

Iutz FRÖHLICH

PostgreSQL



**PRAXISBUCH FÜR
ADMINISTRATOREN
UND ENTWICKLER**



EXTRA: Mit kostenlosem E-Book



Im Internet: Die Programme
und Skripte zum Download

HANSER



Bleiben Sie auf dem Laufenden!

Der Hanser Computerbuch-Newsletter informiert Sie regelmäßig über neue Bücher und Termine aus den verschiedenen Bereichen der IT. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter **www.hanser-fachbuch.de/newsletter**

Lutz Fröhlich

PostgreSQL 9

Praxisbuch für Administratoren
und Entwickler

HANSER

Der Autor:

Lutz Fröhlich, Darmstadt

Kontakt: lutz@lutzfroehlich.de

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

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

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2013 Carl Hanser Verlag München, www.hanser-fachbuch.de

Lektorat: Brigitte Bauer-Schiewek

Herstellung: Irene Weilhart

Copy editing: Manfred Sommer, München

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Stephan Rönigk

Gesamtherstellung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

print-ISBN: 978-3-446-42239-1

e-book-ISBN: 978-3-446-42932-1

Inhalt

1	Einführung und Geschichte	1
1.1	Einführung und Motivation	1
1.2	Die Geschichte von PostgreSQL	2
1.3	Verwendete Version	3
1.4	Konventionen	3
2	Installation und Quellen	5
2.1	Paketinstallation	5
2.2	Installation aus dem Quellcode	7
2.2.1	Installation aus dem Quellcode unter Linux	7
2.2.2	Installation aus dem Quellcode unter Windows	9
2.3	Erste Schritte	10
3	Server und Datenbank konfigurieren und administrieren	15
3.1	Parameter-Einstellungen vornehmen	15
3.1.1	Parameter für System-Ressourcen	16
3.1.2	Parameter für Write Ahead Log (WAL) und Checkpoints	18
3.1.3	Parameter für den Background Writer	19
3.1.4	Parameter für Archivierung, Streaming Replication und Standby-Server	20
3.1.5	Parameter für Verbindung und Authentifizierung	21
3.1.6	Parameter für Client-Standards	22
3.1.7	Parameter für VACUUM	23
3.1.8	Parameter für SQL Tuning	23
3.1.9	Parameter für Logging und Tracing	24
3.1.10	Benutzerspezifische Parameter	26
3.1.11	Parameter für Entwickler	26
3.2	Die WAL-Archivierung einschalten	27
3.3	Authentifizierung von Clients	29
3.4	Einen Connection Pool aufsetzen	31

3.5	Concurrency	33
3.6	Datenbanken verwalten	35
3.7	Wartungsaufgaben	40
3.7.1	VACUUM	40
3.7.2	ANALYZE	44
3.8	Nützliche Skripte und Hinweise	44
3.8.1	Eine Passwort-Datei verwenden	45
3.8.2	Welche Parameter sind Nicht-Standard?	45
3.8.3	Eine Session killen	45
3.8.4	Ein Tabelle nach Excel kopieren	46
3.8.5	Die Datei .psqlrc	47
3.8.6	Einen WAL-Switch manuell auslösen	47
3.8.7	Die PostgreSQL-Server Log-Datei in eine Tabelle laden	48
3.8.8	Automatisches Rotieren von Log-Dateien	48
3.8.9	Nicht verwendete Indexe identifizieren	49
3.8.10	Microsoft Excel als Datenbank-Client	49
3.8.11	Den Inhalt der Kontrolldatei ausgeben	50
3.8.12	Platzverbrauch von Tabellen	51
3.8.13	Die Anzahl von Verbindungen begrenzen	52
3.8.14	Tabellen und Indexe in eine andere Tablespace legen	53
3.8.15	Updateable Views	54
3.8.16	Temporäre Tablespace verwalten	55
3.8.17	Lang laufende SQL-Anweisungen	55
4	Sicherung und Wiederherstellung	57
4.1	SQL Dump	57
4.2	Offline-Sicherung auf Dateisystem-Ebene	61
4.3	Online-Sicherung mit Point-in-time-Recovery	62
4.4	Funktionen zur Sicherung und Wiederherstellung	65
5	Sicherheit und Überwachung	67
5.1	Sicherheit	68
5.1.1	Datenbank-Rollen und -Privilegien	68
5.1.2	Authentifizierung und Zugangskontrolle	74
5.1.3	Rechteverwaltung (Privilegien)	76
5.1.4	Optionen für die Verschlüsselung	81
5.1.5	Out-of-the-box-Sicherheit	84
5.2	Hacker-Attacken abwehren	86
5.2.1	Brute-Force-Attacken	86
5.2.2	Network Spoofing	88
5.2.3	SQL Injection	89
5.3	Überwachung	90

5.3.1	Auditing	91
5.3.2	Monitoring	93
6	Performance Tuning	99
6.1	PostgreSQL Server-Tuning	99
6.1.1	Goldene Regeln für neue Server und Datenbanken	100
6.1.2	Tuning mit „pgtune“	101
6.1.3	Optimierung der Memory-Parameter	102
6.2	Performance-Analyse	105
6.2.1	Analyse mit dem „Statistics Collector“	105
6.2.2	Der Background Writer	108
6.2.3	Analyse mit „pgstatpack“	109
6.3	Partitionierung	114
6.3.1	Vererbung von Tabellen	115
6.3.2	Tabellen partitionieren	116
6.4	Benchmarks	118
6.4.1	Ein TPCC-Benchmark	120
6.4.2	Fazit des Benchmarks	124
7	Hot Standby und Streaming Replication	125
7.1	Eine Standby-Datenbank aufsetzen	126
7.1.1	Vorbereitung und Planung	126
7.1.2	Konfiguration und Aktivierung	127
7.1.3	Streaming Replication einsetzen	130
7.1.4	Die Replikation überwachen	131
7.1.5	Synchrone Replikation	131
7.2	Eine Hot-Standby-Datenbank betreiben	132
7.3	Failover und Switchover	135
8	Entwicklung von Applikationen für PostgreSQL	139
8.1	Client-Programmierung	139
8.1.1	Die PostgreSQL Frontend Library „libpq“	139
8.1.2	PostgreSQL-Applikationen mit Java entwickeln	152
8.2	Server-Programmierung	161
8.2.1	Integrierte Prozedurale Sprachen	161
8.2.2	Trigger	183
8.2.3	SQL erweitern	185
9	Das Regelsystem	191
9.1	Das Regelsystem für SELECT-Anweisungen	192
9.2	Das Regelsystem für DML-Anweisungen	193
9.3	Regeln und Views	196

10	Optimierung von SQL-Anweisungen	199
10.1	Der Query Optimizer	199
10.2	Indexe effektiv einsetzen	201
10.3	Optimierung von SQL-Anweisungen	204
10.3.1	Der EXPLAIN-Befehl	205
10.3.2	Ausführungspläne verstehen	208
11	PostGIS	213
11.1	PostGIS und PostgreSQL	213
11.2	PostGIS installieren	214
11.2.1	Paketorientierte Installation	214
11.2.2	Installation aus dem Quellcode	216
11.3	Erste Schritte mit PostGIS	216
11.4	SQL-Abfragen in PostGIS	222
12	PostgreSQL in die IT-Landschaft einbinden	225
12.1	Funktionen und Features	225
12.2	Datensicherung und Wiederherstellung	226
12.3	Desaster Recovery (DR)	227
12.4	Überwachung	228
12.5	Administrierbarkeit	228
12.6	Verfügbarkeit	229
12.7	Datensicherheit und Auditing	229
12.8	Performance und Skalierbarkeit	230
12.9	Schnittstellen und Kommunikation	231
12.10	Support	231
12.11	Fazit	232
13	Migration von MySQL nach PostgreSQL	233
13.1	Unterschiede zwischen MySQL und PostgreSQL	233
13.2	Eine Migration durchführen	235
14	Von Oracle auf PostgreSQL umsteigen	241
14.1	Den Umstieg planen	241
14.2	Unterschiede zwischen Oracle und PostgreSQL	243
14.2.1	Unterschiede bei Datentypen	243
14.2.2	Steigerung der Kompatibilität von PostgreSQL	247
14.3	Portierung von Oracle PL/SQL	248
14.4	Tools zur Unterstützung der Migration	251
14.5	Technisches Vorgehen	251
14.6	Eine Beispiel-Migration	252
14.6.1	Übernahme der Daten	256

15	Replikation zwischen PostgreSQL und Oracle	259
15.1	Datenbank-Link zwischen Oracle und PostgreSQL	259
15.2	Replikation mit Transaktionslog	265
15.2.1	Replikation mit Oracle Streams Heterogeneous Information Sharing	266
15.2.2	Replikation mit Oracle XStream	270
15.2.3	Replikation von Oracle zu PostgreSQL mit „XStream Out“	271
Index		285

Einführung und Geschichte

■ 1.1 Einführung und Motivation

PostgreSQL ist aktuell die weltweit am stärksten wachsende Datenbank. Immer mehr Unternehmen sind davon überzeugt, dass Open-Source-Produkte in vielen Bereichen gegenüber kommerziellen Produkten klar im Vorteil liegen. Viele Open-Source-Produkte haben bewiesen, dass eine starke Entwickler-Community in der Lage ist, Produkte dynamischer und mit mehr Qualität zu entwickeln und weiter auszubauen. Das Thema Support ist ebenfalls kein Problem. Es gibt genügend IT-Dienstleister, die professionellen Support für Open-Source-Produkte anbieten.

Die zunehmende Popularität von PostgreSQL ist auch darauf zurückzuführen, dass die andere große Open-Source-Datenbank *MySQL* nach der Übernahme durch die Firma Oracle ums Überleben kämpft. Es ist ein zunehmender Umstieg von MySQL auf PostgreSQL zu verzeichnen.

Ein weiterer Vorteil, den PostgreSQL bietet, ist der sehr hohe Kompatibilitätsgrad zur Oracle-Datenbank. Dadurch ist es möglich, mit überschaubarem Migrationsaufwand von Oracle nach PostgreSQL zu migrieren. Darüber hinaus können Oracle- und PostgreSQL-Datenbanken problemlos im Mischbetrieb eingesetzt werden.

Nicht zuletzt wird die zunehmende Verbreitung von PostgreSQL aber auch durch dessen Robustheit in der Transaktionsverarbeitung, hohe Zuverlässigkeit sowie sehr gute Performance unterstützt. Die Version 9 sticht durch neue Features wie eingebauter Replikation, *Hot Standby* und *Streaming Replication*, zusätzlichen Performance-Features, Spalten-Triggern sowie vielen anderen sehr nützlichen Erweiterungen hervor.

Dennoch ist PostgreSQL eine schlanke und sehr gut zu verwaltende Datenbank, die sich auf ihr Kerngeschäft, die Verwaltung von Datenbeständen, konzentriert. Sie ist in den vergangenen Jahren erwachsen geworden und wird heute in vielen Bereichen kommerziell eingesetzt. PostgreSQL läuft auf vielen Plattformen, wie z. B. Linux, MacOS X, Solaris oder Windows. Obwohl es sich um ein Open-Source-Produkt handelt, kann kommerzieller Support zu einem vernünftigen Preis hinzugekauft werden. Einem professionellen Einsatz steht damit nichts im Wege.

Das Buch beschäftigt sich detailliert mit den Themen Administration von und Entwicklung auf PostgreSQL 9. Darüber hinaus finden Sie viele Informationen zum Thema „Migration

von Oracle oder MySQL nach PostgreSQL“ sowie zur Planung und Implementierung von PostgreSQL-Datenbanken in eine bestehende IT-Landschaft. Das schließt den Datenaustausch mit anderen Datenbanksystemen, insbesondere Oracle, mit ein.

Trotz einer hohen Kompatibilitätsrate zu Oracle gibt es für die Migration einiges zu beachten. Im Buch finden Sie viele praktische Erfahrungen, die bei verschiedenen Migrationen gesammelt werden konnten. Für die Replikation zwischen PostgreSQL und Oracle gibt es unterschiedliche Lösungen, die mehr oder weniger stabil laufen und unterschiedliche Kosten verursachen.

Das Buch ist als Schnelleinstieg für IT-Profis geschrieben und setzt grundlegende Kenntnisse von relationalen Datenbanksystemen voraus. Auf eine grundlegende Erläuterung von gängigen Begriffen wird deshalb bewusst verzichtet, auch um den Umfang des Buches überschaubar zu halten. Dennoch finden Sie viele Beispiele und Praxistipps, die auch Einsteigern die Möglichkeit bieten, sich in das Produkt einzuarbeiten.

■ 1.2 Die Geschichte von PostgreSQL

PostgreSQL geht zurück auf das POSTGRES-Projekt, das an der University of California at Berkeley in den 80er-Jahren angesiedelt war. Die erste vorzeigbare Version erschien im Jahre 1987 als *Postgres* Version 1. Als Reaktion auf die ersten Kritiken wurde das noch heute in PostgreSQL vorhandene Rule-System entwickelt. Version 3 erschien im Jahre 1991 mit einer Weiterentwicklung der Abfrageeinheit. 1993 beendete die University of California das Projekt mit der Version 4.2, um die rasant wachsenden Support-Anforderungen nicht mehr tragen zu müssen.

Nach Hinzufügen eines SQL-Abfrage-Interpreters im Jahre 1995 wurde die Software unter dem Begriff *Postgres95* ins Web gestellt, mit dem Quellcode des originalen Berkeley-Postgres. Das Produkt war zu dieser Zeit komplett in ANSI C geschrieben. Durch Verbesserung in den Bereichen Wartbarkeit und Performance lief es schließlich bis zu 50 % schneller als das originale Berkeley-Postgres.

Die Entscheidung, die Jahreszahl aus dem Produktnamen zu entfernen, fiel im Jahre 1996. Damit wurde Postgres95 zu PostgreSQL, und es begann die ständige Weiterentwicklung von PostgreSQL als Open-Source-Produkt. Obwohl Letzteres über viele Jahre ein Schattendasein im Licht der großen kommerziellen Datenbanken, aber auch der durch den Internet-Boom schnell verbreiteten Open-Source-Datenbank MySQL führte, erfolgte seine konsequente Weiterentwicklung durch die Community.

Heute präsentiert sich PostgreSQL als ausgereift und stabil und erfüllt (fast) alle Anforderungen an ein modernes relationales Datenbanksystem. Für viele überraschend: Die Performance ist gleich oder nur unwesentlich schlechter im Vergleich zu so manchem kommerziellen Produkt. Eine positive Überraschung sind auch die neuen Replikations-Mechanismen der Version 9, bis hin zur Hot Standby-Datenbank.

■ 1.3 Verwendete Version

Das Buch bezieht sich auf die während des Schreibens vorliegende aktuelle Version 9.1.0. Schauen Sie nach weiteren Veröffentlichungen, insbesondere für neuere Versionen, auf der Webseite des Verlages oder der Autoren-Webseite. Alles rund um die PostgreSQL-Community finden Sie auf der offiziellen Webseite <http://www.postgresql.org>.

■ 1.4 Konventionen

Begriffe in spitzen Klammern bezeichnen eine zu ersetzende Variable (so ist zum Beispiel der Ausdruck `<VERSION>` durch die aktuelle Version 9.1.0 zu ersetzen).

Die meisten Darstellungen beziehen sich gleichermaßen auf UNIX- und Windows-Betriebssysteme. Die Darstellung der Umgebungsvariablen erfolgt im Wesentlichen im UNIX-Format, d.h. z.B. `$BIN` statt `%BIN%` für Windows. Sie können das Format einfach nach Windows übertragen. Das Gleiche gilt für das Trennzeichen der Pfade: „/“ unter Unix sowie „\“ unter Windows.

2

Installation und Quellen

Die Installation kann sowohl mit vorgefertigten Paketen als auch aus dem Quellcode erfolgen. Wenn Sie eine schnelle Installation bevorzugen, ist die einfachere Paketinstallation zu empfehlen. Allerdings müssen Sie dann mit den Paketen leben, so wie sie vorgegeben sind. Mit der Installation aus dem Quellcode haben Sie die Möglichkeit, Optionen gegebenenfalls hinzuzunehmen.

Die Pakete und den Quellcode finden Sie auf der Seite <http://www.postgresql.org/download>.

■ 2.1 Paketinstallation

Alle erforderlichen Informationen für die Installation befinden sich auf der Seite <http://www.postgresql.org/download> unter dem Abschnitt *Binary Packages*. Wir führen an dieser Stelle die Paketinstallation für ein Windows-Betriebssystem durch. Wählen Sie unter *Binary Packages* den Link *Windows* und danach die Option *Download the one click installer*. Sie werden zur Seite der Firma *EnterpriseDB* weitergeleitet. Dort finden Sie die zu Ihrem Betriebssystem passenden Pakete. Achten Sie darauf, ob es sich um ein 32- oder 64-Bit-Betriebssystem handelt. Starten Sie die heruntergeladene EXE-Datei. Es erscheint der *PostgreSQL Setup Wizzard* (siehe Bild 2.1).

Das Programm führt Sie durch die Installations-Schritte. Wählen Sie zunächst die Verzeichnisse für die Programme (Binaries) sowie die Datenbank-Dateien. Im nächsten Schritt können Sie das Passwort für den Superuser *postgres* vergeben. Das Installationsprogramm legt einen lokalen Benutzer im Windows-Betriebssystem an, den PostgreSQL-Service-Account (siehe Bild 2.2).

Weitere Eingaben für das Installationsprogramm sind die Portnummer sowie Locale Settings. Die Standard-Portnummer von PostgreSQL ist 5432. Nach Abschluss der Installation wurde ein Windows-Dienst eingerichtet. Voraussetzung für die Benutzung von Datenbanken ist der erfolgte Start des Dienstes (siehe Bild 2.3).

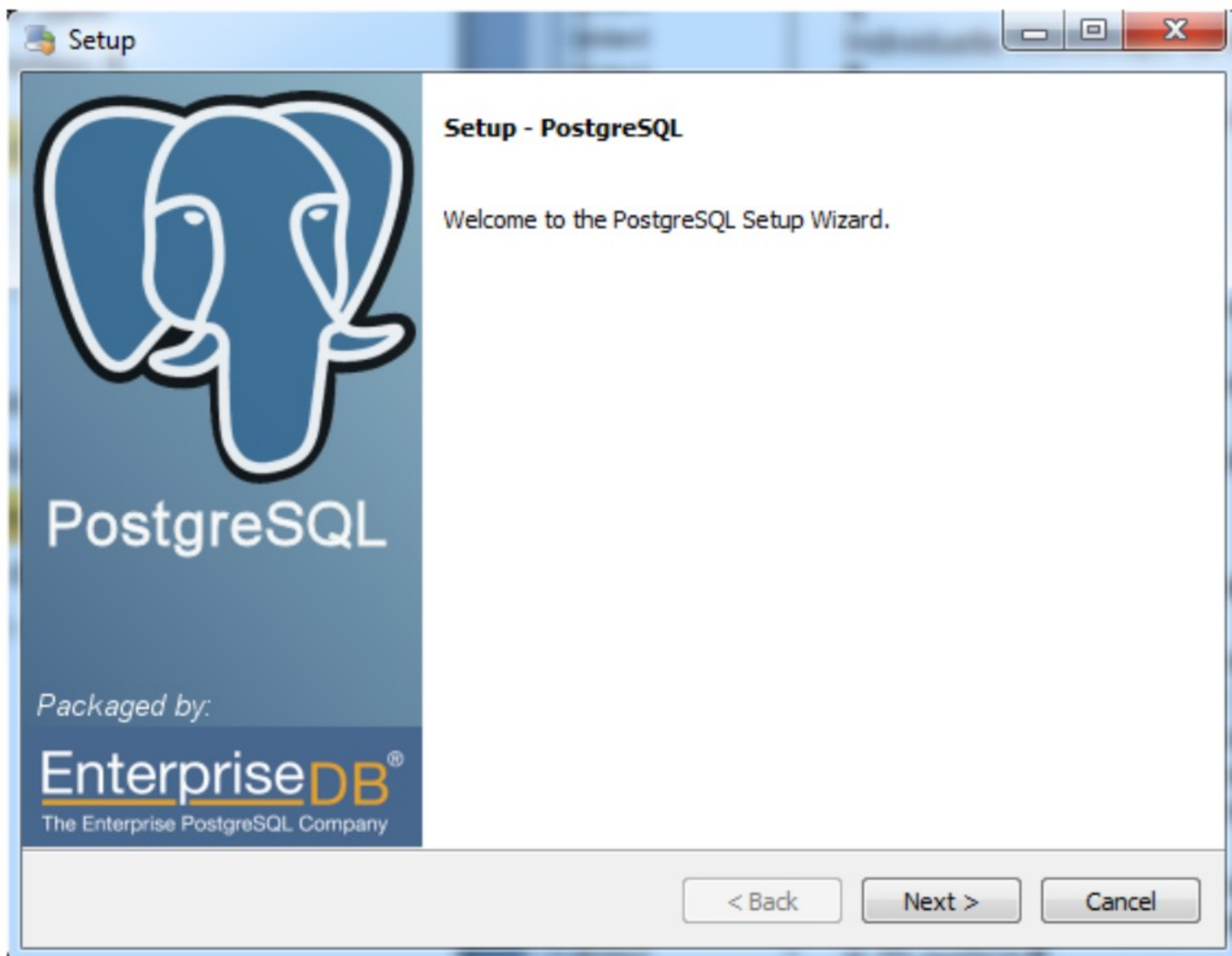


Bild 2.1 Installation des Binary-Packages unter Windows mit dem Setup Wizard

Name	Vollständiger Name	Beschreibung
Administrator		Vordefiniertes Konto für die V...
Gast		Vordefiniertes Konto für Gastz...
HomeGroup...	HomeGroupUser\$	Integriertes Konto für den Zug...
Lutz		
postgres	postgres	PostgreSQL service account

Bild 2.2
Der PostgreSQL-Service-Account in Windows

Name	Beschreibung	Status	Starttyp	Anmelden als
postgresql-x64-9.0	Provides relational databa...	Gestartet	Automa...	.\postgres
Programmkompatibilitäts-Assistent-Dienst	Dieser Dienst bietet Unters...	Gestartet	Automa...	Lokales System
RAS-Verbindungsverwaltung	Verwaltet Einwähl- und V...	Gestartet	Manuell	Lokales System
Remotedesktopdienste	Ermöglicht Benutzern das ...		Manuell	Netzwerkdienst
Remoteprozeduraufruf (RPC)	Der RPCSS-Dienst wird als...	Gestartet	Automa...	Netzwerkdienst
Remoteregistrierung	Ermöglicht Remotebenutz...		Manuell	Lokaler Dienst

Bild 2.3 Der PostgreSQL-Dienst in Windows

Zusammen mit der Kernsoftware für die Datenbank wurde der *pgAdmin III*, ein Programm zur Administration von PostgreSQL-Servern und Datenbanken, entwickelt. pgAdmin ist Teil des Open-Source-Konzepts und somit Bestandteil der Open-Source-Lizenz.

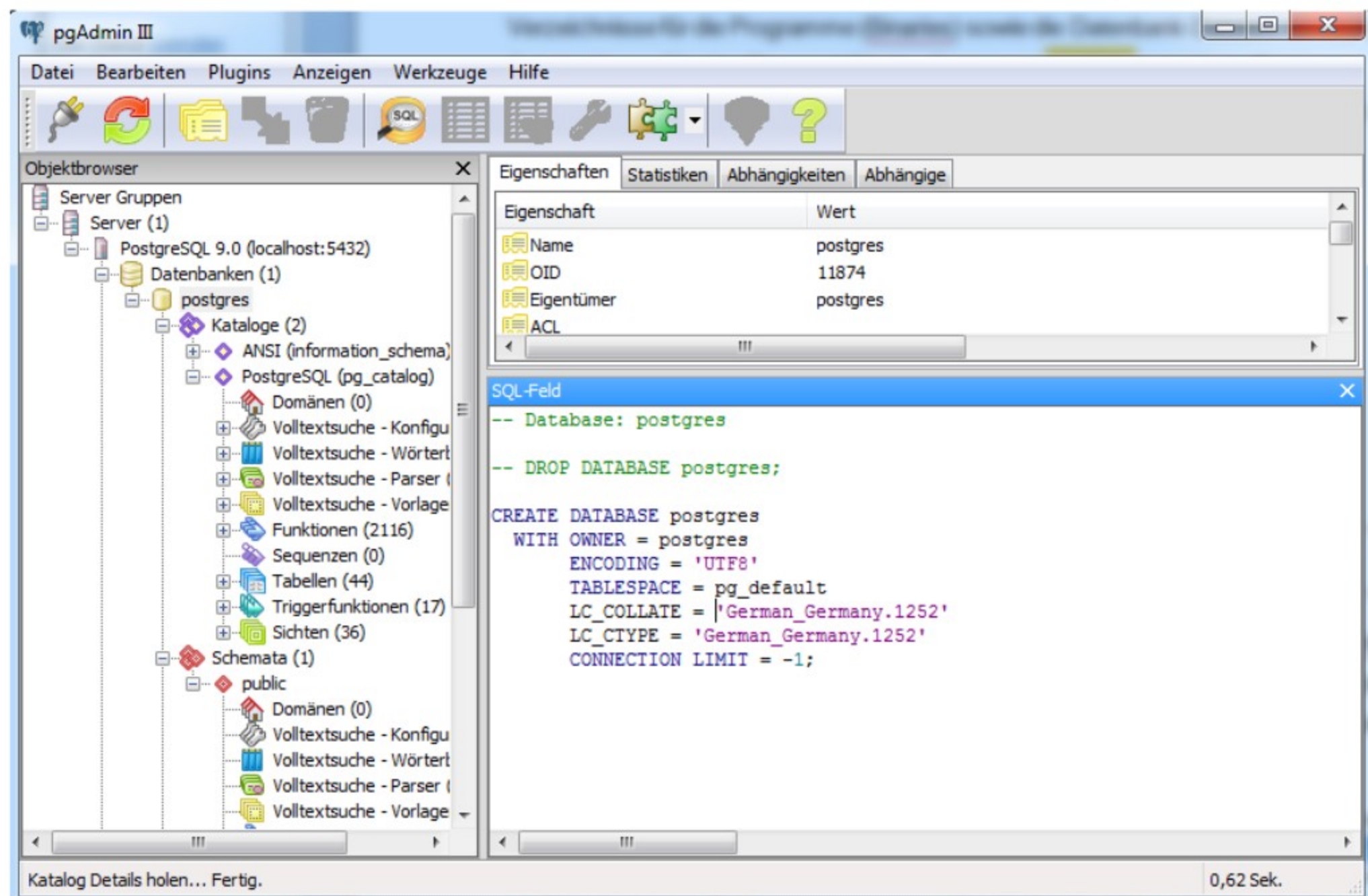


Bild 2.4 Das pgAdmin-Tool zur Verwaltung von PostgreSQL-Datenbanken

Die Paketinstallation ist nun abgeschlossen, und PostgreSQL kann benutzt werden.

■ 2.2 Installation aus dem Quellcode

Weil sich Unix und Windows diesbezüglich stark unterscheiden, stellen wir beide Vorgehensweisen getrennt dar.

2.2.1 Installation aus dem Quellcode unter Linux

Mit der Installation aus dem Quellcode können zusätzliche Optionen hinzugenommen werden. Wir installieren an dieser Stelle zunächst den Standard-Umfang und werden im weiteren Verlauf des Buches darauf hinweisen.

Das folgende Vorgehen beschreibt die Installation unter Red Hat Linux. Führen Sie die folgenden Schritte durch:

1. Laden Sie den Quellcode von der Webseite <http://www.postgresql.org> herunter. Die Datei hat das Format *postgresql-<version>.tar.gz*.

2. Entpacken Sie den Quellcode, und wechseln Sie in das Verzeichnis `postgresql-<version>`.

```
# gunzip postgresql-9.1.1.tar.gz
# tar -xvf postgresql-9.1.1.tar
```

3. Im ersten Schritt der Installation werden die Quellen für das aktuelle System konfiguriert. Das Skript „configure“ führt einige Tests aus, um die Werte einiger systemabhängiger Variablen zu bestimmen. Rufen Sie das Skript folgendermaßen auf:

```
# ./configure
```

4. Starten Sie anschließend den Build-Prozess. Es werden die Libraries und Binaries erstellt. Verwenden Sie das GNU Make Utility. Achten Sie auf die Erfolgsmeldung am Ende des Durchlaufs.

```
# make
. . .
All of PostgreSQL successfully made. Ready to install.
```

5. Der nächste Schritt betrifft die eigentliche Installation in die Verzeichnisse, die in Schritt 3 konfiguriert wurden. Damit die erforderlichen Zugriffsrechte vorhanden sind, sollte dieser Schritt ebenfalls unter dem Benutzer *root* ausgeführt werden. Standardmäßig erfolgt die Installation in das Verzeichnis `/usr/local/pgsql`.

```
# make install
. . .
PostgreSQL installation complete.
```

6. Damit ist die Installation abgeschlossen. In der Nachbereitung werden folgende Schritte durchgeführt:

- a) Einen Benutzer *postgres* anlegen:

```
# adduser postgres
```

- b) Ein Verzeichnis zum Speichern der Datenbankdateien erstellen:

```
# mkdir /usr/local/pgsql/data
# chown postgres /usr/local/pgsql/data
```

- c) Einen Datenbankserver erstellen:

```
# su - postgres
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
. . .
Success. You can now start the database server using:
    /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
or
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

- d) Den Datenbankserver starten. Verifizieren Sie, dass die Prozesse laufen:

```
$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
server starting
$ ps -ef|grep postg
postgres 21009      1  0 18:46 pts/1    00:00:00 /usr/local/pgsql/bin/
postgres
```



```
-D /usr/local/pgsql/data
postgres 21011 21009 0 18:46 ?      00:00:00 postgres: writer process
postgres 21012 21009 0 18:46 ?      00:00:00 postgres: wal writer process
postgres 21013 21009 0 18:46 ?      00:00:00 postgres: autovacuum
launcher
process
postgres 21014 21009 0 18:46 ?      00:00:00 postgres: stats collector
process
```

- e) Eine Testdatenbank mit dem Namen *hanser* anlegen. Überprüfen Sie die Verbindung zur Datenbank:

```
$ /usr/local/pgsql/bin/createdb hanser
$ /usr/local/pgsql/bin/psql hanser
psql (9.1.0)
Type "help" for help.
hanser=# select version();

                                version
-----
PostgreSQL 9.1.1 on x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC)
4.1.2
20080704 (Red Hat 4.1.2-48), 64-bit
(1 row)
```

Damit ist die Installation aus dem Quellcode erfolgreich abgeschlossen. Im folgenden Abschnitt wird die Installation unter Windows beschrieben.

2.2.2 Installation aus dem Quellcode unter Windows

Die Installation aus dem Quellcode auf einem Windows-Betriebssystem erscheint auf den ersten Blick etwas aufwendig, was jedoch eher am Handling der Compiler sowie des Windows-Plattformsupports liegt.

PostgreSQL kann mit Hilfe des Visual C++-Compilers von Microsoft unter Windows übersetzt und erstellt werden.

Voraussetzungen

Verwenden Sie am besten *Microsoft Visual Studio 2008 Express* für das Erstellen der Software aus dem Quellcode. Visual Studio 2010 wird für 64-Bit-Windows noch nicht unterstützt. Die Express-Edition ist kostenlos und kann von der Microsoft-Webseite heruntergeladen und installiert werden. Wenn Sie ein 64-Bit Windows verwenden, benötigen Sie zusätzlich das *Microsoft SDK* („Software Development Kit“ oder auch „Plattform Support“ genannt), da sich darin die für die Kompilierung und das Linken des Quellcodes erforderlichen 64-Bit-Komponenten befinden. Außerdem benötigen Sie einen Perl Interpreter, zum Beispiel ActiveState Perl: <http://www.activestate.com>.

Laden Sie zunächst die notwendigen Komponenten wie Visual Studio, Perl und, sofern erforderlich, das SDK von der Microsoft-Webseite herunter, und installieren Sie diese. Im folgen-

den Beispiel erfolgt die Installation unter Windows 7 auf einer 64-Bit-Plattform. Dazu wurde das SDK 7.01 für Windows 7 (64-Bit) installiert.

Entpacken Sie den heruntergeladenen PostgreSQL-Quellcode. Auch wenn die Datei mit dem Quellcode die Endungtar.gz besitzt, kann sie direkt unter Windows entpackt werden. Alternativ können Sie das Entpacken unter Unix/Linux vornehmen und die Verzeichnisstruktur nach Windows transferieren.

Öffnen Sie über das Windows Startmenü den „Command Prompt“ des Visual Studio, und wechseln Sie in das Verzeichnis mit dem PostgreSQL-Quellcode.



Hinweis

Für eine 64-Bit-Installation muss die Umgebung des Visual Studio auf die 64-Bit-Werkzeuge umgestellt werden. Die kann durch Aufruf des Skripts „vcvarsall.bat amd64“ erfolgen oder einfach durch Setzen der Pfad-Variablen:

```
set PATH=C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\
bin\amd64;%PATH%
```

Beim Build-Prozess wird dann automatisch erkannt, dass es sich um eine 64-Bit-Plattform handelt.

Mit den folgenden Schritten erfolgt die Installation:

1. Wechseln Sie in das Unterverzeichnis `src\tools\msvc`. Rufen Sie das Skript zum Erstellen der Binärprogramme auf:

```
C:\download\build\postgresql-9.1.0\src\tools\msvc> build
```

2. In diesem Schritt erfolgt die Installation. Geben Sie dazu das gewünschte Quellverzeichnis mit ein:

```
C:\download\build\postgresql-9.1.0\src\tools\msvc> install c:\postgresql
```

Damit ist die Installation abgeschlossen und die Software kann verwendet werden.

■ 2.3 Erste Schritte

Wie wir bereits während der Installation erwähnten, sollte PostgreSQL unter einem eigenen User Account im Betriebssystem laufen. So wurde unter Linux der Benutzer *postgres* angelegt und der PostgreSQL-Server unter diesem Account gestartet. Dem Benutzer gehören die Daten, welche durch den Server verwaltet werden. Andererseits können postgres-Daemons von anderen Accounts verwendet werden.

Dasselbe Sicherheitskonzept wird unter Windows praktiziert. Bei der Installation wurde ein lokaler Benutzer *postgres* angelegt. Unter diesem Benutzer wird der Windows-Dienst gestartet (siehe Bild 2.3).

Folgerichtig erfolgt das Starten und Stoppen des Servers durch den Benutzer `postgres`. Für den Start des Servers sowie für viele andere Kommandos benötigt man den Pfad zur Server-Instanz auf der Disk. Um ihn nicht ständig angeben zu müssen, kann er in der Umgebungsvariablen `PGDATA` gesetzt werden. Setzen Sie diese standardmäßig im Benutzerprofil von `postgres`. Setzen Sie auch den Pfad zu den Binärdateien.

Listing 2.1 Den Ausführungspfad sowie die Variable `PGDATA` setzen

```
export PATH=$PATH:/usr/local/pgsql/bin
export PGDATA=/usr/local/pgsql/data
```

Das Starten und Stoppen des Servers erfolgt mit dem Utility `pg_ctl`.

Listing 2.2 Den PostgreSQL-Server starten und stoppen

```
$ pg_ctl start
server starting
LOG: database system was shut down at 2011-04-24 10:06:34 CEST
LOG: autovacuum launcher started
LOG: database system is ready to accept connections
$ pg_ctl stop
waiting for server to shut down....LOG: received smart shutdown request
LOG: autovacuum launcher shutting down
LOG: shutting down
LOG: database system is shut down
done
server stopped
```

In einem Windows-Betriebssystem kann der Server entweder als Windows-Dienst über die Dienstverwaltung oder als normales Programm gestartet werden. Das Starten als Programm erfolgt wie unter Unix mit dem `pg_ctl` utility.

Nachdem der Server gestartet wurde, können Verbindungen zu den Datenbanken hergestellt werden. Verwenden Sie dazu das Kommandozeilen-Utility `psql`. Es wird von der Kommandozeile mit „`psql`“ aufgerufen und mit „`\q`“ verlassen.

Listing 2.3 Die Verbindung zur Datenbank mit `psql` herstellen

```
$ psql
psql (9.1.0)
Type "help" for help.
postgres=# SELECT current_database();
current_database
-----
postgres
(1 row)
postgres=# SELECT version();
version
-----
PostgreSQL 9.1.0 on x86_64-unknown-linux-gnu, compiled by GCC
gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-48), 64-bit
(1 row)
postgres=# \q
```


Aus Sicherheitsgründen ist der Zugriff von einem entfernten Server oder Client abgeschaltet. Der erste Versuch, von einem Client mit der SQL Shell auf den PostgreSQL-Server zuzugreifen, wird deshalb mit der folgenden Fehlermeldung scheitern:

```
Server [localhost]: 192.168.178.201
Database [postgres]:
Port [5432]:
Username [postgres]:
psql: konnte nicht mit dem Server verbinden: Connection refused
(0x0000274D/10061)
    Luft der Server auf dem Host "192.168.178.201" und akzeptiert er
    TCP/IP-Verbindungen auf Port 5432?
Drucken Sie eine beliebige Taste . . .
```

Damit entfernte Clients auf den Server zugreifen konnen, mussen folgende nderungen in der Konfiguration vorgenommen werden:

1. Stoppen Sie den PostgreSQL Server.
2. Wechseln Sie in das Datenverzeichnis (PGDATA). Im vorliegenden Beispiel ist das `/usr/local/pgsql/data`. Hier liegt die zentrale Konfigurationsdatei des Servers. ffnen Sie die Datei `postgresql.conf`,
3. und tragen Sie im Parameter `listen_addresses` den Hostname oder die IP-Adresse des Servers ein. Geben Sie die Portnummer des Servers an.

```
listen_addresses = '192.168.178.201'    # what IP address(es) to listen on;
                                         # comma-separated list of addresses;
                                         # defaults to 'localhost', '*' = all
                                         # (change requires restart)
port = 5432                             # (change requires restart)
```

4. ffnen Sie die Konfigurationsdatei fur „Client Authentication“ `pg_hba.conf`, die sich im selben Verzeichnis befindet. Fugen Sie den folgenden Eintrag hinzu, um eine Verbindung fur alle Clients auf alle Datenbanken zu gestatten. Sie soll uns an dieser Stelle helfen, den initialen Zugriff zu ermoglichen. Weitere Informationen finden Sie in Kapitel 5 „Sicherheit und berwachung“.

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
	host	all	all	0.0.0.0/0	trust

5. Starten Sie den PostgreSQL Server.

Jetzt konnen Sie sich von einem beliebigen Client aus zum Server verbinden. Beachten Sie den folgenden Hinweis.

Umgebungsvariablen

Wenn Sie mit dem `psql`-Utility eine Verbindung zu einem entfernten Server herstellen wollen, mussen Sie die Verbindungsinformationen mitgeben. Dies kann uber die folgenden Umgebungsvariablen erfolgen:

PGHOST	Name oder IP-Adresse des Servers
PGPORT	TCP/IP-Port des Servers
PGDATABASE	Name der Datenbank
PGUSER	Benutzername des Eigentumers des Server-Prozesses

Da jetzt die Verbindungen funktionieren, kann das pgAdmin-Tool auch für den entfernten Server verwendet werden. Wählen Sie im Menü *Datei* den Punkt *Server hinzufügen* aus, und geben Sie die erforderlichen Verbindungsparameter ein (siehe Bild 2.5).



Bild 2.5

Einen entfernten Server in das pgAdmin Tool aufnehmen

Wie Sie vielleicht bemerkt haben, werden die Server-Nachrichten nach stdout geschrieben. In der Standard-Konfiguration ist das Schreiben von Nachrichten in Log-Dateien ausgeschaltet. Ändern Sie die folgenden Parameter in der Konfigurationsdatei `postgresql.conf`, um ein minimales Mitschreiben von Server-Nachrichten und -Fehler einzuschalten. Starten Sie den Server durch, damit die Änderungen wirksam werden.

```
log_destination = 'stderr'          # Valid values are combinations of
                                     # stderr, csvlog, syslog, and eventlog,
                                     # depending on platform.  Csvlog
                                     # requires logging_collector to be on.
logging_collector = on              # Enable capturing of stderr and csvlog
                                     # into log files. Required to be on for
                                     # csvlogs.
                                     # (change requires restart)
log_directory = 'pg_log'           # directory where log files are written,
                                     # can be absolute or relative to PGDATA
```

Nach dem Neustart finden Sie unter PGDATA das Verzeichnis `pg_log`. Darin befindet sich die erste Log-Datei mit Nachrichten vom PostgreSQL-Server.

Listing 2.4 Die Logdatei des PostgreSQL-Servers auslesen

```
$ pwd
/usr/local/pgsql/data/pg_log
$ cat postgresql-2011-04-24_122251.log
LOG:  database system was shut down at 2011-04-24 12:20:52 CEST
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
. . .
```

Falls Sie bereits versucht haben, den Server mit dem Kommando „pg_ctl stop“ herunterzufahren, während noch Benutzer angemeldet waren, haben Sie festgestellt, dass sich der Server nicht stoppen lässt. Die Standard-Option für das Stoppen ist *smart*, das heißt, der Server wird erst gestoppt, wenn sich alle Clients abgemeldet haben. Mit der Option *fast* werden alle Verbindungen getrennt und die Datenbank sauber heruntergefahren.

Listing 2.5 Den PostgreSQL Server mit der Option „fast“ stoppen

```
$ pg_ctl stop -m fast
waiting for server to shut down.... done
server stopped
```

Tabelle 2.1 beschreibt die Optionen zum Stoppen des PostgreSQL-Servers.

Tabelle 2.1 Optionen zum Stoppen der PostgreSQL-Server

Option	Aktion
smart	Stoppen, wenn sich alle Clients abgemeldet haben
fast	Client-Verbindungen trennen und den Server sauber herunterfahren
immediate	Prozesse sofort beenden, ohne den Server sauber herunterzufahren Erfordert ein Recovery beim Neustart

Wenn Sie mit Oracle vertraut sind, werden Sie feststellen, dass bei PostgreSQL ähnliche Begriffe auftauchen, die jedoch ganz anders interpretiert werden. Tabelle 2.2 vergleicht die Shutdown-Optionen zwischen Oracle und PostgreSQL.

Tabelle 2.2 Vergleich der Shutdown-Optionen zwischen Oracle und PostgreSQL

PostgreSQL	Oracle
smart	normal
nicht vorhanden	transactional
fast	immediate
immediate	abort

Damit ist der PostgreSQL Server installiert und für eine erste Inbetriebnahme eingerichtet. Das folgende Kapitel beschäftigt sich detailliert mit der Konfiguration und Administration von Server und Datenbanken.

3

Server und Datenbank konfigurieren und administrieren

In der Datei *postgresql.conf* finden Sie alle Einstellungen für Server und Datenbanken. Die Standardwerte sind für einen niedrigen Ressourcen-Verbrauch ausgelegt und müssen in vielen Fällen den Anforderungen an die Datenbanken sowie den zur Verfügung stehenden Hardware-Ressourcen angepasst werden. Außerdem ist es notwendig, die Sicherheits-Einstellungen für die konkreten Bedürfnisse zu konfigurieren.

■ 3.1 Parameter-Einstellungen vornehmen

Standardmäßig finden Sie die Datei *postgresql.conf* im PGDATA-Verzeichnis. Mit der folgenden Abfrage kann die Lokation der Datei bestimmt werden:

```
postgres=# show config_file;
           config_file
-----
/usr/local/pgsql/data/postgresql.conf
```

Der Aufbau der Datei ist einfach. Die meisten Parameter sind mit dem Zeichen „#“ auskommentiert. In diesem Fall ist der Parameter nicht gesetzt, und falls vorhanden, greifen die Standardwerte. Weitere Konfigurationsdateien können mit dem Schlüsselwort „include“ eingebunden werden:

```
include '/usr/local/pgsql/wal_params.conf'
```

Alle Parameter und deren Werte sind auch in der Tabelle *pg_settings* gespeichert und können darüber abgefragt werden. Das Kommando „show“ liefert nur den aktuellen Wert. Die Abfrage der Tabelle liefert zusätzliche Informationen, wie zum Beispiel den Standard-Wert (*reset_val*).

Listing 3.1 Parameter abfragen

```
postgres=# show shared_buffers;
 shared_buffers
-----
32MB
postgres=# SELECT name,setting,reset_val FROM pg_settings
```



```
postgres=# WHERE name = 'shared_buffers';
      name      | setting | reset_val
-----+-----+-----
shared_buffers | 4096    | 4096
```

Es gibt Parameter, die sich dynamisch, also ohne Neustart des Servers ändern lassen. Welche Werte dynamisch änderbar sind, lässt sich durch die Spalte *context* in der Datei *pg_settings* abfragen. Dabei bedeutet:

- **user:** Jeder Benutzer darf den Wert für seine Session ändern.
- **superuser:** Nur der Administrator darf den Wert für den gesamten Server im laufenden Betrieb anpassen.
- **postmaster:** Die Werte müssen in der Datei *postgresql.conf* gespeichert oder beim Start des Servers mitgegeben werden. Änderungen werden nur durch den Neustart des Servers wirksam.
- **internal:** Es handelt sich um interne Parameter. Änderungen bedingen ein Rebuild des Servers oder müssen mit *initdb* konfiguriert werden.

Änderungen dynamischer Parameter können mit dem Kommando **SET** vorgenommen werden. Das Zurücksetzen auf den Standardwert erfolgt mittels **RESET**.

Listing 3.2 Dynamische Parameter setzen und zurücksetzen

```
postgres=# SET work_mem TO 2048;
SET
postgres=# SELECT name,setting,reset_val FROM pg_settings
postgres=# WHERE name = 'work_mem';
      name      | setting | reset_val
-----+-----+-----
work_mem       | 2048    | 1024
(1 row)
postgres=# RESET work_mem;
RESET
postgres=# SHOW work_mem;
work_mem
-----
1MB
```

Die Datei *postgresql.conf* ist in mehrere Sektionen eingeteilt. Die wichtigsten Parameter jeder Sektion sind in den folgenden Abschnitten beschrieben.

3.1.1 Parameter für System-Ressourcen

Die Parameter für System-Ressourcen finden Sie im Abschnitt „RESOURCE USAGE (except WAL)“. Einer der wichtigsten Parameter ist *shared_buffers*, in unserem Fall mit einem Standard-Wert von 32 MByte, also 4096 Buffer bei einer Blockgröße von 8 KByte. Der Parameter legt die Größe des Shared-Buffer-Pools einer PostgreSQL-Instanz fest. Die Einstellung gilt für den Datenbank-Cluster und wird von allen Datenbanken geteilt. Ein höherer Wert liefert Performance-Vorteile, insbesondere für schreiblastige Systeme. Die Empfehlung ist, 25 bis 33 % des verfügbaren Hauptspeichers zu verwenden. Weitere Informationen zu diesem Thema finden Sie in Kapitel 6 „Performance Tuning“.

Shared Memory-Konfiguration

In einigen Betriebssystemen, wie zum Beispiel Linux oder MacOS X, ist es notwendig, die standardmäßig eingestellte maximale Größe des Shared Memory zu erhöhen. Um genügend Shared Memory betriebssystemseitig zur Verfügung zu stellen, müssen gegebenenfalls die Werte für die Parameter *kernel.shmmax* und *kernel.shmall* in der Datei */etc/sysctl.conf* angepasst werden. Nehmen Sie bei Bedarf die folgenden Änderungen vor:

```
# Controls the maximum shared segment size, in bytes
kernel.shmmax = 68719476736
# Controls the maximum number of shared memory segments, in pages
kernel.shmall = 4294967296
```

Dieser Befehl macht die Änderungen persistent:

```
[root@serv1 ~]# sysctl -p
```

Der Parameter *temp_buffers* legt die Größe des Speichers für temporäre Tabellen fest. Der Standardwert ist 8 MByte, also 1024 bei einer Blockgröße von 8 KByte. Der Wert kann in einer Session dynamisch verändert werden, solange noch keine temporären Tabellen verwendet wurden.



Achtung!

Beachten Sie, dass dieser Buffer eine Obergrenze je Datenbankverbindung darstellt. Viele Sessions können damit einen erhöhten Speicherbedarf verursachen.

Die Größe des verfügbaren Hauptspeichers für Hash-Tabellen, Sortieroperationen oder Joins wird durch den Parameter *work_mem* festgelegt. Der Standard-Wert ist 1 MByte. Der Speicher ist als Maximalwert vergleichbar mit dem Parameter *temp_buffers*, allerdings wird dieser potenziell pro Operation verwendet. Auch hier gilt die Warnung, dass viele Sessions einen erhöhten Speicherbedarf verursachen können. Es gilt, einen Kompromiss zwischen Speicherbedarf und Performance zu finden.

Mit *maintenance_work_mem* wird die Obergrenze von Memory bestimmt, der von Verwaltungsprogrammen zum Erstellen und Ändern von Datenbank-Objekten verwendet wird. Der Standardwert ist 16 MByte. Der Parameter ist auf Session-Ebene dynamisch änderbar.

Mit Hilfe des Parameters *max_prepared_transactions* wird die maximale Anzahl von Transaktionen festgesetzt, die sich gleichzeitig im Status „prepared“ befinden können. Mit dem Wert „0“ (Standard) wird das Feature „Prepared Transactions“ Server-weit ausgeschaltet.

Obsolete Parameter

Die Parameter *max_fsm_pages* und *max_fsm_relations* wurden in der Version 9 entfernt und somit das Free Space Management vereinfacht.

3.1.2 Parameter für Write Ahead Log (WAL) und Checkpoints

Die Sektion ist in der Datei `postgresql.conf` mit „WRITE AHEAD LOG“ gekennzeichnet. Das WAL wird in anderen Datenbank-Systemen häufig als *Transaktionslog* bezeichnet. Informationen der Transaktionen werden in die Datei geschrieben, bevor die eigentlichen Änderungen in den Dateien der Datenbank stattfinden. Dies garantiert die Transaktions-sicherheit einer Datenbank und wird unter anderem beim Recovery herangezogen. So bestätigt man eine mit COMMIT abgeschlossene Transaktion vom Server erst, wenn die zugehörigen Informationen in das WAL geschrieben wurden.

PostgreSQL stellt eine Reihe von Parametern zur Verfügung, die das Verhalten beim Schreiben des WAL beeinflussen.

Ist der Parameter *fsync* auf „on“ gestellt (Standard), sichert der PostgreSQL-Server zu, dass alle Informationen der Transaktion auf die Disk geschrieben wurden. Dies geschieht durch Ausführung des System Calls *fsync()*, oder einer anderen, vom Parameter *wal_sync_method* festgelegten Methode. Damit kann PostgreSQL im Falle eines Crashes ein Recovery garantieren und einen konsistenten Zustand der Datenbanken wiederherstellen.



Achtung!

Durch das Setzen des Parameters auf „off“ erreicht man Performance-Vorteile. Allerdings ist die Wiederherstellbarkeit der Datenbanken nicht mehr garantiert (Datenkorruption). Setzen Sie deshalb den Parameter nur auf „off“, wenn die Wiederherstellung aus einer externen Quelle möglich und der Datenverlust akzeptabel ist.

Der Parameter *wal_sync_method* legt die Methode fest, mit der PostgreSQL den erfolgreichen Abschluss einer Transaktion bestätigt. Die möglichen Werte sind abhängig vom Betriebssystem, der Standard ist *fdatasync* (Verwendung des System Calls *fdatasync()* bei jedem COMMIT). Entnehmen Sie den Betriebssystem-spezifischen Informationen, was der System Call im gewählten Betriebssystem bewirkt.

Der Parameter *wal_level* bestimmt den Umfang der Informationen, die in das WAL geschrieben werden. Der Standardwert ist „minimal“. Damit werden nur Informationen geschrieben, die für das Recovery nach einem Crash oder Shutdown mit der Option „immediate“ benötigt werden. Soll das WAL archiviert und das Archiv für Recovery-Zwecke herangezogen werden, muss man den Parameter auf den Wert „archive“ setzen. Zur Unterstützung von Hot-Standby-Datenbanken muss der Wert auf „hot_standby“ gesetzt werden. Weitere Details zu diesem Thema finden Sie in Kapitel 7 („Hot Standby und Streaming Replication“). Zur Steigerung der Performance wird ein WAL Buffer im Shared Memory verwendet. Die Größe bestimmt der Parameter *wal_buffers*. Der Standardwert ist 64 KByte. Eine Vergrößerung des Buffers kann bei erhöhtem Aufkommen an Transaktionen notwendig sein.

Durch das Setzen des Parameters *synchronous_commit* auf „off“ wird ein asynchrones Schreiben von Transaktionen ermöglicht. Damit erfolgt die Bestätigung des erfolgreichen Transaktionsabschlusses (COMMIT) bereits, bevor die Informationen in das WAL geschrieben wurden. Der Abschluss von Transaktionen wird damit beschleunigt. Allerdings wird

durch die Bestätigung des Abschlusses der Transaktion nicht garantiert, dass alle Informationen in das WAL geschrieben wurden. Im Recovery-Fall werden solche Transaktionen möglicherweise zurückgerollt.

Über den Parameter *wal_writer_delay* lässt sich steuern, in welchen Abständen der WAL-Prozess den WAL Buffer auf Disk schreibt. Der Standardwert ist 200 ms. Im Fall eines hohen Transaktionsaufkommens kann eine Verkleinerung des Wertes erforderlich sein.

Mit dem Parameter *checkpoint_segments* wird die maximale Anzahl von Segmenten im WAL festgelegt. Jedes Segment besitzt eine Größe von 16 MByte. Der Standardwert ist „3“. Sind keine freien Segmente mehr vorhanden, beginnt PostgreSQL die vorhandenen neu zu beschreiben, und es wird ein Checkpoint ausgelöst. In einer Datenbank mit erhöhtem Transaktionsaufkommen kann es dadurch zu einer hohen Checkpoint-Frequenz kommen. Einstellungen von 12 bis 20 Segmenten sind nicht unüblich.

Crash Recovery

Beachten Sie, dass sich mit einer Vergrößerung des Parameters *checkpoint_segments* die Zeit für das Crash Recovery automatisch erhöht. Diese Situation kann jedoch mit dem folgenden Parameter beeinflusst werden.

Der Parameter *checkpoint_timeout* legt die maximale Zeit zwischen zwei Checkpoints fest. Der Standardwert ist 5 Minuten. Wurde der Parameter *checkpoint_segments* erhöht, kann mit diesem Parameter die Zeit für das Crash Recovery verkürzt werden. Auch für Systeme mit sehr geringer Aktivität lässt sich garantieren, dass ein regelmäßiger Checkpoint erfolgt. Mit *checkpoint_completion_target* wird die maximale Dauer eines Checkpoints festgelegt, anteilig zum Checkpoint-Intervall. Er steuert also die Intensität, mit der ein Checkpoint erfolgt, und damit die Systembelastung. Ein Wert von „0“ bedeutet, dass ein Checkpoint mit maximaler Intensität durchgeführt wird. Bei einem Wert von „1“ wird die Durchführung des Checkpoints über den gesamten Zeitraum eines Intervalls verteilt. Der Standardwert ist „0.5“.

In die Logdatei des PostgreSQL-Servers wird eine Warnmeldung geschrieben, falls das Intervall zwischen zwei Checkpoints kleiner als der Wert ist, der vom Parameter *checkpoint_warning* vorgegeben ist. Der Standardwert ist 30 Sekunden. Setzt man den Parameter auf „0“, wird keine Meldung in die Logdatei geschrieben.

3.1.3 Parameter für den Background Writer

Der Background Writer ist ein Server-Prozess des PostgreSQL-Clusters, mit dem Auftrag, „Dirty Buffer“ auf die Disk zu schreiben. Er hat eine wichtige Funktion für die Performance von Datenbank und Cluster. Der Parameter *bgwriter_delay* setzt die Aktivität des Prozesses fest. Der Standardwert von 200 Millisekunden bedeutet, dass der Writer alle 200 msec selbstständig aktiv wird. Unabhängig davon wird der Writer von anderen Ereignissen ausgelöst, zum Beispiel bei einem Checkpoint. Mit dem Parameter *bgwriter_lru_maxpages* wird vorgegeben, wie viele Buffer maximal bei jedem Aufruf auf die Disk geschrieben werden.

Der Standardwert ist 100 Buffer. Die minimale Anzahl von Dirty Buffer, die bei jedem Aufruf auf Disk geschrieben wird, ist abhängig von der Anzahl neuer Buffer, die in letzter Zeit benötigt wurden. Dieser Wert wird mit dem Wert des Parameters *bgwriter_lru_multiplier* multipliziert, um abzuschätzen, wie viele neue Buffer man benötigt, und insofern, wie viele Dirty Buffer mindestens geschrieben werden müssen. Der Standardwert ist „2.0“.

3.1.4 Parameter für Archivierung, Streaming Replication und Standby-Server

Parameter dieser Kategorie finden Sie in der Sektion „WRITE AHEAD LOG“. Mit dem Parameter *archive_mode* wird der Archiv-Modus ein- und ausgeschaltet. Den Archiv-Modus benötigt man, um die Datenbanken zu einem beliebigen Zeitpunkt wiederherstellen zu können (Point-in-time Recovery). Im Archive-Modus werden die WAL-Segmente vor dem automatischen Überschreiben in ein anderes Verzeichnis kopiert. Damit ist ein vollständiges Recovery seit der letzten Sicherung möglich. Der Standardwert ist „off“. Ein Beispiel zum Einschalten der Archivierung finden Sie in Abschnitt 3.1.10.

Der Befehl zur Sicherung der WAL-Segmente im Archiv-Modus wird im Parameter *archive_command* angegeben. Er kann ein Shell-Kommando oder auch ein Skript enthalten. Der Parameter ist nur wirksam, wenn *archive_mode* auf „on“ gestellt ist, andernfalls wird er ignoriert. Im Kommando selbst können zwei Platzhalter verwendet werden:

- %p: Ersetzt Pfad und Namen der zu archivierenden WAL-Datei. Der Pfadname wird relativ zum Verzeichnis PGDATA des Servers gebildet.
- %f: Stellt nur den Dateinamen (ohne Pfad) der zu archivierenden WAL-Datei dar.

PostgreSQL erwartet, dass der aufgerufene Befehl bei erfolgreicher Ausführung einen Exit-Status „0“ zurückliefert.

Das Kommando zur Archivierung wird erst ausgeführt, wenn ein WAL-Segment abgeschlossen ist. Für Systeme mit geringem Transaktionsaufkommen ist die Konsequenz, dass für eine lange Zeit keine Archivierung stattfindet. Im Falle eines Crashes mit Zerstörung der WAL-Segmente würde es zu einem größeren Datenverlust kommen, und das Point-in-time-Recovery ist somit nicht mehr gewährleistet. Mit Hilfe des Parameters *archive_timeout* wird ein Wechsel zum nächsten WAL-Segment in der vorgegebenen Zeit forciert. Die Angabe des Wertes erfolgt in Sekunden. Wird der Parameter zum Beispiel auf 30 Minuten gesetzt, erfolgt spätestens alle 30 Minuten ein WAL-Switch. Bei einem Crash, der den Verlust der WAL-Segmente einschließt, ist dann der Datenverlust nie größer als 30 Minuten.

Größe der Archivdatei

Beachten Sie, dass eine aufgrund des erreichten Timeouts vorzeitig archivierte WAL-Datei die volle Segment-Größe besitzt.

Die folgenden Parameter beziehen sich auf die neuen Features „Streaming Replication“ und „Hot Standby“. Detaillierte Informationen zu diesem Thema finden Sie in Kapitel 7 „Hot Standby und Streaming Replication“.

Der Parameter *max_wal_senders* bestimmt die maximale Anzahl von Sendeprozessen, mit denen sich jeweils ein Slave verbinden kann. Der Standardwert ist „0“. Der Wert von *wal_sender_delay* legt fest, wie lange jeder sendende Prozess wartet, bevor er die bis dahin angefallenen Daten an den Slave schickt. Der Standardwert ist 200 ms.

Bei der Übertragung und Einarbeitung der WAL-Informationen kann es zu Lücken kommen. Wenn dann der Standby-Server die Segmente erneut vom Primärserver anfordert, hat dieser die Dateien möglicherweise bereits gelöscht. Um das zu verhindern, kann der Parameter *wal_keep_segments* gesetzt werden. Der Parameter legt die minimale Anzahl von Segmenten fest, die im Verzeichnis „pg_xlog“ aufbewahrt werden, falls sie vom Standby-Server nachgefahren werden müssen. Der Standardwert ist „0“, das heißt: das Feature ist deaktiviert.

Schauen wir uns nun die Parameter auf der Standby-Seite an. Der Standby-Server befindet sich im Recovery-Modus und kann deshalb nicht wie ein normaler Server verwendet werden. Mit Aktivierung des Parameters *hot_standby* lässt sich die Datenbank gleichzeitig im Read-Only-Modus betreiben. Damit können SQL-Abfragen an die Datenbank gestellt werden. Der Standard ist „off“.

Wenn ein Hot-Standby-Server aktiviert ist, dann legt der Parameter *max_standby_archive_delay* fest, wie lange der Standby-Server warten soll, bevor er SQL-Abfragen abbricht, die einen About-to-be-applied-Konflikt aufweisen. Analog dazu gibt es den Parameter *max_standby_streaming_delay*. Weitere Details finden Sie in Kapitel 7 („Hot Standby und Streaming Replication“).

3.1.5 Parameter für Verbindung und Authentifizierung

Die Parameter für Verbindung und Authentifizierung befinden sich in der Sektion „CONNECTIONS AND AUTHENTICATION“. Mit dem Parameter *listen_addresses* wird der Servername oder die IP-Adresse festgelegt, auf den der PostgreSQL-Server auf Verbindungsanfragen von Clients hört. Es können mehrere, durch Komma getrennte Werte angegeben werden. Der zugehörige TCP/IP-Port wird im Parameter *port* angegeben. Der Standardwert ist 5432. Der Parameter *max_connections* bestimmt die maximale Anzahl von gleichzeitigen Verbindungen zum PostgreSQL-Server mit einem Standardwert von 100.



Praxistipp

Wenn Sie einen Standby-Server einsetzen, muss dort der Wert für *max_connections* größer oder gleich dem Wert auf dem Primärsystem sein. Andernfalls können auf dem Standby-Server keine Abfragen ausgeführt werden.

Mit Setzen des Parameters *superuser_reserved_connections* lassen sich Verbindungen für Superuser reservieren. Ist die Anzahl von *max_connections* minus *superuser_reserved_connections* erreicht, werden nur noch Verbindungen für Superuser aufgebaut. Der Standardwert ist „3“.

Die folgenden Parameter beziehen sich auf Sicherheitsaspekte für die Verbindung zwischen Client und Server. Der Parameter *authentication_timeout* legt die maximale Zeit fest, in der ein Authentifizierungsprozess abgeschlossen werden muss. Der PostgreSQL-Server bricht die Verbindung ab, wenn ein Abschluss des Vorgangs innerhalb der vorgegebenen Zeit nicht erfolgt. Der Standardwert ist eine Minute. Mit dem Parameter *ssl* können SSL-Verbindungen zugelassen werden. Er ist standardmäßig ausgeschaltet.

Ein Passwort kann alternativ verschlüsselt oder unverschlüsselt mit dem Befehl „CREATE USER“ angelegt werden. Dies wird über den Parameter *password_encryption* gesteuert. Die Verschlüsselung ist standardmäßig eingeschaltet.

Der Parameter *db_user_namespace* legt fest, ob Benutzernamen pro Datenbank festgesetzt werden. Der Standardwert ist „off“. Wird er eingeschaltet, müssen die Benutzernamen zusammen mit dem Datenbanknamen angesprochen werden. Das Format ist <user>@<database>.

Weitere Informationen zum Thema Authentifizierung finden Sie in Abschnitt 3.3.

3.1.6 Parameter für Client-Standards

Die Anmeldung eines Clients erfolgt über den PostgreSQL-Server, der die Authentifizierung vornimmt, an eine Datenbank. Damit stellt sich die Frage, auf welche Schemas sich die Anweisungen des Clients beziehen, wenn keine weiteren Angaben vorgenommen werden. Der Parameter *search_path* legt fest, in welcher Reihenfolge die Schemas für die Bezugnahme auf ein Objekt durchsucht werden sollen. Gibt es Objekte mit identischem Namen in verschiedenen Schemas, wird das zuerst Gefundene referenziert.

Im Parameter können mehrere, durch Komma getrennte Werte angegeben werden. Der Standardwert ist „*\$user*“,*public*. „*\$user*“ ist der Benutzername, mit dem der Client am Server angemeldet ist; er kann durch folgende Abfrage ermittelt werden:

