenen Funktionen
pg_type	Liste der Datentypen
pg_operator	Liste der Operatoren
pg_aggregate	Liste der Aggregierungsfunktionen
pg_am	Zugriffsmethoden
pg_amop	Zugriffsoperatoren
pg_amproc	Funktionen für Zugriffsmethoden
pg_opclass	Klassen der Zugriffsoperatoren

Bei den meisten Releases von PostgreSQL kommt es zu kleinen Änderungen in den Systemtabellen. Aus diesem Grund empfehlen wir die genauen Definitionen der Spalten der einzelnen Tabellen in den entsprechenden Header-Files im Source-Code nachzulesen. Die Files sind unter `src/include/catalog` zu finden und exzellent dokumentiert.

Anhang

Inhalt der CD

Auf der beiliegenden CD-ROM finden Sie neben allen Anwendungsbeispielen aus diesem Buch (in `/Beispiele`) einen Teilabzug von `ftp.postgresql.org` (in `/FTP-Teilabzug`) sowie einige Bilder von der O'Reilly Open Source Conference 2002 in San Diego (in `/OSCON 2002`) — damit Sie sich ein Bild davon machen können, wie PostgreSQL-Entwickler so aussehen:).

Sie können durch die ganze CD-ROM navigieren, indem Sie `index.html` im Stammverzeichnis der CD-ROM aufrufen.

/Beispiele

Alle Anwendungsbeispiele liegen in Quellcodeform vor und sind auf Lauffähigkeit geprüft. Es gibt für jedes Kapitel einen eigenen Ordner. Am einfachsten navigieren Sie durch die Verzeichnisse, indem Sie `index.html` unter `/Beispiele` aufrufen.

/FTP-Teilabzug

Der Teilabzug von `ftp.postgresql.org` enthält neben dem Sourcecode für die PostgreSQL-Versionen 6.1 bis 7.2.2 und dem Entwickler-Verzeichnis die vollständige Dokumentation zu den Versionen 7.1 und 7.2 sowie die PostgreSQL-Unterstützung für die ODBC-Schnittstelle u.v.a.m.

/OSCON 2002

Wenn Sie die in diesem Verzeichnis liegende `index.html` aufrufen, können Sie bequem durch die Bilder navigieren.

Stichwortverzeichnis

D

E

F

G

H

I

P

Q

R

S

T

U

V

W

X

Z

... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

InformIT.de, Partner von **Markt+Technik**, ist unsere Antwort auf alle Fragen der IT-Branche.

In Zusammenarbeit mit den Top-Autoren von Markt+Technik, absoluten Spezialisten ihres Fachgebiets, bieten wir Ihnen ständig hochinteressante, brandaktuelle Informationen und kompetente Lösungen zu nahezu allen IT-Themen.

wenn Sie mehr wissen wollen ... **www.InformIT.de**