```
(postgres@[local]:5432) [postgres] > SELECT session_user;
 session_user
-----
 postgres
```

Das Schema mit dem Systemkatalog „pg_catalog“ wird immer durchsucht, auch wenn es nicht im Parameter angegeben ist. Das Anlegen von Objekten erfolgt im ersten Schema des Parameters „*search_path*“. Da in unserem Fall kein Schema mit dem Namen „postgres“ existiert, werden die Objekte im Schema „public“ angelegt.

Der Parameter *default_tablespace* legt den Tablespace fest, in dem Tabellen und Indexe erstellt werden, wenn beim Anlegen explizit kein Tablespace angegeben wird. Der Standardwert ist ein leerer String. Ein leerer String bedeutet: es soll der Default-Tablespace der Datenbank verwendet werden. Analog legt der Parameter *temp_tablespace* den Default für den Temporary Tablespace fest.

Mit dem Parameter *statement_timeout* kann eine Obergrenze für die Laufzeit von SQL-Anweisungen in Millisekunden festgelegt werden. Alle SQL-Anweisungen, die länger laufen, werden abgebrochen. Mit dem Standardwert „0“ wird das Feature abgeschaltet.

3.1.7 Parameter für VACUUM

Vacuum-Operationen untersuchen Tabellen auf gelöschte oder aktualisierte Zeilenversionen und geben den dabei entstandenen freien Speicherplatz frei. Die Parameter zu diesem Thema finden Sie in der Sektion „AUTOVACUUM PARAMETERS“. Der Parameter *autovacuum* legt fest, ob der Autovacuum Daemon gestartet und die Features „Autovacuum“ und „Analyze“ aktiviert werden sollen. Der Standardwert ist „on“. Damit Autovacuum funktioniert, muss der Parameter *track_counts* ebenfalls eingeschaltet werden. Auch hier ist „on“ der Standardwert.

Mit *autovacuum_max_workers* wird die maximale Anzahl von Autovacuum-Prozessen, die parallel laufen können, festgelegt. Der Standardwert ist „3“.

Mit dem Parameter *autovacuum_naptime* wird der minimale Abstand zwischen zwei Läufen vorgegeben. In jedem Lauf werden die Datenbanken von den Vacuum Daemons untersucht und Vacuum- und Analyze-Befehle abgesetzt. Der Standardwert ist eine Minute.

Der Parameter *log_autovacuum_min_durations* legt fest, ob und mit welchem Niveau Loginformationen über die Autovacuum-Operationen aufgezeichnet werden. Mit dem Standardwert von „-1“ erfolgt keine Aufzeichnung. Setzt man den Wert auf „0“, werden alle Aktionen und deren Laufzeiten registriert. Ein Wert größer als „0“ wird als Zeit in Millisekunden interpretiert. Alle Vacuum- und Analyze-Operationen, die länger als vorgegeben laufen, werden aufgezeichnet.

Detaillierte Informationen zu den Vacuum- und Analyze-Features finden Sie in Abschnitt 3.7.

3.1.8 Parameter für SQL Tuning

PostgreSQL verfügt über eine Reihe von Parametern, die das Verhalten des Query Optimizer und somit den Ausführungsplan beeinflussen. Es gibt so genannte Kosten-Parameter, mit denen sich die Kosten von SQL-Anweisungen beeinflussen lassen.

Der Parameter *seq_page_cost* legt die Kosten für das Lesen einer Page von der Disk für sequenzielle Leseoperationen fest. Der Standardwert ist „1.0“. Werte kleiner als „1“ reduzieren die Kosten für das Lesen. Analog funktioniert der Parameter *random_page_cost* für nicht-sequenzielle Leseoperationen mit einem Standardwert von „4.0“. Eine Verkleinerung des Wertes macht die Verwendung von Indexen attraktiver. Bei einer Vergrößerung des Wertes im Verhältnis zu „set_page_cost“ werden Index-Scans und Table-Scans günstiger.



Achtung!

Gehen Sie sensibel mit den Parametern „seq_page_cost“ und „random_page_cost“ um. Sie haben Einfluss auf das Verhalten des Query Optimizer für alle Datenbanken des Cluster. Ein umfangreicher Test der Auswirkungen ist erforderlich.

Der Parameter *cpu_tuple_cost* bestimmt die Kosten für die Verarbeitung eines Satzes. Der Standardwert ist „0.01“. Analog dazu legt *cpu_index_tuple_cost* die Kosten für die Verarbei-

tung eines Index mit dem Standardwert von „0.05“ fest. Der Wert des Parameters *effective_cache_size* überschreibt die Annahme des Optimizer über die Größe des Disk Cache für eine einzelne SQL-Anweisung. Dieser Wert hat indirekt Einfluss auf die Benutzung von Indexen. Der Standardwert ist 128 MByte.

In Tabelle 3.1 finden Sie Parameter, mit deren Hilfe die Verwendung bestimmter Ausführungspläne ein- und ausgeschaltet werden kann.

Tabelle 3.1 Parameter für Ausführungspläne

Parameter	Aktivierung von...	Standard
enable_hashjoin	Hash Joins	on
enable_nestloop	Nested Loop Joins	on
enable_mergejoin	Merge Joins	on
enable_bitmapscan	Bitmap Scans	on
enable_hashagg	Hash Aggregation	on
enable_seqscan	Sequenziellen Table und Index Scans	on
enable_sort	Explizite Schritte für Sortierungen	on
enable_tidscan	TID Scans	on
enable_material	Materialization	on

Alternativ zum Cost Based Optimizer kann der Genetic Query Optimizer(GEQO) verwendet werden. Er erstellt die Ausführungspläne nach bestimmten Regeln ohne Berücksichtigung der Kosten. So lässt sich die Zeit für das Erstellen von komplexen Plänen reduzieren. Der GEQO ist standardmäßig eingeschaltet und kann über den Parameter *geco* ausgeschaltet werden. Mit dem Parameter *geco_threshold* wird eine Grenze festgelegt, ab der der GEQO verwendet werden soll. Dabei handelt es sich um die Anzahl der in der FROM-Klausel verwendeten Objekte. Bei einem Standardwert von „12“ wird der GEQO erst aktiv, wenn mindestens 12 Tabellen verbunden werden. Über den Parameter *geco_effort* kann man einen weiteren Schwellwert einstellen. Dieser legt das Verhältnis zwischen der Zeit für die Erstellung eines Ausführungsplans und der Qualität des Plans fest. Der Wert kann auf einer Skala von „1“ bis „10“ liegen, der Standardwert ist „5“. Ein größerer Wert verlängert die Zeit für die Erstellung des Ausführungsplans, aber auch die Wahrscheinlichkeit von Effizienz.

Detaillierte Informationen zu diesem Thema finden Sie in Kapitel 10 („Optimierung von SQL-Anweisungen“).

3.1.9 Parameter für Logging und Tracing

Die Parameter für Logging und Tracing befinden sich in der Sektion „ERROR REPORTING AND LOGGING“. Sie lassen sich in folgende Kategorien unterteilen:

- Wo sollen die Logdateien liegen?
- Wann soll eine Aufzeichnung stattfinden?
- Was soll aufgezeichnet werden?

3.1.9.1 Wo sollen die Logdateien liegen?

Der Parameter *log_destination* definiert den Ausgabeort für Nachrichten vom PostgreSQL-Server. Mögliche Werte sind „stderr“, „csvlog“ und „syslog“. In einem Windows-Betriebssystem kann zusätzlich der Wert „eventlog“ für die Ausgabe in das Windows-Eventlog angegeben werden. Die Angabe mehrerer, durch Komma getrennter Werte ist möglich.

Setzt man den Parameter *logging_collector* auf „on“, werden die nach stderr versandten Nachrichten in eine Logdatei umgeleitet. Dann lässt sich mit dem Parameter *log_directory* das Verzeichnis festlegen, in dem die Logdatei angelegt wird. Standardmäßig erfolgt das im Verzeichnis „pg_log“ unter \$PGDATA. Ist *logging_collector* eingeschaltet, legt der Parameter *log_filename* den Namen der Logdatei fest. Es kann ein Muster wie im folgenden Beispiel angegeben werden, um Datum und Uhrzeit im Namen unterzubringen:

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

Mit Hilfe von *log_rotation_age* und *log_rotation_size* werden die maximale Lebenszeit und die maximale Größe der Logdatei festgelegt, bevor eine neue Datei geschrieben wird.

3.1.9.2 Wann soll die Aufzeichnung stattfinden?

In diesem Abschnitt finden Sie Parameter, die festlegen, für welche Fälle eine Aufzeichnung erfolgen soll. So legt der Parameter *client_min_messages* die Stufe von Nachrichten fest, die an den Client gesendet werden. Mögliche Werte sind DEBUG1 bis DEBUG5, LOG, NOTICE, WARNING, ERROR, FATAL und PANIC. Der Standardwert ist NOTICE. Analog dazu legt *log_min_messages* die Nachrichtenstufe für die Einträge in das Server-Log fest. Die möglichen Werte sind identisch mit denen des Parameters „client_min_messages“, allerdings ist hier der Standardwert „WARNING“.

Der Parameter *log_min_error_statement* legt fest, welche SQL-Anweisungen, die eine Fehlerbedingung ausgelöst haben, in das Server-Log geschrieben werden. Die möglichen Parameterwerte sind identisch mit den beiden Parametern davor, wobei der Standardwert ERROR ist. Das Ausschalten des SQL-Logging erfolgt mit dem Wert PANIC. Mit dem Parameter *log_min_duration_statement* wird die Laufzeit von SQL-Anweisungen festgelegt, ab der diese in das Server-Log eingetragen werden. Damit können zum Beispiel Langläufer identifiziert werden. Bei einem Wert von „0“ werden alle Anweisungen eingetragen. Mit dem Standardwert „-1“ ist das Logging ausgeschaltet.

3.1.9.3 Was soll aufgezeichnet werden?

Mit dem Parameter *application_name* können Namen von Applikationen vorgegeben werden, für die eine Aufzeichnung im Server-Log erfolgt. Der Applikationsname wird auch in der Spalte „application_name“ im View „pg_stat_activity“ angezeigt.

In Tabelle 3.2 finden Sie eine Reihe von Parametern, die ein- und ausgeschaltet werden können („on“, „off“). Sie legen fest, ob die entsprechende Komponente in das Logging eingebunden werden soll.

Tabelle 3.2 Liste von Logging-Parametern

Parameter	Standard	Aufzeichnung von ...
log_checkpoints	off	... allen Checkpoints. Einige Statistiken des Checkpoints wie zum Beispiel die Anzahl der Buffer und die Länge werden mit ausgewiesen.
log_connections	off	... Verbindungsversuchen zum PostgreSQL-Server. Es werden sowohl erfolgreiche als auch erfolglose Versuche registriert.
log_disconnections	off	... Abmeldungen von einer Session. Die Dauer der Verbindung wird mit ausgewiesen.
log_duration	off	... Ausführungszeit einer jeden SQL-Anweisung.
log_hostname	off	... Hostnamen zusätzlich zur IP-Adresse.
log_lock_waits	off	... Informationen über Sessions, die länger als die im Parameter „deadlock_timeout“ spezifizierte Zeit auf eine Sperre warten.

Der Parameter *log_statement* legt fest, welche SQL-Anweisungen aufgezeichnet werden sollen. Gültige Parameterwerte sind: „none“, „ddl“, „mod“ und „all“. Beachten Sie, dass mit dem Setzen einiger Werte ein intensives Schreiben in die Logdatei hervorgerufen werden kann.

**Hinweis**

Anweisungen mit einfachen Syntax-Fehlern werden nicht aufgezeichnet. Verwenden Sie für solche Fälle den Parameter „log_min_error_statement“.

3.1.10 Benutzerspezifische Parameter

Unter dem Parameter *custom_variable_classes* können Klassennamen für benutzerspezifische Parameter festgesetzt werden. Im Beispiel in Listing 3.3 werden Parameter für das Modul „pg_stat_statements“ gesetzt.

Listing 3.3 Benutzerspezifische Parameter setzen

```
custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

3.1.11 Parameter für Entwickler

PostgreSQL ist eine Open-Source-Datenbank und somit stark auf den Input der Entwicklergemeinschaft angewiesen. Speziell für die Arbeit von Entwicklern wurden deshalb einige Parameter vorgesehen. Sie sollten nicht für produktive Datenbanken eingesetzt werden.

Wenn Sie als PostgreSQL-Entwickler die Funktionalität zu verbessern oder zu erweitern trachten, bieten diese Parameter eine wichtige Unterstützung. Im Folgenden finden Sie eine Auswahl.

Mit dem Einschalten von *allow_system_table_mods* kann die Struktur von Systemtabellen verändert werden. Die Verwendung des Parameters *post_auth_delay* gibt dem Entwickler die Möglichkeit, sich mit einem Debugger in einen Server-Prozess einzuklinken. Wenn Sie den Parameter *trace_notify* einschalten, werden Debug-Informationen der LISTEN- und NOTIFY-Kommandos in die Logdateien geschrieben. Analog können Sie Informationen über das Locking-Verhalten mit dem Parameter *trace_locks* einschalten. Der Parameter *wal_debug* schaltet die Aufzeichnung von Debug-Informationen von WAL-Operationen ein und aus. Voraussetzung ist, dass das Modul WAL_DEBUG in den Build-Prozess des PostgreSQL-Servers mit eingebunden wurde.

■ 3.2 Die WAL-Archivierung einschalten

Die Archivierung der Write-Ahead-Log-Dateien (WAL Archiving) ist zwingende Voraussetzung, um ein Point-in-time-Recovery der Datenbanken zu garantieren. Die Art und Weise der Archivierung von WAL-Dateien ist in PostgreSQL sehr flexibel. Es kann im Parameter „archive_command“ ein beliebiger Shell-Befehl eingestellt werden. Auch der Aufruf eines vorgefertigten Skriptes ist möglich. Die einfachste Form ist die Verwendung eines Kopierbefehls mit Sicherung in ein lokales Verzeichnis oder ein NFS-Dateisystem. Aber auch das Kopieren auf einen anderen Server mit einem SCP-Befehl ist ein gangbarer Weg. Damit befinden sich die Archiv-Dateien auf einem entfernten Server und stehen auch nach einem Crash des Datenbankservers für einen Recovery-Prozess zur Verfügung.

Netzwerk-Performance

Beachten Sie, dass bei einer direkten Archivierung über das Netzwerk (SCP-Befehl oder Kopieren auf NFS) die erforderliche Performance zur Verfügung steht. Andernfalls kann es bei einem hohen Transaktionsaufkommen zu einem Rückstau kommen. In solchen Fällen ist es besser, die WAL-Segmente zuerst in ein lokales Dateisystem zu archivieren und in einem nachgelagerten Prozess auf ein externes Medium zu verschieben.

Der PostgreSQL-Server hat zwei Anforderungen an das Verhalten des Befehls im Parameter „archive_command“:

- Vorhandene Dateien dürfen nicht überschrieben werden.
- Es muss der Status „0“ zurückgeliefert werden, wenn die Datei erfolgreich kopiert wurde. Andernfalls wird ein Status ungleich „0“ erwartet.

In Unix-Betriebssystemen ist die einfachste Version ein cp-Befehl mit der Option „-i“. Damit erfolgt eine Abfrage vor dem Überschreiben einer Datei. Mit einer automatischen Eingabe „</dev/null“ wird diese Abfrage negativ beantwortet. Zur Vereinfachung stellt PostgreSQL

die Platzhalter „%f“ und „%p“ zur Verfügung. „%f“ ersetzt den Dateinamen der zur archivierenden WAL-Datei und „%p“ den kompletten Namen einschließlich Pfad. Dabei handelt es sich um den relativen Pfad unter dem PGDATA-Verzeichnis. Standardmäßig befinden sich die WAL-Dateien im Verzeichnis *pg_xlog*. Ein einfacher Befehl zur Archivierung in ein lokales Dateisystem würde dann so aussehen:

```
cp -i %p /opt/postgresql/archive </dev/null
```

Schlägt das Archivierungskommando fehl (Exit-Status ungleich „0“), dann führt PostgreSQL den Befehl erneut aus, bis ein erfolgreiches Ergebnis zurückkommt. Das Kommando wird durch dem Benutzer ausgeführt, mit dem der PostgreSQL-Server gestartet wurde.

Führen Sie die folgenden Schritte durch, um den Archivelog-Modus zu aktivieren:

1. Ändern Sie den Parameter `archive_command` in der Datei `postgresql.conf`.

```
archive_command = 'cp -i %p /opt/postgres/archive/%f </dev/null'
```

2. Setzen Sie den Parameter zum Einschalten der Archivierung.

```
archive_mode = on
```

3. Damit ist der Archiv-Modus eingeschaltet; allerdings muss noch ein wichtiger Parameter gesetzt werden. Standardmäßig steht „`wal_level`“ noch auf „minimal“. Das heißt: es werden nur die für ein Crash Recovery erforderlichen Informationen in die WAL-Segmente geschrieben. Ein Point-in-time-Recovery unter Hinzunahme der Archivdateien würde nicht funktionieren. Der Parameter muss deshalb auf „archive“ gesetzt werden.

```
wal_level = archive
```

4. Führen Sie einen Restart des Servers durch, um den Archiv-Modus zu aktivieren.

```
$ pg_ctl restart -m fast
waiting for server to shut down..... done
server stopped
server starting
```

Damit ist der Archiv-Modus eingeschaltet, und Sie sehen bereits archivierte Segmente im Archive-Verzeichnis. An dieser Stelle noch ein Hinweis zum Parameter „`archive_timeout`“. Das Setzen des Timeouts bewirkt, dass spätestens nach Erreichen des Wertes ein Switch der WAL-Segmente und somit eine Archivierung erfolgt. Die Archiv-Datei besitzt die volle Größe, auch wenn das WAL-Segment nur teilweise gefüllt ist. Andererseits wird durch die externe Sicherung das Point-in-time-Recovery bis zu einem bestimmten Zeitpunkt garantiert. Gesucht ist also ein Kompromiss zwischen dem Aufkommen an Archivdateien und der Wiederherstellbarkeit. Das folgende Setup ist eine gute Lösung für diesen Konflikt.



Praxistipp WAL-Archivierung

WAL-Dateien verfügen über eine hohe Komprimierungsrate. Mit einer Komprimierung während der Archivierung lässt sich also viel Platz sparen. Sie müssen nur dafür sorgen, dass genügend CPU-Ressourcen zur Verfügung stehen. Damit kann der Konflikt zwischen einem kleinen Wert für „`archive_timeout`“ und erhöhtem Verbrauch von Disk-Kapazität durch die Archiv-Dateien weitgehend gelöst werden.

Setzen Sie den Parameter „archive_command“ wie folgt, um die Archiv-Dateien zu komprimieren:

```
archive_command = 'test ! -f /opt/postgres/archive/%f.gz && gzip -c %p > /opt/postgres/archive/%f.gz'
```

In einem Windows-Betriebssystem können die entsprechenden Shell-Befehle analog angewandt werden. Zu beachten sind die Einschränkungen der Shell im Vergleich zu Unix-Systemen.

Listing 3.4 Beispiel für den Parameter „archive_command“ für Windows-Systeme

```
archive_command = 'copy "%p" "C:\\PostgreSQL\\9.0\\archive\\%f"'
```

■ 3.3 Authentifizierung von Clients

Authentifizierung ist der Prozess, in dem der PostgreSQL-Server die Identität des Clients sowie seine Berechtigungen ermittelt. PostgreSQL stellt eine Reihe verschiedener Authentifizierungsmethoden zur Verfügung. Die Benutzernamen der Datenbanken sind dabei logisch von den Benutzernamen des Betriebssystems getrennt. So kann ein PostgreSQL-Server verschiedene Datenbank-Benutzer kennen, die über keinen Account im Betriebssystem verfügen. Mit anderen Worten: PostgreSQL verfügt über eine eigene Benutzerverwaltung.

Die Authentifizierung von Clients wird über die Konfigurationsdatei *pg_hba.conf* kontrolliert, die sich im Verzeichnis „\$PGDATA“ befindet. Mit der Initialisierung des Servers wird eine Standarddatei erstellt. Jede Zeile repräsentiert einen Verbindungstyp, einen Bereich von IP-Adressen der Clients, die Datenbanknamen, die Benutzernamen sowie die Authentifizierungsmethode. Tabelle 3.3 beschreibt die Optionen für die erste Spalte, den Verbindungstyp.

Tabelle 3.3 Verbindungstypen für die Authentifizierung von Clients

Verb.-Typ	Bedeutung
local	Verbindungen über die Sockets der Unix-Domain. Ermöglicht eine lokale Verbindung auf dem Datenbank-Server.
host	Für Verbindungsanfragen über TCP/IP. Die Verbindungen können vom Typ SSL oder nicht-SSL sein.
hostssl	Verbindungen über TCP/IP, jedoch nur mit SSL-Verschlüsselung.
hostnonssl	Verbindungen über TCP/IP, die nicht mit SSL verschlüsselt sind.

Die Datei „pg_hba.conf“ ist so weit vorkonfiguriert, dass lokale Verbindungen für alle Datenbanken und alle Benutzer generell zugelassen werden. Dies bewirkt der folgende Eintrag:

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
local		all	all		trust

Ist der Eintrag nicht vorhanden (oder auskommentiert), dann ist eine lokale Verbindung, zum Beispiel mit „psql“, wie im folgenden Fall nicht mehr möglich:

```
$ psql
psql: FATAL: no pg_hba.conf entry for host "[local]", user "postgres",
database "hanser"
```

Die überwiegende Anzahl von Verbindungen erfolgt von entfernten Clients über TCP/IP. Mit dem folgenden Eintrag werden alle Verbindungen zugelassen:

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host		all	all	0.0.0.0/0	trust



Hinweis

Beachten Sie, dass eine Verbindung von einem entfernten Client über TCP/IP erst möglich ist, wenn der Parameter „listen_address“ in der Datei „postgresql.conf“ gesetzt wurde. Dieser aktiviert den Listener, der für die Authentifizierung von einem entfernten Client unbedingt erforderlich ist.

In der Spalte „database“ wird festgesetzt, für welche Datenbanken die Verbindung gelten soll. Analog können Benutzernamen in der Spalte „user“ angegeben werden. Für beide Parameter kann man den Wert „all“ angeben. Damit werden alle Datenbanken oder Benutzer zugelassen.

Die Spalte „CIDR-address“ legt den Bereich von Client-IP-Adressen fest, für die Verbindungen zugelassen werden. Der Wert besteht aus zwei Teilen: einer gültigen IP-Adresse sowie der Anzahl durch „/“ getrennter „High Order Bits“. Der Eintrag „0.0.0.0/0“ repräsentiert alle IP-Adressen.

Die letzte Spalte stellt die Authentifizierungsmethode dar. Die wichtigsten Optionen sind in Tabelle 3.4 zusammengefasst.

Tabelle 3.4 Auswahl der wichtigsten Optionen der Authentifizierungsmethode

Option	Beschreibung
trust	Verbindungen werden ohne weitere Einschränkungen zugelassen. Es erfolgt keine Authentifizierung und es wird kein Passwort abgefragt.
reject	Verbindungsanfragen werden ohne weitere Prüfung abgewiesen.
md5	Es wird ein Passwort mit MD5-Verschlüsselung verlangt.
password	Es wird ein nicht verschlüsseltes Passwort verlangt.
ident	Es erfolgt eine Abfrage des Benutzernamens im Betriebssystem des Clients. Verbindungen werden gestattet, wenn dieser mit dem Benutzernamen der Datenbank identisch ist.
ldap	Die Authentifizierung erfolgt über einen LDAP-Server.
cert	Die Authentifizierung erfolgt mit Hilfe von SSL-Zertifikaten.



Hinweis

Die Konfigurationsdatei „pg_hba.conf“ wird beim Start des PostgreSQL-Servers gelesen. Wenn Sie Änderungen in der Datei vornehmen, muss der Server neu gestartet oder die Konfiguration mit dem Kommando `pg_ctl reload` neu geladen werden.

Wenn Sie externe Authentifizierungsmethoden verwenden, muss der Benutzername im Betriebssystem nicht mit dem Benutzer in der Datenbank identisch sein. Diese Namen können über die Datei `pg_ident.conf` zugewiesen werden.

Die am häufigsten angewandten Authentifizierungsmethoden sind „password“ und „md5“. Beide unterscheiden sich nur in der Art und Weise, wie das Passwort über das Netzwerk gesendet wird. Bei „password“ erfolgt dies als Klartext, und bei „md5“ wird eine Verschlüsselung vorgenommen. Beachten Sie, dass das Passwort im Betriebssystem nicht identisch mit dem Passwort des Benutzers in der Datenbank ist. Das Passwort in der Datenbank wird in der Tabelle „pg_authid“ gespeichert.

Listing 3.5 Das Passwort in der Datenbank

```
(postgres@[local]:5432) [hanser] > SELECT rolname,rolpassword
> FROM pg_authid;
rolname |          rolpassword
-----+-----
postgres | md53175bce1d3201d16594cebf9d7eb3f9d
```

Beachten Sie an dieser Stelle, dass das verschlüsselte Passwort nur mit bestimmten Privilegien (Superuser) aus der Datenbank ausgelesen werden kann. Das Speichern des verschlüsselten Passworts stellt somit kein Sicherheitsrisiko dar.



Praxistipp

Die Methode „trust“ ist unsicher und sollte für produktive Datenbanken nicht verwendet werden. Jeder Benutzer, der sich am PostgreSQL-Server anmeldet, einschließlich der Superuser, erhält ohne weitere Authentifizierung Zugriff.

■ 3.4 Einen Connection Pool aufsetzen

Ein Connection Pool bietet den Vorteil, dass eine bestimmte Anzahl von Verbindungen zur Datenbank vorgehalten werden und sich schnell an Clients übergeben lassen. Für die Einbindung von Connection Pools in den Application Stack gibt es verschiedene Konzepte. Einen vorgefertigten Connection Pool liefert „pgBouncer“ im Projekt „pgFoundry“. Der Connection Pool läuft auf dem PostgreSQL Server und ist somit für alle Arten von Verbindungen verwendbar.

Die Parameter des Connection Pool werden in der Datei „etc/pgbouncer.ini“ verwaltet. Diese besteht aus mehreren Sektionen. In der Sektion [databases] werden die Verbindungsparameter zur PostgreSQL-Datenbank eingetragen.

Listing 3.6 Verbindungsparameter zur Datenbank

```
[databases]
; access to dest database will go with single user
postgres = host=192.168.178.100 port=5432 user=postgres password=post1234
client_encoding=UNICODE datestyle=ISO connect_query='SELECT 1'
; use custom pool sizes
nondefaultdb = pool_size=50 reserve_pool=10
```

Die Parameter für die Verbindung zum Connection Pool befinden sich in der Sektion [pgbouncer].

Listing 3.7 Parameter für den Connection Pool

```
[pgbouncer]
;;;
;;; Administrative settings
;;;
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid
;;;
;;; Where to wait for clients
;;;
; ip address or * which means all ip-s
listen_addr = 192.168.178.100
listen_port = 6432
; total number of clients that can connect
max_client_conn = 100
; default pool size. 20 is good number when transaction pooling
; is in use, in session pooling it needs to be the number of
; max clients you want to handle at any moment
default_pool_size = 20
```

Außerdem muss in der Datei „etc/userlist.txt“ mit freigeschalteten Benutzern eingetragen werden:

```
"postgres" "post1234"
```

Starten Sie nun den Connection Pool mit dem folgenden Kommando:

```
$ pgbouncer -d pgbouncer.ini
```

Testen Sie anschließend die Verbindung zu PostgreSQL über den Connection Pool:

```
$ psql -p 6543 -U postgres pgbouncer
```

Die Verbindung zum Connection Pool funktioniert von allen Clients und für alle Programmiersprachen.

■ 3.5 Concurrency

Hinter dem Thema Concurrency steht die Verwaltung, wenn mehrere Clients zur gleichen Zeit auf dieselben Daten zuzugreifen versuchen. PostgreSQL verwendet ein Multiversion-Concurrency-Control-Modell. Das Verhalten der Transaktionen wird Transaction Isolation genannt. In PostgreSQL sind folgende Isolation Level vorgesehen:

1. Read Committed (Standard)
2. Serializable

Eine Änderung des Isolation Level erfolgt mit dem Befehl *SET TRANSACTION*:

```
(postgres@[local]:5432) [hanser] > SET TRANSACTION ISOLATION LEVEL READ
COMMITTED;
SET
```

Bei Isolation Level „Read Committed“ sieht eine Session Änderungen von Daten durch andere Transaktionen nur, wenn diese Transaktionen mit COMMIT abgeschlossen wurden. Eine weitere Eigenschaft ist die Lesekonsistenz. Wenn eine Session eine SELECT-Anweisung abschickt, liefert das Ergebnis den Zustand der Daten, wie er bei Beginn der Transaktion vorgelegen hat, unabhängig von zwischenzeitlichen Änderungen durch andere Sessions.

Wenn Sie das Verhalten von Oracle's SQL*Plus gewöhnt sind, werden Sie bereits Unterschiede im Verhalten zu „psql“ festgestellt haben. In SQL*Plus eröffnet jeder Befehl implizit eine Transaktion, die jederzeit mit ROLLBACK oder COMMIT abgeschlossen werden kann. In „psql“ muss eine Transaktion explizit mit dem Befehl „BEGIN;“ eröffnet werden. Der Abschluss erfolgt ebenfalls mit ROLLBACK oder COMMIT. Der Abschluss einer Transaktion mit „END;“ ist identisch mit der COMMIT-Anweisung, wenn kein Fehler aufgetreten ist.

Innerhalb einer Transaktion kann ein *SAVEPOINT* gesetzt werden. PostgreSQL merkt sich den Status; es ist möglich, bis zum SAVEPOINT zurückzurollen.

Listing 3.8 Zu einem SAVEPOINT zurückrollen

```
(postgres@[local]:5432) [hanser] > BEGIN;
BEGIN
(postgres@[local]:5432) [hanser] *> INSERT INTO test VALUES (1,'BEFORE
SAVEPOINT');
INSERT 0 1
(postgres@[local]:5432) [hanser] *> SAVEPOINT hanser;
SAVEPOINT
(postgres@[local]:5432) [hanser] *> INSERT INTO test VALUES (2,'AFTER
SAVEPOINT');
INSERT 0 1
(postgres@[local]:5432) [hanser] *> ROLLBACK TO hanser;
ROLLBACK
(postgres@[local]:5432) [hanser] *> COMMIT;
COMMIT
(postgres@[local]:5432) [hanser] > SELECT * FROM test;
 id |      text
-----+-----
  1 | BEFORE SAVEPOINT
(1 row)
```


SELECT-Anweisungen setzen keine Sperren auf Sätze und Tabellen. Umgekehrt können SELECT-Anweisungen Sätze lesen, auch wenn sie durch eine andere Session gesperrt sind. Wird ein Satz gesperrt, und eine andere Session versucht, diesen Satz zu ändern oder zu löschen, wartet die zweite Session, bis die erste den Satz durch Abschluss der Transaktion freigegeben hat. Eine Ausnahme besteht, wenn der SQL-Befehl mit der Option *WITH NOWAIT* abgesetzt wird. In diesem Fall bricht die zweite Session die Transaktion ab, wenn die Ressource durch eine andere Session gesperrt ist. Listing 3.9 liefert ein Beispiel.

Listing 3.9 Verhalten von Sessions bei Zugriff auf dieselbe Ressource

```
--Session 1:
(postgres@[local]:5432) [hanser] > BEGIN;
BEGIN
(postgres@[local]:5432) [hanser] *> DELETE FROM test WHERE id=1;
DELETE 1
--Session 2:
(postgres@[local]:5432) [hanser] > DELETE FROM test WHERE id=1;
-- Session 2 wartet, bis die Ressource durch Session 1 freigegeben wird
-- NOWAIT Session 2:
(postgres@[local]:5432) [hanser] > SELECT * FROM test FOR UPDATE NOWAIT;
ERROR:  could not obtain lock on row in relation "test"
```

Ein Deadlock entsteht, wenn sich zwei Sessions durch Sperren auf Sätze gegenseitig sperren. PostgreSQL löst Deadlocks automatisch auf, indem eine Transaktion abgebrochen wird.

Listing 3.10 Eine Deadlock-Situation

```
--Session 1:
(postgres@[local]:5432) [hanser] > BEGIN;
BEGIN
(postgres@[local]:5432) [hanser] *> UPDATE test SET id=11 WHERE id=1;
UPDATE 1
--Session 2:
(postgres@[local]:5432) [hanser] > BEGIN;
BEGIN
(postgres@[local]:5432) [hanser] *> UPDATE test SET id=22 WHERE id=2;
UPDATE 1
--Session 1:
(postgres@[local]:5432) [hanser] *> UPDATE test SET id=22 WHERE id=2;
UPDATE 1
--Session 2:
(postgres@[local]:5432) [hanser] *> UPDATE test SET id=11 WHERE id=1;
ERROR:  deadlock detected
DETAIL:  Process 3455 waits for ShareLock on transaction 838; blocked by
process 3410.
Process 3410 waits for ShareLock on transaction 839; blocked by process
3455.
HINT:  See server log for query details.
```

Das Isolation Level *Serializable* simuliert die Ausführung von Transaktionen, so als würden sie seriell nacheinander ablaufen. Auch hier gilt, dass eine Session nur Änderungen durch andere Sessions sieht, wenn diese ihre Transaktion mit COMMIT abgeschlossen haben. Im Unterschied zu „Read Committed“ sieht die zweite Session beim Absetzen einer SELECT-Anweisung den Zustand der Daten, wie er zu Beginn der Transaktion (nicht der Anweisung) bestanden hat.



Hinweis

An dieser Stelle ein Hinweis zum Transaktionsverhalten des Utility „psql“. In der Version 9 kann „AUTOCOMMIT“ in psql nicht mehr abgeschaltet werden:

```
hanser=# SHOW autocommit;
autocommit
-----
on
(1 row)
hanser=# SET autocommit=off;
ERROR: SET AUTOCOMMIT TO OFF is no longer supported
```

Wenn Sie mit Transaktionen arbeiten wollen, müssen Sie diese explizit mit „BEGIN;“ eröffnen. Der Abschluss einer Transaktion erfolgt dann mit „COMMIT;“ oder „ROLLBACK;“. Der Befehl „END;“ schließt die Transaktion ebenfalls ab (verhält sich wie „COMMIT“).

Es besteht die Möglichkeit, eine Transaction auf „READ ONLY“ zu setzen. Nun können keine Änderungen in den Datenbanken vorgenommen werden.

Listing 3.11 Eine Transaktion auf „READ ONLY“ setzen

```
(postgres@[local]:5432) [hanser] > BEGIN;
BEGIN
(postgres@[local]:5432) [hanser] *> SET TRANSACTION READ ONLY;
SET
(postgres@[local]:5432) [hanser] *> DELETE FROM test;
ERROR: cannot execute DELETE in a read-only transaction
```

■ 3.6 Datenbanken verwalten

Die Architektur von PostgreSQL sieht vor, dass ein Cluster aus einer oder mehreren Datenbanken besteht. An dieser Stelle unterscheidet sich PostgreSQL von Oracle, wo eine Instanz nicht mehr als eine Datenbank bedienen kann. Damit besteht bei PostgreSQL die Aufgabe, die Datenbanken innerhalb eines Clusters zu verwalten.

Die vorhandenen Datenbanken können im Katalog des PostgreSQL-Servers abgefragt werden. Die Abfrage kann mit Hilfe einer SQL-Anweisung oder dem Kurzbehl „\l“ erfolgen.

Listing 3.12 Vorhandene Datenbanken im Cluster abfragen

```
(hanser@[local]:5432) [hanser] > SELECT
> datname,datcollate,datistemplate,dattablespace
> FROM pg_database;
```

datname	datcollate	datistemplate	dattablespace
template1	en_US.UTF-8	t	1663
template0	en_US.UTF-8	t	1663
postgres	en_US.UTF-8	f	1663


```

test      | en_US.UTF-8 | f          |          | 1663
hanser    | en_US.UTF-8 | f          |          | 1663
(hanser@[local]:5432) [hanser] > \l

```

List of databases				
Name	Owner	Encoding	Collation	Ctype
hanser	hanser	UTF8	en_US.UTF-8	en_US.UTF-8
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8
test	postgres	UTF8	en_US.UTF-8	en_US.UTF-8

Neben der Standard-Datenbank „postgres“, die mit der Erstellung des Servers angelegt wurde, gibt es zwei weitere Datenbanken mit den Namen „hanser“ und „test“. Auf die beiden Template-Datenbanken gehen wir später ein.

Das Löschen einer Datenbank erfolgt mit dem Befehl „DROP DATABASE“. Berechtigt zum Löschen sind die Superuser des Clusters sowie der Eigentümer der Datenbank. Mit dem Löschen der Datenbank werden alle Objekte gelöscht.

Listing 3.13 Eine Datenbank löschen

```

(postgres@[local]:5432) [postgres] > DROP DATABASE hanser;
DROP DATABASE

```



Hinweis

Sie können eine Datenbank, mit der Sie gerade verbunden sind, nicht löschen und erhalten die Fehlermeldung: ERROR: cannot drop the currently open database. Melden Sie sich dazu bei einer anderen Datenbank, im Notfall auch bei einer Template-Datenbank an.

Das Anlegen einer Datenbank erfolgt mit dem Befehl „CREATE DATABASE“. Der die Datenbank anlegende Benutzer wird automatisch zum Eigentümer. Der Eigentümer kann nach dem Anlegen geändert werden.

Listing 3.14 Eine Datenbank anlegen und den Eigentümer zuweisen

```

(postgres@[local]:5432) [postgres] > CREATE DATABASE hanser;
CREATE DATABASE
(postgres@[local]:5432) [postgres] > ALTER DATABASE hanser OWNER TO hanser;
ALTER DATABASE

```

Wenn Sie den „CREATE DATABASE“-Befehl anwenden, wird eine existierende Datenbank kopiert. Standardmäßig wird die Datenbank „template1“ als Quelle benutzt. Wenn alle neuen Datenbanken nach einem bestimmten Standard aufgebaut werden sollen, können Sie die Template-Datenbank entsprechend anpassen. Die zweite Template-Datenbank „template0“ enthält alle Objekte von „template1“ nach der Erstellung des Clusters – eine „goldene Quelle“ sozusagen, die man nicht verändern sollte.

Mit Hilfe von Tablespaces kann die Ablage von Tabellen und Indexen kontrolliert werden. Eine kontrollierte Ablage bietet Vorteile sowohl bei der Speicherplatzkontrolle als auch bei der Optimierung der I/O-Performance. Bedenken Sie, dass alle Datenbanken die Ressourcen des Clusters teilen. Dies bedeutet auch, dass sich die Datenbanken gegenseitig den Speicherplatz wegnehmen. Mit einer Trennung in eigene Dateisysteme kann dies verhindert werden. Aus Performance-Sicht bringt eine Trennung von Daten und Indexen oder auch von Tabellenpartitionen signifikante Vorteile.

Die Abfrage von vorhandenen Tablespaces erfolgt mit einer Abfrage von „pg_tablespace“ oder mit dem Kurzbehl „\db“. Die Tablespaces „pg_default“ und „pg_global“ werden standardmäßig beim Erstellen der Datenbank mit angelegt und verweisen auf den Pfad „\$PGDATA“. In der Tablespace „pg_global“ befinden sich die Objekte des Systemkatalogs.

Listing 3.15 Vorhandene Tablespaces abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_tablespace;
```

spcname	spcowner	spclocation	spcACL	spcoptions
pg_default	10			
pg_global	10			

```
(postgres@[local]:5432) [postgres] > \db
```

List of tablespaces

Name	Owner	Location
pg_default	postgres	
pg_global	postgres	

Das Anlegen eines Tablespace erfolgt mit dem Befehl „CREATE TABLESPACE“. Der angegebene Pfad muss im Dateisystem vorhanden und leer sowie durch den PostgreSQL-Server beschreibbar sein.

Listing 3.16 Einen Tablespace anlegen

```
(postgres@[local]:5432) [hanser] > CREATE TABLESPACE ts_hanser LOCATION '/
opt/postgres/hanser';
CREATE TABLESPACE
```

Beim Anlegen einer Tabelle kann der Tablespace explizit angegeben werden:

```
CREATE TABLE test(id INT) TABLESPACE ts_hanser;
```

Wird im CREATE-Befehl keine Tablespace spezifiziert, dann wird das Objekt im Default-Tablespace angelegt. Der Default-Tablespace kann explizit gesetzt werden:

```
SET default_tablespace = ts_hanser;
(postgres@[local]:5432) [hanser] > SHOW default_tablespace;
```

default_tablespace
ts_hanser

```
CREATE TABLE test(id INT);
```

Der Default-Tablespace einer Datenbank kann geändert werden. Beachten Sie, dass damit alle Objekte in den neuen Tablespace gezogen werden. Die Aktion ist deshalb nur möglich, wenn die Datenbank nicht geöffnet ist.

Listing 3.17 Eine Datenbank in einen anderen Tablespace ziehen

```
(postgres@[local]:5432) [postgres] > ALTER DATABASE hanser SET TABLESPACE
ts_hanser;
ALTER DATABASE
(postgres@[local]:5432) [postgres] > SELECT d.datname, t.spclocation, t.oid
> FROM pg_database d, pg_tablespace t
> WHERE t.oid = d.dattablespace;
```

datname	spclocation	oid
template1		1663
template0		1663
postgres		1663
test		1663
hanser2		1663
hanser	/opt/postgres/hanser	16563

Die Organisation eines Tablespace im Betriebssystem ist recht einfach. Im Verzeichnis \$PGDATA befindet sich ein Unterverzeichnis „pg_tblspc“. In diesem Verzeichnis finden Sie einen logischen Link zum Verzeichnis des Tablespace. Der Name des Links ist identisch mit der OID des Tablespace.

Listing 3.18 Organisation eines Tablespace im Betriebssystem

```
$ cd $PGDATA/pg_tblspc
$ ls -l
lrwxrwxrwx 1 postgres postgres 20 May 28 19:03 16563 -> /opt/postgres/hanser
```

Dasselbe Verfahren wird in einem Windows-Betriebssystem angewandt. Auch hier wird ein Link im Verzeichnis angelegt. Ein Beispiel finden Sie in Listing 3.19.

Listing 3.19 Einen Tablespace in Windows anlegen

```
(postgres@:5432) [postgres] > CREATE TABLESPACE ts_hanser LOCATION
'C:/PostgreSQLData/hanser';
CREATE TABLESPACE
C:\PostgreSQL\9.0\data\pg_tblspc>dir
Volume in Laufwerk C: hat keine Bezeichnung.
Volumeseriennummer: 429D-2D45
Verzeichnis von C:\PostgreSQL\9.0\data\pg_tblspc
8.05.2011 19:50 <DIR> .
8.05.2011 19:50 <DIR> ..
8.05.2011 19:50 <VERBINDUNG> 40994 [\\??\C:\PostgreSQLData\hanser]
0 Datei(en), 0 Bytes
3 Verzeichnis(se), 174.564.745.216 Bytes frei
```

Ein leerer Tablespace kann mit dem „DROP TABLESPACE“-Befehl gelöscht werden.

Datumsformat, Sprachen, alphabetische Sortierung und numerische Formate regelt man in PostgreSQL über den *Locale Support*. Die Einstellungen werden automatisch beim Anlegen des Datenbank-Cluster mit „initdb“ vorgenommen. Sie haben die Möglichkeit, bei der Initialisierung eine landesspezifische Einstellung festzulegen. Das Format ist „sp_LA“ mit „sp“ = Sprachkürzel und „LA“ = Landeskürzel. Außer der Locale-Umgebung kann, durch einen Punkt getrennt, der Zeichensatz vorgegeben werden. Die möglichen Einstellungen erhalten Sie auf den meisten UNIX-Systemen mit dem Befehl „locale -a“.

Listing 3.20 Einen Cluster mit Locale-Einstellung initialisieren

```
initdb -locale=en_US
initdb -locale=en_US.UTF8
```

Wenn Sie keine explizite Einstellung angeben, nimmt PostgreSQL die Einstellung des Betriebssystems. Mit dem Java-Programm in können Sie die Standardeinstellung des Betriebssystems herausfinden. Es funktioniert auf allen Plattformen.

Listing 3.21 Java-Programm zur Abfrage der Locale-Einstellung des Betriebssystems

```
import java.util.*;
public class GetLocaleDefault {
    public static void main(String[] args) {
        System.out.println(Locale.getDefault().toString());
    }
}
C:\Temp>java GetLocaleDefault
de_DE
```

Manchmal ist es notwendig, einzelne Einstellungen einer Locale-Umgebung anzupassen, wie zum Beispiel die Verwendung von Punkt und Komma oder das Datumsformat. In Tabelle 3.5 finden Sie die Parameter der Einzelformate.

Tabelle 3.5 Parameter zur Einstellung der Locale-Umgebung

Parameter	Bedeutung
lc-collate	Sortierreihenfolge
lc-ctype	Spezifizierung der Zeichen (Was ist ein Buchstabe, ...)
lc-messages	Sprache der Nachrichten
lc-monetary	Format der Währung
lc-numeric	Formatierung von Zahlen
lc-time	Format für Datum und Uhrzeit

PostgreSQL unterstützt sowohl Single-byte- als auch Multi-byte-Zeichensätze. Single-byte-Zeichensätze sind zum Beispiel „ISO 8859“ oder „WIN 1252“. Ein zugelassener Multi-byte-Zeichensatz ist „UTF8“.

In Listing 3.22 legen wir eine Datenbank mit dem westeuropäischen Windows-Zeichensatz an. Das Template „template0“ muss verwendet werden, wenn „template1“ einen anderen Zeichensatz benutzt.

Listing 3.22 Eine Datenbank mit dem westeuropäischen Windows-Zeichensatz anlegen

```
(postgres@:5432) [hanser] > CREATE DATABASE hanser2 WITH ENCODING 'WIN1252'
TEMPLATE=template0;
CREATE DATABASE
(postgres@:5432) [hanser] > \l
```

```

Liste der Datenbanken
  Name      | Eigentümer | Kodierung | Sortierfolge
-----+-----+-----+-----
hanser      | postgres  | UTF8      | German_Germany.1252
```


hanser2	postgres	WIN1252	German_Germany.1252
postgres	postgres	UTF8	German_Germany.1252
template0	postgres	UTF8	German_Germany.1252
template1	postgres	UTF8	German_Germany.1252

PostgreSQL unterstützt eine automatische Zeichensatzkonvertierung zwischen Client und Server. Es existiert eine Reihe vordefinierter Konvertierungen, die in der Tabelle „pg_conversion“ gespeichert sind. Mit dem Befehl „CREATE CONVERSION“ können Sie weitere Konvertierungen festlegen. Die Kodierung des Clients lässt sich, wie in Listing 3.23 beschrieben, ändern.

Listing 3.23 Die Client-Kodierung ändern

```
(postgres@192.168.178.100:5432) [postgres] > \encoding
UTF8
(postgres@192.168.178.100:5432) [postgres] > SET CLIENT_ENCODING TO
'WIN1252';
SET
(postgres@192.168.178.100:5432) [postgres] > \encoding
WIN1252
```

■ 3.7 Wartungsaufgaben

Als Sie sich mit anderen Datenbanksystemen beschäftigten, konnten Sie sicherlich feststellen, dass jedes System bestimmte regelmäßige Wartungsaufgaben erfordert, um über einen längeren Zeitraum stabil und performant zu laufen. Die Hersteller versuchen, die Wartungsaufgaben weitgehend zu automatisieren. Andererseits werden Datenbanksysteme immer komplexer, sodass das Eingreifen des Datenbank-Administrators an bestimmten Stellen immer noch erforderlich ist und auch bleiben wird. Während manche Wartungsaufgaben eher unkritisch sind, gibt es Aufgaben, die für Stabilität und Performance sehr wichtig sind.

Im Fall von PostgreSQL sind das folgende Aufgaben:

1. VACUUM
2. ANALYZE

3.7.1 VACUUM

Jedes Datenbanksystem muss in der Lage sein, verschiedene Versionen von Sätzen verwalten zu können. So wird ein geänderter Satz erst für andere Sessions sichtbar, wenn die Änderung mit COMMIT abgeschlossen wurde. Andererseits muss auch die Möglichkeit bestehen, offene Transaktionen mit einem ROLLBACK-Befehl zurückrollen zu können. Denken Sie auch an das Prinzip der Lesekonsistenz. So wird bei einem langlaufenden SQL der Zustand der Daten zum Zeitpunkt des Absetzens der Anweisung zurückgeliefert, obwohl zwischenzeitlich Änderungen an den Daten erfolgen.

Oracle löst diese Aufgaben über einen speziellen Tablespace, den UNDO-Tablespace. Er wird sowohl für die Verwaltung von Transaktionen (Rollback) als auch für die Lesekonsistenz herangezogen. Nicht mehr benötigte Undo-Segmente werden ebenso gelöscht wie die früheren Versionen von Sätzen.

PostgreSQL hat einen anderen Ansatz gewählt. Versionen von Datensätzen werden in der Tabelle gespeichert und ältere Versionen entfernt, wenn man sie nicht mehr benötigt. Das Aufräumen dieser Sätze erledigt der VACUUM-Prozess. Werden neue Sätze eingefügt, füllt man zunächst die Lücken, bevor es zu einer Vergrößerung der Tabelle kommt. Ein regelmäßiges Ausführen des VACUUM-Prozesses ist also wichtig, um eine zu starke Fragmentierung zu verhindern.

PostgreSQL verfügt über einen Prozess, der ein automatisches Vacuum durchführt. Für Standard-Installationen von PostgreSQL-Datenbanken kann das automatische Vacuum hinreichend sein. Für anspruchsvolle Datenbanken sind zusätzliche manuelle Vacuum-Aktionen (VACUUM-Befehl) erforderlich.

Listing 3.24 Der automatische Vacuum-Prozess im Betriebssystem

```
$ ps -ef|grep postgres
root      4147   2533   0 10:48 ?        00:00:00 sshd: postgres [priv]
postgres  4150   4147   0 10:48 ?        00:00:00 sshd: postgres@pts/1
postgres  4183   4174   0 10:48 ?        00:00:00 postgres: autovacuum
launcher process
. . .
```

Es existieren zwei Arten von Vacuum: das Standard-Vacuum und das Full-Vacuum. Das Full-Vacuum kann viel Speicherplatz zurückgewinnen, läuft jedoch wesentlich langsamer. Ein weiterer Nachteil des Full-Vacuum ist die Tatsache, dass es für den laufenden Betrieb nicht geeignet ist, da es einen „Exclusive Lock“ auf der Tabelle benötigt. Für den Normalbetrieb sollte deshalb ausschließlich das Standard-Vacuum eingesetzt werden, das Full-Vacuum dagegen nur im Wartungsfenster.



Achtung!

Ein Vacuum-Lauf führt zu einer zusätzlichen Beanspruchung der I/O-Ressourcen, die zu einer Verschlechterung der Performance führen kann. Stellen Sie deshalb die Parameter für das automatische Vacuum entsprechend ein, um den Einfluss möglichst gering zu halten, und führen Sie ein manuelles Vacuum eher zu transaktionsarmen Zeiten durch.

Das Standard-Vacuum gibt ältere Satzversionen, die nicht mehr benötigt werden, frei, indem eine Markierung gesetzt wird. Der Speicherplatz wird dem Betriebssystem nicht zurückgegeben. Im Gegensatz dazu komprimiert das Full-Vacuum alle Sätze zu einer neuen Tabelle und gibt den übrig gebliebenen Speicherplatz an das Betriebssystem zurück.



Praxistipp

Beachten Sie insbesondere für große Tabellen, dass das Full-Vacuum zusätzlich temporären Speicherplatz zum Erstellen der neuen Tabelle benötigt.

Das automatische Vacuum wird stets nur ein Standard-Vacuum und niemals ein Full-Vacuum auslösen. Die entsprechende „Philosophie“ besteht darin, dass die Mehrzahl der Tabellen wächst und der durch das Standard-Vacuum frei gewordene Platz in naher Zukunft wieder benutzt wird und damit eine Rückgabe von Speicherplatz an das Betriebssystem durch ein Full-Vacuum nicht erforderlich ist. Dennoch kann für ausgewählte Tabellen, die starken Änderungen unterliegen, ein Full-Vacuum im Wartungsfenster durchgeführt werden.

Listing 3.25 Ein manuelles Vacuum ausführen (Standard und Full)

```
(postgres@[local]:5432) [postgres] > VACUUM test;
VACUUM
Time: 249.335 ms
(postgres@[local]:5432) [postgres] > VACUUM FULL test;
VACUUM
Time: 3100.747 ms
```

Das Autovacuum ist ein zuverlässiges Mittel um im laufenden Betrieb ständig und automatisch ein Standard-Vacuum durchzuführen. Es besteht aus mehreren Hintergrundprozessen, die durch den Prozess „autovacuum launcher“ aufgerufen werden. Der Launcher-Prozess startet für jede Datenbank einen Vacuum-Worker-Prozess im Abstand, den der Parameter „autovacuum_naptime“ vorgibt. Wenn Sie zum Beispiel fünf Datenbanken in einem Cluster betreiben und der Parameter auf 60 Sekunden gesetzt ist, startet der Launcher-Prozess alle 12 Sekunden einen Worker-Prozess für genau eine Datenbank. Mit dem Parameter „autovacuum_max_workers“ lässt sich festlegen, wie viele Worker-Prozesse parallel laufen dürfen. Damit ist eine Abschirmung vor Überlastung des Clusters durch das Autovacuum möglich. Jeder Worker-Prozess überprüft alle Tabellen, ob ein Vacuum- und ein Analyze-Prozess erforderlich gestartet werden muss.

Wie bereits erwähnt, markiert das Standard-Vacuum die nicht mehr benutzten Sätze (Dead Rows) als wiederverwendbar, gibt jedoch den Speicherplatz nicht an das Betriebssystem zurück. Dies stellt jedoch kein Problem dar, wenn die Tabelle einem Wachstum unterliegt. Die Notwendigkeit, ein VACUUM FULL durchzuführen, besteht nur für wenige Tabellen, zum Beispiel solche, die regelmäßig gelöscht werden.



Praxistipp

Verwenden Sie für Tabellen, die komplett gelöscht werden, wenn möglich den TRUNCATE-Befehl anstelle des DELETE-Befehls. Damit wird die gesamte Tabelle geleert, und es besteht nicht die Notwendigkeit eines FULL VACUUM.

Es gibt allerdings einen anderen Grund, weshalb für Tabellen ein FULL VACUUM durchgeführt werden muss: das *XID Wraparound*.

XID Wraparound

Die Versionierung von Sätzen in PostgreSQL basiert auf der Transaction-ID (XID). Die Transaction-ID ist in jedem Satz vermerkt. Sie hat eine Größe von 32 Bit. Nach 4 Milliarden Transaktionen beginnt der Zähler wieder bei „0“. Damit würden alle Transaktionen, die eigentlich der Vergangenheit angehören, plötzlich in der Zukunft liegen.

Man löst das Problem, indem man PostgreSQL mit einer so genannten *FrozenXID* ausstattet. Bei der FrozenXID geht PostgreSQL davon aus, dass diese immer jünger und niemals älter als die aktuelle Transaction-ID ist, und setzt somit die normalen Vergleichsregeln zur XID außer Kraft. Beim Vergleich wird zur aktuellen XID der Wert 2^{31} addiert und mit der FrozenXID verglichen.

Die „FrozenXID“ gelangt allerdings nicht von allein in die Tabellen, wird jedoch bei einem Vacuum-Lauf aktualisiert. Allerdings überspringt der normale VACUUM-Prozess Sätze, die kein „Dead Flag“ besitzen. Zur Steuerung dieses Mechanismus hält PostgreSQL eine Reihe von Parametern bereit.

Mit dem Parameter *vacuum_freeze_min_age* lässt sich festlegen, wie alt eine XID sein muss, damit sie durch eine FrozenXID ersetzt wird. Je kleiner der Wert, desto mehr Transaktionen können vergehen, bevor die Tabellen einem VACUUM-Lauf unterzogen werden müssen. Der normale Vacuum-Prozess überspringt Sätze die kein Dead Row Flag besitzen. Um sicherzustellen, dass alle XID ersetzt werden, muss Vacuum alle Sätze prüfen. Dies steuert der Parameter *vacuum_freeze_table_age*. Eine vollständige Prüfung der Sätze erfolgt, wenn „vacuum_freeze_table_age“ minus „vacuum_freeze_min_age“ Transaktionen verstrichen sind.

Die Obergrenze für eine Tabelle ohne Vacuum-Lauf ist also zwei Milliarden minus „vacuum_freeze_min_age“. Andernfalls droht ein Datenverlust. Um dies auszuschließen, wird ein Auto-Vacuum spätestens nach der im Parameter *autovacuum_freeze_max_age* vorgegebenen Anzahl von Transaktionen ausgelöst.

Für große Tabellen mit einem großen statischen Anteil von Sätzen besteht das Ziel darin, möglichst viele Transaktionen zwischen den VACUUM-Läufen zuzulassen. Dies erreicht man durch Vergrößerung des Parameters „autovacuum_freeze_max_age“ oder durch Verkleinerung von „vacuum_freeze_min_age“.

Wie kann der Datenbankadministrator das automatische Vacuum überwachen? Einerseits ist natürlich von Interesse, wie viel Zeit die Vacuum-Prozesse benötigen. Dies lässt sich einfach mit dem Parameter „log_autovacuum_min_duration“ realisieren. Nun werden alle Vacuum-Läufe, die diesen Zeitwert überschreiten, in der Logdatei aufgezeichnet.

Andererseits ist es wichtig zu wissen, wann für eine bestimmte Tabelle das letzte Auto-vacuum gelaufen ist. Die Abfrage in Listing 3.26 liefert diese Information.

Listing 3.26 Das Datum des letzten Autovacuum abfragen

```
(postgres@[local]:5432) [postgres] > SELECT
schemaname,relname,last_autovacuum
> FROM pg_stat_all_tables
> ORDER BY 3;
```

schemaname	relname	last_autovacuum
information_schema	sql_languages	
public	test	2011-05-29 12:18:09.013152+02
...		

3.7.2 ANALYZE

Um einen möglichst optimalen Ausführungsplan generieren zu können, benötigt der PostgreSQL-Query-Planer aktuelle und zuverlässige Statistiken über Tabellen und Indexe. Das Sammeln von Statistiken kann mit dem ANALYZE-Befehl manuell angestoßen werden. Interessant ist dabei die Frage, wann eine Aktualisierung der Statistiken wirklich notwendig ist. Als groben Anhaltspunkt lässt sich für PostgreSQL bestimmen, dass eine Aktualisierung dann sinnvoll ist, wenn sich Minimum- und Maximum-Werte von Spalten ändern oder eine signifikante Verschiebung der Histogramme stattfindet. Aber auch Spalten, die häufig in WHERE-Bedingungen auftauchen, sollten besser häufiger als zu wenig aktualisiert werden. Viele Administratoren von Applikationen können recht genau einschätzen, welche Tabellen Kandidaten für eine regelmäßige Aktualisierung der Statistiken sind.

Statistiken können auf Tabellen- und Spaltenebene aktualisiert werden. Damit ist es möglich, kritische Spalten häufig zu aktualisieren, ohne damit eine übermäßige Zusatzbelastung des Systems hervorzurufen. Andererseits erstellt PostgreSQL die Statistiken auf einer relativ kleinen Stichprobe, die nach einem Zufallsverfahren ausgewählt wird. Das Sammeln von Statistiken ist damit weit weniger aufwendig im Vergleich zu anderen Datenbanksystemen, wie zum Beispiel Oracle. In Listing 3.27 wird eine ANALYZE auf einer Tabelle mit 20 Millionen Sätzen ausgeführt. Die Stichprobe besteht aus gerade mal 30 000 Zeilen.

Listing 3.27 Query-Planer-Statistiken auf einer Tabelle sammeln

```
(postgres@:5432) [hanser] > ANALYZE VERBOSE kb;
INFO:  analysiere "public.kb"
INFO:  "kb": 30000 von 285715 Seiten gelesen, enthalten 2100000 lebende
Zeilen und 0 tote Zeilen; 30000 Zeilen in Stichprobe, schätzungsweise
20000050 Zeilen insgesamt
ANALYZE
Zeit: 512,328 ms
```



Hinweis

Ist das automatische Vacuum eingeschaltet (Parameter `autovacuum = on`), löst der Autovacuum Daemon ein ANALYZE aus, wenn sich der Inhalt einer Tabelle verändert hat.

■ 3.8 Nützliche Skripte und Hinweise

Nach den, zugegeben, etwas trockenen (aber notwendigen) Beschreibungen von Parametern in diesem Kapitel stellt der letzte Abschnitt eine Reihe von nützlichen Skripten und Hinweisen zur Verfügung. Sie können die Skripte direkt für die praktische Arbeit einsetzen.

3.8.1 Eine Passwort-Datei verwenden

Für Batch-Verarbeitung muss das Passwort in geeigneter Form auf dem Server hinterlegt werden. Auch wenn es unverschlüsselt in einer Passwortdatei steht, kann der Zugriff auf dieselbe beschränkt werden. Damit ist das Auslesen des Passworts nur durch den Benutzer root möglich. PostgreSQL sucht die Passwortdatei im in der Umgebungsvariablen PGPASS-FILE spezifizierten Pfad oder im Home-Verzeichnis des Benutzers. Der Name der Datei ist `.pgpass` mit folgendem Inhalt:

```
<host>:<port>:<database>:<username>:<password>
```

Schützen Sie den Zugriff auf die Passwortdatei:

```
$ chmod 600 .pgpass
```

3.8.2 Welche Parameter sind Nicht-Standard?

Listing 3.28 Nicht-Standard-Parameter anzeigen

```
postgres=# SELECT name, setting, reset_val
postgres=# FROM pg_settings
postgres=# WHERE source <> 'default' AND source <> 'override';
```

name	setting	reset_val
application_name	psql	psql
DateStyle	ISO, MDY	ISO, MDY
default_text_search_config	pg_catalog.english	pg_catalog.english
lc_messages	en_US.UTF-8	en_US.UTF-8
lc_monetary	en_US.UTF-8	en_US.UTF-8
lc_numeric	en_US.UTF-8	en_US.UTF-8
lc_time	en_US.UTF-8	en_US.UTF-8
listen_addresses	192.168.178.201	192.168.178.201
log_destination	stderr	stderr
log_directory	pg_log	pg_log
log_timezone	Europe/Berlin	Europe/Berlin
logging_collector	on	on
max_connections	100	100
max_stack_depth	2048	2048
port	5432	5432
shared_buffers	131072	131072
TimeZone	Europe/Berlin	Europe/Berlin
timezone_abbreviations	Default	Default

(18 rows)

3.8.3 Eine Session killen

In einigen Situationen ist es erforderlich, eine bestimmte Session in der Datenbank zu killen, um wieder freie Ressourcen zu erhalten. Dies kann zum Beispiel der Fall sein, wenn eine Session einen großen Hauptspeicherverbrauch aufweist. Mit einer Abfrage der Tabelle `pg_stat_activity` kann die Prozess-ID der Session ermittelt werden:


```
postgres=# SELECT procpid, datname, username, client_addr, client_port
postgres=# FROM pg_stat_activity;
 procpid | datname  | username | client_addr | client_port
-----+-----+-----+-----+-----
    4865 | postgres | postgres |              |          -1
    4901 | postgres | postgres | 192.168.178.22 |      49333
    4903 | hanser   | postgres | 192.168.178.22 |      49334
    4906 | postgres | postgres | 192.168.178.22 |      49335
```

Mit der Funktion *pg_cancel_backend()* wird die Ausführung des aktuellen SQL abgebrochen. Die Session selbst wird nicht beendet. Ein Beenden der Session erfolgt mit *pg_terminate_backend()*. Ist dies nicht möglich, muss sie auf Betriebssystemebene mit dem Kill-Befehl beendet werden.

```
postgres=# SELECT pg_terminate_backend(procpid)
postgres=# FROM pg_stat_activity
postgres=# WHERE procpid = 4906;
pg_terminate_backend
-----
t
```

3.8.4 Ein Tabelle nach Excel kopieren

Das Kopieren einer Tabelle nach Excel ermöglicht eine einfache Aufbereitung und Darstellung als Grafik. Dies ist in PostgreSQL einfach mit dem COPY-Befehl möglich. Als Beispiel dient eine einfache Tabelle:

```
postgres=# SELECT * FROM umsatz;
  monat  | wert
-----+-----
2010-01-01 | 54565
2010-01-03 | 34562
2010-01-02 | 64534
2010-01-04 | 44534
```

Mit dem COPY-Befehl wird eine csv-Datei erstellt. Diese lässt sich in Excel einlesen und verarbeiten (siehe Bild 3.1).

```
postgres=# COPY umsatz TO '/home/postgres/umsatz.txt';
COPY 4
```

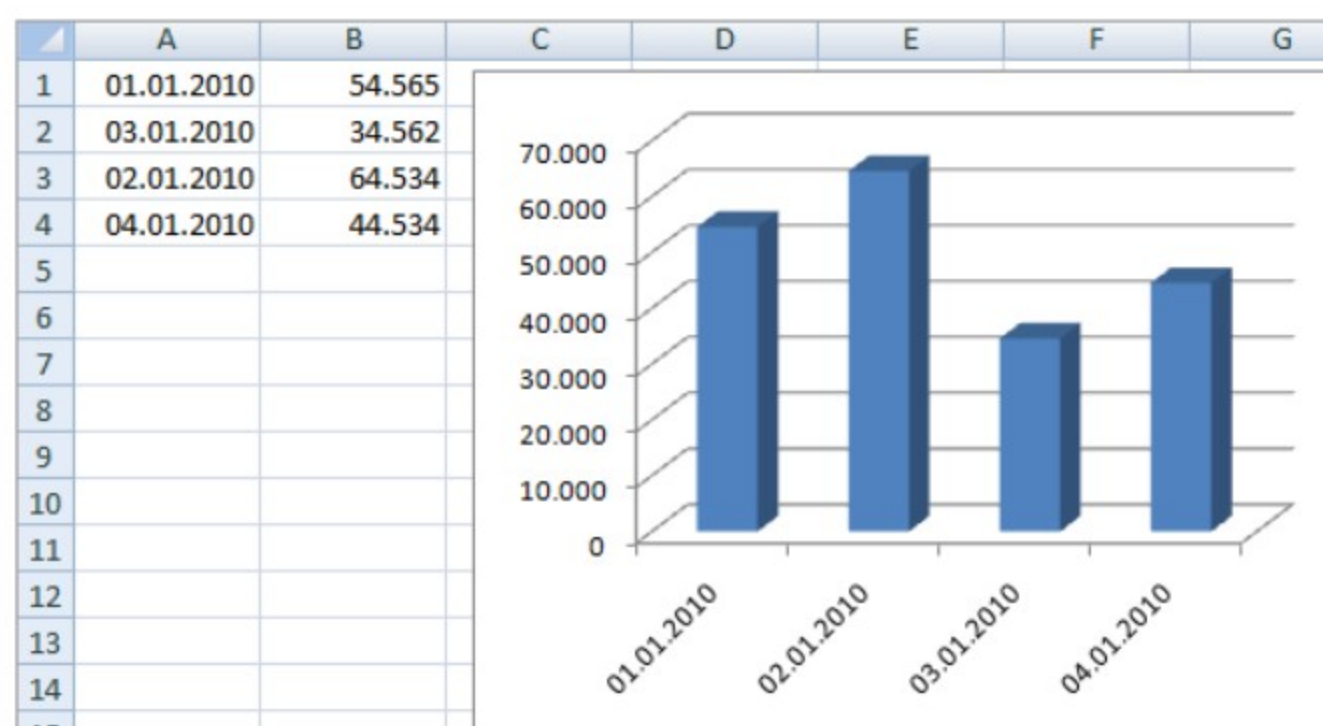


Bild 3.1
Aufbereitung der
csv-Datei in Excel

3.8.5 Die Datei .psqlrc

Für Einstellungen in psql dient die Datei `~/.psqlrc`. Sie wird beim Start von psql gelesen. Im folgenden Beispiel wird ein Prompt für psql definiert.

Listing 3.29 Einstellung für psql in der Datei .psqlrc vornehmen

```
$ vi ~/.psqlrc
\set PROMPT1 '(%n%M:%>) [%/] %x> '
\set PROMPT2 '%x> '
$ psql
psql (9.1.0)
Type "help" for help.
(postgres@[local]:5432) [postgres] >
```

Der so definierte Prompt zeigt an, ob man sich innerhalb einer Transaktion befindet. Innerhalb einer Transaktion erscheint im Prompt zusätzlich ein „*“:

```
(postgres@[local]:5432) [postgres] > BEGIN;
BEGIN
(postgres@[local]:5432) [postgres] *> COMMIT;
COMMIT
(postgres@[local]:5432) [postgres] > COMMIT;
WARNING:  there is no transaction in progress
COMMIT
```

Plattformspezifischer Unterschied

Beachten Sie den Plattform-spezifischen Unterschied: Unter Windows heißt die Konfigurationsdatei *psqlrc.conf* und befindet sich im Verzeichnis `%APPDATA%\postgresql`.

3.8.6 Einen WAL-Switch manuell auslösen

Der Switch zu einem neuen WAL-Segment kann über psql mit der Funktion *pg_switch_xlog* ausgelöst werden. Das ist insbesondere von Vorteil, wenn mit Backup und Restore gearbeitet wird.

Listing 3.30 Einen manuellen WAL-Switch auslösen

```
(postgres@[local]:5432) [postgres] > SELECT pg_switch_xlog();
 pg_switch_xlog
-----
0/5A000000
(1 row)
```

Beachten Sie, dass kein Switch durchgeführt wird, wenn keine Transaktion seit dem letzten Switch stattgefunden hat.

3.8.7 Die PostgreSQL-Server Log-Datei in eine Tabelle laden

Wenn Sie zusätzlich den Wert „csvlog“ im Parameter „log_destination“ angeben, schreibt der PostgreSQL-Server die Logdatei im csv-Format. Dieses Format kann in die Tabelle `postgres_log` (siehe Listing 3.31) geladen werden.

Listing 3.31 Eine Tabelle zum Laden der Logdatei anlegen

```
CREATE TABLE postgres_log (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num));
```

Das Laden der Tabelle kann einfach mit dem COPY-Befehl erfolgen:

```
COPY postgres_log FROM '/usr/local/postgres/data/pg_log/logfile.csv';
```

Um den Ladeprozess praktikabel zu machen, sollten Sie die folgenden Hinweise beachten:

- Setzen Sie den Parameter `log_rotation_size` auf „0“, um den Namen der Logdatei besser definieren zu können.
- Verwenden Sie kein Muster für den Namen der Logdatei (zum Beispiel für Datum und Uhrzeit). Auch das erleichtert die Bestimmung des Dateinamens.

3.8.8 Automatisches Rotieren von Log-Dateien

Obwohl PostgreSQL in Vergleich zu anderen Datenbanksystemen bereits eine sehr gute Unterstützung für die Rotation von Log-Dateien bietet, ist das Stoppen des Servers immer eine lästige Aktion und verringert die Verfügbarkeit der Datenbanken. Eine gute Lösung ist die Verwendung eines externen Programms für die Rotation von Logdateien.

Das Programm *rotatelogs* ist Bestandteil der Apache-Distribution. Sie finden es als Binary im „bin-Verzeichnis“ und können es auf den PostgreSQL-Server kopieren. Die Rotation kann wahlweise in Dateigröße (Megabyte) oder Minuten angegeben werden.

Listing 3.32 Ein externes Programm für die Rotation der Log-Dateien verwenden

```
$ pg_ctl start|/usr/local/apache/bin/rotatelogs $PGDATA/pg_log/pgsql_log 1M
```

3.8.9 Nicht verwendete Indexe identifizieren

Indexe, die von den Applikationen nicht verwendet werden, verursachen nur Kosten in Form von Laufzeit beim Einfügen oder Ändern von Datensätzen sowie zusätzlichem Ressourcenverbrauch, ohne dass die Applikationen einen Nutzen daraus ziehen. Die SQL-Anweisung in Listing 3.33 gibt alle Indexe aus, die weniger als zehnmal benutzt wurden.

Listing 3.33 Nicht verwendete Indexe identifizieren

```
(postgres@[local]:5432) [postgres] > SELECT si.relname, si.indexrelname,
si.idx_scan
> FROM pg_stat_user_indexes si, pg_indexes i, pg_stat_user_tables st
> WHERE si.indexrelname = i.indexname
> AND si.relname = st.relname
> AND si.idx_scan < 10
> AND i.indexdef !~* 'unique'
> ORDER BY 1,2;
 relname | indexrelname | idx_scan
-----+-----+-----
 kb      | index_kb     |         0
```

3.8.10 Microsoft Excel als Datenbank-Client

Microsoft Excel ermöglicht es, eine ODBC-Verbindung zur PostgreSQL-Datenbank herzustellen und SQL-Abfragen abzusetzen. Über das Menü „Daten“, „Aus anderen Quellen“ und „Von Microsoft Query“ (siehe Bild 3.2) können Sie die ODBC-Verbindung herstellen. Vorausgesetzt wird ein installierter PostgreSQL-ODBC-Treiber.

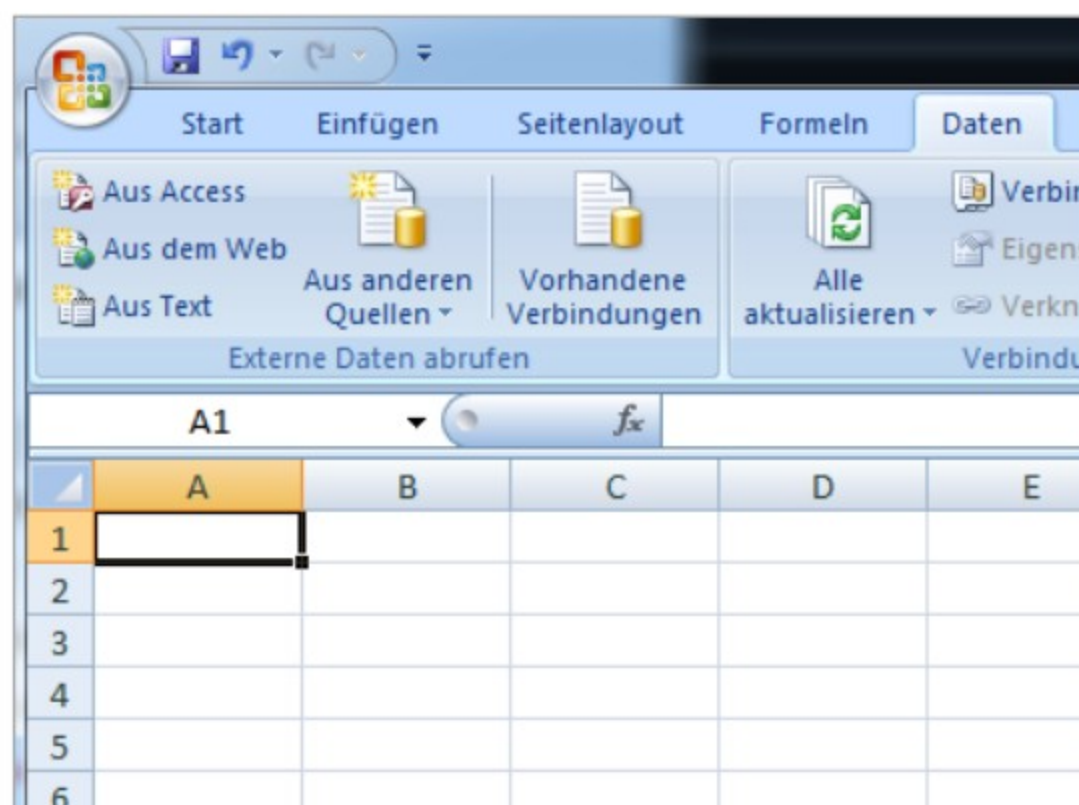


Bild 3.2 Menüauswahl in Excel

Es erscheint der SQL-Abfrage-Assistent, über den eine SELECT-Anweisung abgesetzt werden kann. Diese liefert das Ergebnis in Form einer Excel-Tabelle, die sich direkt bearbeiten lässt (siehe Bild 3.4).

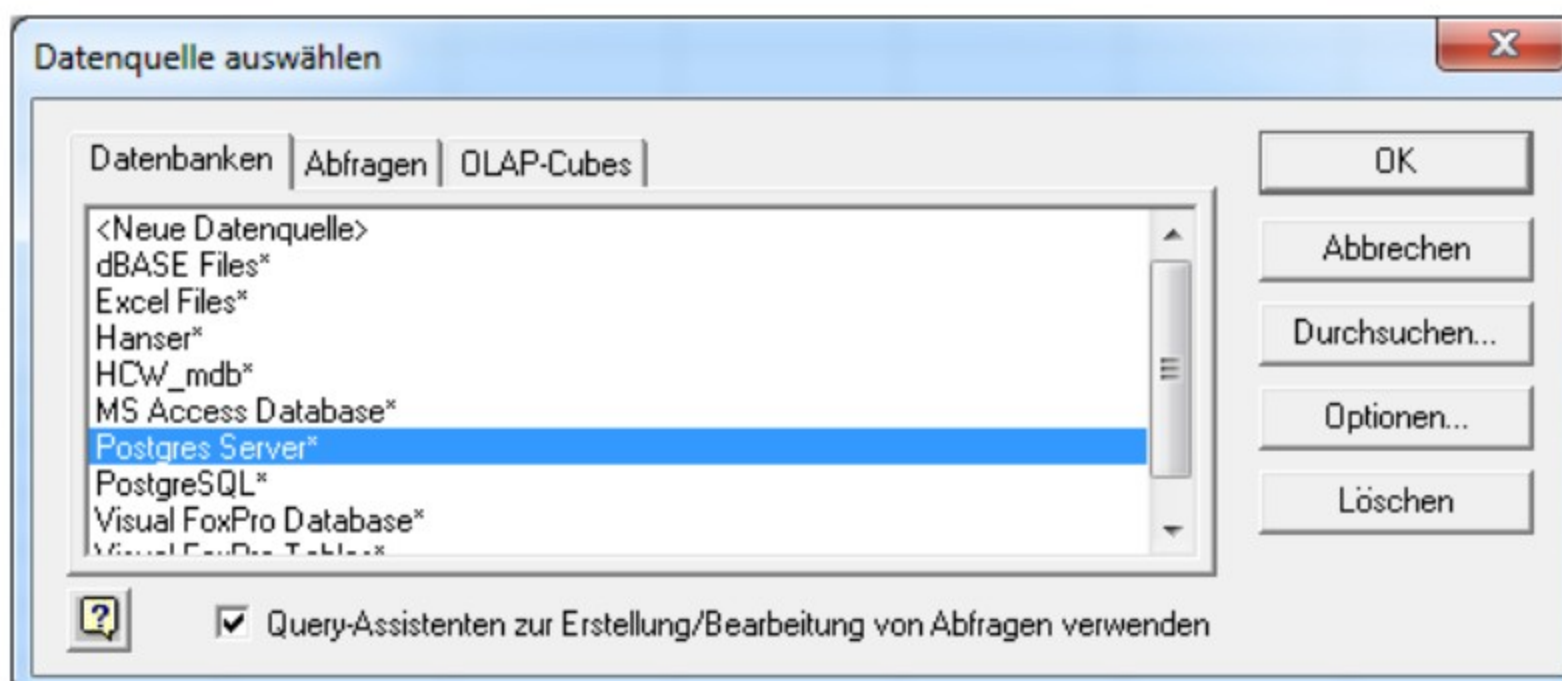


Bild 3.3 Die ODBC-Verbindung auswählen

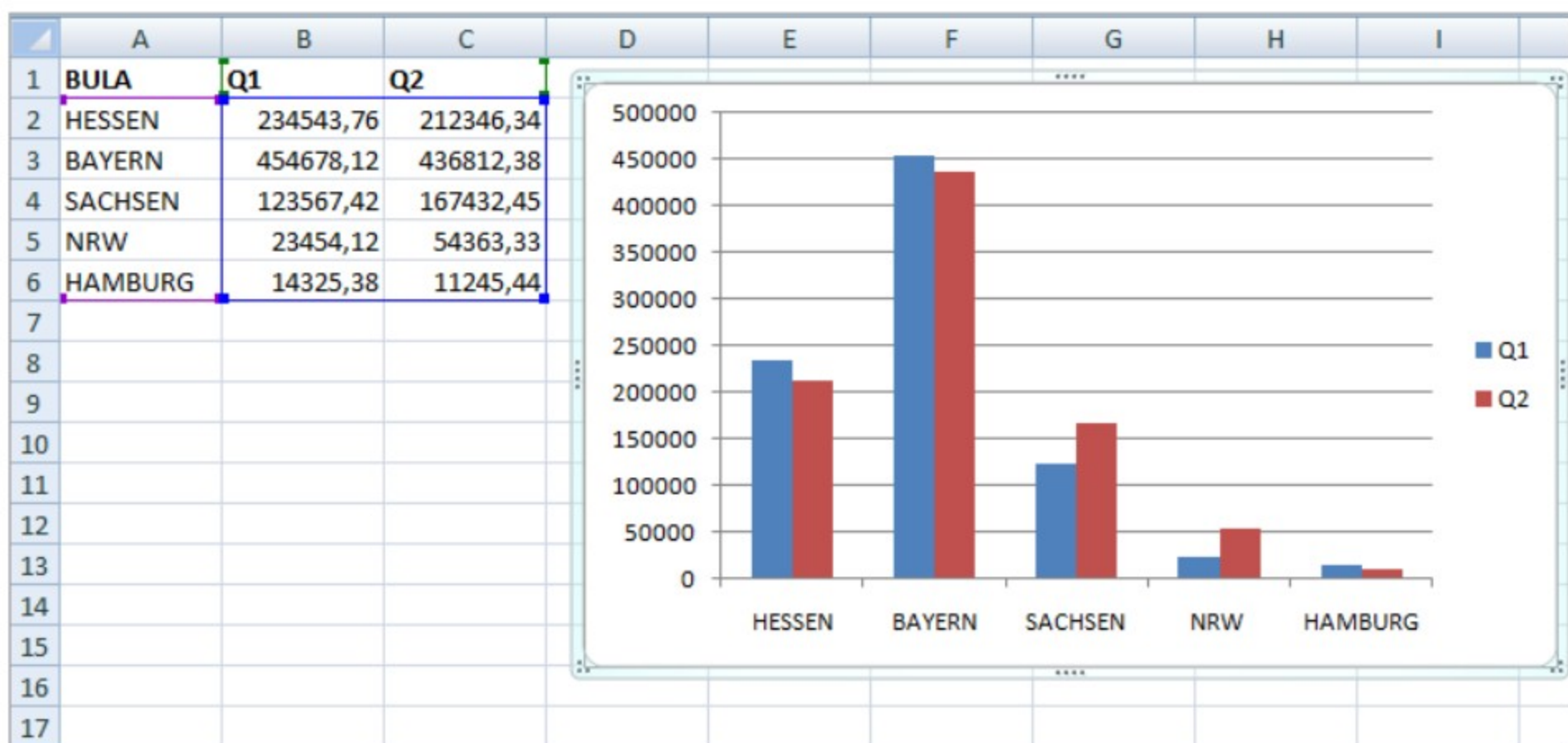


Bild 3.4 Eine Tabelle in Excel verarbeiten

3.8.11 Den Inhalt der Kontrolldatei ausgeben

Wie andere Datenbank-Systeme besitzt auch PostgreSQL eine Kontrolldatei, in der sich zentrale Informationen über Server und Datenbanken befinden. Das Programm *pg_controldata* schreibt den Inhalt der Kontrolldatei aus.

Listing 3.34 Den Inhalt der Kontrolldatei ausgeben

```
$ pg_controldata
pg_control version number:      903
Catalog version number:        201105231
Database system identifier:     5664456093965578734
Database cluster state:         in production
```

```

pg_control last modified:      Sat 12 Nov 2011 05:38:43 PM CET
Latest checkpoint location:    0/5D000020
Prior checkpoint location:     0/5C000160
Latest checkpoint's REDO location: 0/5D000020
Latest checkpoint's TimeLineID: 7
Latest checkpoint's NextXID:   0/1712
Latest checkpoint's NextOID:   24581
Latest checkpoint's NextMultiXactId: 1
Latest checkpoint's NextMultiOffset: 0
Latest checkpoint's oldestXID: 1669
Latest checkpoint's oldestXID's DB: 1
Latest checkpoint's oldestActiveXID: 0
Time of latest checkpoint:     Sat 12 Nov 2011 05:43:38 PM CET
Minimum recovery ending location: 0/0
Backup start location:         0/0
Current wal_level setting:     hot_standby
Current max_connections setting: 100
Current max_prepared_xacts setting: 0
Current max_locks_per_xact setting: 64
Maximum data alignment:        4
Database block size:           8192
Blocks per segment of large relation: 131072
WAL block size:                8192
Bytes per WAL segment:         16777216
Maximum length of identifiers:  64
Maximum columns in an index:    32
Maximum size of a TOAST chunk:  2000
Date/time type storage:         64-bit integers
Float4 argument passing:        by value
Float8 argument passing:        by reference

```

3.8.12 Platzverbrauch von Tabellen

Fragen wie: Wie groß ist eine bestimmte Tabelle? Oder: Was sind die größten Tabellen? werden häufig gestellt und lassen sich einfach beantworten. Die Größe einer Tabelle können Sie mit einer SQL-Anweisung abfragen. Die Ausgabe erfolgt in Byte. Für große Tabellen ist eine Umrechnung in KByte oder Mbyte sinnvoll.

```

(postgres@[local]:5432) [postgres] > SELECT pg_relation_size('kb');
pg_relation_size
-----
1011359744
(postgres@[local]:5432) [postgres] > SELECT pg_relation_size('kb')/1024/1024
AS "MB";
MB
-----
964

```

Die SQL-Anweisung in Listing 35 ermittelt die 20 größten Tabellen, absteigend sortiert.

Listing 3.35 Die 20 größten Tabellen ermitteln

```

(postgres@[local]:5432) [postgres] > SELECT
> table_name, pg_relation_size(table_name)/1024 AS "KB"
> FROM information_schema.tables

```



```
> WHERE table_schema NOT IN ('pg_catalog','information_schema')
> ORDER BY "KB" DESC LIMIT 20;
```

table_name	KB
stock	13611112
order_line	3692312
customer	2600000
kb	987656
history	296304
new_order	45864
item	8248
t	3544
spatial_ref_sys	3184
district	136
warehouse	16
test	8
departments	8
product_data	8
product_log	8
log_table	8
audit_emp	8
pgstatspack_version	8
emp	8
kb3	8

(20 rows)

3.8.13 Die Anzahl von Verbindungen begrenzen

Die maximale Anzahl aller Verbindungen auf einem Cluster kann durch den Parameter „max_connections“ begrenzt werden.

```
(postgres@[local]:5432) [postgres] > SELECT name,setting
> FROM pg_settings WHERE name='max_connections';
```

name	setting
max_connections	1000

In der Regel wählt man den Parameter groß genug, damit alle Clients Platz finden. In bestimmten Datenbanken ist es notwendig, die Anzahl von Sessions zu begrenzen, um eine Überlastung des Datenbank-Clusters zu vermeiden. Es wäre jedoch fatal, wenn bestimmte Sessions, wie zum Beispiel Ladejobs, gegen das Limit laufen und fehlschlagen.

In PostgreSQL besteht die Möglichkeit, die Anzahl von Verbindungen pro Benutzer (Rolle) zu begrenzen. Damit lassen sich zum Beispiel individuelle Accounts in einem Data Warehouse auf eine oder wenige Sessions begrenzen, um zu verhindern, dass zu viele Abfragen parallel abgesetzt werden.

Listing 3.36 Die Anzahl von Verbindungen pro Benutzer begrenzen

```
(postgres@[local]:5432) [postgres] > ALTER ROLE schulze CONNECTION LIMIT 3;
ALTER ROLE
```

Verbindungsversuche, die über das Limit hinausgehen, werden abgewiesen:

```
(postgres@[local]:5432) [postgres] > \c postgres schulze
You are now connected to database "postgres" as user "schulze".
(postgres@[local]:5432) [postgres] > \c postgres schulze
FATAL: too many connections for role "schulze"
```

3.8.14 Tabellen und Indexe in eine andere Tablespace legen

Das Anlegen und Verteilen von Tablespace hat nicht nur organisatorische Hintergründe, sondern kann auch der Verbesserung von Performance und Ausfallsicherheit dienen. Das Anlegen eines Tablespace erfolgt mit dem CREATE TABLESPACE-Befehl:

```
(postgres@[local]:5432) [postgres] > CREATE TABLESPACE app_data
OWNER postgres LOCATION '/usr/local/pgsql/data/app_data';
CREATE TABLESPACE
(postgres@[local]:5432) [postgres] > CREATE TABLESPACE app_index
OWNER postgres LOCATION '/usr/local/pgsql/data/app_index';
CREATE TABLESPACE
```

Das Verschieben von Tabellen und Indexen in eine andere Tablespace erfolgt, indem man Block für Block sequenziell kopiert. Je größer die Objekte, desto länger dauert die Operation. Deshalb empfiehlt sich das Verschieben von großen Objekten in transaktionsarmen Zeiten oder in einem Wartungsfenster. Das Abschalten der WAL-Archivierung würde zwar eine deutliche Performance-Steigerung bringen, kompromittiert jedoch die Garantie des Point-in-time Recovery.

Im Beispiel in Listing 3.37 werden eine Tabelle und ein Index mit 30 Millionen Sätzen in einen anderen Tablespace verschoben.

Listing 3.37 Tabelle und Index in einen anderen Tablespace verschieben

```
(postgres@[local]:5432) [postgres] > BEGIN WORK;
BEGIN
Time: 0.125 ms
(postgres@[local]:5432) [postgres] *> ALTER TABLE order_line SET TABLESPACE
app_data;
ALTER TABLE
Time: 74791.071 ms
(postgres@[local]:5432) [postgres] *> ALTER INDEX order_line_pkey SET TABLE
SPACE app_index;
ALTER INDEX
Time: 59863.471 ms
(postgres@[local]:5432) [postgres] *> COMMIT;
COMMIT
Time: 2429.499 ms
```


**Hinweis**

Beachten Sie, dass die Objekte während des Kopiervorgangs (in diesem Beispiel, bis die Transaktion mit „COMMIT“ abgeschlossen ist) exklusiv gesperrt sind.

```
(postgres@[local]:5432) [postgres] > select pg_class.relname,pg_
locks.mode from pg_class,pg_locks where pg_class.relfilenode=pg_
locks.relation;
relname      |          mode
-----+-----
order_line   | ShareUpdateExclusiveLock
order_line   | AccessExclusiveLock
pg_locks     | AccessShareLock
order_line_pkey | AccessExclusiveLock
```

3.8.15 Updateable Views

Views, die mit der CREATE VIEW-Anweisung angelegt werden, können von Haus aus nicht verändert werden. Eine UPDATE-Anweisung führt zum Fehler, wie das folgende Beispiel zeigt:

```
(postgres@[local]:5432) [postgres] > CREATE VIEW v_warehouse AS
> SELECT w_id,w_name,w_city FROM warehouse;
CREATE VIEW
(postgres@[local]:5432) [postgres] > INSERT INTO v_warehouse
> VALUES(4711,'Speicher','Hamburg');
ERROR: cannot insert into view "v_warehouse"
HINT: You need an unconditional ON INSERT DO INSTEAD rule or an INSTEAD OF
INSERT trigger.
```

Mit Hilfe einer Regel lässt sich die View als veränderbar definieren. Für das vorliegende Beispiel wird eine Regel definiert. Damit können Sie Sätze einfügen.

Listing 3.38 Eine Regel für eine updateable View anlegen

```
(postgres@[local]:5432) [postgres] > CREATE RULE i_v_warehouse AS
> ON INSERT TO v_warehouse
> DO INSTEAD
> INSERT INTO warehouse
> VALUES (NEW.w_id, NEW.w_name, NEW.w_city);
CREATE RULE
```

Die Regel bewirkt, dass eine INSERT-Anweisung entsprechend der Regel auf der Tabelle ausgeführt wird.

```
(postgres@[local]:5432) [postgres] > INSERT INTO v_warehouse
> VALUES(4711,'Speicher','Hamburg');
INSERT 0 1
```

**Hinweis**

Detaillierte Informationen zum Thema „Regeln“ finden Sie in Kapitel 9.

3.8.16 Temporäre Tablespaces verwalten

PostgreSQL verwendet temporäre Dateien zum Speichern von Interims-Daten, wenn diese nicht komplett in den Hauptspeicher passen. Die Größe des Hauptspeichers für temporäre Daten wird durch den Parameter „temp_buffers“ festgesetzt.

Die Lage der Temp-Tablespaces wird vom Parameter „temp_tablespaces“ festgelegt. Ist der Parameter leer, werden die temporären Dateien im Standard-Verzeichnis, also dort, wo die Standard-Tablespaces liegen (\$PGDATA), gespeichert. Dies ist der Standard, wenn ein PostgreSQL Cluster angelegt wird. Führen Sie die folgenden Schritte aus, um einen temporären Tablespace in einem separaten Verzeichnis anzulegen:

1. Erstellen Sie einen Tablespace in einem separaten Verzeichnis.

```
(postgres@[local]:5432) [postgres] > CREATE TABLESPACE temp01 LOCATION '/usr/local/pgsql/data/temp';
CREATE TABLESPACE
```

2. Setzen Sie den Parameter „temp_tablespaces“ auf den Namen der Tablespace.

```
(postgres@[local]:5432) [postgres] > SELECT name,setting FROM pg_settings
WHERE name = 'temp_tablespaces';
name          | setting
-----+-----
temp_tablespaces | temp01
```

3. Überprüfen Sie die Einstellung.

```
(postgres@[local]:5432) [postgres] > SELECT name,setting FROM pg_settings
WHERE name = 'temp_tablespaces';
name          | setting
-----+-----
temp_tablespaces | temp01
```



Praxistipp

Normalerweise werden temporäre Dateien nach Beendigung der Transaktion oder mit dem Shutdown des Cluster gelöscht. Im Falle eines Crashes können diese Dateien liegen bleiben. Die Dateien sind löscher, sobald das Cluster heruntergefahren ist.

3.8.17 Lang laufende SQL-Anweisungen

Lang laufende SQL-Anweisungen stellen häufig eine erhebliche Belastung des Datenbank-servers dar. Es gehört zu den Aufgaben des Datenbankadministrators, diese herauszufinden und entsprechende Maßnahmen zur Optimierung einzuleiten. Mit Hilfe der SQL-Abfrage in Listing 3.39 können solche Langläufer identifiziert werden.

Listing 3.39 Lang laufende SQL-Anweisungen ermitteln

```
((postgres@[local]:5432) [postgres] > SELECT username,client_addr,
> current_query, now()-query_start AS time
> FROM pg_stat_activity
> WHERE current_query NOT LIKE '%pg_stat_activity%';
```

username	client_addr	current_query	time
postgres	192.168.178.117	<IDLE>	00:08:51.290288
postgres	192.168.178.117	select * from order_line	00:01:01.477006

4

Sicherung und Wiederherstellung

Die Anforderungen an die Wiederherstellung einer Datenbank oder von Objekten in einer Datenbank sind in den letzten Jahren gestiegen. So wird die Möglichkeit, eine produktive Datenbank zu einem beliebigen Zeitpunkt wiederherzustellen, inzwischen als Industriestandard gesehen. Auf der anderen Seite steht die Forderung nach einer möglichst hohen Verfügbarkeit der Datenbank.

Die Entwickler-Community von PostgreSQL hat diese Entwicklung erkannt und mit den neuen Features der Version darauf reagiert. Allein die Option einer Standby-Datenbank auf einem entfernten Server bietet viele Möglichkeiten. Detaillierte Informationen zu diesem Thema finden Sie in Kapitel 7 („Hot Standby und Streaming Replication“).

Zurück zum Thema Sicherung und Wiederherstellung. Die Auswahl der Sicherungsmethoden sowie des zugehörigen Schedules hängt von den Anforderungen an die Wiederherstellung und von den Wiederherstellungszeiten ab. Aus diesem Grund wird man die Sicherung für eine Entwicklungsdatenbank anders planen als die für eine unternehmenskritische Produktionsdatenbank. Für IT-Dienstleister sind diese Anforderungen häufig durch Service Level Agreements (SLA) geregelt. Im nächsten Abschnitt erörtern wir die von PostgreSQL angebotenen Sicherungsoptionen.

PostgreSQL bietet die folgenden Verfahren zur vollständigen oder teilweisen Sicherung von Datenbanken:

1. Exportieren von Objekten mit „SQL Dump“
2. Offline-Sicherung auf Dateisystem-Ebene
3. Online-Sicherung mit Point-in-time-Recovery

■ 4.1 SQL Dump

SQL Dump erstellt eine Textdatei mit SQL-Befehlen. Mit diesen Befehlen kann eine Datenbank teilweise oder komplett wiederhergestellt werden. Das Werkzeug heißt *pg_dump*, ist Bestandteil der PostgreSQL-Core-Programme und muss nicht zusätzlich installiert werden. Das Programm befindet sich im bin-Verzeichnis der PostgreSQL-Installation. „pg_dump“ verfügt über eine Reihe von Optionen, die mit `pg_dump --help` angezeigt werden. Ein Beispiel finden Sie in Listing 4.1.

**Praxistipp**

Verwenden Sie für „pg_dump“ immer die Option „-o“. Damit werden die Objekt-Identifizier (OID) mit ausgegeben. Diese sind wichtig für die Wiederherstellung, wenn Sie zum Beispiel Foreign Keys definiert haben. Beachten Sie auch, dass Large Objects nicht automatisch mit ausgegeben werden. Setzen Sie dafür die Option „-b“. Da man im „Worst Case“ davon ausgeht, dass die komplette Datenbank wiederhergestellt werden muss, sollte die Option „-C“ immer mit ausgewählt werden. Mit dieser Option wird ein Befehl zum Erstellen der Datenbank eingefügt.

Listing 4.1 Einen Datenbank-Dump erstellen

```
$ pg_dump -o -b -C -f /opt/postgres/backup/dump_20110512.sql hanser
```

Mit dem folgenden Test wollen wir eine verlorene Datenbank komplett aus dem Dump wiederherstellen. Dazu löschen wir die Datenbank mit dem „DROP DATABASE“-Befehl. Der erste Versuch scheitert jedoch, da noch Clients mit der Datenbank verbunden sind:

```
(postgres@[local]:5432) [postgres] > DROP DATABASE hanser;
ERROR:  database "hanser" is being accessed by other users
DETAIL:  There are 1 other session(s) using the database.
```

In dieser Situation ermitteln wir die Prozess-ID des Clients im Betriebssystem und killen diese. Danach kann die Datenbank gelöscht werden.

Listing 4.2 Eine Session killen und die Datenbank löschen

```
(postgres@[local]:5432) [postgres] > SELECT procpid,username,
application_name,client_addr
> FROM pg_stat_activity
> WHERE datname = 'hanser';
 procpid | username | application_name | client_addr
-----+-----+-----+-----
    10922 | postgres |                  | 192.168.178.21
$ kill -9 10922
(postgres@[local]:5432) [postgres] > DROP DATABASE hanser;
DROP DATABASE
```

Wie kann die Wiederherstellung der Datenbank erfolgen? Schauen Sie sich die ersten Zeilen im Dumpfile an.

Listing 4.3 Die ersten SQL-Befehle im Dumpfile

```
--
-- Name: hanser; Type: DATABASE; Schema: -; Owner: postgres
--
CREATE DATABASE hanser WITH TEMPLATE = template0 ENCODING = 'UTF8'
LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';
ALTER DATABASE hanser OWNER TO postgres;
\connect hanser
. . .
```


Der Kommentar liefert den Hinweis, dass der Eigentümer der Datenbank der Benutzer „postgres“ ist. Er hat damit alle Rechte, und die Anmeldung beim Rückspeichern sollte mit diesem Benutzer erfolgen. Weil wir die Option „-C“ beim Export verwendet haben, wurde ein „CREATE DATABASE“-Kommando eingefügt. Die Datenbank muss also nicht separat angelegt werden. Mit dieser Option ist es ein Leichtes, die Datenbank in einen anderen Server zurück zu speichern. Die nächste Anweisung ist ein CONNECT-Befehl, und es erfolgt die Anmeldung bei der Datenbank „hanser“. Der Benutzer „postgres“ ist Superuser und Eigentümer der Datenbank, so dass keine Authentifizierung über ein Passwort erforderlich ist. Für das Rückspeichern erfolgt die Anmeldung mit dem Benutzer „postgres“ bei der Datenbank „postgres“ (siehe Listing 4.4).

Listing 4.4 Eine Datenbank aus einem Dump wiederherstellen

```
$ psql < /opt/postgres/backup/dump_20110512.sql
SET
SET
SET
SET
SET
SET
SET
CREATE DATABASE
ALTER DATABASE
You are now connected to database "hanser".
SET
. . .
```

Damit wurde die Datenbank „hanser“ komplett wiederhergestellt. Eine einfache und sichere Methode. Die Vor- und Nachteile dieses Verfahrens betrachten wir am Ende dieses Abschnitts. Zunächst einige weitere Optionen von „pg_dump“. Im Idealfall läuft das Skript ohne Fehler durch. Doch was, wenn ein Fehler auftritt? In der Regel will man nicht, dass das Skript weiterlaufen soll, sondern dass es abgebrochen wird. Dafür können Sie sich eine Option von „psql“ zunutze machen:

```
$ psql --set ON_ERROR_STOP=on < /opt/postgres/backup/dump_20110512.sql
```



Praxistipp

Das Utility „pg_dump“ besitzt die Fähigkeit, von Pipes zu lesen und nach Pipes zu schreiben. Damit kann eine Datenbank über das Netzwerk direkt von Server zu Server kopiert werden. Mit dieser Methode wird kein zusätzlicher Platz auf Disk benötigt. Folgen Sie dem Beispiel in Listing 4.5. Darin wird die Datenbank „hanser“ von einem Windows-Server in einen Linux-Server kopiert.

Listing 4.5 Eine Datenbank direkt von Server zu Server kopieren

```
C:\Users\Lutz>pg_dump -C hanser | psql -h 192.168.178.201 -w
AUTOCOMMIT is on
SET
SET
. . .
```



```
ALTER DATABASE
Sie sind jetzt verbunden mit der Datenbank "hanser".
SET
SET
. . .
```

Für Entwicklungsdatenbanken ist der Dateninhalt häufig uninteressant. Entwickler wollen nur ihre Objekte ohne Inhalt sichern. Für diesen Zweck kann die Option „-s“ (schema only) verwendet werden. Das so erstellte Dumpfile weist eine reduzierte Größe auf.

„pg_dump“ ist aber auch gut geeignet, um Daten einer Tabelle zu sichern oder zu übertragen. Im folgenden Beispiel wird die Tabelle „kb“ mit 5 Millionen Sätzen entladen. Die Option „-a“ entlädt nur Daten:

```
$ pg_dump -t kb -a -f /tmp/exp_table_kb.sql
```

Im Dumpfile sieht man den COPY-Befehl zum Laden der Daten. Der COPY-Befehl ist eine effektive Methode zum Laden von Massendaten. Alle Daten werden als so genannter „Bulk Load“ in einem Befehl geladen anstelle von einzelnen Insert-Befehlen.

Listing 4.6 Der COPY-Befehl

```
COPY kb (id, text) FROM stdin;
1      KnowledgeBase
2      Inhaltsverzeichnis
3      -----
. . .
```

Es existiert ein weiteres Werkzeug mit dem Namen *pg_dumpall*. Damit werden alle Datenbanken des Clusters sowie Server-bezogene Daten wie Rollen und Tablespace-Definitionen gesichert. Somit ist die Wiederherstellung des gesamten Clusters möglich.

Die Vor- und Nachteile von SQL Dump sind nachfolgend zusammengefasst:

Vorteile von SQL Dump

- Es können einzelne Objekte einfach gesichert und wiederhergestellt werden.
- Objekte, Datenbanken oder ganze Cluster können zwischen verschiedenen Plattformen, sogar zwischen 32- und 64-Bit-Systemen kopiert werden. Ebenso kann ein Austausch über unterschiedliche Versionen erfolgen.
- Datenbanken können direkt von Server zu Server ohne den Umweg über die Disk kopiert werden.

Nachteile von SQL Dump

- Eine Point-in-time-Wiederherstellung ist nicht möglich. Alle Objekte können in dem Zustand zurückgespeichert werden, in dem sie zum Zeitpunkt der Sicherung vorgelegen haben. Alle nachträglich erfolgten Änderungen gehen verloren.
- Insbesondere für größere Datenbanken ist es nicht möglich, bei Online-Sicherungen einen konsistenten Zustand der Datenbank zu erstellen.

Die Methode ist gut für Test- und Entwicklungsdatenbanken zur Sicherung und einfachen Wiederherstellung von einzelnen Objekten geeignet. Für produktive Datenbanken ist es eine sinnvolle Ergänzung zum gewählten Sicherungsverfahren.

■ 4.2 Offline-Sicherung auf Dateisystem-Ebene

Der Ansatz für die zweite Sicherungsmethode ist, alle Dateien des Datenbank-Clusters mit Mitteln des Betriebssystems zu kopieren. Damit lässt sich ein konsistenter Zustand der Datenbanken sichern. Allerdings gibt es die wichtige Einschränkung, dass für den Kopiervorgang der PostgreSQL-Server heruntergefahren werden muss. Das Verfahren selbst ist allerdings recht einfach. Listing 4.7 beschreibt die notwendigen Schritte für eine Offline-Sicherung.

Listing 4.7 Eine Offline-Sicherung durchführen

```
$ pg_ctl stop -m fast
waiting for server to shut down.... done
server stopped
tar -cvf /opt/postgres/backup/backup_20110512.tar $PGDATA
$ pg_ctl start
```

Die Wiederherstellung erfolgt durch Rückspeichern aus der tar-Datei. Betrachten wir auch hier wieder die Vor- und Nachteile der Methode.

Vorteile der Offline-Sicherung

- Es erfolgt eine konsistente Sicherung des gesamten Cluster.
- Die Methode ist robust und zuverlässig.

Nachteile der Offline-Sicherung

- Auch hier ist wie beim SQL-Dump-Verfahren keine Point-in-time-Recovery möglich. Das Cluster kann nur im Zustand zum Zeitpunkt der Sicherung wiedergestellt werden.
- Einzelne Objekte können nur über den Umweg, den Server auf einer anderen Hardware zu starten, wiederhergestellt werden.
- Eine Wiederherstellung ist nur auf derselben Plattform möglich.
- Der PostgreSQL-Server muss für die Sicherung gestoppt werden und ist für die Anwendungen nicht verfügbar.

Die Methode besitzt eine Reihe von Nachteilen. Allerdings ist sie sehr einfach und zuverlässig, wenn Point-in-time Recovery nicht gefordert ist und die Downtime für die Sicherung kein Problem darstellt. Sie ist für die meisten produktiven Datenbanken nicht geeignet. Eine bessere Option bietet das folgende Verfahren.

■ 4.3 Online-Sicherung mit Point-in-time-Recovery

Diese Methode setzt eine aktivierte WAL-Archivierung voraus. Eine Anleitung zum Einschalten der WAL-Archivierung finden Sie in Abschnitt 3.2. Die folgenden Schritte beschreiben das Verfahren. Um nachzuweisen, dass keine Daten verloren gehen, legen wir eine Tabelle „backup_test“ an und tragen während der diversen Sicherungsphasen Sätze ein.

Listing 4.8 Eine Test-Tabelle anlegen

```
(postgres@[local]:5432) [hanser] > CREATE TABLE backup_test(
> zeit DATE, text VARCHAR(200));
CREATE TABLE
(postgres@[local]:5432) [hanser] > INSERT INTO backup_test VALUES
(current_date,'Before Backup');
INSERT 0 1
```

1. Führen Sie das folgende Kommando als Superuser aus. Es ist unerheblich, mit welcher Datenbank der Superuser verbunden ist, wenn er das Kommando absetzt. Es wird eine Backup-Label-Datei mit dem Namen *backup_label* im Cluster-Verzeichnis erstellt, die Informationen über die Sicherung enthält. Mit dem Absetzen des Kommandos führt der Server einen Checkpoint durch. Die Datenbanken befinden sich damit in einem konsistenten Zustand. Die folgende Sicherung liefert keinen konsistenten Zustand der Datenbanken. Die kann jedoch mit Hilfe der WAL-Segmente bei der Wiederherstellung ausgeglichen werden. Der Rückgabewert liefert die Sequenz-Nummer des aktuellen WAL-Segments („C4“).

```
(postgres@[local]:5432) [hanser] > SELECT pg_start_backup('Daily Backup');
pg_start_backup
-----
0/C4000020
```

2. Sichern Sie das Verzeichnis \$PGDATA mit Betriebssystemmitteln. Fügen Sie einen Satz in die Testtabelle ein. Die Sicherung muss alle Dateien einschließlich der Unterverzeichnisse des Datenbank-Clusters enthalten. Es ist sinnvoll, hier das Unterverzeichnis mit den WAL-Dateien (pg_xlog) bereits auszuschließen, da es später nicht zurückgeschrieben werden sollte.

```
(postgres@[local]:5432) [hanser] > INSERT INTO backup_test VALUES
(current_date,'Backup Mode');
INSERT 0 1
$ tar -cvf /opt/postgres/backup/online_backup.tar -X excl $PGDATA
```

3. An dieser Stelle werden 5 Millionen Sätze in die Tabelle „backup_test“ eingefügt. Dieser Schritt ist optional und nicht Bestandteil des Sicherungsszenarios. Damit kann nachgewiesen werden, dass die Strategie auch bei Belastung der Datenbank funktioniert.
4. Beenden Sie den Backup-Modus der Datenbank, und fügen Sie einen weiteren Satz in die Testtabelle ein. Es erfolgt ein Wechsel zum nächsten WAL-Segment. Damit kann das aktuelle archiviert werden. Mit Beendigung des Backup-Modus werden die Änderungen aus den WAL-Segmenten nachgepflegt. Die aktuelle Sequenz-Nummer ist „E4“.


```
(postgres@[local]:5432) [hanser] > SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL segments have been
archived
 pg_stop_backup
-----
 0/E4BA0598
(postgres@[local]:5432) [hanser] > INSERT INTO backup_test
VALUES(current_date,'after backup');
INSERT 0 1
```

Damit ist die Online-Sicherung abgeschlossen. PostgreSQL hat eine zusätzliche Datei mit der Erweiterung „.backup“ in das WAL-Verzeichnis geschrieben. Dabei handelt es sich um eine Textdatei mit Informationen über die Sicherung (siehe Listing 4.9).

Listing 4.9 Backup-Datei im WAL-Verzeichnis

```
$ cat 000000030000000000000000000000C4.00000020.backup
START WAL LOCATION: 0/C4000020 (file 000000030000000000000000000000C4)
STOP WAL LOCATION: 0/E4BA0598 (file 000000030000000000000000000000E4)
CHECKPOINT LOCATION: 0/C4000020
START TIME: 2011-05-13 09:05:27 CEST
LABEL: Daily Backup
STOP TIME: 2011-05-13 09:30:22 CEST
```

Wenn Sie an dieser Stelle in das Archive-Verzeichnis schauen, können Sie feststellen, dass alle WAL-Dateien bis zum Ende der Sicherung archiviert wurden.

Es folgt ein Test, um nachzuweisen, dass das Point-in-time-Recovery funktioniert. In die Tabelle „backup_test“ wird ein Satz eingefügt, danach ein manueller WAL-Switch erstellt und damit das aktuelle WAL-Segment archiviert. Wir halten die Zeit als Wiederherstellungszeitpunkt fest. Ein Crash des Servers kann am besten durch ein Herunterfahren mit der Option „immediate“ simuliert werden. Anschließend werden alle Dateien im Verzeichnis „\$PGDATA“ gelöscht.

Listing 4.10 Einen Server-Crash simulieren

```
(postgres@[local]:5432) [hanser] > INSERT INTO backup_test
> VALUES(current_date,'Before Crash');
INSERT 0 1
(postgres@[local]:5432) [hanser] > SELECT pg_switch_xlog();
 pg_switch_xlog
-----
 0/EA001810
$ pg_ctl stop -m immediate
waiting for server to shut down... done
server stopped
$ cd $PGDATA
$ rm -Rf *
```

Damit wurde eine Situation hergestellt, die einem Server-Crash mit Totalverlust der Datenbank entspricht. Die folgenden Schritte beschreiben den Prozess zur Wiederherstellung der Datenbank.

1. Speichern Sie das Verzeichnis \$PGDATA aus der Sicherung zurück.

```
$ tar -xvf /opt/postgres/backup/online_backup.tar
```


2. Löschen Sie, falls mitgesichert, alle Dateien im Unterverzeichnis `pg_xlog`. Damit werden alle WAL-Segmente aus dem Backup entfernt.
3. Legen Sie im Verzeichnis „`$PGDATA`“ eine Konfigurationsdatei `recovery.conf` an. Diese sollte mindestens den Parameter a) enthalten:
 - a) Ein Befehl zur Rückspeicherung der archivierten WAL-Dateien. Dieser Befehl ist abhängig vom Speicherort sowie dem Verfahren der Archivierung:

```
restore_command = 'gzip -d -c /opt/postgres/archive/%f.gz > %p'
```

- b) Setzen Sie die Werte für den Wiederherstellungszeitpunkt als Transaktions-ID oder Datum und Uhrzeit. Dies ist nur für ein Recovery zu einem bestimmten Zeitpunkt erforderlich. Wir verzichten darauf an dieser Stelle und gehen bis zum letzten verfügbaren WAL-Archiv.

```
#recovery_target_time = '2010-05-13 10:00:00 CET'
#recovery_target_xid = ''
```

4. Starten Sie den PostgreSQL-Server. Er geht in den Recovery-Modus und wendet die archivierten WAL-Dateien an, bis der Recovery-Zeitpunkt erreicht ist. Danach wird die Datei „`recovery.conf`“ automatisch in „`recovery.done`“ umbenannt. Schauen Sie in die Log-Datei, sie sollte in etwa die folgenden Einträge aufweisen:

```
pg_ctl start
LOG:  database system was interrupted while in recovery at log time 2011-
05-13 09:05:27 CEST
LOG:  restored log file "00000003.history" from archive
LOG:  starting archive recovery
LOG:  restored log file "000000030000000000000000C4" from archive
LOG:  redo starts at 0/C4000078
LOG:  restored log file "000000030000000000000000C5" from archive
LOG:  restored log file "000000030000000000000000C6" from archive
LOG:  restored log file "000000030000000000000000C7" from archive
. . .
LOG:  consistent recovery state reached at 0/E5000000
. . .
gzip: /opt/postgres/archive/000000030000000000000000EB.gz: No such file or
directory
LOG:  could not open file "pg_xlog/000000030000000000000000EB" (log file 0,
segment 235): No such file or directory
LOG:  redo done at 0/EA0017F0
LOG:  last completed transaction was at log time 2011-05-13
09:55:43.576423+02
LOG:  restored log file "000000030000000000000000EA" from archive
gzip: /opt/postgres/archive/00000004.history.gz: No such file or directory
LOG:  selected new timeline ID: 4
LOG:  restored log file "00000003.history" from archive
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

Damit ist die Wiederherstellung abgeschlossen, und die Datenbanken können dem Anwender wieder zur Verfügung gestellt werden. Die Wiederherstellung ist bis zum letzten verfügbaren WAL-Archiv erfolgt. Dies bestätigt der letzte Eintrag in der Tabelle „`backup_test`“:

zeit	text
2011-05-13	Before Crash

**Praxistipp**

Eine Sicherung kann nur als solche angesehen werden, wenn Sie auf ein externes Medium übertragen wird. Im Extremfall geht der gesamte Datenbankserver verloren. Sorgen Sie dafür, dass die Archivierung der WAL-Dateien in regelmäßigen Abständen auf ein externes Medium (NFS Share, Tape, entfernter Server) erfolgt. Letztendlich ist die Wiederherstellbarkeit nach dem Point-in-time-Recovery-Prinzip nur bis zur letzten archivierten WAL-Datei möglich.

Die Methode „Online-Sicherung mit Point-in-time Recovery“ bietet die folgenden Vor- und Nachteile.

Vorteile der PITR-Sicherung

- Zusammen mit den WAL-Dateien erfolgt eine konsistente Sicherung des gesamten Datenbank-Cluster.
- Die Sicherung erfolgt im laufenden Betrieb und trägt damit zu einer erhöhten Verfügbarkeit der Datenbanken bei.
- Die Wiederherstellung der Datenbanken kann zu einem beliebigen Zeitpunkt erfolgen (letztes verfügbares WAL-Segment).

Nachteile der PITR-Sicherung

- Einzelne Objekte können nur über den Umweg, den Server auf einer anderen Hardware zu starten, wiederhergestellt werden.
- Eine Wiederherstellung ist nur auf derselben Plattform möglich.
- Die Wiederherstellung erfordert mehrere Aktivitäten durch den Administrator.

Aus den aufgelisteten Vorteilen ist ersichtlich, dass diese Methode die erste Wahl für produktive Datenbanken ist. Regelmäßige SQL Dumps, auch ohne Daten, stellen eine sinnvolle Ergänzung dar und entschärfen den Nachteil der aufwendigen Wiederherstellbarkeit einzelner Objekte.

■ 4.4 Funktionen zur Sicherung und Wiederherstellung

Die Wiederherstellung aus einer PITR-Sicherung erfordert in einigen Situationen die Kenntnis des WAL-Status. PostgreSQL stellt einige Funktionen zur Verfügung, um diesen abzufragen. Die Funktion *pg_current_xlog_location* liefert die aktuelle Position des Schreibens in das Transaktionslog im internen Format.

```
(postgres@[local]:5432) [hanser] > SELECT pg_current_xlog_location();
pg_current_xlog_location
-----
0/F9000000
```


Auskunft über den zugehörigen Dateinamen gibt die Funktion *pg_xlogfile_name*:

```
(postgres@[local]:5432) [hanser] > SELECT pg_xlogfile_name(pg_current_xlog_location());
 pg_xlogfile_name
-----
00000004000000000000000F8
$ ls -ltr $PGDATA/pg_xlog
. . .
-rw----- 1 postgres postgres 16777216 May 13 21:45 00000004000000000000000F8
```

Die Funktion *pg_is_in_recovery* gibt Auskunft darüber, ob gerade ein Recovery durchgeführt wird.

```
(postgres@[local]:5432) [hanser] > SELECT pg_is_in_recovery();
 pg_is_in_recovery
-----
f
```

Sicherheit und Überwachung sind wichtige Komponenten für den Einsatz von Datenbanken im betrieblichen Umfeld. Für die Datenbank-Sicherheit existieren häufig betriebliche Vorgaben. Große Konzerne versuchen, globale Security-Richtlinien umzusetzen. Neben den betrieblichen Vorgaben, die häufig lokale und interne Befindlichkeiten enthalten, gibt es Vorgaben von weltweit tätigen Sicherheits-Unternehmen oder beratenden Organisationen. Eine breite Akzeptanz haben im Datenbank-Umfeld die folgenden gefunden:

- Health Insurance Portability and Accountability Act (HIPPA)
- Richtlinien des Bundesamts für Sicherheit in der Informationstechnik (BSI): Abschnitt B 5.7
- Sarbanes-Oxley Compliance Model (SOX)

HIPPA war einer der ersten Sicherheits-Standards, der in den USA in einem Industriezweig konsequent umgesetzt wurde. Auch wenn er etwas in die Jahre gekommen ist, liefern die Richtlinien eine gute Basis für Datenbank-Sicherheit (auch für andere Industriezweige).

Die BSI-Richtlinien sind eher auf Deutschland ausgerichtet und werden gerne von der öffentlichen Hand genutzt. Auch sie liefern einen guten Standard, auf dem sich aufbauen lässt.

Das SOX Compliance Model enthält im Vergleich zu anderen Modellen verschärfte Sicherheits-Auflagen und wird von Unternehmen häufig zur Risiko-Minimierung für sensible und wichtige Datenbanken eingesetzt. Während man die BSI-Richtlinien in einigen Bereichen sehr detailliert beschrieben vorfindet (manchmal auch zu detailliert), sind die Regularien im SOX-Modell allgemeiner gehalten, was nicht weniger restriktiv oder weniger Sicherheit bedeutet.



Praxistipp

Wenn in Ihrem Unternehmen keine Richtlinien für Sicherheit und insbesondere für Datenbank-Sicherheit vorhanden sind, empfiehlt es sich, mit einem Standard zu beginnen und diesen systematisch zu erweitern. Auch die hier genannten Modelle sollten nicht eins zu eins übertragen, sondern kritisch betrachtet und gegebenenfalls angepasst oder erweitert werden.

Eine weitere wichtige Voraussetzung für die Umsetzung von Sicherheits-Richtlinien ist ein hoher Standardisierungsgrad im Datenbank-Betrieb. Je höher der Standardisierungsgrad, desto geringer der Aufwand für die Umsetzung und Bewahrung der Richtlinien im betrieblichen Umfeld.

Ein weiterer Sicherheitsaspekt ist der Bereich Überwachung (Monitoring) von Datenbanken. Eine konsequente Überwachung dient nicht nur der Bewältigung von Sicherheitsaufgaben, sondern trägt entscheidend zur Erhöhung der Verfügbarkeit bei. Dies ist das zweite große Thema, mit dem wir uns in diesem Kapitel beschäftigen.

■ 5.1 Sicherheit

PostgreSQL verfügt über einen hohen Standard sowie viele Optionen für einen sicheren Datenbankbetrieb. Mit den wichtigsten Komponenten wollen wir uns in diesem Kapitel beschäftigen. Dabei lernen Sie die einzelnen Sicherheits-Komponenten kennen. Die letzte Entscheidung, die Sie hinsichtlich der Sicherheit Ihrer Datenbanken für relevant halten, liegt natürlich bei Ihnen und Ihrem Unternehmen.



Praxistipp

Häufig ist es sinnvoll, eine Abstufung der eingesetzten Datenbanken vorzunehmen. Nicht alle Datenbanken benötigen denselben Sicherheits-Standard. In der Regel genügt die Datenbank, die den Speiseplan enthält, geringeren Anforderungen als die Datenbank, die zur Produktionssteuerung eingesetzt wird. Betrachten Sie dies auch unter dem Aspekt, dass erhöhte Sicherheit mit erhöhten Kosten verbunden ist. Auch wenn für PostgreSQL keine Zusatzkosten für Lizenzen anfallen (wir nehmen an, dass alle Features mit Open-Source-Software abgedeckt werden kann), verursacht verschärfte Sicherheit zusätzliche organisatorische und IT-technische Maßnahmen.

5.1.1 Datenbank-Rollen und -Privilegien

PostgreSQL verwendet ein Rollenkonzept. Eine Rolle kann ein Datenbank-Benutzer oder auch eine Gruppe von Benutzern sein. Rollen können Objekte in der Datenbank oder Privilegien an Objekten besitzen. Rollen können anderen Rollen zugeordnet werden. Im Vergleich mit anderen Datenbanksystemen besteht der Unterschied nur in der Terminologie. PostgreSQL bezeichnet sowohl einzelne Benutzer als auch Gruppen von Benutzern als Rollen.



Hinweis

Beachten Sie, dass Rollen im PostgreSQL-Server unabhängig von Accounts im Betriebssystem sind. Die Benutzerverwaltung von PostgreSQL ist von der des Betriebssystems entkoppelt, auch wenn die Benutzernamen möglicherweise identisch sind (zum Beispiel Benutzer „postgres“).

Das Anlegen einer Rolle erfolgt mit dem folgenden Befehl:

```
(postgres@[local]:5432) [hanser] > CREATE ROLE hanser;  
CREATE ROLE
```

Die Rolle hat noch keine Rechte, nicht einmal zur Anmeldung an der Datenbank. Ein Verbindungsversuch schlägt deshalb fehl:

```
$ psql -U hanser  
psql: FATAL:  role "hanser" is not permitted to log in
```

Nachdem die Rolle das Attribut „LOGIN“ erhalten hat, kann man sie bei der Datenbank anmelden:

```
(postgres@[local]:5432) [hanser] > ALTER ROLE hanser LOGIN;  
ALTER ROLE
```

In Tabelle 5.1 finden Sie eine Übersicht aller Attribute, die Rollen zugeordnet werden können.

Tabelle 5.1 Attribute von Rollen

Attribut	Beschreibung
LOGIN	Die Rolle kann sich an einer Datenbank anmelden.
SUPERUSER	Es werden alle Prüfungen von Zugriffsrechten umgangen. Die Rolle hat Zugriff auf alle Objekte der Datenbank.
CREATEDB	Die Rolle hat das Privileg, eine Datenbank anzulegen.
CREATEROLE	Der Benutzer darf weitere Rollen anlegen, ändern oder löschen. Er hat keine Rechte zur Bearbeitung von Rollen mit dem Attribut „Superuser“.
PASSWORD	Dem Benutzer wird ein Passwort zugewiesen.

Um Benutzer oder Rollen anlegen zu können, muss man selbst das Superuser-Privileg oder das Privileg „CREATEROLE“ besitzen. Beim Anlegen des Datenbankservers (initdb) wird die vordefinierte Rolle „postgres“ als Superuser angelegt. Diese kann benutzt werden, um weitere Rollen anzulegen.



Hinweis

Der Befehl „CREATE USER“ existiert noch aus Kompatibilitätsgründen und ist nahezu identisch mit dem Befehl „CREATE ROLE“. Ein Unterschied besteht darin, dass bei Anwendung von „CREATE USER“ das Attribut „LOGIN“ automatisch zugewiesen wird.

Wenn ein Projekt erstellt wurde, gehört es dem Besitzer, der es angelegt hat. Damit andere Rollen Zugriff erhalten, müssen diesen explizit Rechte zugewiesen werden. Eine Ausnahme bilden die Superuser, die automatisch Zugriff auf alle Objekte haben. In Listing 5.1 finden Sie einige Beispiele zur Rechte-Bearbeitung:

Listing 5.1 Objekt-Rechte zuweisen und entziehen

```
(hanser@[local]:5432) [hanser] > GRANT SELECT ON order_info TO PUBLIC;
GRANT
(hanser@[local]:5432) [hanser] > GRANT UPDATE ON order_info TO hanser;
GRANT
(hanser@[local]:5432) [hanser] > REVOKE SELECT ON order_info FROM PUBLIC;
REVOKE
(hanser@[local]:5432) [hanser] > REVOKE ALL ON order_INFO FROM PUBLIC;
REVOKE
```

Wie bereits erwähnt, können Rollen auch als Gruppen fungieren – einfach über die Zuweisung von Rollen zu Rollen. So kann man zum Beispiel eine Read-only-Rolle erstellen, die SELECT-Privilegien auf Tabellen erhält und anderen Rollen (Benutzern) zugewiesen wird. Die Rolle benötigt nicht das Attribut „LOGIN“ und auch kein Passwort, weil sich ja niemand mit dieser Rolle anmelden soll. In Listing 5.2 finden Sie ein Beispiel.

Listing 5.2 Eine Rolle mit Gruppenfunktion anlegen

```
(postgres@[local]:5432) [hanser] > CREATE ROLE read_only;
CREATE ROLE
(postgres@[local]:5432) [hanser] > GRANT SELECT ON order_info TO read_only;
GRANT
(postgres@[local]:5432) [hanser] > GRANT SELECT ON orders TO read_only;
GRANT
. . .
(postgres@[local]:5432) [hanser] > GRANT read_only TO hanser;
GRANT ROLE
(postgres@[local]:5432) [hanser] > GRANT read_only TO guest;
GRANT ROLE
. . .
```

Mit der Zuweisung der Rolle „read_only“ erhält der User „hanser“ SELECT-Rechte auf die Tabelle „order_info“. Die Abfrage in Listing 5.3 bestätigt die Zuweisung.

Listing 5.3 Gruppenrollen testen

```
(hanser@192.168.178.100:5432) [postgres] > \conninfo
You are connected to database "postgres" as user "hanser" on host
"192.168.178.100" at port "5432".
(hanser@192.168.178.100:5432) [postgres] > SELECT * FROM order_info;
 id | text
-----+-----
4711 | Order complete.
(hanser@192.168.178.100:5432) [postgres] > UPDATE order_info SET text =
'Order incomplete' WHERE id = 4711;
ERROR: permission denied for relation order_info
```

Rollen können in „psql“ mit dem Kurzbefehl „du“ angezeigt werden. Zusätzliche Informationen liefert das Dictionary-View „pg_roles“.

Listing 5.4 Vorhandene Rolle abfragen

```
(hanser@192.168.178.100:5432) [postgres] > \du
List of roles

```

Role name	Attributes	Member of
hanser		{read_only}
postgres	Superuser, Create role, Create DB, Replication	{}
read_only	Cannot login	{}

```
(hanser@192.168.178.100:5432) [postgres] > SELECT
rolname,rolsuper,rolcanlogin,rolpassword FROM pg_roles;
rolname | rolsuper | rolcanlogin | rolpassword
-----+-----+-----+-----
postgres | t        | t          | *****
hanser   | f        | t          | *****
read_only | f        | f          | *****
```

**Hinweis**

Die Katalog-Sichten „pg_user“ und „pg_shadow“ sind nur noch aus Kompatibilitätsgründen vorhanden und sollten möglichst nicht mehr verwendet werden.

Wie Sie sehen, liefert das View „pg_roles“ Sterne in der Passwort-Spalte, unabhängig davon, ob ein Passwort vergeben ist oder nicht. Das verschlüsselte Passwort befindet sich im View „pg_authid“.

Listing 5.5 Das verschlüsselte Passwort abfragen

```
(postgres@192.168.178.100:5432) [postgres] > SELECT rolname,rolpassword
FROM pg_authid;
rolname | rolpassword
-----+-----
postgres |
read_only |
hanser   | md582fee51651ce30a62e87715d60c3f93e
```

Will man verhindern, dass sich ein User temporär anmeldet, ist es ausreichend, die Login-Rolle zu entfernen. Damit muss der User nicht komplett neu mit all seinen Attributen angelegt werden. Es genügt, das Login-Privileg wieder zu vergeben.

Listing 5.6 Die Login-Rolle temporär entziehen

```
(postgres@192.168.178.100:5432) [postgres] > ALTER ROLE hanser NOLOGIN;
ALTER ROLE
(postgres@192.168.178.100:5432) [postgres] > \c - hanser - -
FATAL:  role "hanser" is not permitted to log in
Previous connection kept
(postgres@192.168.178.100:5432) [postgres] > ALTER ROLE hanser LOGIN;
ALTER ROLE
(postgres@192.168.178.100:5432) [postgres] > \c - hanser - -
You are now connected to database "postgres" as user "hanser".
(hanser@192.168.178.100:5432) [postgres] >
```




Praxistipp

Schwache oder unverschlüsselte Passwörter stellen ein signifikantes Sicherheitsrisiko dar. Deshalb sollte man vermeiden, solche Passwörter in der Datenbank zuzulassen. Neben regelmäßigen Security Scans ist die Implementierung eines Sicherheits-Moduls, das schwache Passwörter beim Anlegen von Rollen ablehnt, die bevorzugte Methode.

Wie bereits erwähnt, können Rollen anderen Rollen (oder Benutzer) zugewiesen werden. Halten Sie sich stets vor Augen, dass es in der Version 9 faktisch keinen Unterschied mehr zwischen Rollen und Benutzern sowie zwischen Gruppenrollen und Nicht-Gruppenrollen gibt. Dabei kann die Mitgliedschaft in einer Gruppenrolle auf zwei verschiedene Arten genutzt werden:

1. Der Benutzer verwendet das Kommando „SET ROLE <rollenname>“; erhält damit temporär exakt die Rechte der Gruppenrolle und verliert die individuellen Rechte des angemeldeten Benutzers. Auch für Datenbank-Objekte, die er in dieser Zeit anlegt, wird die Gruppenrolle als Besitzer eingetragen. Der Benutzer kann seine Originalrechte mit dem Befehl „RESET ROLE“ wiederherstellen.
2. Mitglieder einer Gruppenrolle, die über das Attribut „INHERIT“ verfügen, besitzen automatisch die Rechte der Gruppenrolle sowie alle Rechte, die an die Gruppenrolle vererbt wurden. In Listing 5.7 finden Sie ein Beispiel, wie ein solches Konstrukt angelegt werden kann. In diesem Fall erhält der Benutzer „mitglied“ automatisch das Leserecht auf die Tabelle „version“.

Listing 5.7 Automatische Weitergabe der Rechte einer Gruppenrolle

```
CREATE ROLE mitglied LOGIN INHERIT;
CREATE ROLE gruppen_rolle;
GRANT SELECT ON version TO gruppen_rolle;
GRANT gruppen_rolle TO mitglied;
```

Um das Anlegen schwacher Passwörter zu verhindern, liefert PostgreSQL ein Beispielprogramm, das in den Server eingebunden werden kann. Die können das Programm ändern und der Passwort-Policy Ihres Unternehmens anpassen. Die folgenden Schritte beschreiben, wie ein Passwort-Check in den PostgreSQL-Server eingebunden werden kann:

3. Wechseln Sie als User „root“ in das Installationsverzeichnis der Installation aus dem Quellcode und weiter in das Unterverzeichnis „../contrib/passwordcheck“. Dort befindet sich das C-Programm „passwordcheck.c“. Kompilieren und linken Sie das Programm mit dem Make-Utility.

```
[root]# pwd
/root/postgresql-9.1.1/contrib/passwordcheck
[root]# make
. . .
```

4. Kopieren Sie die Bibliothek „passwordcheck.so“ in das Lib-Verzeichnis des PostgreSQL-Servers.

```
[root]# cp passwordcheck.so /usr/local/pgsql/lib/
```


5. Passen Sie den Parameter „shared_preload_libraries“ in der Konfigurationsdatei „postgresql.conf“ an, und führen Sie einen Neustart des Servers durch.

```
shared_preload_libraries = '/usr/local/pgsql/lib/passwordcheck'
```

Mit diesem Setup wird verhindert, dass Rollen mit schwachen Passwörtern überhaupt angelegt werden.

Listing 5.8 Ablehnung eines schwachen Passworts

```
(postgres@192.168.178.100:5432) [postgres] > CREATE ROLE test PASSWORD
'test1234';
ERROR: password must not contain user name
```

Für den Betrieb von Produktionsdatenbanken empfiehlt sich folgende Konfiguration: Es wird ein Schema-Eigentümer eingerichtet, dem die Objekte der Applikation gehören. Der User wird im Normalbetrieb gesperrt. Für ein neues Deployment oder Upgrade der Applikation wird der Account geöffnet, um die erforderlichen DDL-Anweisungen auszuführen. Der Zugriff der Applikation erfolgt durch einen zusätzlichen User in der Datenbank. Diesem User werden Objekt-Privilegien an den Objekten des Eigentümers zugewiesen. Dieses Vorgehen bietet die folgenden Vorteile:

- Eine unberechtigte Anwendung außerhalb des Change-Request-Verfahrens wird verhindert.
- Das Erlangen weiter reichender Privilegien durch das Upgrade-Skript kann ausgeschlossen werden.

In Listing 5.9 finden Sie ein Beispiel für eine solche Konfiguration. Die Rolle „app_owner“ ist der Eigentümer der Objekte. „app_user“ ist der Account, mit dem sich die Applikation verbindet.

Listing 5.9 Anlegen von Owner und User Account

```
(postgres@192.168.178.100:5432) [postgres] > CREATE ROLE app_owner NOLOGIN
PASSWORD 'start1234';
CREATE ROLE
(postgres@192.168.178.100:5432) [postgres] > CREATE SCHEMA AUTHORIZATION
app_owner;
CREATE SCHEMA
(postgres@192.168.178.100:5432) [postgres] > CREATE ROLE app_user
LOGIN PASSWORD 'start9876';
CREATE ROLE
(postgres@192.168.178.100:5432) [postgres] > CREATE TABLE app_owner.
version(text CHAR(40));
CREATE TABLE
```

Neues Feature

Ein neues Feature in der Version 9 ermöglicht die Zuweisung von Rechten für alle Objekte, die einem User gehören. Eine Aktualisierung ist nicht erforderlich, wenn neue Objekte hinzukommen.

Listing 5.10 Zuweisung von Rechten für alle Objekte eines Users

```
(postgres@192.168.178.100:5432) [postgres] > GRANT SELECT
> ON ALL TABLES IN SCHEMA app_owner TO app_user;
GRANT
(postgres@192.168.178.100:5432) [postgres] > ALTER DEFAULT PRIVILEGES FOR
ROLE app_owner
> GRANT SELECT ON TABLES TO app_user;
ALTER DEFAULT PRIVILEGES
. . .
You are now connected to database "postgres" as user "app_user".
(app_user@192.168.178.100:5432) [postgres] > SELECT * FROM app_owner.version;
 text
-----
11.2.0.3
(1 row)
```

5.1.2 Authentifizierung und Zugangskontrolle

Sendet ein Client eine Verbindungsanfrage an einen Datenbankserver, wird anhand des Benutzernamens entschieden, ob eine Verbindung aufgebaut wird. Ist der Benutzer angemeldet, kann er sich auf dem Datenbankserver entsprechend der ihm zugewiesenen Rechte bewegen.

Vor der Anmeldung findet der Authentifizierungsprozess statt. Dabei wird die Identität des Clients ermittelt und geprüft, ob der Client berechtigt ist, sich am Server anzumelden. Die Art und Weise, wie der Authentifizierungsprozess stattfindet, ist konfigurierbar. Die Konfiguration ist in der Datei „pg_hba.conf“ gespeichert. Die Standard-Auslieferung finden Sie in Listing 5.11.

Listing 5.11 Out-of-the-box-Standard der Konfigurationsdatei „pg_hba.conf“

# TYPE	DATABASE	USER	ADDRESS	METHOD
# "local" is for Unix domain socket connections only				
local	all	all		trust
# IPv4 local connections:				
host	all	all	127.0.0.1/32	trust
# IPv6 local connections:				
host	all	all	::1/128	trust
# Allow replication connections from localhost, by a user with the # replication privilege.				
#local	replication	postgres		trust
#host	replication	postgres	127.0.0.1/32	trust
#host	replication	postgres	::1/128	trust

Die Datei besteht aus fünf Spalten. Bei mehrfachen Einträgen wird der erste passende verwendet. Die erste Spalte reflektiert den Verbindungstyp mit folgenden möglichen Einträgen:

- *local*: Verbindung über Unix-Domain-Socket
- *host*: Verbindung über TCP/IP-Sockets
- *hostssl*: SSL-Verbindungen über TCP/IP-Sockets
- *hostnossl*: Nicht-SSL-Verbindungen über TCP/IP-Sockets

In der Spalte „DATABASE“ kann die Datenbank spezifiziert werden, für die der Eintrag gültig ist. Der Wert „all“ bedeutet, dass der Eintrag für alle Datenbanken gültig ist. Analog ist die Spalte „USER“ für den Login-Namen zu betrachten.

Die Spalte „ADDRESS“ spezifiziert den Hostnamen, die IP-Adresse oder einen Bereich von IP-Adressen, für den der Eintrag gültig ist. Durch einen Schrägstrich getrennt, kann neben der IP-Adresse die CIDR-Maske angegeben werden. Ein typisches Beispiel für einen einzigen Host wäre:

```
192.168.178.100/32
```

Der Eintrag „0.0.0.0/0“ steht für alle IPv4-Adressen. Damit wird der Eintrag für alle Clients als valide angesehen. Weitere gültige Werte:

- *all*: Steht für alle IP-Adressen und hat dieselbe Wirkung wie „0.0.0.0/0“.
- *samehost*: Steht für die IP-Adresse des Servers.
- *samenet*: Alle IP-Adressen im Subnetz des Servers.

Alle sonstigen Namen werden als Hostnamen interpretiert, und PostgreSQL versucht, diese über einen DNS-Eintrag aufzulösen. Die Auflösung erfolgt nicht case-sensitiv.



Praxistipp

Stellen Sie sicher, dass die DNS-Auflösung schnell genug ist, wenn Sie Hostnamen in der Datei „pg_hba.conf“ verwenden. Andernfalls kann es zu einer Verlangsamung der Authentifizierung und somit des Verbindungsaufbaus kommen.

Der Eintrag in der letzten Spalte „METHOD“ legt die Authentifizierungsmethode fest. Die wichtigsten zulässigen Einträge sind:

- *trust*: Verbindungen werden ohne Einschränkung zugelassen. Es ist keine weitere Authentifizierung oder die Angabe eines Passworts erforderlich.
- *reject*: Verbindungsversuche werden ohne weitere Prüfungen abgewiesen. Diese Methode ist sinnvoll, um bestimmte Hosts (Clients) generell auszuschließen.
- *md5*: Der Client muss ein gültiges, mit MD5 verschlüsseltes Passwort liefern, um den Authentifizierungsprozess erfolgreich abzuschließen.
- *password*: Der Client muss ein gültiges, nicht verschlüsseltes Passwort liefern, um den Authentifizierungsprozess erfolgreich abzuschließen.
- *krb5*: Kerberos V5 wird für die Authentifizierung verwendet.
- *ldap*: Die Authentifizierung erfolgt über einen LDAP-Server.
- *ident*: Ident-basierte Authentifizierung.



Hinweis

Beachten Sie, dass ein unverschlüsseltes Passwort auch unverschlüsselt über das Netzwerk übertragen wird. Es stellt somit ein erhöhtes Sicherheitsrisiko dar und sollte nur in Ausnahmefällen, keinesfalls aber in einem nicht-vertrauenswürdigem Netzwerk verwendet werden.

Bei der Authentifizierungsmethode „LDAP“ leitet PostgreSQL die Aufgabe an einen LDAP-Server weiter. Dieses Vorgehen bietet sich an, wenn man Passwörter für mehrere Applikationen vereinheitlichen will (Single Sign On).

Um eine LDAP-Authentifizierung zu konfigurieren, schreibt man – siehe das Beispiel in Listing 5.12 – die LDAP-URL in die Datei „pg_hba.conf“.

Listing 5.12 Eintrag für eine LDAP-basierende Authentifizierung

```
host      all      all      192.168.178.0/24      ldap      "ldap[s]://
servername[:port]/base dn[:prefix[:suffix]]"
```

Das Konzept der Ident-basierten Authentifizierung sieht vor, dass sich der Client bereits beim Betriebssystem authentifiziert hat und somit automatisch das Recht besitzt, sich beim Datenbank-Server anzumelden. Im Authentifizierungsprozess von PostgreSQL muss somit geklärt werden, in Form welchen Users der Client im Betriebssystem angemeldet ist und welche Datenbank-User er zur Anmeldung verwenden darf. Die Art der Ermittlung des Benutzernamens im Betriebssystem ist abhängig vom Verbindungstyp. Im Falle einer Verbindung über Unix-Domain-Sockets fragt der Datenbankserver direkt den Betriebssystemkern ab, bei einer Verbindung über TCP/IP wird der Benutzername über den Ident-Server ermittelt.

Steht die Identität des Betriebssystembenutzers fest, überprüft PostgreSQL, ob eine Verbindung zwischen dem Client und dem Datenbankbenutzer hergestellt werden darf.

5.1.3 Rechteverwaltung (Privilegien)

Der Zugriff auf Datenbank-Objekte wird durch das Rechtesystem kontrolliert. Damit soll in erster Linie ein unberechtigter Zugriff auf die Objekte verhindert werden.

Der Eigentümer eines Objekts ist derjenige, der es erstellt hat. Der Eigentümer hat automatisch das Recht, Änderungen am Objekt vorzunehmen, ohne dass zusätzliche Privilegien vergeben werden müssen. Er kann zum Beispiel auch Änderungen im Inhalt einer Tabelle vornehmen, also so genannte DML-Operationen durchführen. Rechte wie das Löschen, Umbenennen oder Hinzufügen einer Spalte sind ausschließlich dem Eigentümer vorbehalten.

Ein Eigentümer kann nachträglich geändert werden. So gehört die in Listing 5.9 angelegte Tabelle „version“ der Rolle „postgres“. Der Superuser „postgres“ besitzt das Recht, den Eigentümer der Tabelle zu ändern.

Listing 5.13 Den Eigentümer einer Tabelle ändern

```
(postgres@[local]:5432) [postgres] > SELECT schemaname,tableowner,tablename
FROM pg_tables;
      schemaname      | tableowner |      tablename
-----+-----+-----
app_owner             | postgres  | version
(postgres@[local]:5432) [postgres] > ALTER TABLE app_owner.version OWNER TO
app_owner;
ALTER TABLE
```



```
(postgres@[local]:5432) [postgres] > SELECT schemaname,tableowner,tablename
FROM pg_tables;
schemaname      | tableowner |      tablename
-----+-----+-----
app_owner       | app_owner  | version
```

Da Rechte an Objekten an andere Benutzer (Rollen) vergeben werden müssen, hält PostgreSQL Privilegien bereit. Neben granularen Privilegien, wie zum Beispiel das Lesen einer Tabelle, gibt es zusätzlich das Privileg „ALL“, das abhängig vom Objekt-Typ, alle granularen Rechte beinhaltet.

Listing 5.14 Alle Privilegien zuweisen

```
(postgres@[local]:5432) [postgres] > CREATE ROLE dba;
CREATE ROLE
(postgres@[local]:5432) [postgres] > GRANT ALL PRIVILEGES ON DATABASE
postgres TO dba;
GRANT
```

Wie bereits erwähnt, sind Privilegien abhängig vom Objekt. In Tabelle 5.2 finden Sie die Privilegien für Tabellen und Views.

Tabelle 5.2 Privilegien für Tabellen und Views

Privileg	Erklärung
SELECT	Erlaubt das Lesen einer Tabelle mit „SELECT“ oder „COPY FROM“.
INSERT	Recht zum Einfügen mit „INSERT“ oder „COPY TO“.
UPDATE	Erlaubt „UPDATE“, „SELECT FOR UPDATE“, „SELECT FOR SHARE“.
DELETE	Löschen mit „DELETE“ (nicht mit „TRUNCATE“).
TRIGGER	Recht zum Anlegen eines Triggers an der Tabelle.
REFERENCES	Recht, einen Fremdschlüssel zu definieren. Wird an beiden beteiligten Tabellen benötigt.



Hinweis

Hat ein Update- oder ein Delete-Befehl eine Suchbedingung, dann wird zusätzlich das SELECT-Privileg benötigt. „SELECT FOR UPDATE“ und „SELECT FOR SHARE“ benötigen ebenfalls zusätzlich das SELECT-Privileg.

Für Funktionen gibt es genau einen Privilegtyp, das EXECUTE-Recht. Dieses Recht wird bei neu angelegten Funktionen automatisch für „PUBLIC“ gesetzt. Die Privilegtypen für Sequenzen finden Sie in Tabelle 5.3.

Die bisher betrachteten Privilegien galten für einzelne Objekte. PostgreSQL hält aber auch Privilegien bereit, die für ein ganzes Schema oder eine ganze Datenbank gelten (Tab. 5.4).

Tabelle 5.3 Privilegien für Sequenzen

Privileg	Erklärung
SELECT	Die Funktion „curval“ darf verwendet werden.
UPDATE	Erlaubt die Verwendung der Funktionen „nextval“ und „setval“.
USAGE	Die Funktionen „curval“ und „nextval“ dürfen verwendet werden. Das ist das Standard-Privileg für Sequenzen.

Tabelle 5.4 Privilegien für Schemas

Privileg	Erklärung
CREATE	Berechtigung zum Erzeugen neuer Objekte im Schema. Ist der aktuelle Benutzer gleichzeitig Eigentümer des Objekts, schließt das Privileg das Recht ein, das Objekt umzubenennen.
USAGE	Erlaubt die Verwendung der Objekte im Schema.

Tabelle 5.5 Privilegien für Datenbanken

Privileg	Erklärung
CONNECT	Berechtigt zur Verbindung mit der Datenbank. Dieses Recht wird zusätzlich zu den Einträgen in der Datei „pg_hba.conf“ sowie dem Login-Attribut von Rollen geprüft. Es wird bei neuen Datenbanken automatisch auf „PUBLIC“ gesetzt.
CREATE	Gestattet das Anlegen neuer Schemas in der Datenbank.
TEMP	Mit diesem Privileg können temporäre Tabellen in der Datenbank angelegt werden. Es wird bei neuen Datenbanken standardmäßig auf „PUBLIC“ gesetzt.

Schließlich gibt es noch das Privileg „CREATE“ für Tablespaces. Dieses berechtigt zum Anlegen von Datenbanken mit dem Tablespace als Default-Tablespace.

Objektprivilegien kann im Ursprung nur der Eigentümer vergeben. Das Recht zur Vergabe von Privilegien lässt sich mit der Option „WITH GRANT OPTION“ weiterreichen. Im Beispiel in Listing 5.15 hat die Rolle „hr2“ das Recht, das SELECT-Privileg an der Tabelle „employees“ an andere Rollen zu vergeben.

Listing 5.15 Die Rechtevergabe an einem Objekt weiterreichen

```
GRANT SELECT ON TABLE employees TO hr2 WITH GRANT OPTION;
```

Für das Anzeigen von Privilegien auf Tabellen hält das Utility „psql“ das Kürzel „\z“ bereit. Ein Beispiel finden Sie in Listing 5.16. Das Ausgabe-Format ist:

```
GRANTEE=PRIVILEGE/GRANTOR
```

Dabei ist „GRANTEE“ der Empfänger der Privilegien und „GRANTOR“ derjenige, der sie erteilt hat. Die Privilegien selbst sind mit den folgenden Buchstaben verschlüsselt:

- a: APPEND (für INSERT-Befehle)
- w: WRITE (für UPDATE-Befehle)

- d: DELETE
- r: READ (für SELECT-Anweisungen)
- x: REFERENCES
- t: TRIGGER

Ein „*“ hinter dem Buchstaben bedeutet, dass das Privileg mit der GRANT-Option erteilt wurde. Die Rolle hat also das Recht, dieses Privileg weiterzugeben.

Listing 5.16 Tabellen-Privilegien in „psql“ anzeigen

```
(app_owner@[local]:5432) [postgres] > \z
Access privileges
Schema | Name | Type | Access privileges | Column access
privileges
-----+-----+-----+-----+-----
app_owner | version | table | app_owner=arwdDxt/app_owner+
| app_user=ar*wd/app_owner |
```

Basis dieser Abfrage ist das View „pg_class“. Die Abfrage ist auch über eine SELECT-Anweisung möglich.

Listing 5.17 Privilegien über „pg_class“ abfragen

```
(postgres@[local]:5432) [postgres] > SELECT relname,relacl FROM pg_class
> WHERE relname = 'version';
relname | relacl
-----+-----
version | {app_owner=arwdDxt/app_owner,app_user=ar*wd/app_owner}
```

Eine weitere Quelle zum Anzeigen von Privilegien ist das „Information-Schema“. Das Information-Schema ist eine Sammlung von Tabellen, so wie sie im ANSI-SQL vorgeschrieben sind. Das View „table_privileges“ liefert die Informationen über Tabellen-Privilegien in einer etwas übersichtlicheren Form.

Listing 5.18 Objekt-Privilegien über das Information-Schema abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM information_schema.table_
privileges
WHERE table_name = 'version';
grantor | grantee | table_catalog | table_schema | table_name |
privilege_type | is_grantable | with_hierarchy
-----+-----+-----+-----+-----+-----+-----
app_owner | app_owner | postgres | app_owner | version | INSERT
| YES | NO
app_owner | app_owner | postgres | app_owner | version | SELECT
| YES | NO
app_owner | app_owner | postgres | app_owner | version | UPDATE
| YES | NO
app_owner | app_owner | postgres | app_owner | version | DELETE
| YES | NO
app_owner | app_owner | postgres | app_owner | version | TRUNCATE
| YES | NO
app_owner | app_owner | postgres | app_owner | version |
REFERENCES | YES | NO
```


app_owner	app_owner	postgres	app_owner	version	TRIGGER
YES	NO				
app_owner	app_user	postgres	app_owner	version	INSERT
NO	NO				
app_owner	app_user	postgres	app_owner	version	SELECT
YES	NO				
app_owner	app_user	postgres	app_owner	version	UPDATE
NO	NO				
app_owner	app_user	postgres	app_owner	version	DELETE
NO	NO				

Anzeigen und Bearbeiten von Privilegien ist natürlich auch mit dem graphischen Admin-Tool „pgAdmin III“ möglich. Wählen Sie aus dem Kontext-Menü der Tabelle (rechte Maustaste) den Punkt „Eigenschaften“ aus. Im Register „Privilegien“ können die Rechte der Tabelle angezeigt und verändert werden (siehe Bild 5.1).

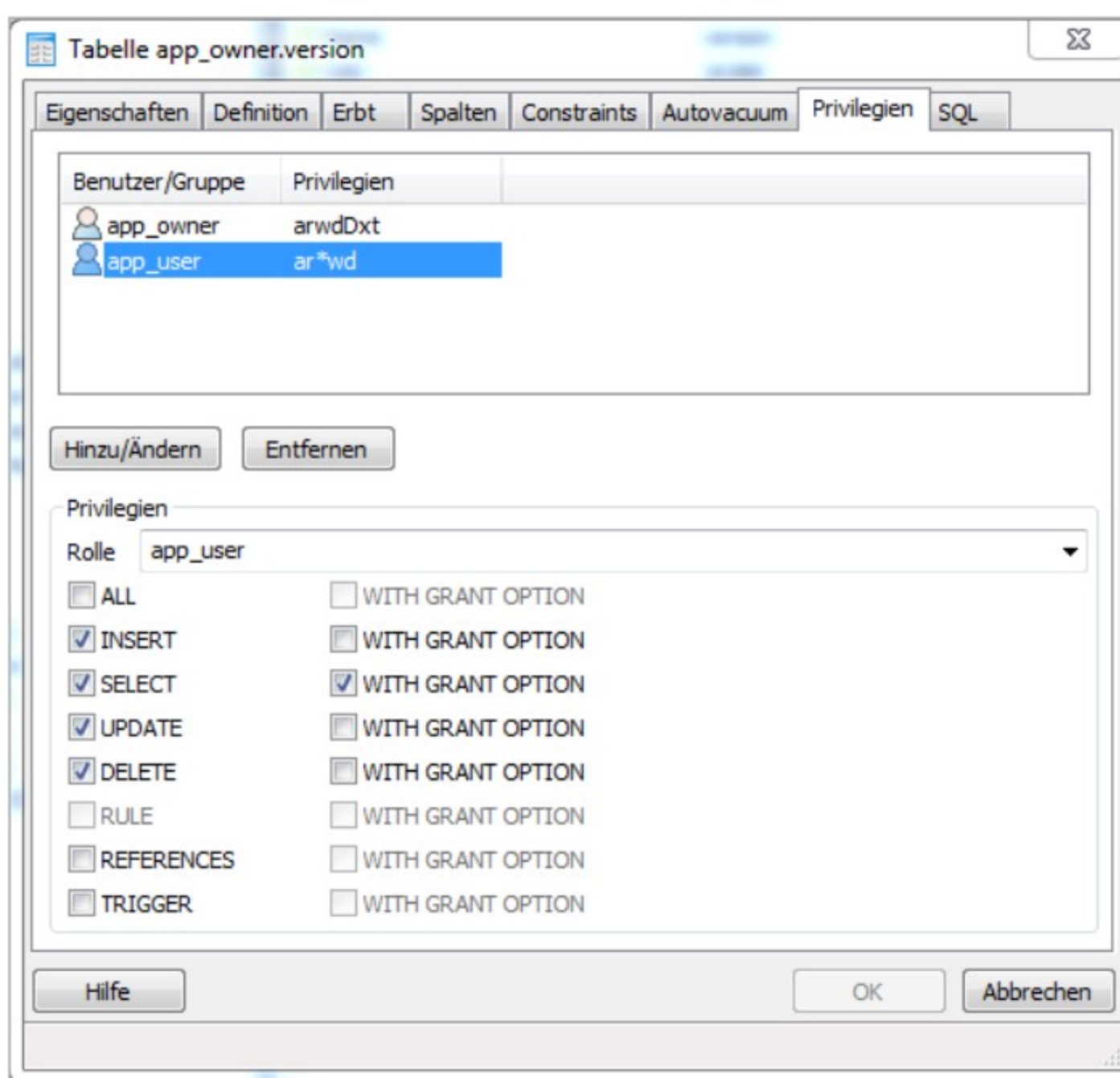


Bild 5.1 Tabellen-Privilegien mit pgAdmin III bearbeiten

Privilegien können mit dem REVOKE-Befehl oder alternativ mit „pgAdmin III“ entzogen werden.

Listing 5.19 Ein Objekt-Privileg entziehen

```
(postgres@[local]:5432) [postgres] > REVOKE SELECT ON TABLE version FROM app_
user;
REVOKE
```

5.1.4 Optionen für die Verschlüsselung

Für ein hohes Maß an Datensicherheit kann eine verschlüsselte Übertragung der Daten zwischen Client und Datenbankserver oder auch bei Verwendung eines Datenbank-Links zwischen Datenbankservern gefordert werden. Dies ist insbesondere wichtig, wenn öffentliche oder gemeinsam benutzte Netzwerkverbindungen benutzt werden. PostgreSQL bietet die folgenden Optionen, die mit wenig Aufwand zu realisieren sind:

1. SSL-Verschlüsselung unter Verwendung von OpenSSL
2. Aufbau eines SSH-Tunnels

5.1.4.1 Sichere Verbindungen mit SSL

PostgreSQL bietet integrierte Unterstützung für SSL-Verbindungen und somit einen erhöhten Sicherheitsstandard in Form einer verschlüsselten Übertragung von Daten zwischen Client und Datenbankserver. Konsequenterweise setzt PostgreSQL dabei auf das Open-Source-Produkt „OpenSSL“. Voraussetzung ist, dass OpenSSL sowohl auf der Server- als auch auf der Client-Seite installiert ist. Außerdem muss die OpenSSL-Option in der PostgreSQL-Distribution enthalten sein. Wenn Sie die Installation von PostgreSQL aus dem Quellcode vornehmen, können Sie die Option leicht einbinden. Die folgende Anleitung beschreibt alle notwendigen Schritte:

1. Laden Sie den Quellcode der aktuellen Version von OpenSSL als tar-Datei von der Webseite <http://www.openssl.org> herunter. Im folgenden Beispiel wurde die Version „OpenSSL 1.0.0g“ verwendet. Installieren Sie die Software unter dem Benutzer „root“ nach Installationsanleitung.

```
# ./configure prefix=/usr/local --openssldir=/usr/local/openssl
# make
# make test
# make install
```

2. Wiederholen Sie die Installation der PostgreSQL-Software, wenn diese nicht mit der Option „openssl“ konfiguriert wurde. Sie können die bestehende Installation überschreiben.

```
# ./configure --with-openssl
# gmake
# gmake install
```

3. Für den Betrieb von SSL ist ein Zertifikat erforderlich. Die folgenden Schritte beschreiben, wie man ein Zertifikat vom Typ „self-signed“ mit OpenSSL erstellt. Führen Sie die Schritte als Benutzer „root“ durch.

- a) Im ersten Schritt muss ein „Private Key“ erstellt werden. Dieser wird in der Datei „server.key“ gespeichert.

```
OpenSSL> genrsa -des3 -out server.key 1024
```

- b) Mit dem „Private Key“ können Sie einen Certificate Signing Request (CSR) erstellen. Diesen müssen Sie normalerweise an eine Firma schicken, die berechtigt ist, Zertifikate auszustellen. Für Testsysteme gibt es die Option, ein Zertifikat vom Typ „self-signed“ zu erstellen.


```
[root]# openssl req -new -key server.key -out server.csr
Enter pass phrase for server.key:
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]:GEState or Province Name (full name)
[Some-State]:Hessen
Locality Name (eg, city) []:Darmstadt
Organization Name (eg, company) [Internet Widgits Pty Ltd]:hanser
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:tt
Email Address []:mail
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:challenge
An optional company name []:optional
```

c) Jetzt wird das Zertifikat generiert:

```
[root]# cp server.key server.key.org
[root@tt data]# openssl rsa -in server.key.org -out server.key
Enter pass phrase for server.key.org:
writing RSA key
[root@tt data]# openssl rsa -in server.key.org -out server.key
Enter pass phrase for server.key.org:
writing RSA key
[root@tt data]# openssl x509 -req -days 365 -in server.csr -signkey server.
key -out server.crt
Signature ok
subject=/C=GE/ST=Hessen/L=Darmstadt/O=hanser/OU=IT/CN=tt/emailAddress=mail
Getting Private key
```

4. Im vorhergehenden Schritt wurden zwei Dateien erzeugt. Die Datei „server.crt“ enthält das Server-Zertifikat und die Datei „server.key“ den Private Key. Kopieren Sie beide Dateien in das Verzeichnis \$PGDATA, und ändern Sie den Eigentümer sowie die Zugriffsrechte.

```
# cp server.crt /usr/local/pgsql/data
# cp server.key /usr/local/pgsql/data
# cd /usr/local/pgsql/data
# chown postgres:postgres server.crt
# chown postgres:postgres server.key
# chmod 600 server.crt
# chmod 600 server.key
```

5. Aktivieren Sie die SSL-Option im PostgreSQL-Server. Damit akzeptiert der Server SSL-Verbindungen. Setzen Sie in der Datei „postgresql.conf“ den Parameter „ssl=on“. Führen Sie einen Neustart des Servers durch.

Der PostgreSQL-Server ist nun bereit, SSL-Verbindungen zu akzeptieren. Auf dem Client müssen Sie eine Datei „root.crt“ als Liste vertrauenswürdiger Zertifikate hinterlegen, und zwar im Verzeichnis „~/.postgresql“ (UNIX) oder „%APPDATA%\postgresql“ (Windows). An dieser Stelle testen wir die SSL-Verbindung direkt auf dem Datenbankserver. Nehmen Sie dazu den folgenden Eintrag in der Datei „pg_hba.conf“ vor, und ersetzen Sie die IP-Adresse durch die Ihres Servers.

hostssl	all	all	192.168.178.100/32	trust
---------	-----	-----	--------------------	-------

Wenn Sie jetzt mit dem Programm „psql“ eine Verbindung über TCP/IP zum Datenbankserver herstellen, erfolgt die Verbindung verschlüsselt. Dies wird im Begrüßungstext (siehe Listing 5.20) angezeigt.

Listing 5.20 Eine SSL-verschlüsselte Verbindung herstellen

```
$ psql -h 192.168.178.100
psql (9.1.1)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
(postgres@192.168.178.100:5432) [postgres] >
```

Um festzustellen, ob eine SSL-Verbindung aufgebaut wurde, gibt es zusätzlich den Modul „sslinfo“, der Bestandteil des PostgreSQL-Contribution-Paketes ist. Die Programme müssen, wie in den folgenden Schritten beschrieben, nachträglich installiert werden.

- Erstellen Sie die Bibliothek „sslinfo.so“ im Verzeichnis „ postgresql-9.1.1/contrib/sslinfo“.

```
[root]# make
```

- Kopieren Sie die Datei „sslinfo.so“ in das Verzeichnis „/usr/local/pgsql/lib“.

- Erstellen Sie die Funktionen, wie zum Beispiel die Funktion „ssl_is_used“:

```
(postgres@[local]:5432) [postgres] > CREATE FUNCTION ssl_is_used() RETURNS
boolean
> LANGUAGE c STRICT
> AS '$libdir/sslinfo', 'ssl_is_used';
CREATE FUNCTION
```

- Überprüfen Sie, ob eine SSL-Verbindung aufgebaut wurde.

```
(postgres@192.168.178.100:5432) [postgres] > SELECT ssl_is_used();
 ssl_is_used
-----
t
```

Der Client baut eine SSL-Verbindung automatisch auf, wenn diese vom Server angeboten wird.

Neben der Verschlüsselung kann SSL auch zu Authentifizierung der Clients verwendet werden. Der PostgreSQL-Server überprüft, ob der Client ein entsprechendes Zertifikat vorweisen kann. Auf dem Client müssen sich folgende Dateien im Verzeichnis „~/.postgresql“ (bzw. „%APPDATA%\postgresql“ für Windows) befinden:

- postgresql.crt: Das Client-Zertifikat
- postgresql.key: Privater Schlüssel

5.1.4.2 Verbindungen mit SSH-Tunneln

SSH-Tunnel stellen eine Alternative für eine verschlüsselte Übertragung dar, wenn OpenSSL nicht verwendet werden kann. Voraussetzung ist, dass auf dem Datenbankserver ein SSH-Server läuft. Ein Tunnel kann mit dem folgenden Kommando auf dem Client hergestellt werden:

Listing 5.21 Einen SSH-Tunnel einrichten

```
ssh -L 65432:localhost:5432 postgres@192.168.178.100
```

Dabei ist „65432“ eine frei wählbare, nicht benutzte Portnummer, die das Ende des Tunnels repräsentiert. Nach dem Servernamen muss die Portnummer, auf die der PostgreSQL-Server hört, eingetragen werden. In diesem Fall ist das der Standard-Port „5432“ – das andere Ende des Tunnels.

**Hinweis**

Die Portnummern 49152 bis 65535 sind für eine benutzerdefinierte Verwendung reserviert.

Die Verbindung zur Datenbank wird dann mit dem lokalen Host, also dem Eingang des Tunnels, hergestellt:

```
psql -h localhost -p 65432 postgres
```

Das Ende des Tunnels verbindet sich dann mit dem Datenbankserver. In den Verbindungsinformationen sieht man nun eine Verbindung, die vom Datenbankserver und nicht vom Client kommt.

5.1.5 Out-of-the-box-Sicherheit

Wichtig für den Einsatz von Datenbank-Systemen im produktiven Umfeld ist, dass der Datenbank-Server nach einer Standard-Installation bereits ein hohes Maß an Sicherheit mitbringt.

Verantwortlich für die Zugriffskontrolle ist die Datei „pg_hba.conf“. Den ausgelieferten Standard sehen Sie in Listing 5.22. Danach werden nur lokale Zugriffe oder Zugriffe über den lokalen Host erlaubt, d.h. die User „postgres“ und „root“ haben Zugriff, ohne Passwortkontrolle. Alle anderen Zugriffe, insbesondere über entfernte Clients, sind nicht zugelassen und müssen durch den Administrator explizit freigeschaltet werden. Damit ist nach der Standard-Installation eine optimale Zugriffskontrolle gewährleistet.

Listing 5.22 Standard der Datei „pg_hba.conf“

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# "local" is for Unix domain socket connections only					
local	all		all		trust
# IPv4 local connections:					
host	all		all	127.0.0.1/32	trust

Schauen wir nun in den Server. Der Eintrag „listen_address“ in der Datei „postgresql.conf“ ist verantwortlich dafür, auf welchem Netzwerk-Interface der PostgreSQL-Server hört. Der Standard ist „localhost“. Damit sind generell nur Verbindungen vom lokalen Host zugelassen. Der Administrator wird hier die IP-Adresse oder den Hostnamen des Netzwerk-Inter-

faces eintragen, über das Verbindungen von außen akzeptiert werden sollen. Auch von dieser Stelle drohen keine Gefahren.



Praxistipp

Der Standard-Port ist „5432“. Dieser ist natürlich bei Angreifern bekannt und sollte deshalb nach der Installation auf einen anderen Wert geändert werden. Je weniger der Angreifer über die Konfiguration des Servers weiß, desto länger braucht er, um in das System einzudringen. Weitere Informationen zu diesem Thema finden Sie in Abschnitt 5.2 „Hacker-Attacken abwehren“.

Standardmäßig führt PostgreSQL keine Kontrolle zur Passwort-Komplexität durch. Schwache Passwörter bieten eine optimale Angriffsfläche und sollten deshalb verhindert werden. Installieren und konfigurieren Sie deshalb die Passwort-Kontrolle mit dem Modul „passwordcheck“, bevor die ersten Benutzer angelegt werden. Eine Anleitung zur Installation des Moduls finden Sie in Abschnitt 5.1.1 („Datenbank-Rollen und -Privilegien“). Eine einfache Prüfung, ob alle Benutzer ein sicheres Passwort besitzen, liefert die SQL-Abfrage in Listing 5.23.

Listing 5.23 Überprüfung der Benutzer auf ein sicheres Passwort

```
(postgres@[local]:5432) [postgres] > SELECT rolname,rolpassword
> FROM pg_authid
> WHERE rolpassword NOT LIKE 'md5%';
 rolname | rolpassword
-----+-----
 test   | test
```

Eine Sicherheits-Schwachstelle des PostgreSQL-Servers ist, dass keine maximale Anzahl von fehlerhaften Login-Versuchen, nach denen der Account gesperrt wird, vorgegeben werden kann. Eine Brute-Force-Attacke kann also direkt auf dem Datenbank-Server gestartet werden. Es sind demgemäß zusätzliche Maßnahmen erforderlich, um solche Attacken zu erkennen und abzuwehren.

Mit dem initialen Erstellen des Servers wird genau eine Rolle, standardmäßig mit dem Namen „postgres“, angelegt. Diese Rolle erhält das Privileg „Superuser“ und damit weitreichende Rechte am Datenbank-Server und den Datenbanken. Da, wie in anderen Datenbanksystemen, keine weiteren Rollen automatisch angelegt werden, ist damit ein hohes Maß an Out-of-the-box-Sicherheit gegeben. Doch gilt auch hier die Empfehlung, den Benutzernamen umzubenennen, um es Angreifern zu erschweren, in das System einzudringen. Das gilt insbesondere in Zusammenhang mit den soeben erwähnten Brute-Force-Attacken.

Als Fazit lässt sich feststellen, dass PostgreSQL über ein hohes Maß an Out-of-the-box-Sicherheit verfügt. Auch wenn für ein Höchstmaß an Sicherheit weitere Maßnahmen erforderlich sind, ist der Out-of-the-box-Standard für die meisten Datenbanken ausreichend.

Im folgenden Abschnitt beschäftigen wir uns mit der Abwehr möglicher Hacker-Attacken.

■ 5.2 Hacker-Attacken abwehren

Für Datenbanken mit hoch-sensiblen Inhalt sind Out-of-the-box-Sicherheit oder auch die Vorgaben von „HIPPA“, „BSI“ oder „SOX“ möglicherweise nicht ausreichend. Für ein Höchstmaß an realer Datenbank-Sicherheit ist es notwendig, die Vorgehensweisen von Hackern zu kennen und entsprechende Gegenmaßnahmen einzuleiten. In diesem Abschnitt lernen Sie einige dieser Methoden kennen, und wir zeigen auf, wie diese in PostgreSQL erkannt und verhindert werden können. Das Ziel ist dabei stets, die Konfiguration so auszubauen, dass ein Angriff unmöglich ist – was allerdings nicht in jedem Fall gelingt. Im gegebenen Fall sollte man zeitnah erkennen, dass ein Angriff stattgefunden hat, um weitere Maßnahmen einzuleiten. Eine der am häufigsten verwendeten Methoden ist die Brute-Force-Attacke.

5.2.1 Brute-Force-Attacken

Insbesondere die Erhöhung der Netzwerk- und Prozessor-Geschwindigkeiten in den vergangenen Jahren haben Brute-Force-Attacken attraktiv gemacht. Es werden entweder alle möglichen Kombinationen aus zugelassenen Buchstaben, Zahlen und Sonderzeichen durchprobiert, oder es wird eine Liste der am häufigsten benutzten Passwörter als Basis herangezogen. Dabei werden zwei verschiedene Methoden angewandt:

1. Direkte Anmeldeversuche am Datenbankserver
2. Diebstahl des verschlüsselten Passworts und Offline-Attacke ohne Verbindung zum Datenbankserver

Direkte Anmeldeversuche am Datenbankserver haben für den Angreifer den Nachteil, dass solche Attacken häufig nicht unbemerkt bleiben und über so genannte Host Intrusion Detection-Systeme (HIDS) erkannt werden. Ein weiterer Nachteil ist die relativ große Latenzzeit des Anmeldeprozesses über das Netzwerk, durch die die Anzahl von Versuchen pro Sekunde deutlich reduziert wird. Das begrenzt die Anzahl der zu probierenden Passwörter deutlich, was insbesondere ein Problem für das Knacken längerer Passwörter darstellt. Aus diesem Grund werden direkte Attacken häufig mit Passwortlisten gefahren und nicht alle möglichen Kombinationen durchgetestet. Damit verringert sich natürlich die Erfolgswahrscheinlichkeit für den Angreifer.



Praxistipp

Weil der PostgreSQL-Server über kein integriertes System verfügt, um die Anzahl nicht-erfolgreicher Anmeldeversuche zu begrenzen, müssen zusätzliche Maßnahmen ergriffen werden. Eine einfache Lösung bietet das folgende Verfahren:

Mit Hilfe einer einfachen Überwachung können direkt Brute-Force-Attacken vermieden werden. Durch Setzen des Parameters „log_connections“ auf „on“ werden erfolgreiche und erfolglose Anmeldeversuche in die Server-Logdatei geschrieben. Der Eintrag hat die folgende Form:


```
[unknown] 2012-02-05 16:56:27 CET [unknown] LOG:  connection received:
host=192.168.178.22 port=50147
postgres 2012-02-05 16:56:27 CET postgres FATAL:  password authentication
failed for user "postgres"
```

Diese Einträge können herausgefiltert und in eine Tabelle geschrieben werden. Lädt man diese Einträge zeitnah in eine Tabelle, ist es ein Leichtes, eine Brute-Force-Attacke zu erkennen. Als Folgeaktion wird der Account im Datenbankserver gesperrt und die Attacke somit verhindert.

Listing 5.24 Einen Account im Datenbankserver sperren

```
(postgres@[local]:5432) [postgres] > ALTER ROLE app_owner NOLOGIN;
ALTER ROLE
```

Voraussetzung für die Offline-Attacke ist der vorhergegangene Diebstahl des verschlüsselten Passworts. Seit der Version 9 ist das Passwort noch einmal besonders geschützt und kann nur von einem Superuser direkt ausgelesen werden. Es befindet sich in der Spalte „rolpassword“ des View „pg_authid“.

Listing 5.25 Das verschlüsselte Passwort auslesen

```
(postgres@[local]:5432) [postgres] > SELECT rolname,rolpassword FROM
pg_authid;
 rolname |          rolpassword
-----+-----
 postgres | md592351ded876996da61d3f79db6605c02
 app_owner | md5207dd91017a1f49ed31abd400d2d55dc
```

Eine weitere Möglichkeit, das Passwort zu ermitteln, ist die Verwendung eines Netzwerk-Sniffers, um sich damit in den Authentifizierungsprozess einzuklinken.

Hat man das verschlüsselte Passwort einmal ermittelt, ist es nur noch eine Frage der Rechenleistung, das unverschlüsselte Passwort auf einem externen Rechner zu ermitteln. In diesen Prozess ist der Datenbank-Server nicht mehr eingebunden. Somit ist es nicht mehr möglich, durch zusätzliche Verschärfung der Sicherheit das Knacken des Passworts zu verhindern.



Praxistipp

Begrenzen Sie den direkten Zugriff auf die View „pg_authid“ sowie den Einsatz des Privilegs „SUPERUSER“, um ein Auslesen des verschlüsselten Passworts zu vermeiden.

Zwar kann das verschlüsselte Passwort nicht direkt entschlüsselt werden, das Verfahren zur Verschlüsselung ist jedoch bekannt. Deshalb nimmt man einfach alle Kombinationen aus den zulässigen Buchstaben, Zahlen und Sonderzeichen her, verschlüsselt das Passwort und vergleicht es mit dem gestohlenen, verschlüsselten Passwort. Stimmt dies überein, hat man das unverschlüsselte Passwort ermittelt.

Die Methoden sind sehr schnell, und somit ist es nur eine Frage der CPU-Geschwindigkeit und der Zeit, bis man das Passwort im Klartext ermittelt. Im folgenden Beispiel verwenden wir das Utility „Hashcat“ – offiziell als Passwort-Wiederherstellungswerkzeug bezeichnet –, um ein verloren gegangenes Passwort wiederzuerlangen. Zunächst ändern wir das Passwort für den Benutzer „app_owner“ in „hanser“.

```
(postgres@[local]:5432) [postgres] > ALTER ROLE app_owner PASSWORD 'hanser';
ALTER ROLE
```

Das verschlüsselte Passwort hat den Eintrag "md5207dd91017a1f49ed31abd400d2d55dc", also eine MD5-Verschlüsselung. Wir tragen den Passwort-Hash mit dem Salt in die Datei hash.txt ein und starten das Programm:

```
echo 207dd91017a1f49ed31abd400d2d55dc:postgres > hash.txt
hashcat-cli64.exe --hash-mode 1 --attack-mode 3 hash.txt
Initializing hashcat v0.38 by atom with 8 threads and 32mb segment size
NOTE: press enter for status-screen
Added hashes from file hash.txt: 1 (1 salts)
Activating quick-digest mode for single-hash with salt
Length....: 5
Index.....: 0/1 (segment), 60466176 (words), 0 (bytes)
Recovered.: 1/1 hashes, 1/1 salts
Speed/sec.: - plains, 13.69M words
Progress...: 60466176/60466176 (100.00%)
207dd91017a1f49ed31abd400d2d55dc:postgres:hanser
All hashes have been recovered
```

Die ganze Aktion hat nur wenige Sekunden gedauert, was sicher der Tatsache geschuldet ist, dass das Passwort nur 5 Stellen aufweist und die Anzahl der möglichen Kombinationen somit vergleichsweise gering ist. Mit der entsprechenden CPU-Kapazität und einem längeren Zeitraum ist es jedoch auch möglich, Passwörter mit mehr Stellen zu knacken. Allerdings vergrößert sich der Aufwand mit jeder zunehmenden Stelle nicht-linear.



Praxistipp

Verwenden Sie Passwörter von mindestens 12 Stellen in einer Kombination von Groß- und Kleinbuchstaben, Zahlen und Sonderzeichen, um das Wiederherstellen aus dem Passwort-Hash zu erschweren.

5.2.2 Network Spoofing

Eine weitere Angriffsmethode ist das „Network Spoofing“, so wie in Bild 5.2 dargestellt. Dabei schaltet der Angreifer einen gefälschten PostgreSQL-Server zwischen den Client und den PostgreSQL-Server und kontrolliert somit den Authentifizierungsprozess. Ein solches Setup erreicht man zum Beispiel durch „Verbiegen“ der Namensauflösung (DNS Server). Der gefälschte PostgreSQL-Server hört dann auf demselben Port und kann die Kommunikation steuern.

Als erste Maßnahme im Authentifizierungsprozess sendet der Client einen Connection Request (1). Der gefälschte Server verlangt das unverschlüsselte Passwort (2), das er im

nächsten Schritt (3) vom Client erhält. Durch eine geschickte Steuerung merkt der Client nicht, dass gerade eine Attacke stattfindet.

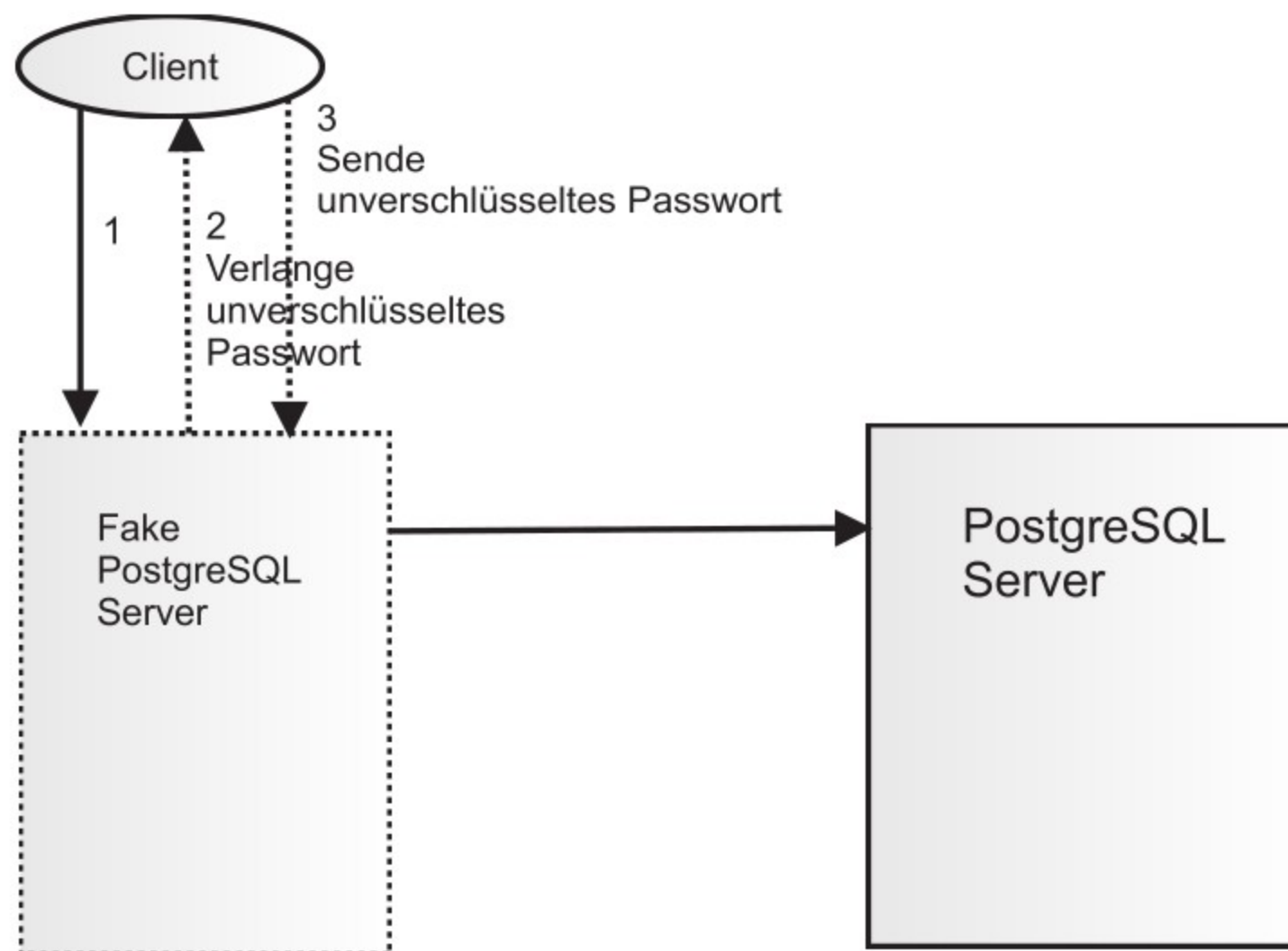


Bild 5.2 Hacker-Attacke Network Spoofing

Ein solcher Angriff lässt sich verhindern, wenn Server und Client mit Authentifizierungszertifikaten arbeiten. Weil der gefälschte PostgreSQL-Server nicht über das Zertifikat (server.crt) verfügt, kann er den Authentifizierungsprozess nicht mehr steuern, und es kommt zum Abbruch.

5.2.3 SQL Injection

SQL Injection ist eine Gefährdung in Zusammenhang mit gespeicherten Funktionen im PostgreSQL-Server. Die Gefährdung tritt häufig in Zusammenhang mit dynamischem SQL auf. Betrachten Sie den SQL-Code in Listing 5.26. Die Funktion macht einen Eintrag in der Tabelle „log_table“. Dabei wird dynamisches SQL verwendet. Die Syntax verlangt, dass die dynamische SQL-Anweisung von zwei Hochkommas eingeschlossen ist.

Listing 5.26 Eine SQL-Funktion mit Gefährdung für SQL Injection

```

(postgres@[local]:5432) [postgres] > CREATE TABLE log_table(message
VARCHAR);
CREATE TABLE
(postgres@[local]:5432) [postgres] > CREATE OR REPLACE FUNCTION log_
func(VARCHAR)
> RETURNS VARCHAR SECURITY DEFINER AS '
> BEGIN
> EXECUTE 'INSERT INTO log_table VALUES('' || $1 || '');';
> RETURN 'Funktion komplett.';
> END ' LANGUAGE 'plpgsql';
CREATE FUNCTION
  
```


Um einen Eintrag in die Log-Tabelle vorzunehmen, wird die SELECT-Anweisung mit der Funktion verwendet:

```
(postgres@[local]:5432) [postgres] > SELECT log_func('Log-Eintrag');
      log_func
-----
Funktion komplett.
(1 row)
(postgres@[local]:5432) [postgres] > SELECT * FROM log_table;
      message
-----
Log-Eintrag
(1 row)
```

Beachten Sie weiterhin, dass die Funktion durch den Superuser „postgres“ eingerichtet und mit Definer-Rechten versehen wurde. Die Ausführungsrechte sind auf den Benutzer „Public“ gelegt, d.h. jeder darf die Funktion ausführen. Ein nicht-privilegierter Benutzer kann sich nun weitreichende Rechte verschaffen. Durch einen Trick wird der Befehl „ALTER ROLE“ untergejubelt und mit den Rechten des Benutzers „postgres“ ausgeführt.

Listing 5.27 Eine Funktion für SQL Injection benutzen

```
(app_owner@[local]:5432) [postgres] > SELECT log_func('TEST'); ALTER ROLE
app_owner SUPERUSER;--');
      log_func
-----
Funktion komplett.
(app_owner@[local]:5432) [postgres] > SELECT rolname,rolsuper FROM pg_roles;
  rolname | rolsuper
-----+-----
app_user  | f
postgres  | t
nagios    | f
app_owner | t
```

Die Möglichkeit der SQL-Injection wurde bei den mitgelieferten Funktionen weitgehend eingeschränkt. Mit den Applikationen können neue Gefährdungen hinzukommen. Deshalb sollte der PL/pgSQL Code nach solchen Gefährdungen durchsucht werden. Eine saubere Rechteverwaltung verringert die Gefährdung. Wenn der Definer einer Funktion das Privileg „ALTER ROLE“ nicht besitzt, kann die Funktion dafür nicht missbraucht werden.

■ 5.3 Überwachung

Zur Überwachung des Datenbank-Servers gehören sowohl das Auditing unter dem Sicherheitsaspekt als auch das allgemeine Monitoring. Auditing wird implementiert, um auch im Nachhinein nachvollziehen zu können, welche Aktionen oder Zugriffe von Benutzern und Clients durchgeführt wurden. Das Monitoring dient der pro-aktiven Überwachung der Datenbanken, um Incidents zu vermeiden und die Verfügbarkeit zu gewährleisten.

5.3.1 Auditing

PostgreSQL besitzt kein spezielles Produkt für das Datenbank-Auditing. Allerdings lässt sich dies mit vorhandenen Features und relativ geringem Aufwand einrichten.

Eine Überwachung von DDL-Anweisungen kann mit Hilfe des Server-Parameters „log_statement“ aktiviert werden. Der Standardwert des Parameters ist „none“. Setzen Sie den Parameter in der Datei „postgresql.conf“ auf den Wert „ddl“, um DDL-Anweisungen zu überwachen.

Listing 5.28 Auditing für DDL-Anweisungen aktivieren

```
log_statement = 'ddl'
```

Die DDL-Anweisungen werden in die Logdatei geschrieben. Ein Standardeintrag sieht wie folgt aus:

```
LOG:  statement: GRANT SELECT ON TABLE version TO app_user;
```

Der Eintrag belegt nicht, zu welchem Zeitpunkt und von welchem Benutzer der Befehl abgesetzt wurde und auf welche Datenbank Bezug genommen wird. Diese Informationen sind für das Auditing allerdings essenziell.



Praxistipp

Setzen Sie den Parameter „log_line_prefix“ mindestens auf den Wert '%u %t %d ', um ein adäquates Auditing von DDL-Anweisungen zu gewährleisten. Damit werden zusätzlich Einträge von Benutzername (%u), Zeitstempel (%t) und Datenbank (%d) vorgenommen. Ein Eintrag in die Log-Datei erfolgt dann in dieser Form:

```
postgres 2012-01-29 20:12:15 CET postgres LOG: statement: ALTER TABLE
version ADD COLUMN version_date DATE;
```

Eine Auswertung der Einträge in die Log-Dateien ist recht unhandlich. Deshalb sollten die Einträge herausgefiltert und in eine Tabelle geladen werden. Ein Herausfiltern kann mit dem grep-Utility, wie in Listing 5.29 dargestellt, erfolgen.

Listing 5.29 Audit-Einträge aus den Logdateien herausfiltern

```
$ pwd
/usr/local/pgsql/data/pg_log
$ grep "statement:" *.log
postgresql-2012-01-29_200443.log:LOG:  statement: GRANT SELECT ON TABLE
version TO app_user;
postgresql-2012-01-29_200443.log:LOG:  statement: CREATE TABLE version2
(text VARCHAR(200));
postgresql-2012-01-29_200443.log:LOG:  statement: CREATE TABLE version3
(text VARCHAR(200));
postgresql-2012-01-29_201126.log:postgres 2012-01-29 20:12:15 CET postgres
LOG:  statement: ALTER TABLE version ADD COLUMN version_date DATE;
```


In Abschnitt 3.7 „Nützliche Skripte und Hinweise“ finden Sie eine Beschreibung, wie der gefilterte Inhalt in eine Tabelle geladen werden kann. In Form einer Tabelle lassen sich die Auditdaten optimal auswerten.

Ein Auditing von Datenveränderungen, also von DML-Anweisungen, kann mit Hilfe von Tabellen-Triggern erfolgen. Im Beispiel in Listing 5.30 werden neben den Veränderungen der Tabelle Benutzername, Zeitstempel und der verwendete Befehl aufgezeichnet. Dabei wird zunächst eine Tabelle zur Aufnahme der Audit-Informationen angelegt. Die Trigger-Funktion wird in PL/pgSQL geschrieben und auf der Originaltabelle implementiert.

Listing 5.30 DML-Auditing an einer Tabelle einrichten

```
(postgres@[local]:5432) [postgres] > CREATE TABLE audit_emp(
> username TEXT,
> timest TIMESTAMP,
> operation CHAR(1),
> empname TEXT,
> salary INTEGER);
CREATE TABLE
(postgres@[local]:5432) [postgres] > CREATE OR REPLACE FUNCTION func_audit_
emp() RETURNS TRIGGER AS $audit_emp$
> BEGIN
> IF (TG_OP = 'DELETE') THEN
> INSERT INTO audit_emp SELECT user, now(), 'D', OLD.*;
> RETURN OLD;
> ELSIF (TG_OP = 'UPDATE') THEN
> INSERT INTO audit_emp SELECT user, now(), 'U', NEW.*;
> RETURN NEW;
> ELSIF (TG_OP = 'INSERT') THEN
> INSERT INTO audit_emp SELECT user, now(), 'I', NEW.*;
> RETURN NEW;
> END IF;
> RETURN NULL;
> END;
> $audit_emp$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [postgres] > CREATE TRIGGER audit_emp
> AFTER INSERT OR UPDATE OR DELETE ON emp
> FOR EACH ROW EXECUTE PROCEDURE func_audit_emp();
CREATE TRIGGER
```

Das Einfügen eines Satzes in die Tabelle „emp“ erzeugt einen Audit-Satz in der Tabelle „audit_emp“.

Listing 5.31 Einen Audit-Eintrag erzeugen

```
(postgres@[local]:5432) [postgres] > INSERT INTO emp VALUES('Max
Meier',3240);
INSERT 0 1
(postgres@[local]:5432) [postgres] > SELECT * FROM audit_emp;
username | timest | operation | empname | salary
-----+-----+-----+-----+-----
postgres | 2012-02-05 13:35:11.436639 | I | Max Meier | 3240
```




Praxistipp

Beachten Sie bei der Auswahl der Informationen, die in das Auditing eingehen sollen, dass es durch den Einsatz der Trigger zu einem Einfluss auf die Online-Performance kommt. Auditing sollte deshalb nur dort eingeschaltet werden, wo es wirklich notwendig ist, und auf Performance-kritischen Tabellen möglichst vermieden werden.

5.3.2 Monitoring

Das Monitoring ist ein unverzichtbarer Baustein, um die Verfügbarkeit des Datenbank-Servers zu gewährleisten. Hauptziel des Monitorings ist es, potenzielle Probleme frühzeitig zu erkennen und deren Ursachen zu beseitigen, möglichst bevor die Probleme überhaupt auftreten. Das schließt sowohl einfache Ereignisse, wie zum Beispiel das Volllaufen eines Tablespace oder einer Disk, als auch komplexere Ereignisse im Bereich Performance-Monitoring ein. Performance-Probleme können zu einer verringerten Verfügbarkeit oder sogar zum Ausfall des Systems führen.

5.3.2.1 Monitoring mit Mitteln des PostgreSQL-Server

Der PostgreSQL-Server hält eine Reihe von Funktionen, Tabellen, Views und Statistiken bereit, die für das Monitoring eingesetzt werden können. Wenn Ihnen kein Monitoring-Framework zur Verfügung steht, können Sie mit Hilfe dieser Objekte und mit einfachen, selbstgebaute Skripten ein effektives Basis-Monitoring mit relativ wenig Aufwand aufsetzen.

Der PostgreSQL-Server beinhaltet eine Reihe von Statistik-Tabellen und -Views für das Monitoring. Die wohl bekannteste ist „pg_stat_activity“. Sie gibt Auskunft über die aktuell laufenden Verbindungen (Sessions). In Listing 5.32 finden Sie ein Beispiel. Neben der eigenen psql-Session ist ein Client mit pgAdmin III angemeldet (RECORD 2). Die Spalten „datid“ und „datname“ geben Auskunft, mit welcher Datenbank die Sitzung verbunden ist. „procpid“ ist die Prozess-ID des zugehörigen Client-Prozesses auf dem Datenbankserver:

```
$ ps -ef|grep 29647
postgres 29647 28763  0 13:36 ?          00:00:00 postgres: postgres postgres
192.168.178.22(50461) idle
```

Als weitere Informationen werden der Benutzername, die IP-Adresse des Clients sowie das verwendete Programm angezeigt. Außerdem sehen Sie die aktuell ausgeführte SQL-Anweisung.

Listing 5.32 Die View „pg_stat_activity“ für das Monitoring abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_stat_activity;
-[ RECORD 1 ]-----+-----
datid              | 12698
datname            | postgres
procpid            | 29609
```



```

usesysid      | 10
username      | postgres
application_name | psql
client_addr   |
client_hostname |
client_port   | -1
backend_start | 2012-02-06 13:32:20.875613+01
xact_start    | 2012-02-06 13:36:28.504894+01
query_start   | 2012-02-06 13:36:28.504894+01
waiting       | f
current_query  | SELECT * FROM pg_stat_activity;
-[ RECORD 2 ]-----+-----
datid         | 12698
datname       | postgres
procpid       | 29647
usesysid      | 10
username      | postgres
application_name | pgAdmin III- Browser
client_addr   | 192.168.178.22
client_hostname |
client_port   | 50461
backend_start | 2012-02-06 13:36:07.034312+01
xact_start    |
query_start   | 2012-02-06 13:36:07.040533+01
waiting       | f
current_query  | <IDLE>

```

Die View „pg_stat_database“ liefert eine Sammlung von Statistiken der einzelnen Datenbanken. Neben der Anzahl der Transaktionen („Commit“ und „Rollback“) liefert die Statistik die Anzahl physikalisch gelesener Blöcke (blks_read) und aus dem Memory Buffer gelesener Blöcke (blks_hit). Daraus lässt sich die durchschnittliche Hit Ratio berechnen, und es kann eine Aussage getroffen werden, ob der Cache vergrößert werden sollte.

Listing 5.33 Statistiken einer Datenbank anzeigen

```

(postgres@[local]:5432) [postgres] > SELECT * FROM pg_stat_database
WHERE datname='postgres';
-[ RECORD 1 ]-----+-----
datid         | 12698
datname       | postgres
numbackends   | 2
xact_commit   | 1794
xact_rollback | 22
blks_read     | 2702
blks_hit      | 80434
tup_returned  | 494634
tup_fetched   | 33458
tup_inserted  | 198
tup_updated   | 13
tup_deleted   | 38
conflicts     | 0
stats_reset   | 2012-01-29 13:52:09.540599+01

```

Eine Reihe von Views liefert Statistiken über Zeilenzugriffe von Tabellen und Indexen. Diese Werte sind sehr interessant für das Performance-Monitoring. Die folgenden Views (eine Auswahl) liefern Zeilenstatistiken:

- pg_stat_user_tables
- pg_stat_user_indexes
- pg_stat_all_tables
- pg_stat_all_indexes

Listing 5.34 liefert ein Beispiel für die Zeilenstatistiken einer Tabelle. In der Spalte „seq_scan“ finden Sie die Anzahl der ausgeführten Full Table Scans. Diese sind auf größeren Tabellen kritisch für die Performance und den Ressourcenverbrauch. Die Spalte „seq_tup_read“ liefert die Anzahl der durch Full Table Scans gelesenen Sätze. Wichtig für das Monitoring sind die Spalten „last_vacuum“, „last_autovacuum“ und „last_analyze“. Mit diesen Statistiken lässt sich feststellen, in welchem Zustand die Tabellen sind. Zum Monitoring gehört auch diese Art der Überwachung, um sicherzustellen, dass die Analyze- und Vacuum-Prozesse regelmäßig angewandt werden.

Listing 5.34 Zeilenstatistiken für eine Tabelle abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_stat_user_tables
> WHERE relname = 'kb';
-[ RECORD 1 ]-----+-----
relid          | 16424
schemaname     | public
relname        | kb
seq_scan       | 2
seq_tup_read   | 10000000
idx_scan       |
idx_tup_fetch  |
n_tup_ins      | 5000000
n_tup_upd      | 0
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup     | 5000060
n_dead_tup     | 0
last_vacuum     |
last_autovacuum |
last_analyze    |
last_autoanalyze | 2012-02-06 14:56:33.323788+01
vacuum_count    | 0
autovacuum_count | 0
analyze_count   | 0
autoanalyze_count | 1
```

Sperren auf Sätze und Tabellen stellen nicht nur ein Performance-Risiko dar, sondern können zum Ausfall von Funktionalitäten der Applikation führen. Aus diesem Grund sollten sie unbedingt in das Monitoring aufgenommen werden. Informationen dazu liefert das View „pg_locks“. Interessant sind dabei nur Sperren vom Typ „relation“. Alle anderen Sperren beziehen sich auf interne Mechanismen des PostgreSQL-Servers und lassen sich von der Applikation nicht beeinflussen.

Listing 5.35 Sperren in der Datenbank abfragen

```
(postgres@[local]:5432) [postgres] > SELECT relation, pid, mode, granted
> FROM pg_locks WHERE locktype = 'relation';
 relation | pid | mode | granted
-----+-----+-----+-----
```



```

16424 | 30862 | RowExclusiveLock | t
16424 | 30847 | RowExclusiveLock | t
(postgres@[local]:5432) [postgres] > SELECT schemaname,relname
> FROM pg_stat_all_tables WHERE relid = '16424';
schemaname | relname
-----+-----
public     | kb

```

Offensichtlich blockiert im Beispiel in Listing 5.35 eine Session die andere. Die Satzsperrung liegt auf derselben Tabelle (Spalte „relation“) und wird von unterschiedlichen Sessions angefordert. Die zugehörigen Sessions lassen sich mit Hilfe des Views „pg_stat_activity“ ermitteln. Um alle notwendigen Informationen in einer Abfrage zu erhalten, vereinigen wir die drei Views in einer SQL-Abfrage. Außerdem lassen wir uns den Client und die SQL-Anweisung ausgeben.

Listing 5.36 Zusätzliche Informationen über eine Lock-Situation abfragen

```

(postgres@[local]:5432) [postgres] > SELECT l.mode, t.schemaname,
relname, a.username, a.client_addr, a.current_query
> FROM pg_locks l, pg_stat_all_tables t, pg_stat_activity a
> WHERE l.locktype='relation'
> AND l.relation = t.relid
> AND l.pid = a.procpid;
mode          | schemaname | relname | username | client_addr |
current_query
-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----
RowExclusiveLock | public    | kb      | postgres | 192.168.178.22 |
UPDATE kb SET text = 'CCCCCCCCCCCCCCCC' WHERE id = 4711;
RowExclusiveLock | public    | kb      | postgres | 192.168.178.22 |
<IDLE> in transaction

```



Hinweis

Ein Zurücksetzen der Statistiken ist mit dem SQL-Befehl „SELECT pg_stat_reset();“ möglich. Dies kann bei bestimmten Analysen hilfreich sein.

Der pgAdmin III bietet begrenzte Unterstützung für das Monitoring. Über die Menüpunkte „Werkzeuge“ und „Serverstatus“ wird das Fenster „Server-Status“ aufgerufen. Hier werden neben den Sessions die aktuellen Sperren angezeigt (siehe Bild 5.3). Es handelt sich um dieselben Informationen, die wir aus den vorhergehenden Abfragen gewonnen haben.

5.3.2.2 Monitoring mit Nagios

Nagios ist ein Open-Source-Produkt zur Überwachung von Servern und Diensten, die auf diesen Servern laufen. Es können Schwellwerte gesetzt werden, die bei Überschreitung einen Alarm auslösen. Das Produkt ist über die Webseite <http://www.nagios.org> frei verfügbar.

Es bietet die Möglichkeit der Erweiterung über Plugins. Auf der Nagios-Webseite finden Sie eine Reihe von Plugins für PostgreSQL. Die Plugins können die Anzahl von Verbindungen, den freien Platz auf der Disk, die Ausführungsdauer von SQL-Anweisungen und vieles mehr überwachen (siehe Bild 5.4).

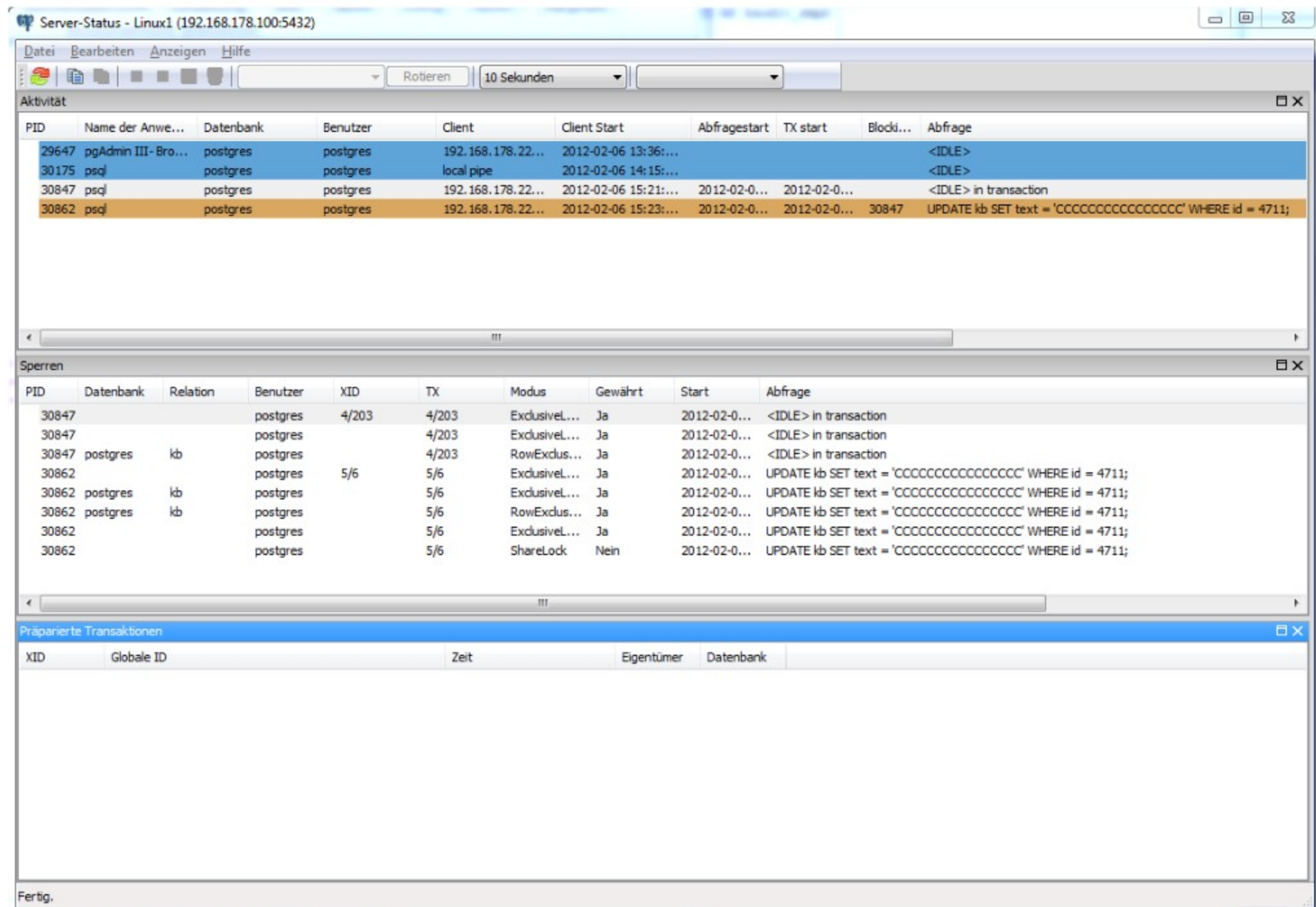


Bild 5.3 Den Server-Status in pgAdmin III anzeigen

Nagios

General

- Home
- Documentation

Current Status

- Tactical Overview
- Map
- Hosts
- Services
- Host Groups
 - Summary
 - Grid
- Service Groups
 - Summary
 - Grid
- Problems
 - Services (Unhandled)
 - Hosts (Unhandled)
 - Network Outages

Reports

- Availability
- Trends
- Alerts
 - History
 - Summary
 - Histogram
- Notifications
- Event Log

System

- Comments
- Downtime
- Process Info
- Performance Info
- Scheduling Queue
- Configuration

Service Information

Last Updated: Mon Feb 6 12:17:36 UTC 2012
 Updated every 90 seconds
 Nagios® Core™ 3.2.3 - www.nagios.org
 Logged in as nagiosadmin

[View Information For This Host](#)
[View Status Detail For This Host](#)
[View Alert History For This Service](#)
[View Trends For This Service](#)
[View Alert Histogram For This Service](#)
[View Availability Report For This Service](#)
[View Notifications For This Service](#)

Service
PostgreSQL Server
 On Host
localhost
(localhost)

Member of
No servicegroups.

127.0.0.1

Service State Information

Current Status: **OK** (for 234d 15h 30m 20s)
 Status Information: postmaster (pid 23694 23691 23690 23625 23598 23589 23483 23480 23474 23455 23448 23445 23422 23406 23393 23366 23364 23321 23309 23211 23208 23202 23197 23126 23100 23032 23027 22987 22897 22868 22852 22849 22836 22831 22766 22701 22613 22610 22304 22298 22156 22140 21957 21856 21850 21664 21435 21180 20866 20850 20787 20754 20311 19884 19282 18658 18439 17859 15884 13339 11810 11516 9839 9838 9837 9835 9788) is running...

Performance Data:
 Current Attempt: 1/5 (HARD state)
 Last Check Time: 02-06-2012 12:13:55
 Check Type: ACTIVE
 Check Latency / Duration: 0.038 / 0.285 seconds
 Next Scheduled Check: 02-06-2012 12:18:55
 Last State Change: 06-16-2011 20:47:16
 Last Notification: N/A (notification 0)
 Is This Service Flapping? **NO** (0.00% state change)
 In Scheduled Downtime? **NO**
 Last Update: 02-06-2012 12:17:34 (0d 0h 0m 2s ago)

Active Checks: **ENABLED**
 Passive Checks: **ENABLED**
 Obsessing: **ENABLED**
 Notifications: **ENABLED**
 Event Handler: **ENABLED**
 Flap Detection: **ENABLED**

Service Commands

- ☒ Disable active checks of this service
- ☒ Re-schedule the next check of this service
- ☒ Submit passive check result for this service
- ☒ Stop accepting passive checks for this service
- ☒ Stop obsessing over this service
- ☒ Disable notifications for this service
- ☒ Send custom service notification
- ☒ Schedule downtime for this service
- ☒ Disable event handler for this service
- ☒ Disable flap detection for this service

Bild 5.4 Das PostgreSQL-Plugin in Nagios

5.3.2.3 Der pgSNMP-Agent

Viele Monitoring-Frameworks verfügen über eine SNMP-Schnittstelle. Agents, die auf den Zielsystemen laufen, laden ihre Informationen als SNMP-Traps zum zentralen Monitoring Tool. Wenn Sie also bereits ein Monitoring-Framework einsetzen, kann der Health-Status der PostgreSQL-Server über diese Schnittstelle geladen werden.

Das Projekt „pgsnmp“ ist Teil des pgFoundry-Frameworks. Sie finden es auf der Seite „<http://pgsnmpd.projects.postgresql.org>“.

6

Performance Tuning

Die vorhergehenden Kapitel haben unter anderem nachgewiesen, dass PostgreSQL die wesentlichen Voraussetzungen für einen professionellen Einsatz erfüllt. Doch wie sieht es mit der Performance aus? Kommerziellen Datenbanksystemen wie Oracle, DB2, Microsoft SQL Server oder Sybase traut man ohne Bedenken zu, dass eine Datenbank sehr groß werden kann, viele Benutzer damit arbeiten können und dabei eine akzeptable Performance erreicht wird. Bei Open-Source-Produkten bestehen da schon eher Bedenken. Insbesondere stellen sich die folgenden Fragen:

- Ist die Performance vergleichbar mit kommerziellen Datenbanksystemen?
- Wie kann ein PostgreSQL Server optimiert werden?
- Welche regelmäßigen Wartungsaufgaben sind notwendig?
- Wo liegen die Stärken und Schwächen des Systems?
- Wie steht es um die Skalierbarkeit?
- Wie reagiert PostgreSQL auf Datenbankwachstum und eine zunehmende Anzahl von Benutzern?

Diese und andere Fragen rund um die Performance beantworten wir in diesem Kapitel.

■ 6.1 PostgreSQL Server-Tuning

Mit der Installation eines neuen Servers werden zunächst die Standard-Parameter übernommen. Diese sind für viele Datenbanken hinreichend gut, doch für keine optimal. Nun stellt sich die Frage: Welche Anpassungen sollten initial beim Aufsetzen eines neuen Servers respektive einer neuen Datenbank vorgenommen werden? Hierfür gibt es eine Reihe von Regeln, die man unabhängig von Größe und Typ der Datenbank beachten sollte.

6.1.1 Goldene Regeln für neue Server und Datenbanken

1. Legen Sie die Größe des Buffer Cache fest. Der Buffer Cache ist eines der wichtigsten Instrumente für eine Steigerung der Performance. Je mehr Daten darin gehalten werden können, desto geringer ist der Aufwand für physikalische I/O-Operationen. Die Größe des Buffer Cache wird mit dem Parameter „shared_buffers“ festgelegt. Setzen Sie den Wert auf mindestens 25 % des gesamt verfügbaren Hauptspeichers, wenn es sich um einen dedizierten Datenbankserver handelt. Ein Wert von 50 % kann durchaus sinnvoll sein. Beachten Sie, dass der Memory-Verbrauch auch von der Anzahl der Client-Verbindungen und den dort eingestellten Parametern, wie zum Beispiel „work_mem“, abhängig ist. Setzen Sie deshalb den Parameter für den Buffer Cache nur so hoch, dass noch genügend Speicher für die Clients und für das Betriebssystem übrig bleibt. Auf dem Server sollte kein Paging infolge unzureichender Memory-Kapazitäten auftreten.
2. Bestimmen Sie die maximale Anzahl von Verbindungen, und geben Sie einen Puffer dazu. Tragen Sie diesen Wert in den Parameter „max_connections“ ein. Wird er im laufenden Betrieb erreicht, können Clients keine Verbindungen mehr herstellen, und ein Neustart des Servers ist zur Erhöhung des Parameterwertes erforderlich.
3. Der Parameter „effective_cache_size“ ist eine Schätzung für die Größe des Betriebssystem-Cache. Es handelt sich um eine Einschätzung, wie viel Speicher im Betriebssystem und dem PostgreSQL-Server zur Verfügung stehen. Dieser Wert ist wichtig für den Query-Planer. Ist er zu klein, werden vorhandene Indexe möglicherweise nicht benutzt. Die Empfehlung liegt in einem Bereich von 1/2 bis 3/4 des insgesamt vorhandenen Hauptspeichers. Wählen Sie für einen konservativen Ansatz die Hälfte und für eine aggressive Strategie drei Viertel.
4. Dividieren Sie die angenommene Größe des Betriebssystem-Cache durch die Anzahl der maximalen Verbindungen und diesen Wert durch zwei. Nehmen Sie das Ergebnis als Wert für den Parameter „work_mem“. Das folgende Beispiel illustriert eine solche Abschätzung:
 - Der Datenbankserver hat einen Memory von 4 GByte.
 - Der Parameter „shared_buffers“ wird auf 25 %, also 1 GByte gesetzt.
 - Es werden maximal 200 gleichzeitige Verbindungen zum PostgreSQL Server erwartet. Der Parameter „max_connections“ wird auf den Wert „250“ gesetzt.
 - Die Größe des Betriebssystem-Cache wird auf 500 MByte geschätzt, entsprechend wird der Parameter „effective_cache_size“ auf 2,5 GByte gesetzt.
 - Die Größe des Betriebssystem-Cache durch die Anzahl der maximalen Verbindungen durch zwei ergibt einen Wert von 1 MByte. Dies ist der Schätzwert für den Parameter „work_mem“.
5. Setzen Sie den Parameter „maintenance_work_mem“ auf 50 MByte pro Gigabyte des zur Verfügung stehenden Hauptspeichers.
6. Vergrößern Sie den Wert des Parameters „checkpoint_segments“. Der Standardwert von drei ist für viele Datenbanken zu klein, insbesondere wenn viele Daten geladen werden oder eine hohe Transaktionsrate existiert. Ein Wert von „10“ ist für die meisten Datenbanken angemessen.

7. Setzen Sie den Parameter „wal_buffers“ auf 16 MByte. Der Standardwert von 64 KByte ist zu klein für viele Datenbanken. Insbesondere Datenbanken mit einem höheren Transaktionsaufkommen profitieren von einem größeren Puffer.

Mit dieser Einstellung haben Sie die Basis für eine gute Performance des Datenbankservers geschaffen.

Seit der Version 8.4 steht über das *pgFoundry-Projekt* ein Programm mit dem Namen *pgtune* zur Verfügung. „pgFoundry“ ist eine Plattform der PostgreSQL Development-Gruppe zur Entwicklung und Publizierung von Software für PostgreSQL, die nicht Bestandteil des Kernprodukts ist. Wir werden noch weitere Tools aus diesem Programm vorstellen. Der folgende Abschnitt beschreibt den Einsatz von „pgtune“ als Werkzeug zur Datenbank-Optimierung.

6.1.2 Tuning mit „pgtune“

Sie finden das Projekt auf der pgFoundry-Webseite unter dem Link <http://pgfoundry.org/projects/pgtune>. Laden Sie die tar-Datei herunter, und entpacken Sie diese. Kopieren Sie das Programm „pgtune“ sowie die Settings-Dateien in das Verzeichnis „/usr/bin“. Es handelt sich um ein Python-Skript, das nicht installiert werden muss. Den Aufruf des Programms sehen Sie in Listing 6.1. Unter den Optionen „-i“ und „-o“ werden Ein- und Ausgabedateiname der Konfigurationsdatei des Servers angegeben.

Listing 6.1 Das Programm „pgtune“ ausführen

```
$ pgtune -i /usr/local/pgsql/data/postgresql.conf
-o /usr/local/pgsql/data/postgresql.conf.new
```

Das Programm kommentiert die Parameter, die geändert werden, und fügt am Ende der Datei die geänderten Werte ein. In Listing 6.2 finden Sie ein Beispiel.



Praxistipp

Auch wenn „pgtune“ nicht immer die beste Entscheidung trifft, ist es besser, mit den durch „pgtune“ gewählten Parametern als mit der Standard-Konfiguration zu beginnen. Dies garantiert eine gute Performance für den Start. Mit angeschlossener Applikation können dann weitere Tuning-Maßnahmen durchgeführt werden.

Listing 6.2 Durch „pgtune“ geänderte Parameter in der Datei postgresql.conf

```
#-----
# CUSTOMIZED OPTIONS
#-----
#custom_variable_classes = ''          # list of custom variable class names
default_statistics_target = 50 # pgtune wizard 2011-05-10
maintenance_work_mem = 192MB # pgtune wizard 2011-05-10
```



```
checkpoint_completion_target = 0.9 # pgtune wizard 2011-05-10
effective_cache_size = 2304MB # pgtune wizard 2011-05-10
work_mem = 20MB # pgtune wizard 2011-05-10
wal_buffers = 8MB # pgtune wizard 2011-05-10
checkpoint_segments = 16 # pgtune wizard 2011-05-10
shared_buffers = 768MB # pgtune wizard 2011-05-10
max_connections = 80 # pgtune wizard 2011-05-10
```

6.1.3 Optimierung der Memory-Parameter

Der Buffer Cache nimmt einen großen Teil des Shared Memory ein und spielt eine zentrale Rolle. Doch welchen Performance-Gewinn erreicht man mit einem größeren Buffer Cache, und wie wird er aktuell genutzt? Antworten darauf liefert das Werkzeug „pg_buffercache“, das Teil von PostgreSQL Contribution ist. Es kann wie der PostgreSQL Server selbst aus dem Quellcode installiert werden. Die notwendigen Schritte finden Sie in Listing 6.3.

Listing 6.3 Das Utility „pg_buffercache“ installieren

```
# cd /root/postgresql-9.1.0/contrib/pg_buffercache
# make
. . .
# make install
/bin/mkdir -p '/usr/local/pgsql/lib'
/bin/mkdir -p '/usr/local/pgsql/share/contrib'
/bin/sh ../../config/install-sh -c -m 755 pg_buffercache.so
/usr/local/pgsql/lib/pg_buffercache.so'
/bin/sh ../../config/install-sh -c -m 644 ./uninstall_pg_buffercache.sql
'/usr/local/pgsql/share/contrib'
/bin/sh ../../config/install-sh -c -m 644 pg_buffercache.sql
'/usr/local/pgsql/share/contrib'
```

Schließlich müssen Sie das Modul noch in der Datenbank installieren. Folgen Sie dem Beispiel in Listing 6.4.

Listing 6.4 pg_buffercache in der Datenbank installieren

```
$ psql -d hanser -f /usr/local/pgsql/share/contrib/pg_buffercache.sql
SET
CREATE FUNCTION
CREATE VIEW
REVOKE
REVOKE
```

Die View pg_buffercache enthält eine Zeile für jeden Buffer, der sich im Cache befindet. Beachten Sie, dass der Buffer Cache von allen Datenbanken des Servers gemeinsam benutzt wird. Die Abfrage in Listing 6.5 liefert die zwölf Objekte mit der höchsten Buffer-Belegung für die aktuelle Datenbank.

Listing 6.5 Anzeige der Belegung im Buffer Cache

```
(postgres@[local]:5432) [postgres] > SELECT c.relname, count(*) AS buffers
> FROM pg_buffercache b
> INNER JOIN pg_class c
```

```
> ON b.relfilename = pg_relation_filename(c.oid) AND
> b.reldatabase IN
> (0, (SELECT oid FROM pg_database
> WHERE datname = current_database()))
> GROUP BY c.relname
> ORDER BY 2 DESC
> LIMIT 12;
```

relname	buffers
test	22728
pg_proc	59
pg_depend	45
pg_attribute	24
pg_description	21
pg_proc_prname_args_nsp_index	14
pg_operator	13
pg_rewrite	12
pg_statistic	12
pg_class	10
pg_depend_reference_index	10
pg_type	10

(12 rows)

Um die Effekte des Buffer Cache besser verstehen zu können, unternehmen wir einen Ausflug in das Disk-Layout von PostgreSQL. Die Tabellen werden unter dem Verzeichnis `$PGDATA/base` gespeichert. Zu diesem Zweck legen wir eine neue Tabelle an:

```
(postgres@[local]:5432) [postgres] > CREATE TABLE neu
(sn SERIAL, wert INTEGER);
NOTICE: CREATE TABLE will create implicit sequence "neu_sn_seq" for serial
column "neu.sn"
CREATE TABLE
```

Im Verzeichnis `$PGDATA/base` befinden sich weitere Unterverzeichnisse, die jeweils zu einer Datenbank gehören. Der Verzeichnisname ist identisch mit dem Object Identifier (OID) der Datenbank. In Listing 6.6 finden Sie die OID der Datenbanken sowie die zugehörigen Verzeichnisse.

Listing 6.6 Die Verzeichnisstruktur der Datenbanken

```
(postgres@[local]:5432) [postgres] > SELECT oid, datname FROM pg_database;
```

oid	datname
1	template1
11866	template0
11874	postgres
16384	hanser

(4 rows)

```
drwx----- 2 postgres postgres 12288 Apr 22 18:45 1
drwx----- 2 postgres postgres 4096 Apr 22 18:42 11866
drwx----- 2 postgres postgres 4096 May 10 20:42 11874
drwx----- 2 postgres postgres 4096 May 10 17:26 16384
drwx----- 2 postgres postgres 4096 May 10 20:31 pgsql_tmp
```

Analog erhält jede Tabelle einen OID, der identisch mit dem Dateinamen ist. In unserem Beispiel wurde eine leere Datei mit dem Namen „16404“ angelegt (siehe Listing 6.7).

Listing 6.7 Die Relation zwischen Tabelle und Dateinamen herstellen

```
(postgres@[local]:5432) [postgres] > SELECT oid,relname
> FROM pg_class WHERE relname = 'neu';
 oid | relname
-----+-----
 16404 | neu
(1 row)
$ ls -l $PGDATA/base/11874/16404
-rw----- 1 postgres postgres 0 May 10 20:42
/usr/local/pgsql/data/base/11874/16404
```

Wenn Sie nun einen Satz in die Tabelle einfügen, dann wird ein Block im Buffer Cache allokiert und ein Block auf der Disk reserviert. Dieser Block wird als „dirty“ markiert, da sich die aktuelle Version im Cache und nicht auf der Disk befindet. Aus Listing 6.8 ist ersichtlich, dass der Block nach einem Checkpoint nicht mehr als „dirty“ markiert ist.

Listing 6.8 Status eines Blocks vor und nach einem Checkpoint

```
(postgres@[local]:5432) [postgres] > SELECT
relblocknumber,isdirty,usagecount
> FROM pg_buffercache
> WHERE relfilenode = 16404;
 relblocknumber | isdirty | usagecount
-----+-----+-----
              0 | t       |          1
(postgres@[local]:5432) [postgres] > checkpoint;
CHECKPOINT
(postgres@[local]:5432) [postgres] > SELECT
relblocknumber,isdirty,usagecount
> FROM pg_buffercache
> WHERE relfilenode = 16404;
 relblocknumber | isdirty | usagecount
-----+-----+-----
              0 | f       |          1
```

Die Abfrage in Listing 6.9 liefert das Verhältnis von Buffer mit dem Status „dirty“.

Listing 6.9 Verhältnis der Status im Buffer Cache

```
(postgres@[local]:5432) [postgres] > SELECT isdirty,count(*)
> FROM pg_buffercache
> GROUP BY isdirty;
 isdirty | count
-----+-----
 t       | 369957
 f       |      25
 f       | 23234
(3 rows)
```

Ein weiterer Parameter, der die Performance stark beeinflussen kann, ist „wal_buffers“. Insbesondere Applikationen mit einer hohen Transaktionsrate erzeugen ein großes Volumen an WAL-Daten. Die Empfehlung lautet, den Wert mindestens auf 1 MByte und für stärker belastete Server auf 16 MByte zu setzen.

■ 6.2 Performance-Analyse

Für die Performance-Analyse können Sie „Statistics Collector“ und „pgstatspack“ verwenden. Der „Statistics Collector“ ist ein System, um Informationen über die Aktivitäten des PostgreSQL-Servers zu sammeln. „pgstatspack“ ist Teil des Projekts „pgFoundry“ und dient der nachträglichen Performance-Analyse durch Auswertung der Snapshots und eines Standard-Berichts.

6.2.1 Analyse mit dem „Statistics Collector“

Der Statistics Collector unterstützt das Sammeln und die Auswertung von Statistiken über die Aktivitäten des PostgreSQL-Servers. Er sammelt zum Beispiel Informationen über den Zugriff auf Tabellen und Indexe, über Vacuum und Analyze sowie über Funktionen und deren Laufzeiten.

Das Sammeln von Statistiken hat natürlich Einfluss auf den Ressourcen-Verbrauch des Servers. Deshalb kann das Sammeln ein- und ausgeschaltet werden. In Tabelle 6.1 finden Sie eine Übersicht der zugehörigen Parameter.

Tabelle 6.1 Parameter für das Sammeln von Statistiken

Parameter	Standard	Bedeutung
track_activities	on	Schaltet das Sammeln von Informationen über gerade ausgeführte SQL-Anweisungen auf Session-Ebene ein und aus.
track_activity_query_size	1024	Anzahl von reservierten Bytes zur Aufzeichnung von aktuell ausgeführten Kommandos auf Session-Ebene.
track_counts	on	Schaltet das Sammeln von Statistiken auf Datenbank-Ebene ein und aus.
track_functions	none	Ein- und Ausschalten der Überwachung von Funktionen (Aufrufe und Ausführungszeit). „pl“ für das Aufzeichnen von prozeduralen Funktionen, „all“ für das Aufzeichnen aller Funktionen.

Für die Auswertung der Statistik-Daten stellt PostgreSQL eine Reihe von mitgelieferten Views zur Verfügung. Sie besitzen den Präfix „pg_stat_“. In Tabelle 6.2 finden Sie eine Auswahl von Views des Statistics Collector.



Praxistipp

Mit Version 9 wurde die Funktion „pg_stat_reset()“ eingeführt. Damit werden alle Statistiken für die aktuelle Datenbank zurückgesetzt.

Tabelle 6.2 Views des Statistics Collector

View	Beschreibung
pg_stat_activity	Enthält Informationen des Client/Server-Prozesses sowie der damit verbundenen Session-Informationen im PostgreSQL-Server. Dient als Prozessübersicht mit einer Zeile pro Prozess.
pg_stat_database	Liefert Statistiken kumuliert auf Datenbank-Ebene. Dazu gehören die Anzahl von Transaktionen, Datenblöcke, die aus dem Cache und von der Disk gelesen wurden.
pg_stat_bgwriter	Liefert die Statistiken des Background Writers. Eine Zeile pro Cluster.
pg_stat_all_tables	Listet Statistiken für jede Tabelle. Dazu gehören Informationen wie die Anzahl von Full Table Scans sowie die Anzahl von Sätzen, die mittel Table Scan oder Index Scan gelesen wurden.
pg_stat_all_indexes	Enthält Statistiken über alle Indexe, wie zum Beispiel die Anzahl der ausgeführten Index Scans oder die Anzahl der zurückgelieferten Einträge pro Index-Scan.
pg_stat_database_conflicts	Statistik über die Anzahl nicht erfolgreich ausgeführter SQL-Anweisungen. Enthält eine Zeile pro Datenbank.
pg_stat_replication	Liefert Statistiken über Aktionen beim Betrieb von Standby-Datenbanken und über den Status der einzelnen Prozesse. Es wird ein Satz pro WAL Sender-Prozess angezeigt.

Für die Performance-Analyse ist es wichtig festzustellen, auf welchen Tabellen Table Scans laufen. Diese Informationen erhalten Sie aus dem View „pg_stat_user_tables“.

Listing 6.10 Das Verhältnis von Table und Index Scans anzeigen

```
(postgres@[local]:5432) [postgres] > SELECT schemaname,relname,seq_scan,
idx_scan
> FROM pg_stat_user_tables
> ORDER BY 3 DESC;
```

schemaname	relname	seq_scan	idx_scan
public	kb	233	
public	pgstatpack_statements	1	111
public	audit_emp	1	
. . .			

Die Statistik in Listing 6.10 sagt nur aus, wie viele Table und Index Scans aufgetreten sind. Interessant ist natürlich darüber hinaus, wie viele Sätze durch diese Scans tatsächlich verarbeitet wurden. Auch diese Informationen sind verfügbar (siehe Listing 6.11).

Listing 6.11 Statistik über verarbeitete Sätze

```
(postgres@[local]:5432) [postgres] > SELECT relname,
seq_tup_read,idx_tup_fetch
> FROM pg_stat_user_tables;
```

relname	seq_tup_read	idx_tup_fetch
pgstatpack_sequences	0	0
pgstatpack_bgwriter	0	0


```

pgstatspack_tables |          0 |          0
kb                 | 15000000 |
. . .

```

Wissenswert ist auch das Verhältnis von INSERT-, UPDATE- und DELETE-Anweisungen für eine Tabelle. Tabellen, die häufigen UPDATE-Anweisungen unterworfen sind, sind ein Kandidat für regelmäßigen Neuaufbau der Indexe. Dasselbe gilt für Tabellen, die einen hohen Anteil von DELETE-Anweisungen aufweisen. Außerdem lassen sich sogenannte „append only“-Tabellen herausfinden, die besonders behandelt werden können. In Listing 6.12 finden Sie die SQL-Anweisung zur Abfrage der Statistik.

Listing 6.12 Das Verhältnis von INSERT-, UPDATE- und DELETE-Anweisungen abfragen

```

(postgres@[local]:5432) [postgres] > SELECT relname,
> CAST(n_tup_ins AS NUMERIC) / (n_tup_ins + n_tup_upd + n_tup_del)
AS pct_insert,
> CAST(n_tup_upd AS NUMERIC) / (n_tup_ins + n_tup_upd + n_tup_del)
AS pct_update,
> CAST(n_tup_del AS NUMERIC) / (n_tup_ins + n_tup_upd + n_tup_del)
AS pct_delete
> FROM pg_stat_user_tables
> WHERE relname = 'kb';
 relname | pct_insert | pct_update | pct_delete
-----+-----+-----+-----
kb       | 0.9999998000 | 0.00000019999996 | 0.0000000000000000

```

Analog zu den Statistiken für Tabellen lassen sich Statistiken für Indexe auswerten. Listing 6.13 liefert eine Statistik über den Anteil von Index-Blöcken, die aus dem Cache gelesen wurden. Die Hit Ratio ist ein wichtiger Performance-Indikator für die Reduzierung der I/O-Performance.

Listing 6.13 I/O-Statistik für Indexe: Buffer Hit Ratio

```

(postgres@[local]:5432) [postgres] > SELECT indexrelname, idx_blks_hit,
idx_blks_read,
> (CAST(idx_blks_hit AS NUMERIC) / (idx_blks_hit + idx_blks_read)) * 100
AS hit_ratio
> FROM pg_statio_user_indexes
> WHERE indexrelname = 'i_kb'
> ORDER BY hit_ratio DESC;
indexrelname | idx_blks_hit | idx_blks_read | hit_ratio
-----+-----+-----+-----
i_kb         |          900 |          400 | 69.23076923076923076900

```

Die View „pg_stat_activity“ liefert eine Übersicht aller aktuellen Verbindungen zum PostgreSQL-Server. Enthalten ist auch die Prozess-ID des Client/Server-Prozesses auf dem Datenbankserver. Darüber lassen sich weitere prozess-spezifische Performance-Analysen durchführen.

Listing 6.14 Einen Client/Server-Prozess identifizieren

```

(postgres@[local]:5432) [postgres] > SELECT
procpid,username,application_name,client_addr
> FROM pg_stat_activity;
 procpid | username | application_name | client_addr

```



```

-----+-----+-----+-----+
 4624 | postgres | psql | |
 6146 | postgres | pgAdmin III- Browser | 192.168.178.22
 6159 | postgres | pgAdmin III- Abfragewerkzeug | 192.168.178.22
$ ps -ef|grep 6159
postgres 6159 4615 0 13:42 ? 00:00:00 postgres: postgres postgres
192.168.178.22(49855) idle

```

Außerdem zeigt die View die SQL-Anweisung an, die gerade in der Session läuft.

Listing 6.15 Die aktuelle SQL-Anweisung anzeigen

```

(postgres@[local]:5432) [postgres] > SELECT procpid, current_query
> FROM pg_stat_activity;
procpid | current_query
-----+-----
 6224 | SELECT procpid, current_query+
      | FROM pg_stat_activity;
 6146 | <IDLE>
 6159 | select * from kb

```

6.2.2 Der Background Writer

Der Background Writer hat wesentlichen Anteil an einer erfolgreichen Gesamt-Performance des Datenbank-Servers. Statistiken dazu liefert die View „pg_stat_bgwriter“; sie enthält Statistiken über das Verhalten der Datenblöcke sowie des angebundenen Checkpoint-Prozesses. Besonders interessant sind die folgenden Informationen:

- wie viele Daten im Durchschnitt in einem Checkpoint und
- wie viel Prozent der Daten bei einem Checkpoint geschrieben werden;
- das Verhältnis zwischen durch eine Aktivität und ein Zeitintervall ausgelöste Checkpoints.

Die SQL-Abfrage in Listing 6.16 liefert die wichtigsten Statistiken über das Checkpoint-Verhalten.

Listing 6.16 Statistiken über Checkpoints abrufen

```

(postgres@[local]:5432) [postgres] > SELECT (100 * checkpoints_req)/
(checkpoints_timed + checkpoints_req) AS req_pct,
> pg_size_pretty (buffers_checkpoint * block_size / (checkpoints_timed +
checkpoints_req)) AS avg_write,
> pg_size_pretty (block_size * (buffers_checkpoint + buffers_clean + buffers_
backend)) AS total_write,
> 100 * buffers_checkpoint / (buffers_checkpoint + buffers_clean + buffers_
backend ) AS chkpt_pct,
> 100 * buffers_backend / (buffers_checkpoint + buffers_clean + buffers_
backend) AS write_pct
> FROM pg_stat_bgwriter,
> (SELECT CAST(current_setting('block_size') AS integer) AS block_size) as bs;
req_pct | avg_write | total_write | chkpt_pct | write_pct
-----+-----+-----+-----+-----
      8 | 8119 kB  | 4104 MB    |      47   |      48

```


6.2.3 Analyse mit „pgstatspack“

Für die nachträgliche Performance-Analyse steht das Produkt *pgstatspack* zur Verfügung. Es ist Teil des Projekts „pgFoundry“ und ist auf der pgFoundry-Webseite unter der URL <http://pgfoundry.org/projects/pgstatspack> zu finden. Die Installation erfolgt wie gewohnt durch Entpacken der tar-Datei unter dem Verzeichnis `/usr/local/pgsql/share`. Die Installation in den Datenbanken erfolgt durch Aufruf des Skripts „install_pgstats.sh“. Die Installation erfolgt in allen Datenbanken mit Ausnahme der System-Datenbank „postgres“.

Listing 6.16 Das Utility „pgstatspack“ in den Datenbanken installieren

```
$ ./install_pgstats.sh
/usr/local/pgsql/share/pgstatspack /usr/local/pgsql/share/pgstatspack
Results for database hanser
Installing Statistics Package for database hanser
psql:sql/pgstatspack_create_tables.sql:7: NOTICE:  table "pgstatspack_snap"
does not exist, skipping
DROP TABLE
CREATE TABLE
psql:sql/pgstatspack_create_tables.sql:15: NOTICE:  table "pgstatspack_
database" does not exist, skipping
DROP TABLE
CREATE TABLE
. . .
INSERT 0 1
CREATE FUNCTION
CREATE FUNCTION
/usr/local/pgsql/share/pgstatspack
```

„pgstatspack“ benötigt das Modul *pg_stat_statements*, das Ausführungsstatistiken für alle SQL-Anweisungen des PostgreSQL-Servers sammelt. Es ist Teil der PostgreSQL Contribution und bereits im Quellcode des PostgreSQL-Kerns enthalten. Führen Sie die folgenden Schritte aus, um „pg_stat_statements“ zu installieren und in den Server einzubinden:

1. Wechseln Sie als Benutzer „root“ in das Verzeichnis mit den Installationsquellen. Wir erstellen das Modul aus dem Quellcode.

```
# cd /root/postgresql-9.1.0/contrib/pg_stat_statements
```

2. Führen Sie die Kommandos „make“ für das Erstellen der Library sowie „make install“ für das Kopieren der Dateien in die Verzeichnisse des PostgreSQL-Servers aus.

```
[root@serv1 pg_stat_statements]# make
sed 's,MODULE_PATHNAME,$libdir/pg_stat_statements,g'
pg_stat_statements.sql.in >pg_stat_statements.sql
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-
statement -Wendif-labels -fno-strict-aliasing -fwrapv -fpic -I. -I. -
I../src/include -D_GNU_SOURCE -c -o pg_stat_statements.o
pg_stat_statements.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-
statement -Wendif-labels -fno-strict-aliasing -fwrapv -fpic -shared -o
pg_stat_statements.so pg_stat_statements.o -L../src/port -Wl,-
rpath,'/usr/local/pgsql/lib',--enable-new-dtags
[root@serv1 pg_stat_statements]# make install
```



```

/bin/mkdir -p '/usr/local/pgsql/lib'
/bin/mkdir -p '/usr/local/pgsql/share/contrib'
/bin/sh ../../config/install-sh -c -m 755 pg_stat_statements.so
'/usr/local/pgsql/lib/pg_stat_statements.so'
/bin/sh ../../config/install-sh -c -m 644
./uninstall_pg_stat_statements.sql '/usr/local/pgsql/share/contrib'
/bin/sh ../../config/install-sh -c -m 644 pg_stat_statements.sql '/usr/local/
pgsql/share/contrib'

```

3. Melden Sie sich jetzt als Benutzer „postgres“ an, und installieren Sie die Library in der Datenbank.

```

$ psql -d hanser -f /usr/local/pgsql/share/contrib/pg_stat_statements.sql
SET
CREATE FUNCTION
CREATE FUNCTION
CREATE VIEW
GRANT
REVOKE

```

4. Sie müssen das Modul „pg_stat_statements“ als Preload Shared Library deklarieren, um es bereits beim Start des PostgreSQL-Servers als Shared Library zu laden. Dies erfolgt mit dem Parameter *shared_preload_libraries*. Zusätzlich müssen zwei Parameter für das Modul eingetragen werden. Führen Sie die folgenden Einträge in der Konfigurationsdatei „postgresql.conf“ durch, und starten Sie den Server neu.

```

# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'
custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all

```

Damit lässt sich „pgstatspack“ mit allen Features verwenden. Snapshots können manuell oder automatisch erstellt werden. Ein manuelles Snapshot erfolgt mit dem Aufruf der Funktion „pgstatspack_snap“ (siehe Listing 6.17).

Listing 6.17 Einen manuellen Snapshot mit „pgstatspack“ erzeugen

```

(postgres@[local]:5432) [hanser] > SELECT pgstatspack_snap('Erstes
Snapshot');
 pgstatspack_snap
-----
                5
(1 row)

```

Das automatische Erstellen von Snapshots kann über einen beliebigen Scheduler erfolgen, zum Beispiel mit „crontab“:

```

*/10 * * * * /usr/local/pgsql/share/pgstatspack/bin/snapshot.sh
1> /usr/local/pgsql/share/logs/pgstatspack.log 2>&1

```

Die Snapshots werden nun automatisch erzeugt, und die Berichte können im Nachhinein abgerufen werden.

```

(postgres@[local]:5432) [hanser] > SELECT * FROM pgstatspack_snap;
 snapid |          ts          | description
-----+-----+-----

```



```

5 | 2011-05-11 13:21:24.973512 | Erstes Snapshot
6 | 2011-05-11 13:30:01.208728 | cron based snapshot
7 | 2011-05-11 13:40:01.331015 | cron based snapshot

```

Ein Bericht wird mit dem Shell-Skript *pgstatpack_report.sh* erstellt. Es können die Parameter Benutzername und Datenbank direkt mitgegeben werden. In der Option „-f“ kann der Dateiname für die Ausgabedatei des Berichts angegeben werden.

Listing 6.18 Einen pgstatpack-Bericht erstellen

```

$ cd /usr/local/pgsql/share/pgstatpack/bin
$ ./pgstatpack_report.sh -u postgres -d hanser -f /tmp/sp_repl.txt
/usr/local/pgsql/share/pgstatpack/bin
snapid |          ts          | description
-----+-----+-----
      7 | 2011-05-11 13:40:01.331015 | cron based snapshot
      6 | 2011-05-11 13:30:01.208728 | cron based snapshot
      5 | 2011-05-11 13:21:24.973512 | Erstes Snapshot
(3 rows)
Enter start snapshot id : 6
Enter stop snapshot id  : 7
. . .

```

Welche Aussagen liefert ein pgstatpack-Bericht? Am Anfang findet sich eine Statistik über alle Datenbanken. Obwohl der Bericht für eine Datenbank erstellt wird, ist immer zu beachten, dass viele Ressourcen für alle Datenbanken des Servers geteilt werden. Die Datenbank-Statistik sagt aus, in welchem Umfang die Ressourcen von den einzelnen Datenbanken benutzt wurden. In Listing 6.19 ist zu sehen, dass im vorliegenden Zeitraum die überwiegende Last von der Datenbank „hanser“ verursacht wurde. Neben der Hit-Rate des Buffer Pool finden Sie die Anzahl von Transaktionen pro Sekunde, die Anzahl von physischen und logischen I/O-Operationen pro Sekunde sowie die Rollback-Operationen. Alle Werte sind Durchschnittswerte über den Zeitraum zwischen den gewählten Snapshots. Auffällig ist die schlechte Hit-Rate für die Datenbank „hanser“.

Listing 6.19 Datenbank-Statistik im pgstatpack-Bericht

```

Database statistics
database | tps | hitrate | lio_ps | pio_ps | rollbk_ps
-----+-----+-----+-----+-----+-----
hanser   | 0.56 | 4.00 | 24297.96 | 23133.81 | 0.00
postgres | 0.24 | 98.00 | 7.53 | 0.11 | 0.19
template1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00
template0 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00

```

Es folgen weitere Statistiken wie Tabellen mit dem stärksten Wachstum, das Verhältnis zwischen Tabellen- und Indexzugriffen. Außerdem gibt es eine Übersicht für Indexe, geordnet nach den häufigsten Index-Scans. Weiter unten finden Sie eine Tabelle mit SQL-Anweisungen, absteigend geordnet nach der Laufzeit. Langläufer lassen sich damit gut identifizieren (siehe Listing 6.20).

Listing 6.20 Top SQL-Anweisungen im pgstatpack-Bericht

```

Top 20 SQL statements ordered by total_time
calls | total_time | rows | query
-----+-----+-----+-----

```



```

2 | 0.000042 | 0 | SELECT groname FROM pg_group ORDER BY groname
2 | 0.000040 | 2 | SELECT count(*) FROM public.pgstatspack_table
2 | 0.000040 | 2 | SELECT count(*) FROM public.pgstatspack_datab
2 | 0.000040 | 2 | SELECT count(*) FROM public.pgstatspack_setti
2 | 0.000040 | 2 | SELECT count(*) FROM public.pgstatspack_versi
2 | 0.000038 | 2 | SELECT count(*) AS rows FROM ONLY orders
. . .

```

Den Abschluss bildet eine Übersicht der gesetzten Server-Parameter.

Wir erhöhen die Shared Buffers, um die Hit-Rate des Buffer-Pools zu verbessern, und erstellen einen neuen Bericht. Nach der Vergrößerung erhöhte sich die Hit-Rate auf 52 %.

Weitere Optionen zur Auswertung und Überwachung der Performance bietet der *Statistics Collector*. Es handelt sich um ein Subsystem, das viele wertvolle Informationen über die Server-Aktivitäten sammelt. So werden zum Beispiel die Zugriffe auf Tabellen und Indexe gezählt oder die Anzahl von Sätzen in einer Tabelle überwacht. Das Sammeln der Statistiken kann ein- und ausgeschaltet werden. Dazu dienen die folgenden Parameter:

- „track_counts“ steuert das Sammeln von Statistiken über den Zugriff auf Tabellen und Indexe.
- „track_functions“ schaltet die Statistiken über „Benutzerdefinierte Funktionen“ ein und aus.
- „track_activities“ steuert die Überwachung der aktuell ausgeführten Befehle.

PostgreSQL besitzt zwei Arten von Statistik-Views. Die erste speichert Statistiken in Zähler, die mit dem Wert „0“ starten, wenn das Datenbank-Cluster erstellt wird. Ein Beispiel für diese Views ist *pg_stat_database*, das Statistiken auf Datenbank-Ebene sammelt. Die Statistik-Werte sind kumulativ.

Listing 6.21 Das Statistik-View „pg_stat_database“

```

(postgres@[local]:5432) [hanser] > SELECT
  datid,datname,xact_commit,blks_read,blks_hit
> FROM pg_stat_database;
 datid | datname  | xact_commit | blks_read | blks_hit
-----+-----+-----+-----+-----
      1 | template1 |           0 |          0 |          0
 11866 | template0 |           0 |          0 |          0
 11874 | postgres  |        5319 |    129996 |   9833956
 16384 | hanser    |        5538 |   20246984 |   6947948
(4 rows)

```

Die zweite Kategorie bilden die Live-Views, die aktuelle Zustände in der Datenbank erfassen. Ein Beispiel ist die View *pg_stat_activity*, die Informationen über die aktuellen Sessions in den Datenbanken liefert.

Listing 6.22 Übersicht der Sessions in den Datenbanken

```

(postgres@[local]:5432) [hanser] > SELECT
  datname,username,client_addr,backend_start
> FROM pg_stat_activity;
 datname | username | client_addr | backend_start
-----+-----+-----+-----
 postgres | postgres | 192.168.178.21 | 2011-05-11 15:58:56.421705+02

```



```

hanser | postgres | 192.168.178.21 | 2011-05-11 15:59:01.949992+02
hanser | postgres |                | 2011-05-11 16:03:55.85309+02
hanser | postgres | 192.168.178.21 | 2011-05-11 16:10:47.369705+02
hanser | postgres | 192.168.178.21 | 2011-05-11 16:15:50.115524+02
(5 rows)

```

Interessant für Performance-Analysen sind immer I/O-Statistiken über Tabellen. Mit Hilfe der Views „pg_statio_user_tables“ können Sie nicht nur Informationen darüber gewinnen, wie viele Blöcke von welcher Tabelle gelesen werden, sondern auch die Hit-Raten für die Top-Tabellen bestimmen. Die SQL-Anweisung in Listing 6.23 liefert diese Werte.

Listing 6.23 Top-Tabellen und deren Hit-Rate ermitteln

```

(postgres@[local]:5432) [hanser] > SELECT relname,
> ROUND(CAST(heap_blks_hit AS NUMERIC) / (heap_blks_hit +
heap_blks_read),2) AS hit_ratio,
> heap_blks_hit, heap_blks_read
> FROM pg_statio_user_tables
> ORDER BY heap_blks_read DESC;

```

relname	hit_ratio	heap_blks_hit	heap_blks_read
orders	0.25	6673870	20241583
pgstatpack_indexes	0.98	7290	130
pgstatpack_settings	0.97	1349	38
pgstatpack_tables	0.97	1241	34
pgstatpack_statements	0.94	515	32
pgstatpack_database	0.94	127	8
pgstatpack_snap	0.94	81	5
pgstatpack_sequences	0.86	30	5
pgstatpack_version	0.33	1	2

```

(9 rows)

```

Wichtig für die Analyse der aktuellen Situation in den Datenbanken ist die Überwachung von Sessions in der Datenbank. Die View „pg_stat_activity“ liefert diese Informationen. Mit Hilfe der SQL-Anweisung in Listing 6.24 können Langläufer identifiziert und die zugehörigen Sessions ermittelt werden.

Listing 6.24 Abfrage von Top Sessions

```

(postgres@[local]:5432) [hanser] > SELECT procpid,waiting,
> current_timestamp - LEAST(query_start,xact_start) AS run_time,
> current_query
> FROM pg_stat_activity
> WHERE NOT procpid = pg_backend_pid();

```

procpid	waiting	run_time	current_query
5975	f	01:07:08.997046	<IDLE>
5977	f	00:55:42.105917	<IDLE>
6042	f	00:54:56.222	<IDLE>
6061	f	00:00:01.220954	select count(*) from orders

In der Spalte „procpid“ finden Sie die Prozess-ID des Clients auf dem Datenbankserver. Wenn eine Session zwangsweise aus der Datenbank entfernt werden muss, kann der Prozess auf dem Datenbank-Server durch den Benutzer postgres „gekillt“ werden (siehe Listing 6.25).

Listing 6.25 Eine Session in der Datenbank killen

```
$ ps -ef|grep 6061
postgres 6061 5778 0 16:15 ?          00:00:02 postgres: postgres hanser
192.168.178.21(51761) idle
$ kill -9 6061
```

Datenbank-Applikationen sind nicht immer effektiv programmiert, und es kommt im produktiven Betrieb zu Wartezuständen, weil andere Sessions Sperren auf Sätze und Tabellen halten. Locking wird damit zum Performance-Problem in der Datenbank. Alle Informationen über aktuell gehaltene Sperren im Server können über die View *pg_locks* abgefragt werden. Die SQL-Abfrage in Listing 6.26 verbindet die View „pg_locks“ mit dem View „pg_stat_activity“. Darüber lassen sich Informationen der geblockten und der blockierenden Session herausfinden, und es können Maßnahmen zur Auflösung der Situation ergriffen werden.

Listing 6.26 Blockierende und geblockte Sessions identifizieren

```
(postgres@[local]:5432) [hanser] > SELECT w.pid AS waiter_pid, b.pid AS
blocker_pid,
> ws.username AS waiting_user, bs.username AS blocking_user,
> w.transactionid, w.locktype
> FROM pg_locks w, pg_locks b, pg_stat_activity ws, pg_stat_activity bs
> WHERE b.granted=true AND w.granted=false
> AND b.pid=ws.procpid AND w.pid=bs.procpid
> AND w.transactionid=b.transactionid;
 waiter_pid | blocker_pid | waiting_user | blocking_user | transactionid |
locktype
-----+-----+-----+-----+-----+
6438 | 6439 | postgres | postgres | 865 |
transactionid
```

■ 6.3 Partitionierung

PostgreSQL unterstützt die Partitionierung von Tabellen. Eine große Tabelle kann damit in mehrere kleinere logische Fragmente unterteilt werden. Partitionierung bietet die folgenden Vorzüge:

- Verbesserung der Performance, insbesondere bei lesendem Zugriff auf die Tabelle. Abhängig von der SELECT-Anweisung und deren Ausführungsplan muss nicht auf die gesamte Tabelle, sondern nur auf einzelne Partitionen zugegriffen werden. Full Table Scans reduzieren sich auf Partition Scans. Dasselbe gilt für Index Scans.
- Erhöhung der Wartungsfreundlichkeit von großen Tabellen.
- Partitionen können physikalisch im I/O-Subsystem verteilt werden.
- Bietet zusätzliche Möglichkeiten für die Archivierung von Daten. Die Auslagerung ins Archiv kann durch Entfernen von Partitionen erfolgen. Ältere, wenig benutzte Daten können auf kostengünstigeren Storage ausgelagert werden.

PostgreSQL unterstützt gegenwärtig zwei Arten von Partitionen:

- **Range Partitioning:** Der Partition Key kann aus einer oder mehreren Spalten bestehen. Die Unterteilung erfolgt durch definierte Bereiche des Partition Key. Mit dieser Methode ist zum Beispiel eine Partitionierung nach Datumsinformationen möglich, eine der am häufigsten verwendeten Partitionierungsmethoden. Eine Partitionierung auf der Zeitachse bietet, wie bereits erwähnt, zusätzliche Möglichkeiten für die Archivierung der Daten.
- **List Partitioning:** Die Vorgabe für den Partition Key erfolgt über Einzelwerte, die in Listen zusammengefasst sind. Jede Liste entspricht einer Partition. Die Listen müssen nicht sortiert sein. So kann zum Beispiel eine Partitionierung nach geografischen Gesichtspunkten erfolgen.

Technisch erfolgt die Umsetzung der Partitionierung mit Hilfe des Features „Tabellen-Vererbung“. Dabei müssen die Partitionen als Kind-Tabellen angelegt werden. Die Eltern-Tabelle ist leer. Schauen wir uns das Feature etwas genauer an, bevor wir uns mit der Partitionierung von Tabellen beschäftigen.

6.3.1 Vererbung von Tabellen

Tabellen-Vererbung ist ein Konzept aus der Objekt-Orientierung. In der Praxis werden Objekt-orientierte Methoden in relationalen Datenbanksystemen selten genutzt, was nicht zuletzt am mangelnden Angebot Objekt-orientierter Features liegt. Viele Datenbanken verfügen über Ansätze, eine konsequente Umsetzung ist jedoch nicht vorhanden. Dies gilt auch für PostgreSQL.

Schauen wir uns die Tabellen-Vererbung anhand eines Beispiels an. In einer HR-Applikation werden zwei Arten von Mitarbeitern verwaltet: angestellte und externe. Eine Option wäre, zwei verschiedene Tabellen zu erstellen, die sich durch eine zusätzliche Spalte „comp_id“ unterscheiden, die nur bei externen Mitarbeitern auftritt.

Listing 6.27 Verwandte Tabellen anlegen

```
(postgres@[local]:5432) [postgres] > CREATE TABLE employees (
> first_name    VARCHAR(30),
> last_name     VARCHAR(30),
> hr_id         INT,
> salary        INT);
CREATE TABLE
(postgres@[local]:5432) [postgres] > CREATE TABLE contractor (
> first_name    VARCHAR(30),
> last_name     VARCHAR(30),
> hr_id         INT,
> salary        INT,
> comp_id       INT);
CREATE TABLE
```

Aus Sicht der Applikationsentwicklung ist ein solches Design natürlich schwierig, da ständig unterschieden werden muss, ob es sich um angestellte oder um externe Mitarbeiter handelt. Lösen wir die Aufgabe mit Vererbung. Im Beispiel in Listing 6.28 wird die Tabelle „contractor“ aus der Tabelle „employees“ vererbt und erhält die zusätzliche Spalte „comp_id“.

Listing 6.28 Eine Tabelle vererben

```
CREATE TABLE contractor (
  comp_id      INT)
INHERITS (employees);
(postgres@[local]:5432) [postgres] > INSERT INTO contractor
> VALUES ('Max','Mueller',317,12000,4711);
INSERT 0 1
```

Was passiert, wenn man eine Abfrage auf die Tabelle „employees“ stellt? Weil die Tabelle „contractor“ aus der Tabelle „employees“ gebildet wurde, werden die Sätze der Tabelle „contractor“ mit angezeigt. Sollen ausschließlich die Sätze der Tabelle „employees“ angezeigt werden, müssen Sie die Option „ONLY“ verwenden.

Listing 6.29 Vererbte Tabellen abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM employees;
 first_name | last_name | hr_id | salary
-----+-----+-----+-----
Max         | Mueller   | 317   | 12000
(1 row)
(postgres@[local]:5432) [postgres] > SELECT * FROM ONLY employees;
 first_name | last_name | hr_id | salary
-----+-----+-----+-----
(0 rows)
```

6.3.2 Tabellen partitionieren

Die folgende Anleitung beschreibt das Anlegen einer partitionierten Tabelle.

1. Erstellen Sie die Master-Tabelle, von der alle Partitionen abgeleitet werden. Diese Tabelle wird keine Daten enthalten. Aus diesem Grund erhält sie keine Indexe zugewiesen.
2. Erstellen Sie mehrere Kind-Tabellen durch Vererbung von der Master-Tabelle. Die Kind-Tabellen besitzen dieselbe Tabellen-Struktur und keine zusätzlichen Spalten.
3. Fügen Sie den partitionierten Tabellen Constraints hinzu. Diese Constraints definieren in der Partition zulässige Werte.
4. Erstellen Sie für jede Partition einen Index über den Partition Key sowie über weitere Spalten.
5. Definieren Sie einen Trigger, der das Einfügen von Sätzen in die richtige Partition dirigiert.
6. Stellen Sie sicher, dass der Parameter „constraint_exclusion“ in der Datei „postgresql.conf“ nicht ausgeschaltet ist.

Führen wir das Erstellen einer partitionierten Tabelle an einem Beispiel durch. Ein Auftragssystem speichert Auftragseingänge. Diese sollen nach dem Zeitstrahl partitioniert werden.

7. Die Tabelle „order_entry“ enthält die Spalte „entry_date“, über die partitioniert werden soll.

```
(postgres@[local]:5432) [postgres] > CREATE TABLE order_entry(
> order_id      INT,
```



```
> customer      INT,
> article       INT,
> amount        INT,
> entry_date    DATE NOT NULL);
CREATE TABLE
```

8. Legen Sie nun mit Hilfe der Vererbung die einzelnen Partitionen an.

```
CREATE TABLE order_entry_201201 () INHERITS (order_entry);
CREATE TABLE order_entry_201202 () INHERITS (order_entry);
CREATE TABLE order_entry_201203 () INHERITS (order_entry);
CREATE TABLE order_entry_201204 () INHERITS (order_entry);
CREATE TABLE order_entry_201205 () INHERITS (order_entry);
CREATE TABLE order_entry_201206 () INHERITS (order_entry);
```

9. Jetzt gilt es für jede Partition den Partition Constraint zu definieren. Die Bereiche dürfen sich nicht überlappen.

```
CREATE TABLE order_entry_201201 (
CHECK ( entry_date >= DATE '2012-01-01' AND entry_date < '2012-02-01')
) INHERITS (order_entry);
```

10. Erstellen Sie nun die Indexe für die Partition Keys.

```
CREATE INDEX i1_order_entry_201201 ON order_entry_201201 (entry_date);
CREATE INDEX i1_order_entry_201202 ON order_entry_201202 (entry_date);
CREATE INDEX i1_order_entry_201203 ON order_entry_201203 (entry_date);
CREATE INDEX i1_order_entry_201204 ON order_entry_201204 (entry_date);
CREATE INDEX i1_order_entry_201205 ON order_entry_201205 (entry_date);
CREATE INDEX i1_order_entry_201206 ON order_entry_201206 (entry_date);
```

11. Um einzufügende Sätze in die richtige Partition zu leiten, erstellen wir einen Trigger auf der Master-Tabelle.

```
(postgres@[local]:5432) [postgres] > CREATE OR REPLACE FUNCTION tr_
insert_
order_entry()
> RETURNS TRIGGER AS $$
> BEGIN
> IF ( NEW.logdate >= DATE '2012-01-01' AND NEW.logdate < DATE '2012-02-
01' )
THEN
> INSERT INTO order_entry_201201 VALUES (NEW.*);
> ELIF ( NEW.logdate >= DATE '2012-02-01' AND NEW.logdate < DATE '2012-03-
01' ) THEN
> INSERT INTO measurement_201202 VALUES (NEW.*);
> ELIF ( NEW.logdate >= DATE '2012-03-01' AND NEW.logdate < DATE '2012-04-
01' ) THEN
> INSERT INTO measurement_201203 VALUES (NEW.*);
> ELIF ( NEW.logdate >= DATE '2012-04-01' AND NEW.logdate < DATE '2012-05-
01' ) THEN
> INSERT INTO measurement_201204 VALUES (NEW.*);
> ELIF ( NEW.logdate >= DATE '2012-05-01' AND NEW.logdate < DATE '2012-06-
01' ) THEN
> INSERT INTO measurement_201205 VALUES (NEW.*);
> ELIF ( NEW.logdate >= DATE '2012-06-01' AND NEW.logdate < DATE '2012-07-
01' ) THEN
> INSERT INTO measurement_201206 VALUES (NEW.*);
> ELSE
```



```

> RAISE EXCEPTION 'Unzulaessiges Datum';
> END IF;
> RETURN NULL;
> END;
> $$
> LANGUAGE plpgsql;
CREATE FUNCTION

```

Damit ist die partitionierte Tabelle voll funktionsfähig und kann durch die Applikation verwendet werden. Im Applikations-Code ist es ausreichend, mit der Tabelle „order_entry“ zu arbeiten. Ist der Parameter „constraint_exclusion“ eingeschaltet, sorgt PostgreSQL automatisch dafür, dass SQL-Anweisung nur die relevanten Partitionen verwenden. In Listing 6.30 sehen Sie den Ausführungsplan einer SELECT-Anweisung. PostgreSQL liest ausschließlich die Indexe der Partitionen für Januar und Februar.

Listing 6.30 Ausführungsplan für eine partitionierte Tabelle

```

(postgres@[local]:5432) [postgres] > EXPLAIN
> SELECT COUNT(*) FROM order_entry
> WHERE entry_date < DATE '2012-02-15';

                                QUERY PLAN
-----
Aggregate  (cost=53.21..53.22 rows=1 width=0)
  -> Append  (cost=0.00..50.49 rows=1087 width=0)
        -> Seq Scan on order_entry  (cost=0.00..0.00 rows=1 width=0)
              Filter: (entry_date < '2012-02-15'::date)
        -> Bitmap Heap Scan on order_entry_201201 order_entry
      (cost=8.46..25.25 rows=543 width=0)
            Recheck Cond: (entry_date < '2012-02-15'::date)
          -> Bitmap Index Scan on il_order_entry_201201
        (cost=0.00..8.32 rows=543 width=0)
              Index Cond: (entry_date < '2012-02-15'::date)
        -> Bitmap Heap Scan on order_entry_201202 order_entry
      (cost=8.46..25.25 rows=543 width=0)
            Recheck Cond: (entry_date < '2012-02-15'::date)
          -> Bitmap Index Scan on il_order_entry_201202
        (cost=0.00..8.32 rows=543 width=0)
              Index Cond: (entry_date < '2012-02-15'::date)

```

Auch wenn sich hinter dem Aufsetzen der Partitionen ein gewisser manueller Aufwand verbirgt, ist die Partitionierung eine sehr gute Methode zur effektiven Verwaltung und Performance-Verbesserung für große Tabellen.

■ 6.4 Benchmarks

Mit Hilfe von Benchmarks kann die Performance zwischen verschiedenen Datenbanken, die auf verschiedener Hardware laufen, verglichen werden. Für solche Benchmarks gibt es eine offizielle Stelle, den Transaction Processing Performance Council (TPC), der auf der Webseite <http://www.tpc.org> zu finden ist. Für uns ist an dieser Stelle vor allem interessant, wie sich die Performance von PostgreSQL-Datenbanken im Vergleich zu anderen Datenbank-Betriebssystemen verhält. Da wir in diesem Buch aufgrund des hohen Kompatibilitätsgrades

von PostgreSQL gegenüber Oracle die Themen „Migration nach Oracle“ oder auch „Mischbetrieb von Oracle und PostgreSQL“ an vielen Stellen diskutieren, liefert natürlich ein direkter Performance-Vergleich wichtige Informationen für eine Entscheidungsgrundlage.

Für einen direkten Vergleich der Performance zwischen zwei Datenbank-Systemen liefern Benchmarks unbestreitbare Ergebnisse. Neben einem allgemeinen Vergleich, der häufig durch die Anzahl von Transaktionen pro Zeiteinheit gemessen wird, können Stärken und Schwächen der Systeme aufgedeckt werden. Auch Aussagen für die Skalierbarkeit oder das Verhalten bei Wachstum der Datenbank lassen sich ableiten.

Für relationale Datenbank-Betriebssysteme gibt es hauptsächlich zwei Einsatzgebiete: „Online Transaction Processing (OLTP)“ und „Data Warehousing (DWH)“. Obwohl es in der Praxis sehr häufig zu einer Durchmischung der beiden Typen kommt, ist es sinnvoll, Benchmarks strikt zu trennen. Nicht nur die Datenbank-Schemas unterscheiden sich, auch die optimale Konfiguration der Datenbank ist kontrovers. Der TPC hält ebenfalls zwei verschiedene Benchmarks bereit: „TPCC“ für OLTP und „TPCH“ für Data Warehouse-Datenbanken.

Wir führen in diesem Abschnitt jeweils einen TPCC-ähnlichen Test für eine PostgreSQL und eine Oracle-Datenbank durch, die beide auf derselben Hardware laufen. Die Tests werden als „ähnlich“ beschrieben, da die TPC-Anforderungen sehr strikt und umfangreich sind und den Umfang dieses Abschnitts sprengen würden. Aber auch mit den weniger aufwendigen Benchmarks, die wir hier durchführen, lässt sich ein hinreichend genaues Ergebnis für einen Performance-Vergleich ziehen.

Zum Einsatz kommt das „Benchmark Utility“ (siehe Bild 6.1), das die Simulation von mehreren Tausend Sessions in der Datenbank ermöglicht. Es handelt sich um ein Java-Programm, das für Oracle entwickelt wurde, jedoch für verschiedene Datenbanksysteme eingesetzt werden kann.

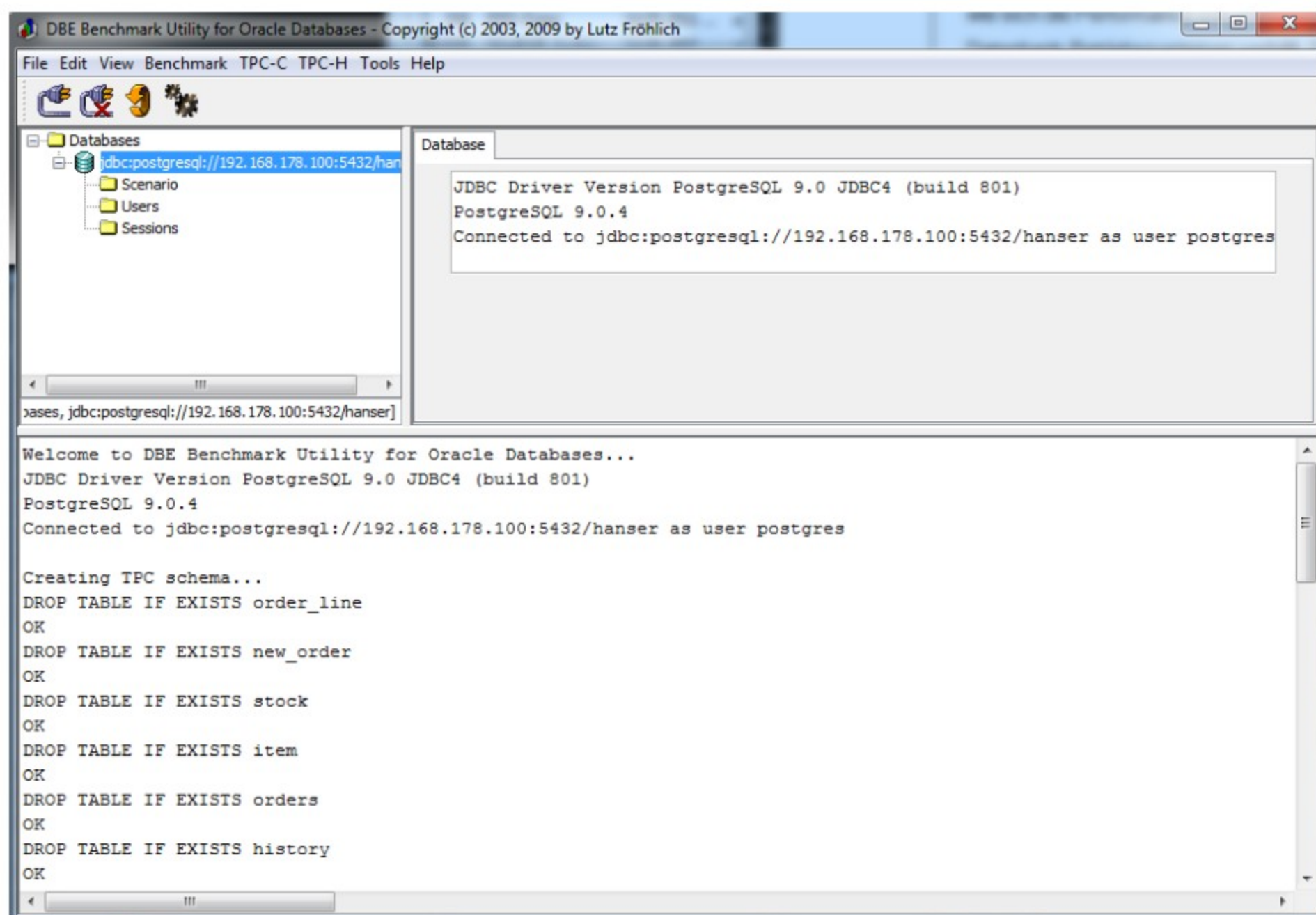


Bild 6.1 Das „Benchmark Utility“

6.4.1 Ein TPCC-Benchmark

Wie bereits erwähnt, erfüllt dieser Benchmark nicht alle Anforderungen des TPC und ist deshalb ein TPCC-ähnlicher Benchmark. Das Schema wird durch das Benchmark-Utility erstellt und gefüllt. Das verwendete Schema finden Sie in Bild 6.2. Es wurde ein Skalierungsfaktor von 100 gewählt, d.h. es befinden sich 100 Sätze in der Tabelle „warehouse“, 1000 Sätze in der Tabelle „district“, 10 Millionen Sätze in der Tabelle „stock“ und so weiter.

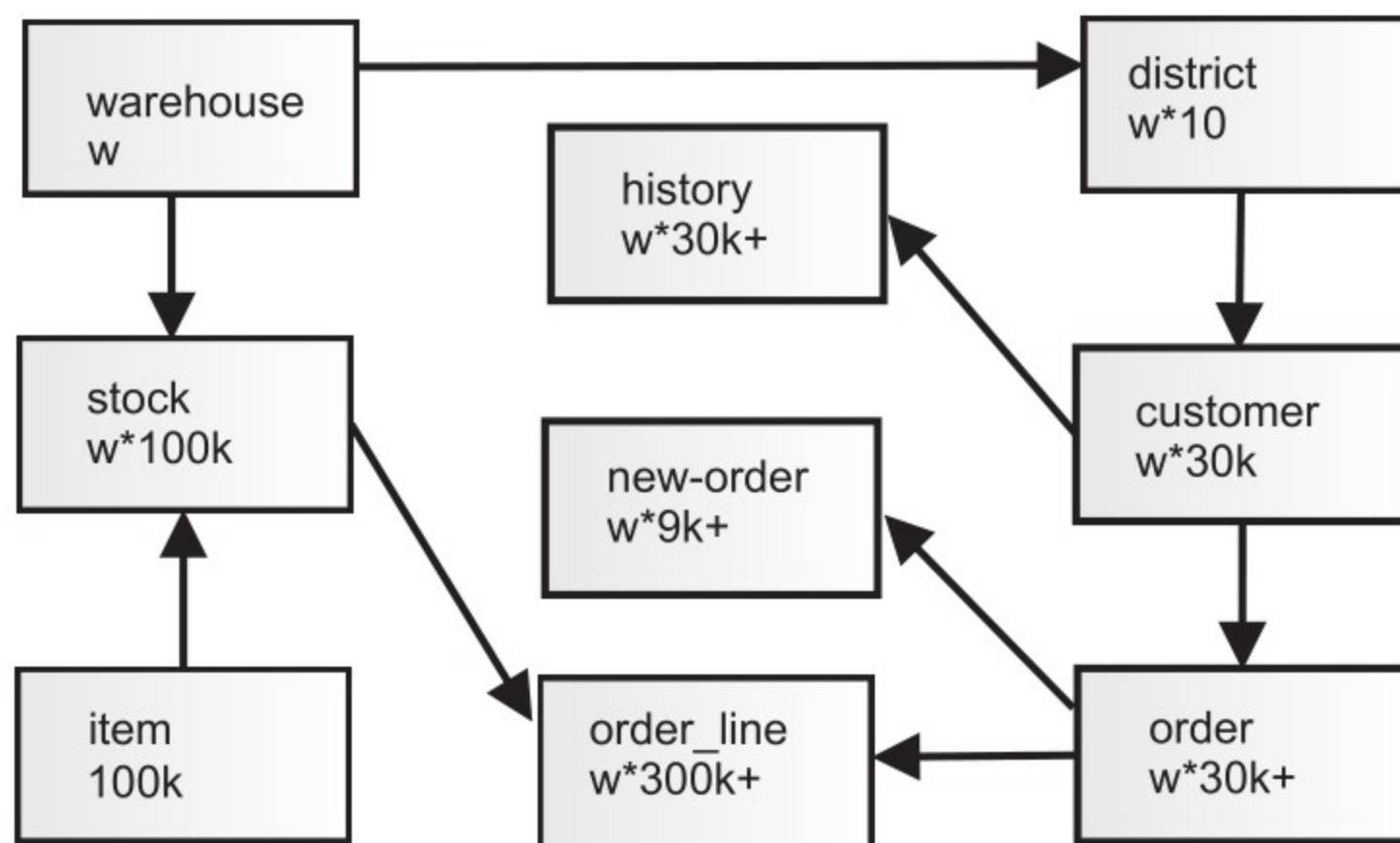


Bild 6.2 Das TPCC-Schema für den Benchmark

Die Größe der Datenbank beträgt nach dem Laden einschließlich Indexe ungefähr 11,2 Gigabyte.

```
(postgres@[local]:5432) [postgres] > SELECT pg_database_size('postgres');
pg_database_size
-----
11.288.986.424
```

Wir vergleichen nun die Ergebnisse der einzelnen Tests mit einem Performance-Test auf einer Oracle-Datenbank. Die Oracle-Datenbank läuft auf derselben Hardware und erhält vergleichsweise dieselben Ressourcen zugewiesen. Wir können also von nahezu identischen Ressourcen für CPU, Memory und I/O ausgehen.

Zunächst untersuchen wir die SELECT-Performance und das Verhalten einer einzelnen Session. Es werden SELECT-Anweisungen über einen Index durchgeführt. Das zugehörige PL/pgSQL-Programm finden Sie in Listing 6.31. Unter Oracle wird ein adäquates PL/SQL-Programm ausgeführt.

Listing 6.31 Scenario 1: Einfache SELECT-Anweisung

```
DO language plpgsql $$
DECLARE
x BIGINT;
v_last VARCHAR(30);
BEGIN
FOR x IN 1..3000000 LOOP
```



```

SELECT c_last INTO v_last FROM customer WHERE c_id = x;
END LOOP;
END
$$;

```

Die Anzahl von Sessions wird so lange erhöht, bis eine Sättigung der vorhandenen Ressourcen und somit die maximale Anzahl von Transaktionen erreicht ist.

Tabelle 6.3 Performance-Test SELECT-Anweisung mit PostgreSQL

Anzahl Sessions	Laufzeit [sec]	Transaktionen pro Minute (tpm)
1	41	4,39 Mio
5	98	9,18 Mio
10	182	9,89 Mio
15	311	8,68

Führen wir nun den analogen Test mit einer Oracle-Datenbank durch. Das zugehörige PL/SQL-Programm finden Sie in Listing 6.32.

Listing 6.32 Einfache SELECT-Anweisung

```

DECLARE
x NUMBER;
v_last VARCHAR2(30);
BEGIN
FOR x IN 1..3000000 LOOP
SELECT c_last INTO v_last FROM customer WHERE c_id = x;
END LOOP;
END;

```

Das Ergebnis des Tests finden Sie in Tabelle 6.4.

Tabelle 6.4 Performance-Test SELECT-Anweisung mit Oracle

Anzahl Sessions	Laufzeit [sec]	Transaktionen pro Minute (tpm)
1	144	1,25 Mio
5	311	2,89
10	672	2,68
15	931	2,90

Der Vergleich beider Tests ist in Bild 6.3 grafisch dargestellt. Für lesende Transaktionen hat PostgreSQL deutlich die Nase vorn. Der Durchsatz verhält sich auch mit wachsender Anzahl von Transaktionen und Benutzern relativ konstant.

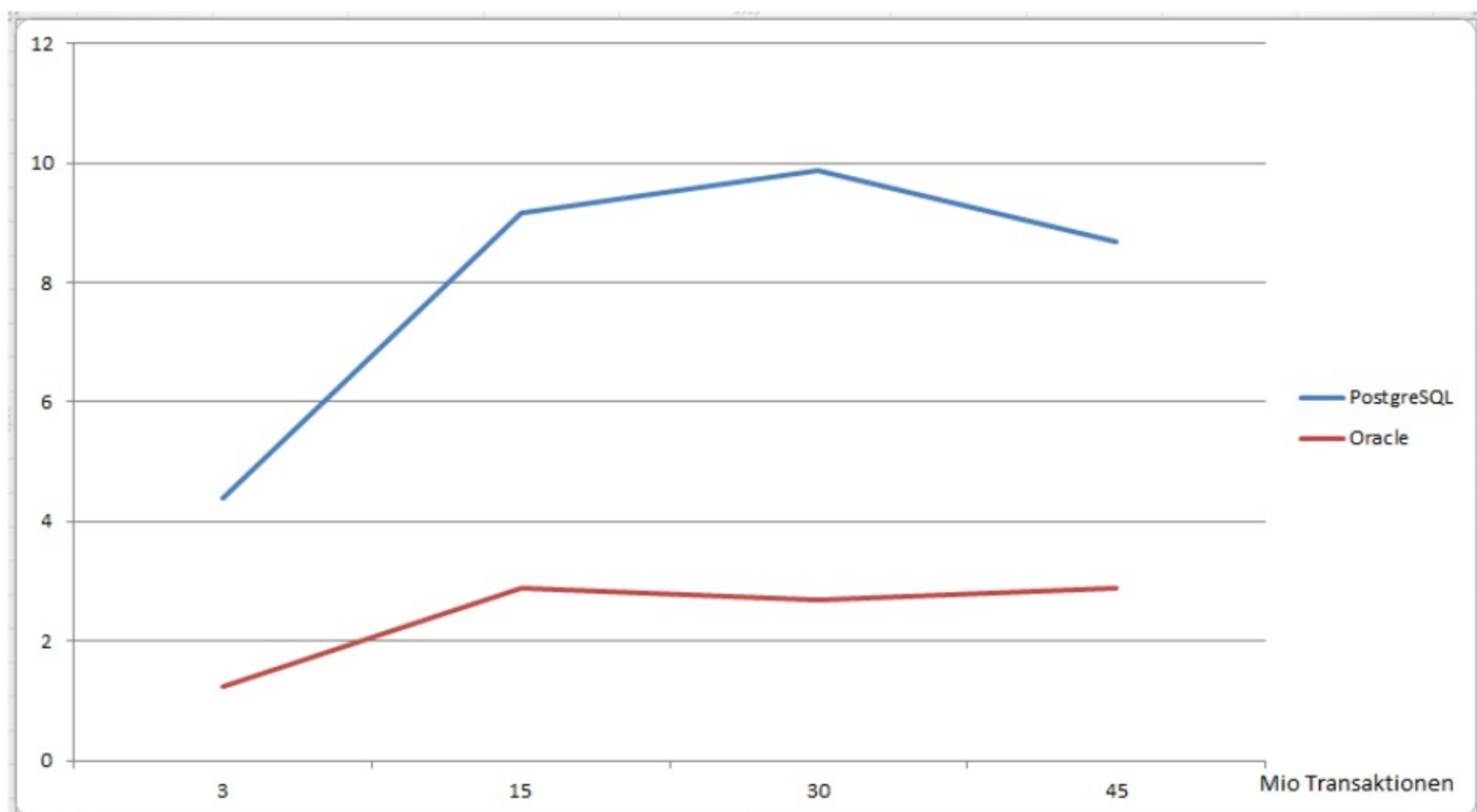


Bild 6.3 Vergleich Performance-Test lesende Transaktionen

Führen wir nun den Vergleichstest für schreibende I/O-intensive Transaktionen durch. Entscheidend für eine optimale Performance ist das Verhalten des Background Writers (Database Writers) und der Verarbeitung der WAL-Records (Redo Log Records), da eine sehr große Anzahl von COMMIT-Anweisungen zu verarbeiten ist.

Listing 6.33 Einfacher UPDATE-Test in PostgreSQL

```
DO language plpgsql $$
DECLARE
x BIGINT;
BEGIN
FOR x IN 2..10000000 LOOP
UPDATE stock SET s_quantity = s_quantity + 0.01 WHERE s_i_id = x;
UPDATE stock SET s_quantity = s_quantity - 0.01 WHERE s_i_id = x - 1;
END LOOP;
EXCEPTION
WHEN no_data_found THEN
RAISE NOTICE 'No data exception caught';
END;
$$;
```

Der Update-Test lieferte die folgenden Ergebnisse:

- PostgreSQL: Laufzeit: 1701 sec, entspricht 0,35 tpm
- Oracle: Laufzeit: 4143 sec, entspricht 0,14 tpm

Diese Ergebnisse sind nicht unerwartet. Es ist bekannt, dass PostgreSQL eine schlankere Datenbank-Engine besitzt und damit bei kleinen Transaktionen bessere Antwortzeiten ausweist.

Einen sehr guten Vergleich der OLTP-Eigenschaften eines Datenbanksystems liefert die „Delivery-Transaktion“. Dabei handelt es sich um gut gemischte OLTP-typische SQL-Anweisungen, die alle typischen Eigenschaften von Datenbanksystem abfordern. Das zugehörige Programm finden Sie in Listing 6.34.

Listing 6.34 Die Delivery-Transaktion

```

DO language plpgsql $$
DECLARE
x BIGINT;
y BIGINT;
z BIGINT;
v_no_o_id BIGINT;
v_o_c_id BIGINT;
v_ol_amount BIGINT;
oid BIGINT;
oid2 BIGINT;
cust BIGINT;
BEGIN
  y := 1;
  oid2 := 1;
  x := 1;
  oid := 1;
  cust := 1;
  FOR z IN 1..10000000 LOOP
    cust := cust + 1;
    IF cust > 3000000 THEN
      cust := 1;
    END IF;
    oid := oid + 1;
    IF oid > 900000 THEN
      oid := 1;
    END IF;
    oid2 := oid2 + 1;
    IF oid2 > 3000000 THEN
      oid2 := 1;
    END IF;
    y := y + 1;
    IF y > 100 THEN
      y := 1;
    END IF;
    x := x + 1;
    IF x > 1000 THEN
      x := 1;
    END IF;
    SELECT no_o_id INTO v_no_o_id FROM new_order
      WHERE no_o_id = oid AND no_w_id = y;
    SELECT o_c_id INTO v_o_c_id FROM orders
      WHERE o_id = oid2 AND o_d_id = x;
    UPDATE orders SET o_carrier_id = 4
      WHERE o_id = oid2 AND o_d_id = x AND o_w_id = y;
    UPDATE order_line SET ol_delivery_d = current_date
      WHERE ol_o_id = z AND ol_d_id = x AND ol_w_id = y;
    SELECT SUM(ol_amount) INTO v_ol_amount FROM order_line
      WHERE ol_o_id = z AND ol_d_id = x AND ol_w_id = y;
    UPDATE customer SET c_balance = c_balance
      WHERE c_id = cust AND c_d_id = x AND c_w_id = y;
  END LOOP;
EXCEPTION
  WHEN no_data_found THEN
    RAISE NOTICE 'No data exception caught';
END;
$$;

```


Der Test mit der Delivery-Transaktion lieferte die folgenden Ergebnisse:

- PostgreSQL: Laufzeit: 3092 sec, entspricht 0,19 tpm
- Oracle: Laufzeit: 3762 sec, entspricht 0,16 tpm

Bei der Delivery-Transaktion liegen Oracle und PostgreSQL wesentlich näher beieinander. Oracle spielt seine Stärken aus, wenn es um komplexere Transaktionen geht oder aber auch im Bereich Multi-User-Fähigkeit.

Auf einen TPC-H-Benchmark wollen wir an dieser Stelle verzichten. Hier kann die Oracle Datenbank mit Features punkten, die in PostgreSQL fehlen. Deshalb wäre der Vergleich aus Performance-Sicht auch nicht fair. Es lässt sich dennoch sagen, dass PostgreSQL auch im Data Warehouse-Umfeld sehr gute Performance-Werte erzielt.

6.4.2 Fazit des Benchmarks

Man kann sagen, dass die allgemeine Performance einer PostgreSQL-Datenbank denen der Datenbanken anderer Hersteller in nichts nachsteht. Allerdings darf man nicht verschweigen, dass in PostgreSQL einige Features fehlen, um beispielsweise in einigen SQL-Abfragen des TPC-H-Tests (ein Test mit einer Data-Warehouse-Datenbank) eine vergleichbare Performance zu anderen Herstellern zu erreichen. Wenn Sie das Ziel verfolgen, ein großes Data Warehouse aufzubauen, sollten Sie in Betracht ziehen, eher auf kommerzielle Produkte wie Oracle, DB2 oder SQL Server zurückzugreifen.

Dennoch lässt sich sagen, dass PostgreSQL auch im Data-Warehouse-Umfeld hinsichtlich der Performance-Werte gut abschneidet. Insbesondere die Möglichkeit der Partitionierung von Tabellen trägt dazu bei. PostgreSQL ist deshalb als Plattform für Reporting-Datenbanken und für DWH-typische Abfragen gut geeignet.

Für einige – spezielle – Typen von Web-Applikationen stehen einer guten Performance die Features der Datenintegrität des PostgreSQL-Servers im Wege. Diese lassen sich nicht ohne Weiteres beseitigen. Hier erreichen Sie wahrscheinlich bessere Performance-Werte mit Datenbanken wie zum Beispiel MySQL, die weniger Restriktionen aufweisen.

Für einen Großteil der kleineren bis mittelgroßen Datenbanken stellt PostgreSQL eine äußerst sinnvolle Alternative dar. Wenn wir von kleinen bis mittelgroßen Datenbanken sprechen, dann sind das Datenbanken, die im Bereich von wenigen bis mehreren Hundert Gigabyte liegen. Hier kann PostgreSQL seine Stärken ausspielen und kommt in den Performance-Werten sehr nahe an Oracle- oder SQL-Server-Datenbanken heran bzw. kann in bestimmten Konstellationen diese sogar übertreffen.

Hot Standby und Streaming Replication

„Standby-Datenbank“ und „Streaming Replication“ sind neue Features der Version 9, die PostgreSQL in der Beurteilung als Datenbank für Enterprise-Lösungen einen entscheidenden Schritt vorangebracht hat. Zusätzlich wurde mit dem Release 9.1 die synchrone Replikation bereitgestellt.

Zum Thema Hochverfügbarkeit von Datenbanken gibt es unterschiedliche Ansprüche und Lösungen. Aktiv-Passiv-Lösungen basieren auf der Spiegelung der zur Datenbank gehörenden Dateien auf Dateisystem- oder Storage-Ebene. So besteht im Open Source-Umfeld die Möglichkeit der Spiegelung auf Block-Device-Ebene mit DRBD, die in vielen Bereichen erfolgreich eingesetzt wird. Das Spiegeln auf Storage Ebene birgt die Gefahr des Spiegeln von Fehlern. Für beide Spiegelungen gilt, dass ein Performance-Impact für die Datenbank vorhanden sein kann. Während viele Standard-Datenbanken in dieser Architektur problemlos laufen, kann es bei Performance-kritischen Datenbanken zu Problemen kommen. Verursacher ist dabei häufig die erhöhte Latenzzeit für das Schreiben auf den Spiegel. Aus diesem Grund sollten die Spiegel des Storage nicht zu weit auseinander liegen.

Auf Trigger basierende Replikationslösungen werden immer seltener eingesetzt, da sie einen direkten Einfluss auf die Performance von Transaktionen haben und sehr schlecht skalieren.

Replikations-Methoden auf der Basis von Transaktionslog-Shipping besitzen die wenigsten Nachteile und bieten gleichzeitig eine hohe Sicherheit. Einerseits verursachen die asynchronen Prozesse den geringsten Impact auf die Performance der Primär-Datenbank, andererseits gestattet die Technologie eine sichere Übertragung auch bei hohem Transaktionsaufkommen. Außerdem kann die Replikation problemlos über größere Entfernungen durchgeführt werden.

Seit der Version 9 ist eine solche Replikation in PostgreSQL integriert und lässt sich durch Konfiguration der Standby-Datenbank aufsetzen. Darüber hinaus ist es möglich, die Standby-Datenbank im Read-only-Modus zu nutzen. So kann zum Beispiel das Reporting auf der Standby-Datenbank durchgeführt und somit die Primär-Datenbank entlastet werden. Dieser Modus wird „Hot Standby“ genannt. Damit bleiben die Ressourcen auf der Standby-Seite nicht ungenutzt.

Mit dem Release 9.1 ist ein weiteres Feature hinzugekommen: die synchrone Replikation. Diese kann durch einfache Konfigurationsänderung aktiviert werden. Dabei gilt eine Transaktion als abgeschlossen, wenn sie auf der Standby-Datenbank erfolgreich angewandt wurde. Diese Architektur liefert ein hohes Maß an Transaktionssicherheit.

Damit bieten sich die folgenden Einsatzmöglichkeiten für Standby-Datenbanken:

- Als Disaster-Recovery(DR)-Lösung mit asynchroner Replikation und ggf. minimalem Datenverlust.
- Als DR-Lösung mit synchroner Replikation ohne Datenverlust. Alle abgeschlossenen Transaktionen befinden sich in der Standby-Datenbank.
- Geplanter Rollentausch zum Patchen oder für Wartungsarbeiten.
- Zusätzliche Nutzung der Standby-Datenbank für Reporting und Abfragen zur Entlastung des Primärsystems (Hot Standby).

■ 7.1 Eine Standby-Datenbank aufsetzen

In diesem Abschnitt liefern wir ein Beispiel zum Aufsetzen einer Standby-Datenbank. Die Replikation erfolgt zwischen zwei Linux-Servern, lässt sich jedoch analog auf andere Betriebssysteme übertragen.

7.1.1 Vorbereitung und Planung

Primär- und Standby-Server sollten so gleichartig wie möglich aufgesetzt werden. Insbesondere müssen die Pfade zu den Tablespaces identisch sein. Obwohl die Hardware nicht identisch sein muss, wird die Administration umso einfacher, je weniger Unterschiede bestehen. Insbesondere müssen folgende Übereinstimmungen vorliegen:

- Die Pfade zu den Tablespaces müssen existieren. Legt man auf der Primärdatenbank einen Tablespace an, wird dies durch die Replikation übertragen und muss auf dem Standby-Server nachvollziehbar sein.
- Wegen der notwendigen Übereinstimmung der Pfade für die Tablespaces ist eine Replikation zwischen Windows- und Unix-Betriebssystemen nicht möglich.
- Die Hardware-Architektur muss identisch sein, ein Mischen von 32-Bit- und 64-Bit-Systemen ist nicht möglich.
- Primär- und Standby-Datenbank müssen dasselbe Major Release besitzen. Da es Policy der Development Group ist, zwischen Minor Releases keine Änderungen in den Disk-Formaten vorzunehmen, besteht keine Gefahr, wenn es Abweichungen beim Minor Release gibt. Wann immer es möglich ist, sollten jedoch die Versionen identisch sein, um Probleme auszuschließen.

Neben der Planung der Server ist es wichtig, hinreichend Netzwerk-Kapazität zur Verfügung zu stellen. Planen Sie dabei nicht nur die durchschnittliche Übertragungsrate, sondern beachten Sie den notwendigen Durchsatz in der Spitze. Kann dieser nicht garantiert werden, kommt es zu einer Übertragungslücke und im Bedarfsfall zu Datenverlust.

7.1.2 Konfiguration und Aktivierung

Die Standby-Datenbank wendet permanent die WAL Records, die sie vom Primärsystem erhält, an und befindet sich damit im Recovery-Modus. Der Standby-Server ist in der Lage, WAL Records aus dem lokalen WAL-Archiv oder direkt vom Primär-Server über die TCP/IP-Verbindung zu lesen. Die letzte Methode wird *Streaming Replication* genannt. Darüber hinaus versucht der Standby-Server natürlich die WAL-Records im lokalen pg_xlog-Verzeichnis zu verarbeiten. Damit ist gewährleistet, dass auch nach einer Verzögerung der Recovery-Prozess an den aktuellen Status herangeführt werden kann.

Bevor wir mit dem Aufsetzen der Replikation beginnen, werfen wir einen Blick auf die Arbeitsweise des Anwendungs-Prozesses auf dem Standby-Server. Nach dem Start des Standby-Servers beginnt PostgreSQL unter Verwendung des Befehls im Parameter „*restore_command*“ alle WAL Records aus dem Archivlog-Verzeichnis zurückzuspeichern. Sind alle WAL-Dateien im Archivlog-Verzeichnis abgearbeitet, versucht PostgreSQL die WAL Records im pg_xlog-Verzeichnis zu verarbeiten. Ist „Streaming Replication“ konfiguriert, dann versucht der Server im nächsten Schritt, Verbindung mit dem Primärserver aufzunehmen und mit der direkten Übertragung zu beginnen. Andernfalls liest er das Archivlog-Verzeichnis wieder aus. Diese Schleife wird durchgeführt, solange der Standby-Server läuft. Es ist ein einfaches, aber wirkungsvolles Prinzip, mit dem in jeder Situation wieder aufgesetzt werden kann.

Voraussetzung für den Betrieb von Standby-Datenbanken ist, dass die WAL-Archivierung konfiguriert ist. Eine detaillierte Beschreibung zur Aktivierung finden Sie in Abschnitt 3.2. Für das Beispiel wiederholen wir diesen Vorgang auf der Primär- und der Standby-Datenbank. Die Schritte können analog für ein Windows-Betriebssystem angewandt werden.

1. Erstellen Sie ein Verzeichnis, das als Speicherort für die Archiv-Dateien dienen soll.

```
# cd /usr/local/pgsql
# mkdir archive
# chown postgres:postgres archive
```

2. Setzen Sie die notwendigen Parameter zur Aktivierung der Archivierung:

```
archive_mode = on
archive_command = 'cp -i %p /usr/local/pgsql/archive/%f </dev/null'
wal_level = archive
```

3. Führen Sie einen Neustart des PostgreSQL-Servers durch.

Auf dem Standby-Server muss nun der Recovery-Befehl verfügbar gemacht werden. Legen Sie dazu eine Datei *recovery.conf* im Verzeichnis *\$PGDATA* an. Diese Datei muss den Parameter *restore_command* enthalten:

```
restore_command = 'cp /usr/local/pgsql/archive/%f %p'
```

Weitere Details zum Restore-Befehl finden Sie in Kapitel 4 („Sicherheit und Wiederherstellung“).

Hilfreich ist in diesem Zusammenhang noch der Parameter *archive_cleanup_command*. Mit seiner Hilfe können ältere, nicht länger vom Standby-Server benötigte archivierte WAL-Dateien automatisch gelöscht werden. Der Parameter „%r“ steht für den Dateinamen mit

dem letzten gültigen Zeitpunkt für einen erfolgreichen Neustart. Der Parameter wird ebenfalls in der Datei „recovery.conf“ gesetzt.

```
archive_cleanup_command = 'pg_archivecleanup /usr/local/pgsql/archive %r'
```



Praxistipp

Wenn Sie planen, die Konfiguration als Disaster-Recovery-Umgebung einzusetzen oder wenn ein Rollentausch zur Durchführung von Wartungsarbeiten durchgeführt werden soll, dann fungiert der Primär-Server auch als Standby-Server und umgekehrt. Führen Sie deshalb die Vorbereitungsmaßnahmen unter diesem Blickwinkel durch. Alle Konfigurationen, die Sie auf dem Standby-Server vornehmen, sollten deshalb auch auf dem Primär-Server erfolgen (und umgekehrt), um einen Rollentausch erfolgreich durchführen zu können.

Schließlich müssen Sie dem Standby-Server noch mitteilen, dass er im Recovery-Modus arbeiten soll. Dies geschieht mit Hilfe des Parameters *standby_mode*. Die Datei „recovery.conf“ würde damit für unser Beispiel wie folgt aussehen:

Listing 7.1 Beispiel für die Konfigurationsdatei recovery.conf

```
standby_mode=on
restore_command = 'cp /usr/local/pgsql/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /usr/local/pgsql/archive %r'
```

Schließlich muss noch das WAL Shipping aktiviert werden. Dies übernimmt der PostgreSQL-Server. Setzen Sie dazu den Parameter „archive_command“ wie folgt:

```
archive_command = 'cp -i %p /usr/local/pgsql/archive/%f </dev/null; scp %p
192.168.178.212:/usr/local/pgsql/archive/%f </dev/null'
```

Damit sind alle Vorbereitungen für den Betrieb der Standby-Datenbank getroffen. Für das initiale Setup muss nun der Standby-Server auf den Stand des Primär-Servers gebracht werden. Die sicherste Methode ist, ein Backup auf den Standby-Server zu übertragen. Führen Sie dazu die folgenden Schritte durch:

1. Versetzen Sie den Primär-Server in den Backup-Modus.

```
(postgres@[local]:5432) [postgres] > SELECT pg_start_backup('standby');
pg_start_backup
-----
0/7000020
```

2. Kopieren Sie den Inhalt des Verzeichnisses \$PGDATA auf den Standby-Server. Sichern Sie vorher auf dem Standby-Server die Konfigurationsdateien „postgresql.conf“ und „recovery.conf“.

```
$ cd $PGDATA
$ tar -cvf prim.tar *
$ scp prim.tar 192.168.178.212:$PGDATA
. . .
tar -xvf prim.tar
```



```
cp postgresql.conf.ori postgresql.conf
$ cp recovery.conf.ori recovery.conf
```

3. Beenden Sie den Backup-Modus auf dem Primär-Server.

```
(postgres@[local]:5432) [postgres] > SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_stop_backup
-----
0/7000094
```

4. Starten Sie den Standby-Server.

```
$ pg_ctl start
pg_ctl: another server might be running; trying to start server anyway
server starting
$ LOG:  database system was interrupted; last known up at 2011-10-17
16:22:35 CEST
LOG:  entering standby mode
LOG:  restored log file "00000001000000000000000007" from archive
LOG:  redo starts at 0/7000070
LOG:  consistent recovery state reached at 0/8000000
cp: cannot stat `/usr/local/pgsql/archive/00000001000000000000000008': No
such file or directory
LOG:  unexpected pageaddr 0/4000000 in log file 0, segment 8, offset 0
```

Damit befindet sich die Standby-Datenbank im permanenten Recovery-Modus. Die archivierten WAL-Dateien werden zum Standby-Server übertragen. Mit dem folgenden Test lässt sich feststellen, ob die Replikation funktioniert.

5. Führen Sie auf der Primär-Datenbank eine DML-Anweisung durch.

```
(postgres@[local]:5432) [postgres] > INSERT INTO test VALUES(1,'FROM PRIMARY');
INSERT 0 1
```

6. Da eine Replikation erst mit der Übertragung der nächsten WAL-Datei stattfindet, forcieren wir an dieser Stelle einen Log Switch.

```
(postgres@[local]:5432) [postgres] > SELECT pg_switch_xlog();
pg_switch_xlog
-----
0/8000110
```

7. Simulieren Sie eine Failover-Situation, und aktivieren Sie die Standby-Datenbank. Starten Sie dazu die Datenbank mit dem Parameter „standby_mode=off“.

```
$ pg_ctl stop
$ pg_ctl start
. . .
$ psql
(postgres@[local]:5432) [postgres] > SELECT * FROM test;
 id |      text
-----+-----
  1 | FROM PRIMARYpostgres=# SELECT * FROM test;
 id |      text
-----+-----
  1 | FROM PRIMARY
```


Damit ist der Nachweis erbracht, dass die Replikation funktioniert. Ein Nachteil dieser Methode ist, dass eine Anwendung der WAL Records auf dem Standby-Server erst nach Übertragung einer archivierten WAL-Datei erfolgt. Damit ist im Havarie-Fall ein Datenverlust denkbar. Dieser Nachteil wird durch das Feature „Streaming Replication“ ausgeglichen.

7.1.3 Streaming Replication einsetzen

Beim Einsatz von „Streaming Replication“ erfolgt die Übertragung der WAL Records nicht erst beim Log Switch, wenn eine neue WAL-Datei geöffnet wird, sondern direkt, mit einer minimalen Verzögerung beim Schreiben des WAL Records. Es handelt sich in der Standard-Konfiguration immer noch um einen asynchronen Prozess, so dass der Performance-Impact auf die Primär-Datenbank sehr klein ist. Andererseits erfolgt bei einer gut geplanten Infrastruktur die Übertragung zum Standby-Server durchschnittlich in weniger als einer Sekunde.

Als Aufsatzpunkt für die Einrichtung der „Streaming Replication“ dient die Konfiguration mit Datei-basierender WAL-Übertragung aus dem vorhergehenden Abschnitt. Für die Verwendung von „Streaming Replication“ ist eine Verbindung vom Standby- zum Primär-Server erforderlich. Die Verbindung sollte über einen so genannten „technischen User“ erfolgen. Das Passwort eines technischen Users darf niemals ablaufen. Für eine funktionierende Verbindung sind zwei Schritte erforderlich:

- Den technischen User im Primär-Server anlegen:

```
(postgres@[local]:5432) [postgres] > CREATE ROLE rep_admin
> LOGIN REPLICATION PASSWORD 'rep1234';
CREATE ROLE
```

- Die Verbindung in der Datei „pg_hba.conf“ gestatten:

#	TYPE	DATABASE	USER	ADDRESS	METHOD
host	replication		rep_admin	192.168.178.212/32	md5

Zusätzlich muss der Parameter *max_wal_senders*, der die maximale Anzahl von gleichzeitigen Standby-Verbindungen festlegt, auf einen Wert größer als „Null“ gesetzt werden. Führen Sie anschließend einen Neustart des Servers durch, damit die Änderungen wirksam werden.

Der Prozess wird aktiviert durch Setzen des Parameters *primary_conninfo* in der Datei „recovery.conf“ auf dem Standby-Server. Im Beispiel sieht die Datei wie folgt aus:

```
standby_mode=on
restore_command = 'cp /usr/local/pgsql/archive/%f %p </dev/null'
archive_cleanup_command = 'pg_archivecleanup /usr/local/pgsql/archive %r'
primary_conninfo = 'host=192.168.178.100 port=5432 user=rep_admin
password=rep1234'
```

Starten Sie jetzt den Standby-Server. Er wird sich zum Primär-Server verbinden, wenn alle archivierten WAL-Dateien abgearbeitet sind und in den Modus „Streaming Replication“ übergehen. Sie sollten als Bestätigung die folgende Meldung im Server-Logfile oder auf dem Bildschirm finden:

```
LOG: streaming replication successfully connected to primary
```


Weiterhin sollte auf dem Primär-Server ein Sender- und auf der Standby-Seite ein Receiver-Prozess laufen:

```
$ ps -ef|grep wal
postgres 6083 5937 0 22:05 ?          00:00:00 postgres: wal sender process
rep_admin 192.168.178.212(24389) streaming 0/16000070
. . .
$ ps -ef|grep wal
postgres 4118 4101 0 22:38 ?          00:00:00 postgres: wal receiver
process   streaming 0/16000070
```

7.1.4 Die Replikation überwachen

Eine einfache, aber zuverlässige Methode für die Überwachung ist die Anzahl von WAL Records, die auf dem Primär-Server generiert und noch nicht auf dem Standby-Server eingearbeitet wurden. Während die Funktion *pg_current_xlog_location* den aktuellen WAL Record anzeigt, erhält man mit der View *pg_stat_replication* den Status des Receiver-Prozesses. Größere Differenzen zwischen dem aktuellen und dem übertragenen WAL Record (Spalte „sent_location“) bedeuten, dass es bei der Übertragung zu Verzögerungen im Streaming Replication-Prozess gekommen ist. Im Beispiel in Listing 7.2 ist die Replikation synchronisiert.

Listing 7.2 Überwachung des Streaming-Replication-Prozesses

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_current_xlog_location();
pg_current_xlog_location
-----
0/16000070
(postgres@[local]:5432) [postgres] > SELECT
application_name,state,sync_state,sent_location,write_location
> FROM pg_stat_replication;
 application_name |   state   | sync_state | sent_location | write_location
-----+-----+-----+-----+-----
 walreceiver      | streaming | async     | 0/16000070    | 0/16000070
```

7.1.5 Synchrone Replikation

Bisher haben wir das Streaming-Replication-Feature als asynchronen Prozess kennengelernt. Dabei sind wir uns bewusst, dass im Falle eines Crashes des Primär-Servers möglicherweise wenige abgeschlossene Transaktionen, die noch nicht auf den Standby-Server übertragen wurden, verloren gehen können. Wir haben dies in Kauf genommen und einen sinnvollen Kompromiss zwischen Ausfallsicherheit und Performance-Impact erzielt.

Falls für eine Datenbank die Ausfallsicherheit eine höhere Priorität als die Performance hat, empfiehlt sich die synchrone Replikation. Dieses Feature steht mit der Version 9.1 zur Verfügung. Die Bestätigung eines COMMIT-Befehls wird erst erfolgreich erteilt, wenn die Transaktion erfolgreich in das Transaktions-Logfile auf dem Primär- und dem Standby-Server geschrieben wurde. Damit ist ein Datenverlust für abgeschlossene Transaktionen bei Ausfall des Primär-Servers ausgeschlossen.

Eingeschaltet wird die synchrone Replikation durch Setzen des Parameters *synchronous_standby_names*. Der Parameter enthält durch Komma getrennte Namen des Applikationsnamens der Standby-Server. Der Applikationsname lässt sich durch folgende SQL-Abfrage herausfinden:

```
(postgres@[local]:5432) [postgres] > SELECT application_name FROM pg_stat_
replication;
 application_name
-----
walreceiver
```

Alternativ ist eine Wildcard zulässig in der Form *synchronous_standby_names*= '*'. Außerdem muss der Parameter *synchronous_commit* auf „on“ gesetzt sein, allerdings ist dies Standard.

Eine Bestätigung, dass die synchrone Replikation eingeschaltet ist, liefert ebenfalls die View „pg_stat_replication“ (siehe Listing 7.3).

Listing 7.3 Den Status der Replikation abfragen

```
(postgres@[local]:5432) [postgres] > SELECT application_name, sync_state
> FROM pg_stat_replication;
 application_name | sync_state
-----+-----
walreceiver      | sync
```

Es stellt sich die Frage, wie groß der Impact auf die Performance ist. In einem Vergleich zwischen synchroner und asynchroner Replikation werden eine Million Sätze in eine Tabelle eingefügt. Die Laufzeit mit asynchroner Replikation war ca. 12 Sekunden, mit synchroner ca. 15 Sekunden. Der Unterschied ist eher klein, insbesondere da der Standby-Server weniger Ressourcen zur Verfügung hat. Allerdings stehen die Server nebeneinander, so dass die Netzwerk-Roundtrip Zeiten sehr klein sind. Im Falle einer Replikation über größere Entfernungen werden die Unterschiede größer sein. Störfaktor ist dabei weniger die Durchsatzrate, sondern die Latenzzeit. Die kleinstmögliche Verzögerung pro Transaktion ist die Zeit für einen Netzwerk-Roundtrip.

■ 7.2 Eine Hot-Standby-Datenbank betreiben

Eine Hot-Standby-Datenbank verfügt auch über die Fähigkeit, SQL-Abfragen ausführen zu können, während die Datenbank im Recovery-Modus läuft. Damit ist also ein paralleler Read-only-Betrieb möglich. Die Nutzungsmöglichkeiten sind zahlreich. Häufig werden solche Datenbanken für das Reporting und Ad-hoc-Abfragen eingesetzt und entlasten somit den Primär-Server. Transaktionen, die auf der Hot-Standby-Datenbank gestartet werden, erhalten aufgrund des Read-only-Modus keine Transaktions-ID. Demzufolge können nur Transaktionen ausgeführt werden, die keine Veränderungen vornehmen.



Hinweis

Bei der Datenübertragung zwischen Primär- und Standby-Datenbank kann es zu Verzögerungen kommen. Aus diesem Grund können Abfragen, die auf beiden Systemen gleichzeitig abgesetzt werden, unterschiedliche Ergebnisse liefern. Deshalb sollte man davon ausgehen, dass der Datenbestand auf dem Standby-Server nur als nahezu identisch zu betrachten ist. Die ist bei der Planung des Einsatzes der Standby-Datenbank als Quelle für das Reporting zu berücksichtigen. In vielen Fällen ist jedoch diese kurzfristige Differenz unerheblich.

Der Hot-Standby-Modus wird eingeschaltet, wenn der Parameter *hot_standby* auf den Wert „on“ gesetzt wird und eine Datei „recovery.conf“ existiert. Darüber hinaus muss der Parameter *wal_level* in der Primär-Datenbank auf den Wert „hot_standby“ gesetzt werden. Nach Aktivierung der Hot-Standby-Datenbank finden Sie im Logfile die folgende Meldung:

```
LOG: database system is ready to accept read only connections
LOG: standby "walreceiver" is now the synchronous standby with priority 1
```

Auf der Host Standby-Datenbank ist der Parameter *transaction_read_only* stets auf „on“ gesetzt und lässt sich nicht verändern:

```
(postgres@[local]:5432) [postgres] > show transaction_read_only;
transaction_read_only
-----
on
```

Schauen wir uns an dieser Stelle weitere Parameter an, die Einfluss auf die Konfiguration einer Hot-Standby-Umgebung haben. In der Primär-Datenbank sind folgende Parameter einzustellen:

- *wal_level*: Dieser Parameter legt fest, wie viele Informationen in die WAL-Dateien geschrieben werden sollen. Bei „minimal“ werden gerade so viele Daten geschrieben, wie notwendig sind, um das Crash-Recovery der Primärdatenbank zu gewährleisten. Um eine Hot-Standby-Datenbank betreiben zu können, muss der Parameter auf den Wert „hot_standby“ gesetzt werden.
- *vacuum_defer_cleanup_age*: Der Parameter legt die Anzahl von Transaktionen fest, um die die Beseitigung von Dead-Row-Versionen verzögert wird. Eine Vergrößerung des Wertes ermöglicht dem Standby-Server, mehr SQL-Abfragen zu bedienen, ohne dass es zu Konflikten mit zu früh entfernten Row-Versionen kommt. Der Standardwert ist „0“ und sollte beim Betrieb einer Hot-Standby-Umgebung in Abhängigkeit vom Transaktions-Aufkommen nach oben gesetzt werden.

Auf dem Standby-Server haben folgende Parameter Einfluss auf den Betrieb:

- *max_standby_archive_delay*: Ist die Hot-Standby-Datenbank eingeschaltet, legt dieser Parameter fest, wie lange der Standby-Server warten soll, bis er die Abfrage wegen eines Konflikts (About-to-be-applied) abbricht. Dieser Parameter bezieht sich auf eine Situation, wenn der Standby-Server aus einer archivierten WAL-Datei liest.
- *max_standby_streaming_delay*: Dieser Parameter besitzt eine analoge Funktion wie der Parameter „max_standby_archive_delay“, in einer Situation, in der die Standby-Datenbank WAL-Daten über „Streaming Replication“ empfängt.

**Praxistipp**

Beachten Sie beim Festlegen der Parameter, dass die Datenbank-Server beide Rollen, Primary und Standby übernehmen können.

In Zusammenhang mit SQL-Abfrage auf einer Hot-Standby-Datenbank kann es zu Konflikten kommen. Ein typisches Beispiel ist das Absetzen eines DROP TABLE-Befehls auf der Primär-Datenbank, während eine SELECT-Anweisung auf dem Standby-Server läuft. Die Abfrage auf dem Standby-Server könnte nicht zu Ende laufen, wenn der DROP TABLE-Befehl abgesetzt würde. Da der Primär-Server keine Informationen darüber hat, welche SQL-Abfragen auf dem Standby-Server laufen, wird er den DROP TABLE-Befehl direkt ausführen. Andererseits würde der Standby-Server in seinem APPLY-Prozess zurückfallen, wenn er auf das Ende der SELECT-Anweisung warten würde.

Um einen Kompromiss herbeizuführen, können Sie die beiden soeben erwähnten Parameter „max_standby_streaming_delay“ und „max_standby_archive_delay“ setzen.

Eine Zusammenfassung von aufgetretenen Konflikten liefert die Tabelle *pg_stat_database_conflicts*.

Listing 7.4 Konflikte auf der Hot Standby-Datenbank abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_stat_database_conflicts;
 datid | datname | confl_tablespace | confl_lock | confl_snapshot | confl_
bufferpin | confl_deadlock
-----+-----+-----+-----+-----+-----
      1 | template1 |                  |            |                | 
0 |                  0
 12690 | template0 |                  |            |                | 
0 |                  0
 12698 | postgres  |                  |            |                | 
0 |                  0
```

Da wir nun in der Lage sind, Abfragen an den Standby-Server zu schicken, ist ein direkter Vergleich der angewandten WAL Records möglich. Es lässt sich also feststellen, ob eine Anwendungslücke im Recovery-Prozess vorliegt (siehe Listing 7.5).

Listing 7.5 Den Status von Primär- und Standby-Server vergleichen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_current_xlog_location();
pg_current_xlog_location
-----
0/500000A0
postgres=# SELECT * FROM pg_last_xlog_receive_location();
pg_last_xlog_receive_location
-----
0/500000A0
```


■ 7.3 Failover und Switchover

Während „Switchover“ einen geplanten Rollentausch zwischen Primär- und Standby-Datenbank bezeichnet, ist ein „Failover“ die Übernahme der Primär-Rolle durch den bisherigen Standby-Server bei Ausfall des Primärservers.

PostgreSQL stellt keine Standard-Software für die Verwaltung von Switchover und Failover zur Verfügung. Wird die Standby-Datenbank für Disaster-Recovery-Zwecke eingesetzt, dann ist das Failover in der Regel ein manueller Prozess. Nach dem Crash des Primär-Servers wird die Standby-Datenbank als Primär-Datenbank aktiviert, und der Client verweist auf den neuen Server durch Änderung des Servernamens oder der IP-Adresse. Führen Sie die folgenden Schritte für ein manuelles Failover aus:

1. Führen Sie einen INSERT-Befehl durch, um die Replikation zu überprüfen.

```
(postgres@192.168.178.100:5432) [postgres] > INSERT INTO test VALUES
(1000001,'Before Crash');
INSERT 0 1
```

2. Simulieren Sie einen Crash des Primär-Servers mit dem folgenden Kommando:

```
$ pg_ctl stop -m immediate
$ LOG: received immediate shutdown request
```

3. Die Verbindung vom Client zum Server ist abgebrochen.

```
(postgres@192.168.178.100:5432) [postgres] > SELECT * FROM test WHERE id =
1000001;
WARNING: terminating connection because of crash of another server process
```

4. Aktivieren Sie den bisherigen Standby-Server.

```
$ pg_ctl promote
LOG: received promote request
LOG: redo done at 0/5700130C
LOG: last completed transaction was at log time 2011-11-12 15:28:32.53834+01
LOG: archive recovery complete
. . .
LOG: autovacuum launcher started
LOG: database system is ready to accept connections
```

5. Verbinden Sie sich mit dem Client zum soeben aktivierten Server. Überprüfen Sie, ob die letzte Transaktion gespeichert ist.

```
C:\Users\Lutz>psql -h 192.168.178.212 -U postgres -W
(postgres@192.168.178.212:5432) [postgres] > SELECT * FROM test WHERE id =
1000001;
   id   |      text
-----+-----
1000001 | Before Crash
```

Damit ist das Failover abgeschlossen, und der Standby-Server hat die Funktion des Primär-Servers übernommen.

Der Prozess lässt sich vereinfachen, indem man einen sogenannten „Logischen Host“ definiert und dabei eine virtuelle IP-Adresse verwendet, die mit dem Netzwerk-Interface verbunden wird. Diese legt man mit dem Failover auf das Interface des bisherigen Standby-Servers. Damit muss die Konfiguration des Clients nicht geändert werden, es genügt, die Verbindung wiederherzustellen. Die folgenden Schritte liefern ein Beispiel für ein Failover mit einem Logischen Host.

6. Konfigurieren Sie den Primärserver für den Betrieb mit einem logischen Hostnamen (Virtuelle IP-Adresse). Legen Sie die virtuelle IP-Adresse auf das Netzwerk-Interface und starten Sie den PostgreSQL-Server neu, wobei der Parameter `listen_addresses` auf die virtuelle IP-Adresse verweist. Die Änderungen des Netzwerk-Interfaces müssen unter dem Benutzer „root“ ausgeführt werden.

```
# ifconfig eth0:0 192.168.178.222
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:24:81:B2:F9:73
          inet addr:192.168.178.100  Bcast:192.168.178.255
          . . .
eth0:0    Link encap:Ethernet  HWaddr 00:24:81:B2:F9:73
          inet addr:192.168.178.222  Bcast:192.168.178.255
# su - postgres
$ cd $PGDATA
$ vi postgresql.conf → listen_addresses = '192.168.178.222'
```

7. Führen Sie einen Neustart des Servers durch.

```
$ pg_ctl restart
```

8. Verbinden Sie sich von einem Client unter Verwendung der virtuellen IP-Adresse, und fügen Sie einen Testsatz ein.

```
C:\Users\Lutz>psql -h 192.168.178.222 -U postgres -W
(postgres@192.168.178.222:5432) [postgres] > INSERT INTO test
VALUES(2000000,'Before Crash 2');
INSERT 0 1
```

9. Simulieren Sie den Crash des Datenbank-Servers.

```
$ pg_ctl stop -m immediate
```

10. Entfernen Sie die virtuelle IP-Adresse vom Primär-Server.

```
# ifconfig eth0:0 down
```

11. Legen Sie die virtuelle IP-Adresse auf das Netzwerk-Interface des Standby-Servers.

```
# ifconfig eth0:0 192.168.178.222
```

12. Aktivieren Sie den Standby-Server, und starten Sie diesen mit der virtuellen IP-Adresse.

```
$ pg_ctl promote
$ pg_ctl restart
```

13. Verbinden Sie den Client mit der unveränderten Konfiguration.

```
(postgres@192.168.178.222:5432) [postgres] > \c
You are now connected to database "postgres" as user "postgres".
(postgres@192.168.178.222:5432) [postgres] > SELECT * FROM test WHERE id =
2000000;
```


id	text
2000000	Before Crash 2

Der Vorteil dieser Methode besteht darin dass die Konfiguration der Clients nicht geändert werden muss. In der Regel liegen die Zeiten für das Failover bei wenigen Sekunden. Dieses Verfahren kann in Skripten hinterlegt werden. Es genügt also ein einfaches „Re-Connect“, um die Verbindung zur Datenbank wiederherzustellen.

Ein Switchover ist ein geplanter Rollentausch zwischen Primär- und Standby-Server. Damit kann man die Verfügbarkeit des Datenbank-Servers im Falle von Wartungsarbeiten am Betriebssystem oder der Hardware erhöhen. Ein einfaches Switchover wird von PostgreSQL noch nicht unterstützt. Dennoch lässt sich die Konfiguration so aufsetzen, dass ein Rollentausch mit wenig Konfigurationsaufwand möglich ist.

Die Idee basiert auch hier auf dem Prinzip des logischen Hosts (virtuelle IP-Adressen), das Sie vom Failover bereits kennen. Allerdings verwendet man hier zwei virtuelle IP-Adressen – eine für den Primär-Server und eine für den Standby-Server. Mit dem Rollentausch werden die virtuellen IP-Adressen sowie die Konfigurationsdateien „postgresql.conf“ zwischen beiden Servern getauscht. Die Vorgehensweise ist folgende:

1. Herunterfahren der Primär-Datenbank (Option „fast“).
2. Austausch der virtuellen IP-Adressen und Konfigurationsdateien.
3. Neustart und Aktivierung des bisherigen Standby-Servers.
4. Neuaufbau des bisherigen Primär-Servers als Standby-Server.

8

Entwicklung von Applikationen für PostgreSQL

Wie fast alle Datenbanken bietet auch PostgreSQL die Möglichkeit, Programme aller verbreiteten Programmier- und Skriptsprachen anzubinden. Dazu gehören C/C++, Java, C#, Perl, Tcl und Python sowie alle Programmiersprachen, die eine ODBC-Schnittstelle bedienen können. Nicht vergessen sollte man die eigene Sprache PL/pgSQL, die Server-seitig eingesetzt werden kann.

PostgreSQL unterscheidet zwischen Client- und Serverprogrammierung. Client-Programme können auf Basis der PostgreSQL-eigenen Library „libpq“ oder über Standardschnittstellen JDBC und ODBC mit dem PostgreSQL-Server kommunizieren. Server-Programme lassen sich direkt integrieren. PostgreSQL bietet die Option, prozedurale Programme verschiedener Sprachen in der Datenbank zu speichern und auszuführen.

■ 8.1 Client-Programmierung

PostgreSQL wurde nach dem Client/Server/Modell entwickelt. Client-Programme senden ihre Anforderungen über das Netzwerk zum Datenbankserver und erhalten ein Ergebnis zurück. Neben den Standard-Schnittstellen JDBC oder ODBC ist der Zugriff über die Bibliothek „libpq“ möglich. Dabei handelt es sich um eine native Verbindung zum PostgreSQL-Server.

8.1.1 Die PostgreSQL Frontend Library „libpq“

Die Bibliothek „libpq“ ist Bestandteil der Standard-Distribution und befindet sich in jeder PostgreSQL-Umgebung. Programme, die über „libpq“ auf PostgreSQL-Datenbanken zugreifen, müssen in „C“ oder „C++“ geschrieben werden. Dabei muss die Kopfdatei „libpq-fe.h“ (fe = Frontend) eingebunden werden. Die Bibliothek selbst befindet sich im Verzeichnis `/usr/local/pgsql/lib` und muss beim Linken mit eingebunden werden. Die Bibliothek enthält vorgefertigte Funktionen, die Sie im Anwenderprogramm benutzen können. Betrachten Sie das Beispiel in Listing 8.1, das eine Verbindung zu einer PostgreSQL-Datenbank herstellt.

Listing 8.1 C-Programm „pgconn.c“ für Verbindungsaufbau zur Datenbank

```
#include <stdlib.h>
#include <libpq-fe.h>
int main() {
    PGconn *myconn = PQconnectdb("");
    if (PQstatus(myconn) == CONNECTION_OK)
        printf("Verbindung hergestellt\n");
    else
        printf("Verbindung fehlgeschlagen\n");
    PQfinish(myconn);
    return EXIT_SUCCESS;
}
```

An die Funktion `PQconnectdb` werden keine Parameter übergeben. In diesem Fall werden die Parameter aus den Umgebungsvariablen des Betriebssystems genommen.

Tabelle 8.1 Verbindungsoptionen für die Funktion „`PQconnectdb`“

Option	Umgebungsvariable	Bedeutung
dbname	\$PGDATABASE	Name der Datenbank
host	\$PGHOST	Hostname des Cluster (Standard: localhost)
user	\$PGUSER	Benutzername
password	\$PGPASSWORD	Passwort
port	\$PGPORT	TCP/IP Port (Standard: 5432)

Im Folgenden führen wir das Übersetzen und das Linken des C-Programms in einer Unix-/Linux-Umgebung mit dem GCC-Compiler durch. Die Syntax finden Sie in Listing 8.2. Die Option „-I“ setzt den Pfad zu den Header-Dateien und „-L“ zu den Bibliotheken. Für das Programm benötigt man die Header-Datei „`libpq-fe.h`“ sowie die Bibliothek „`libpq.so`“.

Listing 8.2 Ein C-Programm mit „libpq“ übersetzen und linkern

```
$ gcc -o pgconn -I/usr/local/pgsql/include pgconn.c -L/usr/local/pgsql/lib -lpq
$ ./pgconn
Verbindung hergestellt
```

**Hinweis**

Sollten Sie beim Ausführen des Programms den folgenden Fehler erhalten, findet das Programm die Bibliothek „libpq“ nicht.

```
./pgconn: error while loading shared libraries: libpq.so.5: cannot
open shared object file: No such file or directory
```

Nehmen Sie den Pfad mit der Bibliothek in die Umgebungsvariable auf:

```
$ export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
```


C-Programme werden üblicherweise mit Hilfe einer Make-Datei übersetzt und gelinkt. Darin finden sich die Parameter aus dem gcc-Befehl wieder. Der Dateiname ist „makefile“, der Aufruf erfolgt mit dem Kommando „make“.

Listing 8.3 Eine Make-Datei für die Übersetzung verwenden

```
CC=gcc
INC=/usr/local/pgsql/include
LIB=/usr/local/pgsql/lib
CFLAGS=-I$(INC)
LDLIBS=-L$(LIB) -lpq
ALL = pgconn
all: $(ALL)
clean: @rm -f *.o *~ $(ALL)
```

Kann die Verbindung zur Datenbank nicht hergestellt werden, gibt uns das Programm nur eine allgemeine Fehlermeldung zurück. Selbstverständlich gibt die Bibliothek Fehlerinformationen zurück, die wir weitergeben können. Fügen Sie die Zeile aus Listing 4 ein.

Listing 8.4 Fehlerauswertung für Verbindungsprobleme einfügen

```
. . .
printf("Fehler: %s\n", PQerrorMessage(myconn));
. . .
$ make
gcc -I/usr/local/pgsql/include pgconn.c -L/usr/local/pgsql/lib -lpq -o
pgconn
$ ./pgconn
Verbindung fehlgeschlagen
Fehler: FATAL: database "hanser3" does not exist
```

Um die Programme auf einem Windows Client laufen zu lassen, wird die DLL-Datei „libpq.dll“ benötigt.



Praxistipp

Soll der Windows-Rechner nur als Datenbank-Client für libpq-Programme dienen, ist es nicht notwendig, die gesamte PostgreSQL-Software zu installieren. Es genügt, einen sogenannten „Thin Client“ zu konfigurieren. Die folgende Anweisung beschreibt, wie ein Client installiert wird.

1. Erzeugen Sie den PostgreSQL-Client aus dem Quellcode. Wechseln Sie in der Quellcode-Distribution in das Verzeichnis „..\postgresql-9.1.1\src“, und führen Sie den Make-Befehl aus.

Listing 8.5 Die Datei „libpq.dll“ für Windows-Clients erstellen

```
C:\Users\Lutz\Downloads\postgresql-9.1.1\src>nmake /f win32.mak
Microsoft (R) Program Maintenance Utility, Version 10.00.30319.01
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.
    cd include
    if not exist pg_config.h copy pg_config.h.win32 pg_config.h
    if not exist pg_config_os.h copy port\win32.h pg_config_os.h
```



```

cd ..
cd interfaces\libpq
nmake /f win32.mak
Microsoft (R) Program Maintenance Utility, Version 10.00.30319.01
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.
Building the Win32 static library...
Using default OpenSSL Include directory: C:\OpenSSL\include
Using default OpenSSL Library directory: C:\OpenSSL\lib\VC
Using default Kerberos Include directory: C:\kfw-2.6.5\inc
Using default Kerberos Library directory: C:\kfw-2.6.5\lib\i386
cd ..\..
echo All Win32 parts have been built!
All Win32 parts have been built!

```

2. Die Datei „libpq.dll“ wurde im Verzeichnis „\postgresql-9.1.1\src\interfaces\libpq\Release“ erstellt. Kopieren Sie diese nach „C:\Windows\System32“. Damit wird sie von jeder Applikation gefunden. Sie funktioniert analog zur Shared Library unter Unix/Linux.

Das C-Programm aus Listing 8.16 können Sie mit einem beliebigen Windows-Compiler übersetzen und linken. Wir verwenden für die folgenden Programme den Microsoft Visual C++ Compiler. Zum Linken muss die Bibliothek „libpqdll.lib“ eingebunden werden.



Hinweis

Die Microsoft Visual C++ Express Edition können Sie kostenlos von der Microsoft-Webseite herunterladen und nutzen.

Listing 8.6 Ein C-Programm für „libpq“ unter Windows kompilieren und linken

```

C:\Users\Lutz> cl.exe /nologo /I C:\PostgreSQL\9.1\include
C:\Users\Lutz\Downloads\postgresql-9.1.1
\src\interfaces\libpq\Release\libpqdll.lib pgconn.c
C:\Users\Lutz >set PGHOST=192.168.178.100
C:\Users\Lutz >set PGUSER=postgres
C:\Users\Lutz >pgconn.exe
Verbindung hergestellt

```

Für das Ausführen von SQL-Anweisungen, gleich welcher Art, stellt „libpq“ die Funktion *PQexec* zur Verfügung. Mit „PQexec“ können sowohl SELECT-Anweisungen als auch DML- oder DDL-Befehle ausgeführt werden. Der Rückgabewert ist vom Datentyp „PGresult“. Auch hier ist es notwendig, die Strukturen wieder freizugeben. Dies erfolgt mit der Funktion *PQclear*.



Hinweis

Beachten Sie, dass die Ergebnisse der SQL-Ausführung selbst dann in „PGresult“ erhalten bleiben, wenn die Verbindung zur Datenbank beendet wurde. Die Struktur wird erst mit „PQclear“ aufgelöst.

SQL-Anweisungen werden nach dem Client/Server-Prinzip gestellt. Die Anweisung wird an den Server übergeben. Der Server verarbeitet sie und liefert ein Ergebnis zurück. Das Ergebnis kann das Resultset einer SELECT-Anweisung, das „OK“ für die erfolgreiche Ausführung einer DDL- oder DML-Anweisung oder aber ein Fehler sein. Um dies zu unterscheiden, kann der Status der Ausführung abgefragt werden. Die Funktion *PQresultStatus* liefert diesen Status. Die möglichen Werte finden Sie in Tabelle 8.2.

Tabelle 8.2 Rückgabewerte der Funktion „PQresultStatus“

Status	Bedeutung
PGRES_COMMAND_OK	Die SQL-Anweisung wurde erfolgreich ausgeführt, liefert jedoch keine Daten zurück (zum Beispiel bei einer DDL-Anweisung).
PGRES_TUPLES_OK	Die SQL-Anweisung wurde erfolgreich ausgeführt und liefert Daten zurück (zum Beispiel SELECT-Anweisung). Es kann sich dabei um null, eine oder mehrere Zeilen handeln.
PGRES_FATAL_ERROR, PGRES_NONFATAL_ERROR	Bei der Ausführung der SQL-Anweisung ist ein Fehler aufgetreten.
PGRES_BAD_RESPONSE	Die Antwort vom Server wurde nicht verstanden. Es ist ein Fehler aufgetreten, den der Client nicht verarbeiten kann.
PGRES_EMPTY_QUERY	Es ist kein Zugriff auf die Datenbank erforderlich (zum Beispiel: leere SQL-Anweisung).

Das Programm in Listing 8.7 führt eine beliebige SQL-Anweisung, die als Parameter übergeben wird, aus und wertet das Ergebnis aus.

Listing 8.7 Auswertung des Ergebnisses einer SQL-Anweisung

```
#include <stdlib.h>
#include <libpq-fe.h>
int main(int argc, char *argv[]) {
    printf("SQL-Anweisung: %s\n", argv[1]);
    PGresult *res;
    PGconn *myconn = PQconnectdb("");
    if (PQstatus(myconn) == CONNECTION_OK)
        printf("Verbindung hergestellt\n");
    else {
        printf("Verbindung fehlgeschlagen\n");
        printf("Fehler: %s\n", PQerrorMessage(myconn));
    }
    res = PQexec(myconn, argv[1]);
    switch(PQresultStatus(res)) {
        case PGRES_COMMAND_OK:
            printf("Anweisung erfolgreich ausgeführt\n");
            break;
        case PGRES_TUPLES_OK:
            printf("Anweisung mit Resultset\n");
            break;
        case PGRES_EMPTY_QUERY:
            printf("Leere Anweisung\n");
            break;
        case PGRES_FATAL_ERROR:
```



```

        printf("Fehler\n");
        break;
    }
    PQfinish(myconn);
    return EXIT_SUCCESS;
}
$ ./pgstat
SQL-Anweisung: (null)
Verbindung hergestellt
Fehler
$ ./pgstat ""
SQL-Anweisung:
Verbindung hergestellt
Leere Anweisung
$ ./pgstat "SELECT * FROM test"
SQL-Anweisung: SELECT * FROM test
Verbindung hergestellt
Anweisung mit Resultset
$ ./pgstat "CREATE TABLE hanser(id INT, text VARCHAR(100))"
SQL-Anweisung: CREATE TABLE hanser(id INT, text VARCHAR(100))
Verbindung hergestellt
Anweisung erfolgreich ausgeführt

```

Die Ausführung einer SELECT-Anweisung erfolgt, wie bereits erwähnt, ebenfalls mit der Funktion „PQexec“. Das Resultset enthält dann alle Informationen über die zurückgelieferten Daten. In der Regel weiß man nicht, wie viele Zeilen und Spalten von der Ausführung der Anweisung zurückgeliefert werden. Diese Informationen werden vom Programm für die Aufbereitung des Ergebnisses benötigt und lassen sich mit Hilfe weiterer Funktionen der Bibliothek „libpq“ ermitteln. In Tabelle 8.3 finden Sie eine Beschreibung.

Tabelle 8.3 „libpq“-Funktionen zur Auswertung eines Resultsets

Funktion	Beschreibung
<code>int PQntuples(const PGresult *result);</code>	Anzahl der im Resultset zurückgegebenen Zeilen ermitteln.
<code>int PQnfields(const PGresult *result);</code>	Anzahl der Spalten des Resultsets ermitteln.
<code>char *PQfname(const PGresult *result, int index);</code>	Den Namen einer Spalte abfragen. Der Spaltenindex beginnt bei „0“.
<code>int PQfsize(const PGresult *result, int index);</code>	Die Größe einer Spalte im Resultset erfragen.
<code>int PQfnumber(const PGresult *result, const char *fieldname);</code>	Den Spaltenindex eines Spaltennamens im Resultset ermitteln.

Das Programm in Listing 8 verwendet diese Funktionen und liefert eine Ausgabe von Struktur und Daten des Resultsets.

Listing 8.8 Aufbereitung des Resultsets

```

#include <stdlib.h>
#include <libpq-fe.h>
int main(int argc, char *argv[]) {

```



```

PGresult *res;
int i=0;
int j=0;
int rows=0;
int cols=0;
PGconn *myconn = PQconnectdb("");
if (PQstatus(myconn) == CONNECTION_OK)
    printf("Verbindung hergestellt\n");
else {
    printf("Verbindung fehlgeschlagen\n");
    printf("Fehler: %s\n", PQerrorMessage(myconn));
}
res = PQexec(myconn, argv[1]);
switch(PQresultStatus(res)) {
    case PGRES_TUPLES_OK:
        rows = PQntuples(res);
        cols = PQnfields(res);
        printf("Anzahl Zeilen: %d\n", rows);
        printf("Anzahl Spalten: %d\n", cols);
        printf("Tabellenstruktur:\n");
        for(i = 0; i < cols; i++) {
            printf("%s:%d\n", PQfname(res, i), PQfsize(res, i));
        }
        printf("Daten:\n");
        for (i = 0; i < rows; i++) {
            for (j = 0; j < cols; j++)
                printf("%s,", PQgetvalue(res, i, j));
            printf("\n");
        }
        break;
    case PGRES_EMPTY_QUERY:
        printf("Leere Anweisung\n");
        break;
    case PGRES_FATAL_ERROR:
        printf("Fehler\n");
        break;
}
PQfinish(myconn);
return EXIT_SUCCESS;
}
$ ./pgselect "SELECT * FROM orders LIMIT 5"
Verbindung hergestellt
Anzahl Zeilen: 5
Anzahl Spalten: 6
Tabellenstruktur:
orderid:4
orderdate:4
customerid:4
netamount:-1
tax:-1
totalamount:-1
Daten:
1,2004-01-27,7888,313.24,25.84,339.08,
2,2004-01-01,4858,54.90,4.53,59.43,
3,2004-01-17,15399,160.10,13.21,173.31,
4,2004-01-28,17019,106.67,8.80,115.47,
5,2004-01-09,14771,256.00,21.12,277.12,

```


Mit den bereits verwendeten Funktionen wurde das gesamte Resultset komplett auf den Client übertragen und in einer C-Programmstruktur gespeichert. Diese Vorgehensweise kann bei großen Ergebnissen von SQL-Abfragen problematisch werden und an die Grenzen der Systemressourcen des Client stoßen. Das gesamte Resultset wird auf dem Client im Memory gehalten.

Mit „libpq“-Mitteln ist es möglich, Cursor zu bilden, so wie das aus dem PL/pgSQL-Umfeld bekannt ist. Neben dem Vorteil der einzelnen Satzbearbeitung ist es möglich, die Resultsets in einzelne Pakete zu zerlegen. Damit werden größere Ergebnisse von SQL-Abfragen besser beherrschbar.

Das Programm in Listing 8.9 verwendet einen Cursor und liefert alle Sätze der Tabelle in einem Resultset zurück. Dies wird durch den Befehl " FETCH ALL IN cust_cursor" ausgelöst.

Listing 8.9 Verwendung eines Cursors mit „libpq“

```
#include <stdlib.h>
#include <libpq-fe.h>
void runSQL(PGconn *conn, char *sqlcomm)
{
    PGresult *result;
    printf("%s\n", sqlcomm);
    result = PQexec(conn, sqlcomm);
    printf("\nStatus: %s\n", PQresStatus(PQresultStatus(result)));
    printf("Number of rows: %s\n", PQcmdTuples(result));
    printf("Result: %s\n", PQresultErrorMessage(result));
    switch(PQresultStatus(result)) {
    case PGRES_TUPLES_OK:
    {
        int i,j;
        int rows = PQntuples(result);
        int cols = PQnfields(result);
        for ( i = 0; i < rows; i++) {
            for ( j = 0; j < cols; j++)
                printf(" %s = %s,", PQfname(result,j), PQgetvalue(result,i,j));
            printf("\n");
        }
    }
    }
}

int main()
{
    PGresult *result;
    PGconn *conn;
    conn = PQconnectdb("");
    if(PQstatus(conn) == CONNECTION_OK) {
        printf("Connection successful\n");
        runSQL(conn, "BEGIN WORK");
        runSQL(conn, "DECLARE cust_cursor CURSOR FOR SELECT firstname, lastname
FROM customers");
        runSQL(conn, "FETCH ALL IN cust_cursor");
        runSQL(conn, "CLOSE cust_cursor");
        runSQL(conn, "COMMIT WORK");
    } else {
        printf("Connection failed: %s\n", PQerrorMessage(conn));
    }
}
```



```

    PQfinish(conn);
    return EXIT_SUCCESS;
}
$ ./pg_cursor_all
Connection successful
BEGIN WORK
Status: PGRES_COMMAND_OK
Number of rows:
Result:
DECLARE cust_cursor CURSOR FOR SELECT firstname, lastname FROM customers
Status: PGRES_COMMAND_OK
Number of rows:
Result:
FETCH ALL IN cust_cursor
Status: PGRES_TUPLES_OK
Number of rows: 20000
Result:
firstname = Sven, lastname = Meier,
firstname = Horst, lastname = Schulze,
. . .
CLOSE cust_cursor
Status: PGRES_COMMAND_OK
Number of rows:
Result:
COMMIT WORK
Status: PGRES_COMMAND_OK
Number of rows:
Result:

```

Durch die Verwendung eines Cursors ist es möglich, das Resultset in Stücken (Batches) anzufordern. Das ist sinnvoll, wenn nicht alle Daten sofort angezeigt werden oder in der Applikation auf Bedieneranforderung weiterlaufen sollen. Dieses Vorgehen spart nicht nur Ressourcen, insbesondere Memory, sondern sichert auch die Performance der Applikation für sehr große Tabellen. Es muss also kein Unterschied mehr in Abhängigkeit von der Anzahl der Sätze gemacht werden.

Das Programm in Listing 8.10 ruft das Resultset in Stücken von 12 Sätzen ab.

Listing 8.10 Das Resultset stückweise abrufen

```

#include <stdlib.h>
#include <libpq-fe.h>
void printResult(PGresult *result)
{
    int i,j;
    int nrow = PQntuples(result);
    int ncols = PQnfields(result);
    for ( i=0; i < nrow; i++ ) {
        for ( j=0; j < ncols; j++)
            printf("%s - %s\n", PQfname(result, j), PQgetvalue(result, i, j));
        }
    printf("\n");
}
void runSQL(PGconn *conn, char *sqlcomm)
{
    PGresult *result;
    printf("%s\n", sqlcomm);
}

```



```

    result = PQexec(conn, sqlcomm);
    printf("\nStatus: %s\n", PQresStatus(PQresultStatus(result)));
    printf("Number of rows: %s\n", PQcmdTuples(result));
    printf("Result: %s\n", PQresultErrorMessage(result));
    switch(PQresultStatus(result)) {
    case PGRES_TUPLES_OK:
    {
        printResult(result);
        break;
    }
    }
}
int main()
{
    PGresult *result;
    PGconn *conn;
    int ntups;
    conn = PQconnectdb("");
    if(PQstatus(conn) == CONNECTION_OK) {
        printf("Connection successful\n");
        runSQL(conn, "BEGIN WORK");
        runSQL(conn, "DECLARE cust_cursor CURSOR FOR SELECT firstname, lastname
FROM customers");
        do {
            result = PQexec(conn, "FETCH 12 IN cust_cursor");
            if ( PQresultStatus(result) == PGRES_TUPLES_OK) {
                ntups = PQntuples(result);
                printResult(result);
                PQclear(result);
            } else ntups = 0;
        } while(ntups);
        runSQL(conn, "CLOSE cust_cursor");
        runSQL(conn, "COMMIT WORK");
    } else {
        printf("Connection failed: %s\n", PQerrorMessage(conn));
    }
    PQfinish(conn);
    return EXIT_SUCCESS;
}

```

Alle bisherigen Beispiele verwenden die Funktionen der Bibliothek „libpq“ im synchronen Modus. Jeder Programmaufruf einer Funktion wartet, bis dieser einen Wert zurückliefert. Datenbankabfragen liefern jedoch nicht immer eine zeitnahe Antwort zurück. Zum Beispiel kann eine Abfrage an ein Data Warehouse mehrere Minuten dauern. Das Programm muss also warten, bis die Funktion „PQexec“ eine Antwort zurückgibt und ist in dieser Zeit blockiert. Das ist eine unbefriedigende Situation, insbesondere für Anwendungen mit grafischer Benutzeroberfläche.

Alternativ kann die Anwendung asynchron programmiert werden. Dabei ist sie so lange nicht mehr blockiert, wie eine Datenbank-Anfrage läuft. Dabei handelt es sich um ein sogenanntes Call Back-Verfahren. Die Anwendung erhält eine Nachricht, sobald ein Ereignis in der Datenbank stattgefunden hat. Nicht-blockierende oder asynchrone Programme bestehen aus einer äußeren Schleife, in der die Nachrichten abgefragt und verarbeitet werden.

Damit das Programm nicht auf die Funktionen *PQsendQuery* und *PQgetResult* der „libpq“ wartet, muss die Verbindung als „nicht-blockierend“ definiert werden. Dies erfolgt mit Hilfe

der Funktion *PQsetnonblocking*. Der aktuelle Modus kann mit der Funktion *PQisnonblocking* abgefragt werden. Ein Wert ungleich „Null“ bedeutet, dass sich die Verbindung im nicht-blockierenden Modus befindet. Die Funktion „PGgetResult“ liefert bei jedem Aufruf einen Wert, bis alle Ergebnisse der Abfrage geliefert wurden.

Listing 8.11 Eine asynchrone oder „nicht-blockierende“ SQL-Verbindung verwenden

```
#include <stdlib.h>
#include <libpq-fe.h>
int main()
{
    PGresult *result;
    PGconn *conn;
    conn = PQconnectdb("");
    if(PQstatus(conn) == CONNECTION_OK) {
        int ret = PQsetnonblocking(conn, 1);
        int ret2 = PQisnonblocking(conn);
        printf("Non-Blocking? %d\n", ret2);
        PQsendQuery(conn, "SELECT * FROM test WHERE id = 19000000");
        while ( result != PQgetResult(conn)) {
            printf("%s\n", PQgetvalue(result, 1, 1));
        }
        PQclear(result);
    }
    PQfinish(conn);
}
```

Wie Sie sicher bereits bemerkt haben, zeichnen sich C-Programme, welche die „libpq“-Funktionen verwenden durch eine sehr gute Performance aus. Dies liegt einerseits an der Sprache C selbst, andererseits daran, dass es sich bei den Funktionen der Bibliothek um Programmcode handelt, der relativ weit unten, also nahe an der PostgreSQL Core Engine angesiedelt ist und deshalb wenig Overhead produziert. Diese Eigenschaft kann man sich zunutze machen, um bestimmte Prozesse der Applikation zu beschleunigen. Aus diesem Grund fällt in Zusammenhang mit der „libpq“-Programmierung auch immer wieder das Thema *Large Objects*.

Die Forderung nach der Speicherung von Large Objects in der Datenbank ist vorwiegend mit den Anforderungen durch Internet-Applikationen gekommen, die Bilder, Movies oder auch Textbausteine in geordneter Form speichern müssen. PostgreSQL besitzt eine Schnittstelle für Large Objects, die sich alle in einer zentralen System-Tabelle mit dem Namen *pg_largeobject* wiederfinden. In einem Feld können dabei bis zu 2 GByte untergebracht werden. In Tabelle 8.4 finden Sie eine Übersicht der Funktionen.

Das Beispiel in Listing 8.12 lädt die als Parameter angegebene Datei als Large Object in die Datenbank.

Tabelle 8.4 Funktionen der Bibliothek „libpq“ für Large Objects

Funktion	Beschreibung
<code>oid lo_creat(PGconn *conn, int mode);</code>	Erstellt ein neues Large Object. Rückgabewert ist der Objekt-Identifizier des Objekts (OID). Der Modus kann lesend oder schreibend sein (INV_READ; INV_WRITE).
<code>oid lo_import(PGconn *conn, const char *filename);</code>	Importiert ein Large Object in die Datenbank aus der spezifizierten Datei (filename). Rückgabewert ist ein neuer OID.
<code>oid lo_import_with_oid(PGconn *conn, const char *filename, oid lobjId);</code>	Importiert ein Large Object in einen neuen, selbst gewählten OID. Rückgabewert ist der neue OID oder „InvalidOid“ (Null), falls der gewünschte OID bereits vergeben ist.
<code>int lo_export(PGconn *conn, Oid lobjId, const char *filename);</code>	Exportiert ein Large Object aus dem angegebenen OID in eine Datei (filename). Rückgabewert ist „1“ bei Erfolg und „-1“ im Fehlerfall.
<code>int lo_write(PGconn *conn, int fd, const char *buf, size_t len);</code>	Schreibt eine Anzahl von Bytes (len) in ein Large Object mit dem Handle „fd“. Der Handle wurde im vorhergehenden „lo_open“ zurückgegeben. Quelle ist der Speicherbereich, auf den „*buf“ verweist.
<code>int lo_read(PGconn *conn, int fd, char *buf, size_t len);</code>	Funktioniert analog zu „lo_write“, in die andere Richtung.
<code>int lo_seek(PGconn *conn, int fd, int offset, int whence);</code>	Verstellt den aktuellen Positionierungszeiger auf den Wert von „offset“.
<code>int lo_tell(PGconn *conn, int fd);</code>	Liefert als Rückgabewert die aktuelle Position des Zeigers.
<code>int lo_truncate(PGconn *conn, int fd, size_t len);</code>	Schneidet den Inhalt des Large Objects bis zur Länge „size_t“ ab.

Listing 8.12 Ein Large Object aus einer Datei in die Datenbank laden

```

#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
int main(int argc, char **argv) {
    char *filename;
    PGconn *conn;
    PGresult *result;
    Oid lo_id;
    if (argc != 2) {
        printf("Usage: %s filename.\n", argv[0]);
        exit(1);
    }
    filename = argv[1];
    conn = PQconnectdb("");
    if (PQstatus(conn) == CONNECTION_OK) {
        printf("Importing file %s as Large Object to database...\n", filename);
        result = PQexec(conn, "BEGIN");
        PQclear(result);
    }
}

```

```

    lo_id = lo_import(conn, filename);
    printf("Object ID is: %d\n", lo_id);
    result = PQexec(conn, "END");
    PQfinish(conn);
} else {
    printf("%s\n", "Connection to database could not be established.");
    printf("%s\n", PQerrorMessage(conn));
}
}
$ ./load_lo 14_01.bmp
Importing file 14_01.bmp as Large Object to database...
Object ID is: 16468

```

Das Objekt findet sich unter dem OID in der System-Tabelle „pg_largeobject“ wieder. Wie Sie in Listing 8.13 sehen erfolgt die Speicherung in einzelnen Seiten. Dies verbessert die Performance bei der Positionierung.

Listing 8.13 Large Objects in der Datenbank

```

(postgres@[local]:5432) [hanser] > \d pg_largeobject
Table "pg_catalog.pg_largeobject"
Column | Type      | Modifiers
-----+-----+-----
loid    | oid       | not null
pageno  | integer   | not null
data    | bytea     |
Indexes:
    "pg_largeobject_loid_pn_index" UNIQUE, btree (loid, pageno)
(postgres@[local]:5432) [hanser] > SELECT loid, COUNT(*)
> FROM pg_largeobject GROUP BY loid;
 loid | count
-----+-----
16468 | 6511
(1 row)

```

Der Ansatz ist also ein anderer als bei vergleichbaren Datenbank-Systemen, bei denen die Large Objects direkt in der Applikations-Tabelle gespeichert wird. In PostgreSQL erfolgt die Speicherung zentral, und die Anwendung referenziert über den OID (Beispiel in Listing 8.14).

Listing 8.14 Die Objekt-Referenz in der Applikationstabelle speichern

```

(postgres@[local]:5432) [hanser] > CREATE TABLE images(
> id          int,
> image_name  varchar(30),
> image_id    oid);
CREATE TABLE
(postgres@[local]:5432) [hanser] > INSERT INTO images
VALUES(1, 'Big Tree', 16468);
INSERT 0 1

```


8.1.2 PostgreSQL-Applikationen mit Java entwickeln

Datenbankapplikationen, die in Java geschrieben sind, bieten ein hohes Maß an Übertragbarkeit. Die JDBC-Schnittstelle garantiert weitgehende Datenbank-Unabhängigkeit. Eine Portierung von Java-Applikationen von Oracle nach PostgreSQL sollte deshalb wenig Aufwand bereiten. Aber auch hier muss untersucht werden, inwieweit Vendor-spezifische Features benutzt wurden. Für eine Applikation, die mit einem Thin JDBC-Treiber auskommt, kann davon ausgegangen werden, dass sie weitgehend Datenbank-unabhängig ist, da der Treiber nur die Standard-JDBC-Funktionen enthält. Dagegen verfügt der OCI-Treiber über viele Oracle-spezifische Erweiterungen, die für andere Datenbanksysteme nicht zur Verfügung stehen.

Den JDBC-Treiber für PostgreSQL können Sie von der Seite <http://jdbc.postgresql.org> herunterladen. Achten Sie darauf, dass der Treiber zur Version des PostgreSQL Servers passt. Den zur Version 9.1.1 passenden Treiber liefert die Datei *postgresql-9.1-901.jdbc4.jar*.

Um eine Verbindung zur Datenbank erstellen zu können, muss sich der JDBC-Treiber mit dem Driver Manager registrieren. Dies erfolgt über die Methode „Class.forName“. Danach kann die Verbindung zur Datenbank hergestellt werden. Die sogenannte URL enthält die notwendigen Informationen über Server und Datenbank: Servername, Portnummer sowie Datenbank. Als zweiter und dritter Parameter werden Benutzername und Passwort mitgegeben. Das Programm in Listing 8.15 registriert den Treiber und verbindet sich zur Datenbank

Listing 8.15 Eine JDBC-Verbindung zur PostgreSQL-Datenbank herstellen

```
import java.sql.*;
public class TestConn {
    public static void main(String[] argv) {
        Connection conn = null;
        DatabaseMetaData dmd = null;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Treiber konnte nicht geladen werden.");
            System.out.println(ex.toString());
        }
        try {
            conn = DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/hanser","postgres","post_123");
        } catch (SQLException cex) {
            System.out.println("Verbindung konnte nicht hergestellt werden.");
            System.out.println(cex.toString());
        }
        if ( conn != null ) {
            System.out.println("\nVerbindung zur PostgreSQL-Datenbank erfolgreich
hergestellt");
            try {
                dmd = conn.getMetaData();
                System.out.println("Verbunden als Benutzer " +
dmd.getUserName() + " zur Datenbank " + dmd.getURL());
            } catch ( SQLException sqllex ) {
                System.out.println(sqllex.toString());
            }
        }
    }
}
```



```

    }
    }
}
C:\JDev>"C:\program files\Java\jdk1.6.0_21\bin\javac"
-cp .;../postgresql-9.1-901.jdbc4.jar TestConn.java
C:\JDev>java -cp .;../postgresql-9.1-901.jdbc4.jar TestConn
Verbindung zur PostgreSQL-Datenbank erfolgreich hergestellt
Verbunden als Benutzer postgres zur Datenbank
jdbc:postgresql://localhost:5432/hanser

```



Hinweis

Wenn Sie ein Java-Programm ausführen, das Daten verändert, dann werden Sie feststellen, dass die Änderungen wirksam werden, ohne dass die Transaktion explizit mit „COMMIT“ abgeschlossen wurde. Die Standardeinstellung im PostgreSQL-JDBC-Treiber ist „AUTOCOMMIT=true“. Damit wird jede SQL-Anweisung automatisch mit COMMIT beendet. Diese Einstellung ist für die Programmierung von Client-Programmen eher ungünstig. Setzen Sie deshalb „AUTOCOMMIT“ am besten aus. Das Ausschalten erfolgt durch die Variable vom Typ „Connection“: `conn.setAutoCommit(false)`; Die COMMIT-Anweisung kann ebenfalls mit der Funktion „executeUpdate“ abgesetzt werden: `stmt.executeUpdate("COMMIT")`;

Die JDBC-Schnittstelle liefert Standard-Funktionen für die Metadaten des PostgreSQL-Servers und der Datenbank. Die Metadaten sind bedeutsam für die Navigation durch die Datenbank, insbesondere dann, wenn Hersteller-unabhängig programmiert wird. In Listing 8.16 finden Sie ein Beispiel für die Auswertung von Metadaten.

Listing 8.16 Die JDBC-Metadaten auswerten

```

import java.sql.*;
public class postgresMetaData {
    public static void main(String[] argv) {
        Connection conn = null;
        DatabaseMetaData dmd = null;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn =
DriverManager.getConnection("jdbc:postgresql://192.168.178.201:5432/
hanser","postgres","post1234");
            dmd = conn.getMetaData();
            System.out.println("Database Product: " + dmd.getDatabaseProductNa-
me());
            System.out.println("Product Version: " + dmd.getDatabaseProductVersi-
on());
            System.out.println("Driver Name: " + dmd.getDriverName());
            System.out.println("Driver Version: " + dmd.getDriverVersion());
            System.out.println("Connection URL: " + dmd.getURL());

```



```

        System.out.println("Supports Transactions?: " + dmd.supportsTrans
actions());
        System.out.println("Isolation Level: " + dmd.getDefaultTransaction
Isolation());
        conn.close();
    } catch (SQLException sqllex) {
        System.out.println(sqllex.toString());
    }
}
}
C:\JDev>java -cp .;. \postgresql-9.1-901.jdbc4.jar postgresMetaData
Database Product: PostgreSQL
Product Version: 9.1.1
Driver Name: PostgreSQL Native Driver
Driver Version: PostgreSQL 9.1 JDBC4 (build 901)
Connection URL: jdbc:postgresql://192.168.178.201:5432/hanser
Supports Transactions?: true
Isolation Level: 2

```

SELECT-Anweisungen liefern ein Resultset zurück. Für die Verarbeitung des Resultsets steht eine Java-Klasse zur Verfügung. Die Funktion *executeQuery* liefert das Resultset. Im Beispiel in Listing 8.17 werden die Metadaten des Resultsets ausgewertet. Damit ist das Programm für eine beliebige SELECT-Anweisung lauffähig.

Listing 8.17 Ein JDBC Resultset verarbeiten

```

import java.sql.*;
public class result {
    public static void main(String[] argv) {
        Connection conn = null;
        ResultSet rs;
        ResultSetMetaData rsmd;
        Statement stmt;
        int ncols;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.201:5
432/postgres","postgres","post1234");
            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT * FROM customers");
            rsmd = rs.getMetaData();
            ncols = rsmd.getColumnCount();
            while ( rs.next() ) {
                for ( int i=1; i <= ncols; i++ ) {
                    System.out.print(rs.getString(i) + "\t");
                }
                System.out.println();
            }
            conn.close();
        } catch (SQLException sqllex) {
            System.out.println(sqllex.toString());
        }
    }
}

```

Für Datenmanipulationen steht die JDBC-Klasse „Statement“ mit der Funktion *executeUpdate* zur Verfügung. Man kann sie für INSERT-, UPDATE und DELETE-Anweisungen verwenden. Zunächst muss ein Statement erstellt werden. Der Funktion „executeUpdate“ wird die SQL-Anweisung als String mitgegeben. Rückgabewert ist die Anzahl der geänderten Sätze.

Listing 8.18 Datenmanipulation mit JDBC

```
import java.sql.*;
public class dml {
    public static void main(String[] argv) {
        Connection conn = null;
        Statement stmt;
        int rcount;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.201:5432/hanser","postgres","post1234");
            stmt = conn.createStatement();
            rcount = stmt.executeUpdate("UPDATE test SET text = 'Update by JDBC'
WHERE id < 10");
            System.out.println(rcount + " rows changed.");
            conn.close();
        } catch (SQLException sqlex) {
            System.out.println(sqlex.toString());
        }
    }
}
C:\JDev>java -cp .;.\postgresql-9.1-901.jdbc4.jar dml
9 rows changed.
```

Programme sollen häufig auch Batch-Ladeprozesse durchführen. Ein Testprogramm mit dem folgenden Abschnitt fügt 100 000 Sätze in eine Tabelle ein und benötigt dafür 2:50 Minuten:

```
for ( i = 1; i < 100001; i++ ) {
    rcount = stmt.executeUpdate("INSERT INTO test2 VALUES (" + i + ", 'Insert
test records...'");
}
```

Die Laufzeit ist überraschend hoch. Ein Grund dafür ist sicher die Tatsache, dass „Autocommit“ nicht ausgeschaltet wurde und somit nach jeder Ausführung einer einzelnen INSERT-Anweisung ein „COMMIT“ durchgeführt wird. Wir korrigieren das und schalten „Autocommit“ aus:

```
conn.setAutoCommit(false);
```

Nun beträgt die Laufzeit 1:50 Minuten. Das ist schon besser, aber immer noch vergleichsweise lang. Eine weitere Verbesserung lässt sich mit Hilfe von *SQL Batches* erreichen. Dabei werden alle Anweisungen in das Statement geladen und danach in einem Batch ausgeführt. Das Programm in Listing 8.19 verwendet „SQL Batches“ und benötigt nur noch 9,1 Sekun-

den! Das Prinzip, einzelne Statements als Batch an den Datenbankserver zu übergeben, führt also zu signifikanten Laufzeitverbesserungen.

Listing 8.19 Laden von Daten mit „SQL Batches“

```
import java.sql.*;
public class batch1 {
    public static void main(String[] argv) {
        Connection conn = null;
        Statement stmt;
        int rcount;
        int i;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.201:5432/hanser","postgres","post1234");
            conn.setAutoCommit(false);
            stmt = conn.createStatement();
            for ( i = 1; i < 100001; i++ ) {
                stmt.addBatch("INSERT INTO test2 VALUES (" + i + ", 'Insert test records...'");
            }
            stmt.executeBatch();
            stmt.executeUpdate("COMMIT");
            conn.close();
        } catch (SQLException sqlex) {
            System.out.println(sqlex.toString());
        }
    }
}
```



Praxistipp

Verwenden Sie für das Laden oder Ändern von Massendaten SQL Batches. Damit erzielen Sie signifikante Laufzeitverbesserungen.

Ein weiteres Feature zur Laufzeitverbesserung, das nahezu allen Programmiersprachen zur Verfügung steht, ist die Verwendung von Bindevariablen. Die Arbeit mit Bindevariablen bietet Vorteile für die Performance, wenn dieselbe Anweisung mehrfach mit unterschiedlichen Werten ausgeführt wird. Das „Parsen“ muss nur einmal erfolgen und spart somit wertvolle Zeit bei jeder Ausführung. Es wird ein „Prepared Statement“ an den Server übergeben, das nach dem „Parsen“ zur Ausführung bereitsteht. Vor jeder Ausführung müssen dann nur noch die Bindevariablen mit den entsprechenden Werten gefüllt werden. In Listing 8.20 finden Sie ein Beispielprogramm für die Verwendung von Bindevariablen. Mit einer Ausführungszeit von 0:58 Minuten lässt sich feststellen, dass es sich dabei nicht um ein Feature für die Batch-Verarbeitung handelt, sondern für eine Umgebung mit mehreren Benutzern, die sich dasselbe SQL teilen. Es demonstriert jedoch die Laufzeitvorteile gegenüber der herkömmlichen Verarbeitung.

Listing 8.20 Verwendung von Bindevariablen

```

import java.sql.*;
public class batch2 {
    public static void main(String[] argv) {
        Connection conn = null;
        PreparedStatement pstmt;
        Statement stmt;
        int rcount;
        int i;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.201:5432/hanser","postgres","post1234");
            conn.setAutoCommit(false);
            pstmt = conn.prepareStatement("INSERT INTO test2 VALUES (?, ?)");
            pstmt.setString(2, "Insert test records...");
            for ( i = 1; i < 100001; i++ ) {
                pstmt.setInt(1, i);
                pstmt.executeUpdate();
            }
            stmt = conn.createStatement();
            stmt.executeUpdate("COMMIT");
            conn.close();
        } catch (SQLException sqllex) {
            System.out.println(sqllex.toString());
        }
    }
}

```

Large Objects der PostgreSQL-Datenbank können auch mit Java-Programmen bearbeitet werden. Betrachten wir an dieser Stelle zwei Varianten. Im Unterschied zum Beispiel in Listing 8.21 wird die Image-Datei zunächst direkt in der Tabelle „images“ gespeichert. Im zweiten Beispiel wird Image selbst in die zentrale Tabelle für Large Objects geladen und nur die Referenz auf die Objekt-ID in die Tabelle „images2“ eingetragen. Legen Sie zunächst die beiden Tabellen an.

Listing 8.21 Tabellen für das Speichern von Large Objects anlegen

```

(postgres@[local]:5432) [hanser] > CREATE TABLE images(
> id int,
> image_name varchar(30),
> img bytea);
CREATE TABLE
(postgres@[local]:5432) [hanser] > CREATE TABLE images2(
> id int,
> image_name varchar(30),
> image_id oid);
CREATE TABLE

```

Das erste Beispielprogramm finden Sie in Listing 8.22. Zum Setzen der Bindevariablen verwendet man die Funktion „setBinaryStream“. Damit wird der zum Laden der Datei aus dem

Betriebssystem verwendete `BinaryStream` direkt weitergeleitet. Beachten Sie, dass der zweite Parameter mit „int“ ge-castet werden muss, da die Funktion „length()“ einen Long-Wert zurückliefert.

Listing 8.22 Speichern einer Image-Datei direkt in der Tabelle

```
import java.sql.*;
import java.io.*;
public class pgLargeObjects {
    public static void main(String[] argv) {
        Connection conn = null;
        DatabaseMetaData dmd = null;
        String FileName = null;
        FileInputStream fis = null;
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.100:5432/hanser","postgres","post1234");
            conn.setAutoCommit(false);
        } catch (SQLException sqllex) {
            System.out.println(sqllex.toString());
        }
        FileName = argv[0];
        System.out.println("Importing file " + FileName + " as Large Object to database...\n");
        File file = new File(FileName);
        try {
            fis = new FileInputStream(file);
        } catch (IOException ioex) {
            System.out.println(ioex.toString());
        }
        try {
            PreparedStatement ps = conn.prepareStatement("INSERT INTO images VALUES (?, ?, ?)");
            ps.setInt(1, 1);
            ps.setString(2, file.getName());
            ps.setBinaryStream(3, fis, (int)file.length());
            ps.executeUpdate();
            conn.commit();
        } catch (SQLException sqllex) {
            System.out.println(sqllex.toString());
        }
        try {
            conn.close();
        } catch (SQLException sqllex) {
            System.out.println(sqllex.toString());
        }
    }
}

C:\JDev>java -cp ../postgresql-9.1-901.jdbc4.jar pgLargeObjects 14_01.bmp
Importing file 14_01.bmp as Large Object to database...
C:\JDev>psql
(postgres@[local]:5432) [hanser] > SELECT id, image_name FROM images;
 id | image_name
```

```

-----+-----
1 | 14_01.bmp
(1 row)

```

Das Image wurde also in einem Byte-Array direkt in der Tabelle gespeichert. Der Nachteil dieser Vorgehensweise besteht darin, dass die Tabelle aus sehr langen Zeilen besteht, was für die weitere Verarbeitung einen erheblichen Einfluss auf die Performance hat. Des Weiteren ist zu beachten, dass eine Spalte vom Typ „bytea“ nicht größer als 1 GByte werden kann. Die bessere Methode ist, die Large Object-Funktionalität der PostgreSQL-Datenbank zu nutzen. Dabei wird das Image als Large Object zentral in der Datenbank gespeichert. Die Tabelle „images2“ enthält nur die Objekt-ID, also eine Referenz auf das Large Objekt. Ein Large Object kann bis zu 2 GByte groß sein. Ein Beispiel für diese Vorgehensweise finden Sie in Listing 8.23.

Listing 8.23 Speichern der Image-Datei als Large Object

```

import java.sql.*;
import org.postgresql.largeobject.*;
import java.io.*;
public class pgLargeObjects2 {
    public static void main(String[] argv) {
        Connection conn = null;
        DatabaseMetaData dmd = null;
        String FileName = null;
        FileInputStream fis = null;
        LargeObjectManager lobj;
        LargeObject obj;
        long oid;
        int i,j;
        byte buf[] = new byte[204800];
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        FileName = argv[0];
        System.out.println("Importing file " + FileName + " as Large Object to
database...\n");
        File file = new File(FileName);
        try {
            fis = new FileInputStream(file);
        } catch (IOException ioex) {
            System.out.println(ioex.toString());
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.100:
5432/hanser","postgres","post1234");
            conn.setAutoCommit(false);
            lobj = ((org.postgresql.PGConnection)conn).getLargeObjectAPI();
            oid = lobj.createLO(LargeObjectManager.READ | LargeObjectManager.
WRITE);
            obj = lobj.open(oid, LargeObjectManager.WRITE);
            j = 0;
            try {
                while (( i = fis.read(buf, 0, 204800)) > 0 ) {
                    obj.write(buf, 0, j);

```



```

        j += i;
    }
    obj.close();
} catch (IOException ioex) {
    System.out.println(ioex.toString());
}
PreparedStatement ps = conn.prepareStatement("INSERT INTO images2
VALUES (?, ?, ?)");
ps.setInt(1, 1);
ps.setString(2, file.getName());
ps.setLong(3, oid);
ps.executeUpdate();
conn.commit();
conn.close();
} catch (SQLException sqllex) {
    System.out.println(sqllex.toString());
}
}
}
C:\JDev>psql
(postgres@[local]:5432) [hanser] > SELECT * FROM images2;
 id | image_name | image_id
-----+-----+-----
  1 | 14_01.bmp  |    16495
(1 row)

```

Im Programm wird das Large Object-API verwendet. Es ist im JDBC-Treiber enthalten und muss importiert werden (`import org.postgresql.largeobject.*`).

Das Programm in Listing 8.24 liest ein Large Object aus der Datenbank mit Hilfe der Large Object-API. Die Objekt-ID wird aus der Tabelle „images2“ ermittelt.

Listing 8.24 Ein Large Object aus der Datenbank lesen

```

import java.sql.*;
import org.postgresql.largeobject.*;
import java.io.*;
public class pgRetrieveObject {
    public static void main(String[] argv) {
        Connection conn = null;
        LargeObjectManager lobj;
        LargeObject obj;
        long oid;
        int i,j;
        byte buf[] = new byte[204800];
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Driver could not be loaded.");
        }
        try {
            conn = DriverManager.getConnection("jdbc:postgresql://192.168.178.100:
5432/hanser","postgres","post1234");
            conn.setAutoCommit(false);
            lobj = ((org.postgresql.PGConnection)conn).getLargeObjectAPI();
            PreparedStatement ps = conn.prepareStatement("SELECT image_id FROM
images2 WHERE image_name = ?");
            ps.setString(1, "14_01.bmp");

```

```

        ResultSet rs = ps.executeQuery();
        if ( rs.next() ) {
            oid = rs.getLong(1);
            obj = lobj.open(oid, LargeObjectManager.READ);
// Verarbeitung
        }
        rs.close();
        conn.close();
    } catch (SQLException sqlex) {
        System.out.println(sqlex.toString());
    }
}
}

```

■ 8.2 Server-Programmierung

Unter dem Begriff „Server-Programmierung“ werden alle Programm-Architekturen verstanden, die im Datenbank-Server integriert sind. Zur Verfügung stehen verbreitete prozedurale Programmier- und Skriptsprachen sowie SQL-Erweiterungen.

8.2.1 Integrierte Prozedurale Sprachen

PostgreSQL bietet die Möglichkeit, prozedurale Sprachen in die Datenbank zu integrieren und als benutzerdefinierte Funktionen auszuführen. Dabei hat der PostgreSQL-Server selbst keine Kenntnis darüber, wie der Quelltext der Sprache zu interpretieren ist. Der Quellcode wird an einen „Handler“ übertragen. Der Handler übernimmt alle Aufgaben bis hin zur Programmausführung. Von Haus aus sind bereits Handler für die Sprachen PL/pgSQL, PL/Perl, PL/Python und PL/Tcl eingebaut. Handler für weitere Sprachen sind erhältlich, allerdings nicht im Kernpaket von PostgreSQL vorhanden.

Führen Sie die folgende Abfrage aus, um festzustellen, welche Sprachen installiert sind:

```

(postgres@[local]:5432) [hanser] > SELECT lanname,lanispl,lanpltrusted
> FROM pg_language;

```

lanname	lanispl	lanpltrusted
internal	f	f
c	f	f
sql	f	t
plpgsql	t	t

Wenn wie in diesem Fall eine Sprache fehlt, wurde sie möglicherweise bei der Distribution nicht angegeben. In diesem Fall müssen die Binaries noch einmal erstellt und installiert werden.

8.2.1.1 Perl

Wenn Sie Perl als Skriptsprache einsetzen wollen, können Sie zwischen zwei Optionen wählen:

- Perl als PostgreSQL-Sprache
- Perl mit DBI-Schnittstelle

In einem PostgreSQL-Cluster ist Perl als Sprache bereits implementiert. Sie kann wie jede andere Sprache direkt ausgeführt werden. Weit verbreitet sind die DBI-Module. Der Vorteil bei der Verwendung von DBI ist die Möglichkeit, die Schnittstelle austauschen zu können und damit ein gewisses Maß an Unabhängigkeit gegenüber dem Datenbankprodukt zu erzielen.

8.2.1.1.1 Perl als PostgreSQL-Sprache

Wenn Sie PostgreSQL aus dem Quellcode installiert haben, muss Perl in die Konfiguration mit eingebunden werden. In Listing 8.25 erstellen wir die Distribution mit einem Handler für die Sprache Perl.

Listing 8.25 Eine Distribution mit Perl erstellen

```
# ./configure --with-perl
# make
# make install
```

Nun müssen Sie die Sprache noch installieren. Melden Sie sich dazu als Benutzer „postgres“ an, und führen Sie den Befehl in Listing 8.26 aus.

Listing 8.26 Die Sprache Perl zur Datenbank hinzufügen

```
$ createlang plperl hanser
$ psql
psql (9.1.0)
(postgres@[local]:5432) [hanser] > SELECT lanname,lanispl,lanpltrusted
> FROM pg_language;
 lanname | lanispl | lanpltrusted
-----+-----+-----
 internal | f       | f
 c       | f       | f
 sql     | f       | t
 plpgsql | t       | t
 plperl  | t       | t
```

Für PL/Perl können anonyme Funktions-Blöcke erstellt werden. Die Syntax lautet:

```
DO $$
# Perl-Befehle>
. . .
$$ LANGUAGE plperl;
```

Perl Code kann als Funktion in der Datenbank gespeichert werden. Ein Beispiel finden Sie in Listing 8.27.

Listing 8.27 Eine PL/Perl-Funktion in der Datenbank anlegen

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION pl_min
(INTEGER, INTEGER)
> RETURNS INTEGER AS $$
> my ($a, $b) = @_;
> return $a if $a < $b;
> return $b;
> $$ LANGUAGE plperl;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT pl_min(1,99);
 pl_min
-----
      1
```

Für den Zugriff auf die Datenbank stehen vorgefertigte Funktionen zur Verfügung, zum Ausführen von SELECT-Anweisungen die Funktion *spi_exec_query*:

```
spi_exec_query(<SQL-Anweisung>, [Satzlimit])
```

Die Rückgabe erfolgt in einem Array. Listing 8.28 liefert ein Beispiel für die Verwendung von „spi_exec_query“.

Listing 8.28 Die Funktion spi_exec_query in ein Perl-Programm einbinden

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION perl_select()
> RETURNS SETOF kb AS $$
> my $rv = spi_exec_query('SELECT * FROM kb;',5);
> my $status = $rv->{status};
> my $nrows = $rv->{processed};
> foreach my $res (0 .. $nrows-1) {
>   my $row = $rv->{rows}[$res];
>   return_next($row);
> }
> return undef;
> $$ LANGUAGE plperl;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT perl_select();
      perl_select
-----
(1,Knowledgebase)
(2,Inhaltsverzeichnis)
. . .
```

8.2.1.1.2 Perl mit DBI

Bei Verwendung der DBI-Schnittstelle muss Perl als Sprache im PostgreSQL-Cluster nicht zwangsläufig installiert sein. Die Ausführung erfolgt über einen externen Perl-Interpreter.

Neben der generischen DBI-Schnittstelle gibt es Vendor-spezifische DBD-Schnittstellen. Es ist also möglich, Perl-Skripte weitgehend unabhängig vom eingesetzten Datenbank-System zu programmieren.

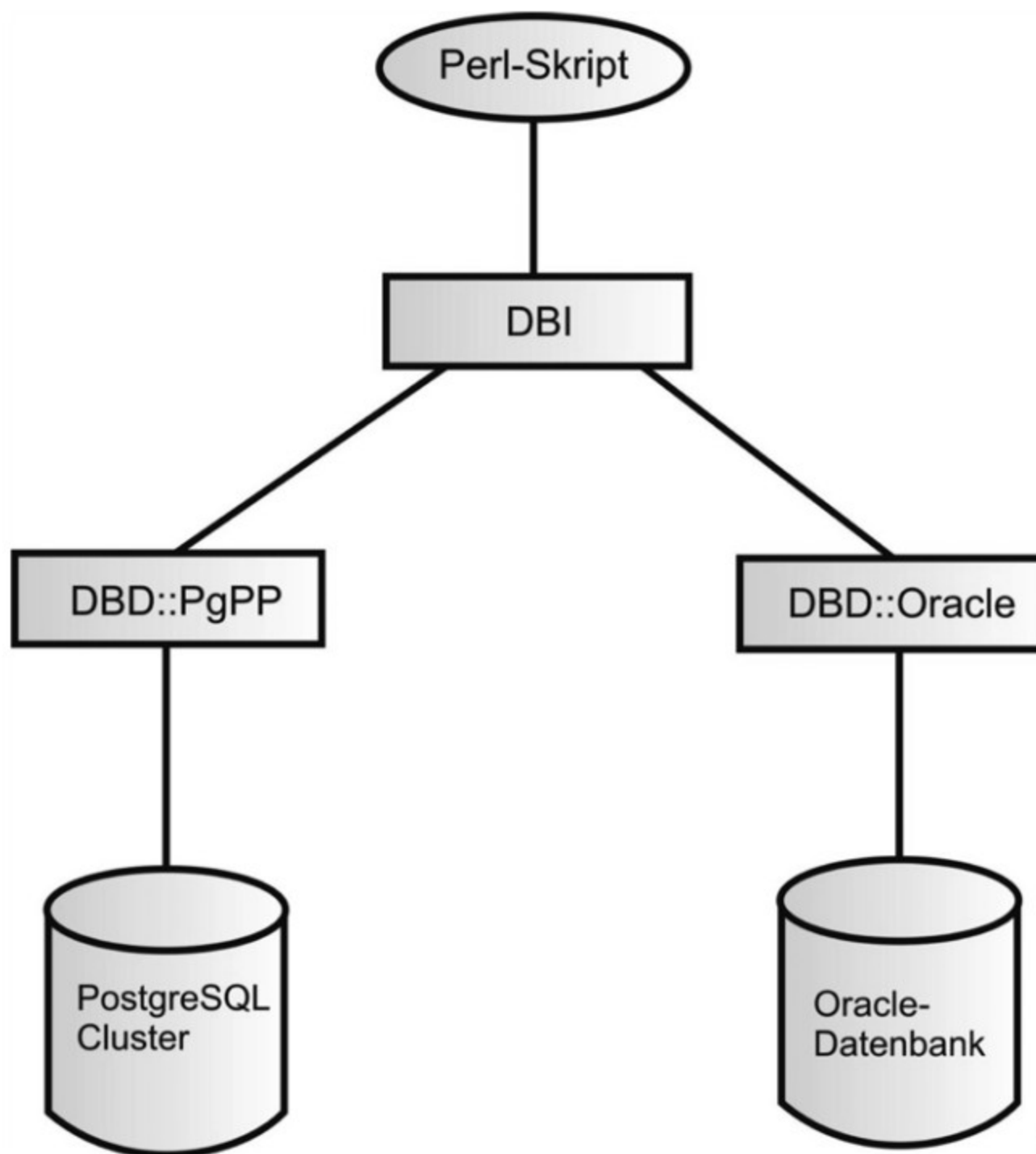


Bild 8.1 Die Perl-DBI-Architektur

Unix- und Linux-Systeme bringen einen Perl-Interpreter mit. Sofern nicht implementiert, installieren Sie einfach das entsprechende Paket. Falls Sie eine Installation vom Quellcode vornehmen wollen, ist das CPAN-Modul zu empfehlen. Für Windows-Systeme empfehlen wir „Active State Perl“ (<http://www.activestate.com>). Dabei handelt es sich um bereits kompilierte und verbundene Programme.

Den Quellcode für eine Linux-Installation (CPAN) finden Sie auf der Webseite <http://www.cpan.org>. Der komprimierte Tarball enthält die Versionsnummer. Die folgenden Schritte beschreiben die Installation:

1. Entpacken Sie den Quellcode:

```
# gzip -d perl-5.16.1.tar.gz
# tar -xvf perl-5.16.1.tar
```

2. Konfigurieren Sie die Installation:

```
# cd perl-5.16.1
# ./Configure -des -Dprefix=$HOME/localperl
```

3. Führen Sie den Make-Befehl aus, um die Module zu erstellen:

```
# make
```

4. Im finalen Schritt werden die Dateien erstellt und in die Zielverzeichnisse kopiert:

```
# make install
```

Unter Windows rufen Sie zur Installation von Active State Perl einfach den msi-Installer auf. Achten Sie darauf, den Perl Package Manager (ppm) mit zu installieren. Sie können ihn zur Installation des DBI- sowie des DBD-Moduls für PostgreSQL verwenden.

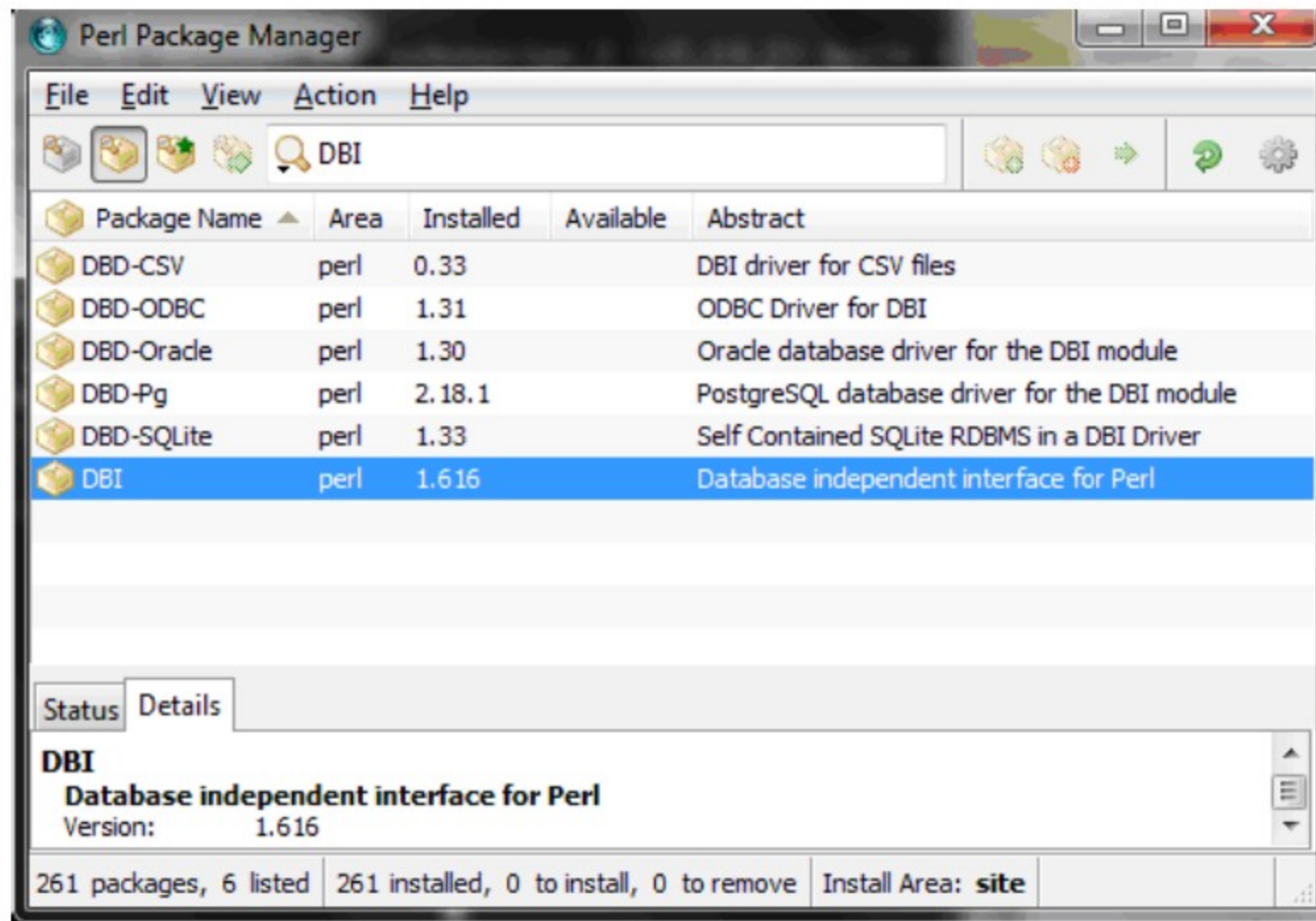


Bild 8.2 Das DBI-Modul mit dem Perl Package Manager (ppm) installieren

Zuerst soll ein Skript erstellt werden, das eine Verbindung zu einer Datenbank herstellt. Dazu verwenden wir die Funktion „DBI->connect“. Parameter der Funktion sind DBD-Treiber, Name und Lokation der Datenbank, Benutzername, Passwort sowie optionale, zusätzliche Attribute. Eine typische Connect-Funktion zu einem PostgreSQL-Cluster sieht wie folgt aus:

```
$dbp = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432","postgres",
"post1234");
```

Zusätzlich können mit dem Verbindungsaufbau Attribute gesetzt werden. Im Fall von PostgreSQL ist es möglich, Autocommit ein- und auszuschalten. Zum Ausschalten muss das Attribut auf „0“, zum Einschalten auf „1“ gesetzt werden. Der Standard ist „Autocommit on“.

Listing 8.29 Autocommit in einem Perl-Skript ausschalten

```
$dbp = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432","postgres",
"post1234",{ AutoCommit => 0 });
```

Attribute können auch für bestehende Verbindungen geändert werden:

```
$dbp{ AutoCommit } = 1;
```


**Hinweis**

Die Namen der Attribute sind Case-sensitiv. Benutzen Sie für „AutoCommit“ die entsprechenden Groß- und Kleinbuchstaben. Beachten Sie weiterhin, dass nicht alle DBD-Treiber das dynamische Ändern von Attributen unterstützen. Außerdem schränken Sie damit die Übertragbarkeit der Skripte auf verschiedene Datenbanksysteme ein.

Neben „AutoCommit“ sind folgende Attribute von Interesse:

- *PrintError*: Fehlermeldungen werden auf „stderr“ ausgegeben. Der Schalter ist in der Testphase hilfreich.
- *RaiseError*: Beim Auftreten eines Fehlers wird das Programm abgebrochen und liefert keinen Fehlerstatus an das Skript zurück. Der Schalter hat jedoch keinen Einfluss auf Fehler, die innerhalb eines eval-Blocks auftreten.

In Listing 8.30 finden Sie ein einfaches Skript, in dem eine Datenbankverbindung auf- und wieder abgebaut wird.

Listing 8.30 Aufbau einer Datenbankverbindung mit Perl DBI

```
#!/usr/bin/perl
use DBI;
use strict;
my $dsn = $ARGV[0];
my $user = $ARGV[1];
my $password = $ARGV[2];
my $conn = DBI->connect($dsn, $user, $password);
printf "Status: %s\n", $conn->state;
$conn->disconnect;
. . .
$ perl conntest.pl DBI:Pg:dbname=postgres;host=localhost postgres post1234
Status: OK
```

Für DBI-Module können Umgebungsvariablen gesetzt werden. Die wichtigsten finden Sie in Tabelle 8.5.

Tabelle 8.5 Umgebungsvariablen für DBI-Module

Variable	Bedeutung
DBI_DRIVER	Name des DSN-Driver. Kann im DSN-String weggelassen werden, wenn hier spezifiziert
DBI_DSN	DSN-String im Format „DBI:driver:optionen“. „driver“ kann leer sein, wenn in der Variablen „DSN_DRIVER“ angegeben.
DBI_USER	Benutzername im PostgreSQL Cluster
DBI_PASS	Passwort des Benutzers

Im Fall eines Verbindungsfehlers ist es sinnvoll, mit dem „die“-Befehl das Skript abbrechen. Ein Beispiel liefert Listing 8.31.

Listing 8.31 Fehlerbehandlung in DBI

```
#!/usr/bin/perl
use DBI;
use strict;
my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status: %s\n", $conn->state || "OK";
$conn->disconnect;
```

Die Ausführung von SQL-Anweisung wird vom DBI-Treiber in drei Phasen aufgeteilt:

- Vorbereitungsphase (Preparation)
- Ausführungsphase (Execution)
- Rückgabephase (Fetch, falls Ergebnisse zurückgeliefert werden)

Aus Datenbanksicht erfolgt in der Vorbereitungsphase das Parsing des SQL und in der Execution-Phase die Ausführung. Eine solche Vorgehensweise verwendet man auch in anderen Sprachen, insbesondere wenn Bindevariablen verwendet werden. Die Rückgabephase ist vergleichbar mit dem Fetch-Kommando zur Übergabe des Resultsets an den Client. Das „prepare“-Kommando liefert ein Statement Handle, mit dem die execute-Anweisung durchgeführt wird. Im Skript in Listing 8.32 finden Sie ein Beispiel für die Ausführung von DML-Befehlen.

Listing 8.32 DML-Anweisungen mit Perl DBI ausführen

```
#!/usr/bin/perl
use DBI;
use strict;
sub runDML {
    my ($conn, $command) = @_;
    print $command, "\n";
    my $sth = $conn->prepare($command);
    my $zeilen = $sth->execute;
    print "SQL-Status: ", $conn->err, "\n" if $conn->err;
    print "SQL-Fehler: ", $conn->errstr, "\n" if $conn->err;
    print "Zeilen verarbeitet: ", $zeilen, "\n";
}
my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status: %s\n", $conn->state || "OK";
runDML($conn, "INSERT INTO employees(employee_id,name,salary) VALUES
(4711,'Joe Ackermann',34000)");
runDML($conn, "COMMIT");
$conn->disconnect;

. . .
$ perl dml.pl
Status: OK
INSERT INTO employees(employee_id,name,salary) VALUES (4711,'Joe
Ackermann',34000)
Zeilen verarbeitet: 1
COMMIT
Zeilen verarbeitet: 0
```


Bei Ausführung einer SELECT-Anweisung liefert das Execute-Kommando die Anzahl der Sätze im Resultset zurück. Allerdings wird dieser Wert nicht als zuverlässig angesehen, so dass es besser ist, durch das Resultset zu laufen, bis man am Ende angekommen ist. Die Funktion „DBI->fetchrow_array“ erzeugt jeweils den nächsten Satz aus dem Resultset als Liste von Spaltenwerten. Das Statement Handle enthält Attribute mit näheren Informationen zum Resultset.

Tabelle 8.6 Attribute des Resultset

Attribut	Inhalt
NAME	Verweist auf ein Feld, das die Spaltennamen enthält
NUM_OF_FIELDS	Anzahl der Spalten
NULLABLE	Verweist auf ein Feld als Indikator, ob eine Spalte NULL enthalten darf
TYPE	Verweist auf ein Feld mit den Datentypen der Spalten
Statement	Eine Zeichenkette mit dem SQL-Befehl

In Listing 8.33 finden Sie ein Beispiel für die Verarbeitung eines Resultset.

Listing 8.33 Verarbeitung eines Resultset mit Perl DBI

```
#!/usr/bin/perl
use DBI;
use strict;
sub runQuery {
    my ($conn, $command) = @_;
    print $command, "\n";
    my $sth = $conn->prepare($command);
    my $zeilen = $sth->execute;
    print "SQL-Status: ", $conn->err, "\n" if $conn->err;
    print "SQL-Fehler: ", $conn->errstr, "\n" if $conn->err;
    print "Zeilen im Resultset: ", $zeilen, "\n";
    print "Anzahl von Spalten: ", $sth->{NUM_OF_FIELDS}, "\n";
    while ( my @row = $sth->fetchrow_array ) {
        print " ", join(" ",@row), "\n";
    }
    $sth->finish;
}
my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
    "postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status: %s\n", $conn->state || "OK";
runQuery($conn, "SELECT * FROM inventory LIMIT 10");
$conn->disconnect;
.
.
.
$ perl query.pl
Status: OK
SELECT * FROM inventory LIMIT 10
Zeilen im Resultset: 10
Anzahl von Spalten: 3
1 138 9
2 118 19
3 228 11
4 279 12
```

```

5 382 13
6 109 14
7 90 10
8 357 6
9 215 27
10 105 16

```

Wie in anderen Programmier- und Skriptsprachen üblich, erlaubt auch Perl DBI die Verwendung von Bindevariablen. Platzhalter für die Werte in der SQL-Anweisung ist das Fragezeichen. Das Binden der Variablen erfolgt mit Hilfe der DBI-Funktion „bind_param“. Im Beispiel in Listing 8.34 werden Bindevariablen in einem INSERT-Befehl verwendet.

Listing 8.34 Perl DBI-Skript mit Bindevariablen

```

#!/usr/bin/perl
use DBI;
use strict;
sub runDML {
    my ($conn) = @_;
    my $sth = $conn->prepare("INSERT INTO employees (employee_id,name,salary)
VALUES (?, ?, ?)");
    $sth->bind_param(1, 815);
    $sth->bind_param(2, "Max Mustermann");
    $sth->bind_param(3, 24600);
    my $zeilen = $sth->execute;
    print "Zeilen verarbeitet: ", $zeilen, "\n";
    $sth->bind_param(1, 777);
    $sth->bind_param(2, "Not Sure");
    $sth->bind_param(3, 11400);
    $zeilen = $sth->execute;
    print "Zeilen verarbeitet: ", $zeilen, "\n";
    $sth = $conn->prepare("COMMIT");
    $zeilen = $sth->execute;
}
my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status: %s\n", $conn->state || "OK";
runDML($conn);
$conn->disconnect;
. . .
$ perl bind1.pl
Status: 0
Zeilen verarbeitet: 1
Zeilen verarbeitet: 1

```

Perl DBI unterstützt auch das so genannte „Binding by reference“. Damit ist es möglich, Perl-Variablen als Bindevariablen zu benutzen. Der Skriptteil würde dann wie in Listing 8.35 dargestellt aussehen.

Listing 8.35 Perl DBI „Binding by reference“

```

sub runDML {
    my ($conn) = @_;
    my $sth = $conn->prepare("INSERT INTO employees (employee_id,name,salary)
VALUES (?, ?, ?)");
    my $emp_id;

```



```

my $emp_name;
my $emp_salary;
$sth->bind_param_inout(1, \$emp_id, 10);
$sth->bind_param_inout(2, \$emp_name, 30);
$sth->bind_param_inout(3, \$emp_salary, 10);
$emp_id = 815;
$emp_name = "Max Mustermann";
$emp_salary = 32600;
my $zeilen = $sth->execute;
print "Zeilen verarbeitet: ", $zeilen, "\n";
$sth = $conn->prepare("COMMIT");
$zeilen = $sth->execute;
}

```

Das Binden von Variablen kann auch für das Resultset von SELECT-Anweisungen verwendet werden. Die DBI-Funktion „bind-col“ verknüpft die Spalte des Resultset mit der Perl-Variablen.

Listing 8.36 Bindevariablen für Resultsets

```

#!/usr/bin/perl
use DBI;
use strict;
sub runQueryBind {
    my ($conn, $command) = @_;
    print $command, "\n";
    my $sth = $conn->prepare($command);
    my $zeilen = $sth->execute;
    print "SQL-Status: ", $conn->err, "\n" if $conn->err;
    print "SQL-Fehler: ", $conn->errstr, "\n" if $conn->err;
    print "Zeilen im Resultset: ", $zeilen, "\n";
    print "Anzahl von Spalten: ", $sth->{NUM_OF_FIELDS}, "\n";
    my ( $name, $sal );
    $sth->bind_col(1, \$name);
    $sth->bind_col(2, \$sal);
    while ($sth->fetch) {
        print "Name = ", $name, ", Salary = ", $sal, "\n";
    }
    $sth->finish;
}

my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
    "postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";

printf "Status: %s\n", $conn->state || "OK";
runQueryBind($conn, "SELECT name, salary FROM employees");
$conn->disconnect;

. . .
$ perl bind3.pl
Status: OK
SELECT name, salary FROM employees
Zeilen im Resultset: 2
Anzahl von Spalten: 2
Name = Max Mustermann, Salary = 32600
Name = Not Sure, Salary = 11400

```



Praxistipp

Perl DBI-Programme können zur Laufzeit mit mehreren Datenbanksystemen gleichzeitig verbunden sein. Das ist zum Beispiel nützlich bei Migrationen oder Datenübertragungen zwischen PostgreSQL und Oracle oder anderen Datenbanksystemen.

Im Beispielprogramm in Listing 8.37 wird mit Hilfe von „DBMS_METADATA“ eine SQL-Anweisung zum Erstellen einer Tabelle aus der Oracle-Datenbank ausgelesen und in der PostgreSQL-Datenbank angewandt.

Listing 8.37 Parallele Verbindung für PostgreSQL und Oracle

```
#!/usr/bin/perl
use DBI;
use strict;
my $conn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status PostgreSQL Connection: %s\n", $conn->state || "OK";
my $oraconn = DBI->connect("DBI:Oracle:OEM","hr","hr");
printf "Status Oracle Connection: %s\n", $oraconn->state || "OK";
$oraconn->{LongReadLen} = 512 * 1024;
my $retval = "";
my $sqlcommand="";
my $sth = $oraconn->prepare(q{
    BEGIN
    DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_
TRANSFORM,'STORAGE',false);
    DBMS_METADATA.SET_TRANSFORM_PARAM(TRANSFORM_HANDLE=>DBMS_METADATA.SESSION_
TRANSFORM,name=>'SEGMENT_ATTRIBUTES', value=>false);
    END;});
$sth->execute;
$sth = $oraconn->prepare("SELECT
dbms_metadata.get_ddl('TABLE','EMPLOYEES','HR') FROM dual");
$sth->execute;
$sth->bind_col(1, \ $retval);
while ($sth->fetch) {
    print $retval, "\n";
    $sqlcommand = $sqlcommand + $retval;
}
print $sqlcommand;
# . . . Umwandlung DDL . . .
my $sthpost = $conn->prepare($sqlcommand);
$sthpost->execute;
$conn->disconnect;
$ perl trans.pl
Status PostgreSQL Connection: 25P01
Status Oracle Connection: OK
CREATE TABLE "HR"."EMPLOYEES"
(
    "EMPLOYEE_ID" NUMBER,
    "NAME" VARCHAR2(100),
    "STRASSE" VARCHAR2(20),
    SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS,
    SUPPLEMENTAL LOG DATA (UNIQUE INDEX) COLUMNS,
```



```

SUPPLEMENTAL LOG DATA (FOREIGN KEY) COLUMNS
)
0

```

Eine parallele Verbindung zwischen Oracle- und PostgreSQL-Datenbank ist aber auch für die Übertragung oder Migration von Daten vorteilhaft. Man benötigt keinen zusätzlichen Platz auf der Disk, und die Performance wird im Wesentlichen durch die Netzwerk-Kapazität bestimmt. Sie müssen nicht das langsame Medium „Disk“ verwenden. Im Beispielprogramm in Listing 8.38 wird eine Tabelle mit 1 Million Sätze migriert (ca. 150 Mbyte). Die Laufzeit betrug 1:53 min.

Listing 8.38 Migration einer Tabelle mit zwei parallelen Datenbankverbindungen

```

#!/usr/bin/perl
use DBI;
use strict;
my $pgconn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status PostgreSQL Connection: %s\n", $pgconn->state || "OK";
my $oraconn = DBI->connect("DBI:Oracle:OEM","system","manager");
printf "Status Oracle Connection: %s\n", $oraconn->state || "OK";
$oraconn->{LongReadLen} = 512 * 1024;
my $col1 = "";
my $col2 = "";
my $sqlcommand = "";
my $sthora = $oraconn->prepare("SELECT id,text FROM kb");
my $sthpost = $pgconn->prepare("INSERT INTO kb VALUES(?,?)");
$sthora->execute;
$sthora->bind_col(1, \$col1);
$sthora->bind_col(2, \$col2);
while ($sthora->fetch) {
    $sthpost->bind_param_inout(1, \$col1, 10);
    $sthpost->bind_param_inout(2, \$col2, 30);
    $sthpost->execute;
}
$sthpost = $pgconn->prepare("COMMIT");
$sthpost->execute;
$pgconn->disconnect;

```

Das Skript in Listing 8.38 führt eine Einzelsatz-Verarbeitung durch. In einer Schleife wird mit Hilfe einer Fetch-Funktion jeder Satz einzeln aus dem Resultset gelesen, die Werte werden den Bindevariablen für die INSERT-Anweisung zugeordnet, und die Anweisung wird ausgeführt. Aus anderen Sprachen ist bekannt, dass Bulk-Operationen für die Verarbeitung von Massendaten wesentliche Performance-Vorteile bringen. Die DBI-Module halten zwar solche Funktionen bereit, allerdings erreicht man damit keinen Performance-Gewinn. Einige Operationen sind sogar geringfügig langsamer. Wir wollen diese Funktionen trotzdem erläutern, weil sie die Programmierung vereinfachen und Programme flexibler gestalten.

Mit Hilfe der Funktion „fetchall_arrayref“ kann das gesamte Resultset in ein Array geladen und von dort aus weiterverarbeitet werden. In Listing 8.39 wird diese Methode anstelle einzelnen Fetchens von Datensätzen verwendet.

Listing 8.39 Migration einer Tabelle mit „Bulk Load“

```
#!/usr/bin/perl
use DBI;
use strict;
my $pgconn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
printf "Status PostgreSQL Connection: %s\n", $pgconn->state || "OK";
my $oraconn = DBI->connect("DBI:Oracle:OEM","system","manager");
printf "Status Oracle Connection: %s\n", $oraconn->state || "OK";
$oraconn->{LongReadLen} = 512 * 1024;
my $sth = $oraconn->prepare("SELECT id,text FROM kb");
my $sth2 = $pgconn->prepare("INSERT INTO kb VALUES(?,?)");
$sth->execute();
my $array_ref = $sth->fetchall_arrayref();
foreach my $row(@$array_ref) {
    my ($id, $text) = @$row;
    $sth2->bind_param_inout(1, \$id, 10);
    $sth2->bind_param_inout(2, \$text, 30);
    $sth2->execute;
}
$sth2 = $pgconn->prepare("COMMIT");
$sth2->execute;
$pgconn->disconnect;
```

Während das Programm in Listing 8.38 eine Laufzeit von 1:53 Minuten auswies, benötigte das Programm in Listing 8.39 1:59 Minuten. Es konnte also keine Verbesserung der Laufzeit erreicht werden.

**Praxistipp**

Beachten Sie, dass die Funktion „fetchall_arrayref“ das Array im Hauptspeicher aufbaut. Stellen Sie sicher, dass beim Lesen von großen Tabellen genügend Speicher zur Verfügung steht.

Eine weitere Vereinfachung in der Programmierung lässt sich mit Hilfe der Funktion „execute_for_fetch“ erreichen. Die SELECT-Anweisung wird als Unterprogramm definiert; „execute_for_fetch“ führt die INSERT-Anweisungen durch und übernimmt das Binden der von der SELECT-Anweisung gelieferten Werte. Listing 8.40 präsentiert ein Beispiel. Die Laufzeit lag hier bei 2:05 Minuten, also wurde keine Performance-Verbesserung erreicht. Die Programmierung vereinfachte sich gegenüber den vorherigen Programmen ein weiteres Mal.

Listing 8.40 Migration einer Tabelle mit „execute_for_fetch“

```
#!/usr/bin/perl
use DBI;
use strict;
my $pgconn = DBI->connect("DBI:Pg:dbname=postgres;host=localhost;port=5432",
"postgres", "post1234",{ AutoCommit => 0}) || die "Fehler $DBI::err
[$DBI::errstr]";
```



```

printf "Status PostgreSQL Connection: %s\n", $pgconn->state || "OK";
my $oraconn = DBI->connect("DBI:Oracle:OEM","system","manager");
printf "Status Oracle Connection: %s\n", $oraconn->state || "OK";
$oraconn->{LongReadLen} = 512 * 1024;
my $sth = $oraconn->prepare("SELECT id,text FROM kb");
$sth->execute();
my $sth2 = $pgconn->prepare("INSERT INTO kb VALUES(?,?)");
my $fetch_tuple_sub = sub { $sth->fetchrow_arrayref };
my @tuple_status;
my $rc = $sth2->execute_for_fetch($fetch_tuple_sub, \@tuple_status);
$sth2 = $pgconn->prepare("COMMIT");
$sth2->execute;
$pgconn->disconnect;

```

Nachdem Sie beide Optionen zur Einbindung von Perl kennengelernt haben, können Sie für eine von beiden entscheiden. Während Perl als integrierte Sprache des PostgreSQL-Cluster die Vorteile der Integration nutzen kann, überzeugt DBI mit komfortablen Funktionen und einem hohen Grad an Datenbank-Unabhängigkeit. Allerdings spricht auch nichts dagegen, mit beiden Optionen zu arbeiten.

8.2.1.2 PL/pgSQL

PL/pgSQL ist die prozedurale Sprache von PostgreSQL – eine ladbare Sprache, die für Applikationen entwickelt wurde, die mit in der Datenbank gespeicherten Funktionen arbeiten wollen. Seit Version 9 wird diese Sprache automatisch installiert. Eine in PL/pgSQL geschriebene Funktion besteht aus einem Block und jeder Block aus einer Erklärungs-Sektion und einer Ausführungssektion. Der Ausführungsteil wird (wie im Beispiel in Listing 8.41 durch „BEGIN“ und „END“ eingeschlossen. Blöcke können geschachtelt, und innerhalb eines Blocks lässt sich ein weiterer Block einbinden.

Listing 8.41 Die Struktur von PL/pgSQL

```

(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION beispiel()
  RETURNS integer AS $$
> DECLARE
> var INTEGER := 30;
> BEGIN
>   var := 10;
>   RAISE NOTICE 'Der Wert ist %', var;
>   RETURN var;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT beispiel();
NOTICE:  Der Wert ist 10
beispiel
-----
      10

```

Alle im Ausführungsteil des Blocks verwendeten Variablen müssen im Deklarationsteil erklärt werden. Einzige Ausnahme bildet die Iterationsvariable in einer „FOR-LOOP“-Anweisung. Variablen in PL/pgSQL können alle definierten SQL-Datentypen annehmen. Es ist möglich, Variablen bei der Deklaration Initialwerte zuzuweisen. Konstanten werden mit dem Schlüsselwort „CONSTANT“ gekennzeichnet. In Listing 8.42 finden Sie einige Beispiele.

Listing 8.42 Beispiel für die Deklaration von Variablen in PL/pgSQL

```
CREATE FUNCTION bsp_dekl() RETURNS text AS $$
DECLARE
    const CONSTANT INTEGER := 5;
    var1 INTEGER NOT NULL := 10;
    var2 CHAR DEFAULT 'X';
BEGIN
    return var2;
END;
$$ LANGUAGE 'plpgsql';
```

Häufig verwendete Datentypen in PL/pgSQL-Programmen:

- CHAR
- TEXT
- INTEGER
- DOUBLE PRECISION
- DATE
- TIME
- BOOLEAN

Mit der „SELECT INTO“-Anweisung können SQL-Befehle direkt abgesetzt und das Ergebnis Variablen zugeordnet werden.

Listing 8.43 Der Befehl SELECT INTO

```
(postgres@[local]:5432) [hanser] > CREATE FUNCTION get_emp_name (integer)
RETURNS text AS $$
> DECLARE
>   emp_id ALIAS FOR $1;
>   v_first TEXT;
>   v_last TEXT;
> BEGIN
>   SELECT INTO v_first, v_last
>   first_name, last_name
>   FROM employees WHERE employee_id = emp_id;
>   RETURN v_first || ' ' || v_last;
> END;
> $$ LANGUAGE 'plpgsql';
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT get_emp_name('100');
get_emp_name
-----
Steven King
```

Mit Hilfe von Attributen lässt sich die Programmierung vereinfachen. So liefert „%TYPE“ den Typ eines Objekts, zum Beispiel einer Spalte. „%ROWTYPE“ steht für die Struktur, die einer Tabellenzeile entspricht. Der Vorteil bei der Verwendung von Attributen liegt nicht nur darin, dass man programmieren kann, ohne die Datentypen im Detail kennen zu müssen. Änderungen in der Struktur der verwendeten Tabelle bedingen keine Anpassung des Programmcodes. Im Beispiel in Listing 8.44 werden die Attribute verwendet.

Listing 8.44 Verwendung von Attributen in PL/pgSQL

```

(postgres@[local]:5432) [hanser] > CREATE FUNCTION get_emp_id (varchar,
varchar) RETURNS int AS $$
> DECLARE
> v_firstname ALIAS FOR $1;
> v_lastname ALIAS FOR $2;
> rec_emp employees%ROWTYPE;
> BEGIN
> SELECT INTO rec_emp * FROM employees WHERE first_name = v_firstname
> AND last_name = v_lastname;
> RETURN rec_emp.employee_id;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT get_emp_id('Steven','King');
get_emp_id
-----
100

```

Eine Variable vom Typ *RECORD* ist identisch mit einer Variablen, die als „%ROWTYPE“ definiert wurde, nur dass sie keine vordefinierte Struktur besitzt. Die Struktur wird bei der ersten Verwendung durch eine SELECT- oder FOR-Anweisung definiert. Es handelt sich also um keinen realen Datentyp, sondern um einen Platzhalter.

Wie andere prozedurale Sprachen verfügt auch PL/pgSQL über Anweisungen, die den Programmfluss steuern. Die erste ist die einfache Verzweigung mit „IF“. Der ELSE-Zweig ist optional und kann weggelassen werden:

```

IF <Bedingung> THEN
    <Anweisungen>;
ELSE
    <Anweisungen>;
END IF;

```

In der einfachen LOOP-Anweisung gibt es eine Abbruchbedingung. Die Schleife wird verlassen, wenn diese erfüllt ist:

```

LOOP
    <Anweisungen>;
    EXIT WHEN <Bedingung>;
END LOOP;

```

Beim WHILE-LOOP wird die Schleife verlassen, wenn die Bedingung nicht mehr erfüllt ist:

```

WHILE <Bedingung> LOOP
    <Anweisungen>;
END LOOP;

```

Die FOR-Schleife gestattet sowohl die Abarbeitung in einem Zahlenbereich als auch das Durchlaufen des Cursors einer SELECT-Anweisung:

```

FOR var IN 1..100000 LOOP
    <Anweisungen>;
END LOOP;
FOR rec_var IN <SELECT-Anweisung> LOOP
    <Anweisungen>;
END LOOP;

```

Listing 8.45 liefert ein Beispiel für eine Schleife durch eine SELECT-Anweisung (Cursor).

Listing 8.45 Beispiel für eine LOOP-Anweisung (Cursor)

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION
get_phonenumber() RETURNS text AS $$
> DECLARE
>   text_output TEXT := E'\n';
>   row_emp employees%ROWTYPE;
> BEGIN
>   FOR row_emp IN SELECT * FROM employees LOOP
>     text_output := text_output || row_emp.last_name || ':' ||
row_emp.phone_number || E'\n';
>   END LOOP;
>   RETURN text_output;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT get_phonenumber();
get_phonenumber
-----
+
King:515.123.4567 +
Kochhar:515.123.4568 +
De Haan:515.123.4569 +
```

Extrem wichtig für eine Datenbanksprache ist die Möglichkeit der Fehlerbehandlung. Der RAISE-Befehl sendet ein spezielles Signal an den PostgreSQL-Server, der eine Fehlerbehandlung auslöst. Die Syntax ist: RAISE <Fehler-Level> 'Nachricht';. Die folgenden Fehler-Level können ausgelöst werden:

- **DEBUG:** Die Nachricht wird in die PostgreSQL-Logdatei geschrieben und an den Client übertragen. Die Nachricht wird ignoriert, wenn die Datenbank im Produktionsmodus läuft.
- **NOTICE:** Die Nachricht wird in die PostgreSQL-Logdatei geschrieben und an den Client übertragen, unabhängig vom Operationsmodus der Datenbank.
- **EXCEPTION:** Die Nachricht wird in die PostgreSQL-Logdatei geschrieben und an den Client übertragen; die Transaktion wird abgebrochen.

In Listing 8.46 finden Sie ein Beispiel für die Verwendung des RAISE-Befehls.

Listing 8.46 Der RAISE-Befehl

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION err_test ()
RETURNS integer AS $$
> DECLARE
>   var INTEGER = 1;
> BEGIN
>   RAISE DEBUG 'Start der Funktion...';
>   var = var + 1;
>   RAISE NOTICE 'Variable ist: %',var;
>   RAISE EXCEPTION 'Transaction abgebrochen';
>   RETURN 0;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT err_test();
NOTICE: Variable ist: 2
ERROR: Transaction abgebrochen
```


In PL/pgSQL-Blöcken ist es möglich, dynamische Kommandos zu verwenden. Die Anweisung wird erst zur Laufzeit gebildet. Die Syntax lautet:

```
EXECUTE <Kommando> [ INTO [STRICT] <Ziel> ] [ USING <Ausdruck> ];
```

<Kommando> ist ein Text, der den auszuführenden Befehl enthält. Die Angabe eines <Ziel> ist optional und kann eine Zeilenvariable, eine Liste von Variablen oder ein RECORD-Typ sein. In der Option „USING“ können Werte stehen, die <Kommando> verwendet. Mit der Option „STRICT“ wird ein Fehler generiert, wenn nicht genau ein Satz zurückgegeben wird. In Listing 8.47 finden Sie ein Beispiel für dynamisches SQL. Der Wert in der WHERE-Klausel wird zur Laufzeit aus einer Steuertabelle ermittelt und durch die Option „USING“ übergeben.

Listing 8.47 Ein PL/pgSQL-Block mit dynamischem SQL

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION cnt_test()
RETURNS VARCHAR AS $$
> DECLARE
> txt VARCHAR := '';
> var int;
> BEGIN
> SELECT INTO var * FROM ct1;
> EXECUTE 'SELECT text FROM kb WHERE id = $1' INTO txt USING var;
> RETURN txt;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT cnt_test();
cnt_test
-----
Knowledge Base - Inhaltsverzeichnis
```

Das Kommando *RETURN* haben wir bereits verwendet, um den Rückgabewert einer Funktion zu liefern. Dabei kann eine Variable oder ein Ausdruck zurückgegeben werden. Der Rückgabewert kann auch ein ganzer Satz oder das Resultset einer SQL-Abfrage, also mehrere Sätze sein. Für diese Fälle gibt es folgende Variationen des RETURN-Befehls:

```
RETURN QUERY <SELECT-Anweisung>;
RETURN QUERY EXECUTE . . .;
RETURN NEXT <Ausdruck>;
```

In Listing 8.48 finden Sie ein Beispiel für das Ergebnis einer SELECT-Anweisung als Rückgabewert einer Funktion.

Listing 8.48 Ein Resultset als Rückgabewert einer Funktion

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION fn_department
p_depart VARCHAR)
> RETURNS SETOF employees
> AS $$
> DECLARE
> v_depid INT;
> BEGIN
> SELECT INTO v_depid department_id FROM departments WHERE
> department_name = p_depart;
> RETURN QUERY
```



```

> SELECT * FROM employees
> WHERE department_id = v_depid;
> END;
> $$
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT fn_department('IT');
                                fn_department
-----
(103,Alexander,Hunold,AHUNOLD,590.423.4567,2006-01-03,IT_PROG,,,102,60)
(104,Bruce,Ernst,BERNST,590.423.4568,2007-05-21,IT_PROG,,,103,60)
(105,David,Austin,DAUSTIN,590.423.4569,2005-06-25,IT_PROG,,,103,60)
(106,Valli,Pataballa,VPATABAL,590.423.4560,2006-02-05,IT_PROG,,,103,60)
(107,Diana,Lorentz,DLORENTZ,590.423.5567,2007-02-07,IT_PROG,,,103,60)

```

Alternativ zu einer SELECT-Anweisung können Sie auch dynamisches SQL mit dem EXECUTE-Befehl verwenden, so wie Sie ihn bereits kennengelernt haben.

Wenn Sie in einer Funktion einen Cursor verwenden, können Sie mit dem „RETURN NEXT“-Befehl jeweils den aktuellen Satz zurückgeben. In Listing 8.49 finden Sie ein Beispiel.

Listing 8.49 Rückgabe von Zeilen mit „RETURN NEXT“

```

(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION
> fn_department2 (p_depart VARCHAR)
> RETURNS SETOF employees
> AS $$
> DECLARE
> v_depid INT;
> emp_row employees%ROWTYPE;
> BEGIN
> SELECT INTO v_depid department_id FROM departments WHERE
> department_name = p_depart;
> FOR emp_row IN
> SELECT * FROM employees
> LOOP
> IF emp_row.department_id = v_depid THEN
> RETURN NEXT emp_row;
> END IF;
> END LOOP;
> END;
> $$
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT fn_department2('IT');
                                fn_department2
-----
(103,Alexander,Hunold,AHUNOLD,590.423.4567,2006-01-03,IT_PROG,,,102,60)
(104,Bruce,Ernst,BERNST,590.423.4568,2007-05-21,IT_PROG,,,103,60)
(105,David,Austin,DAUSTIN,590.423.4569,2005-06-25,IT_PROG,,,103,60)
(106,Valli,Pataballa,VPATABAL,590.423.4560,2006-02-05,IT_PROG,,,103,60)
(107,Diana,Lorentz,DLORENTZ,590.423.5567,2007-02-07,IT_PROG,,,103,60)

```

SQL-Abfragen können so komplex werden, dass bestimmte Abfragen mit einer einzelnen Anweisung nicht mehr darstellbar sind. In solchen Fällen greift man auf PL/pgSQL und die Verwendung von Cursors zurück. Ein Cursor kann in einer Schleife Satz für Satz ausgewertet und weiterverarbeitet werden. Damit ergeben sich wesentlich mehr Darstellungsmöglichkeiten als in einer einzelnen SQL-Anweisung.

Implizit haben Sie bereits Cursor verwendet, nämlich in der FOR-Schleife mit einer SELECT-Anweisung. Diese Form eines Cursor wird deshalb impliziter Cursor genannt. Ein expliziter Cursor wird im Deklarationsteil des PL/pgSQL-Blocks erklärt. Er muss im Verarbeitungsteil explizit geöffnet und geschlossen werden. In einer Schleife sorgt ein FETCH-Befehl dafür, dass der nächste Satz gelesen wird. Im Beispiel in Listing 8.50 wird ein Cursor explizit erklärt, geöffnet und in einer Schleife durchlaufen. Die Abbruchbedingung ist durch „EXIT WHEN NOT FOUND“ definiert. Damit wird die Schleife verlassen, wenn kein Satz mehr gefunden wird, das heißt, das Ende des Cursors erreicht ist.

Listing 8.50 Verwendung eines expliziten Cursors in PL/pgSQL

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION f_salary()
> RETURNS VOID AS $$
> DECLARE
>   v_empid employees.employee_id%TYPE;
>   v_first employees.first_name%TYPE;
>   v_last  employees.last_name%TYPE;
>   v_sal   employees.salary%TYPE;
>   c_emp CURSOR FOR
>   SELECT employee_id, first_name, last_name, salary FROM employees;
> BEGIN
>   OPEN c_emp;
>   LOOP
>     FETCH c_emp INTO v_empid, v_first, v_last, v_sal;
>     EXIT WHEN NOT FOUND;
>     IF v_sal > 5000 THEN
>       INSERT INTO salary VALUES (v_empid, v_first, v_last, v_sal);
>     END IF;
>   END LOOP;
>   CLOSE c_emp;
> END;
> $$ LANGUAGE plpgsql;
(postgres@[local]:5432) [hanser] > SELECT f_salary();
(postgres@[local]:5432) [hanser] > SELECT * FROM salary;
 emp_id | first_name | last_name | salary
-----+-----+-----+-----
    100 | Steven    | King      | 24000.00
    101 | Neena     | Kochhar   | 17000.00
    102 | Lex       | De Haan   | 17000.00
. . .
```

Cursor können von einer Funktion zurückgegeben werden. Man spricht dann von einem *Ref Cursor*. Der Ref Cursor lässt sich vom aufrufenden Programm verarbeiten.

Listing 8.51 Einen Ref Cursor in PL/pgSQL verwenden

```
(postgres@[local]:5432) [hanser] > CREATE FUNCTION f_ref(refcursor)
> RETURNS refcursor
> AS $$
> BEGIN
>   OPEN $1 FOR SELECT department_name, manager_id FROM departments;
>   RETURN $1;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > BEGIN;
```

```

BEGIN
(postgres@[local]:5432) [hanser] *> SELECT f_ref('testcursor');
      f_ref
-----
testcursor
(postgres@[local]:5432) [hanser] *> FETCH ALL IN testcursor;
 department_name | manager_id
-----+-----
Administration   |          200
Marketing         |          201
Purchasing       |          114
. . .
(postgres@[local]:5432) [hanser] *> COMMIT;
COMMIT

```



Praxistipp

Für die Verwendung von Anführungszeichen im PL/pgSQL-Block gelten spezielle Regeln. Diese sind in den folgenden Beispielen beschrieben.

Wenn Sie das doppelte \$-Zeichen (\$\$) als Begrenzer für den PL/pgSQL-Block verwenden, können im Block einzelne Hochkommas als Begrenzer für Konstanten verwendet werden. Das Programm in Listing 8.52 zeigt ein Beispiel.

Listing 8.52 Verwendung von Hochkommas als Begrenzer für Strings

```

(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION simple()
> RETURNS char AS $$
> BEGIN
> RETURN 'return_message';
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > select simple();
      simple
-----
return_message

```

Bei Verwendung eines einfachen Hochkommas (') als Begrenzer für den PL/pgSQL-Block müssen Hochkommas innerhalb des Blocks doppelt geschrieben werden (siehe Listing 8.53).

Listing 8.53 Doppelte Hochkommas als String-Begrenzer

```

(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION simple2()
> RETURNS char AS '
> BEGIN
> RETURN ''return_message'';
> END;
> ' LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT simple2();
      simple2
-----
return_message

```


Für Hochkommas innerhalb eines Strings, der selbst mit einfachen Hochkommas begrenzt ist, muss, wie in Listing 8.54 dargestellt, eine weitere Verdoppelung der Hochkommas erfolgen.

Listing 8.54 Hochkommas innerhalb eines Strings

```
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION simple3()
> RETURNS char AS '
> DECLARE
> stmt CHAR(100);
> BEGIN
> stmt := 'SELECT usesysid FROM pg_user WHERE '
> stmt := stmt || ' username = ''''postgres''''';
> RETURN stmt;
> END;
> ' LANGUAGE plpgsql;
CREATE FUNCTION
(postgres@[local]:5432) [hanser] > SELECT simple3();
                                     simple3
-----
SELECT usesysid FROM pg_user WHERE username = 'postgres';
```

Seit der Version 9 erlaubt PL/pgSQL anonyme Blöcke. Damit wird eine einfache Ausführung von PL/pgSQL-Programmen ermöglicht, ohne dass diese in der Datenbank gespeichert und ein Name vergeben werden müssen. Die Syntax erklärt das Beispiel in Listing 8.55.

Listing 8.55 Ein anonymer PL/pgSQL-Block

```
(postgres@[local]:5432) [postgres] > DO LANGUAGE plpgsql $$
> DECLARE
> v_rec RECORD;
> BEGIN
> FOR v_rec IN SELECT tablename FROM pg_tables
> WHERE schemaname <> 'pg_catalog'
> AND schemaname <> 'information_schema'
> LOOP
> EXECUTE 'GRANT UPDATE ON ' || v_rec.tablename || ' TO app_user';
> END LOOP;
> END
> $$;
DO
```

Den RAISE-Befehl zum Erzeugen von Informationen oder Fehlermeldungen haben Sie bereits kennengelernt. Die Behandlung von Fehlern gliedert sich in die Struktur des PL/pgSQL-Blocks ein und wird durch die Klausel „EXCEPTION“ eingeleitet. Tritt ein Fehler auf, oder wird ein Fehler mit dem RAISE-Befehl erzeugt, springt das Programm in den EXCEPTION-Teil. Innerhalb des Programmcodes können einzelne Fehler-Bedingungen mit der WHEN-Klausel behandelt werden. Die Syntax finden Sie in Listing 8.56.

Listing 8.56 Syntax für die EXCEPTION-Klausel in PL/pgSQL

```
[ DECLARE
. . . ]
BEGIN
. . .
EXCEPTION
```

```

    WHEN bedingung THEN
    . . .
    [ WHEN bedingung THEN
      . . . ]
    . . .
END;
```



Hinweis

Eine komplette Liste der Exceptions (Fehler-Codes) finden Sie in der offiziellen PostgreSQL-Dokumentation.

Listing 8.57 Beispiel für eine Exception-Behandlung

```

(postgres@[local]:5432) [postgres] > DO LANGUAGE plpgsql $$
> DECLARE
> v_i BIGINT;
> BEGIN
> UPDATE stock SET s_quantity = s_quantity * 2 WHERE s_i_id = 900000000000;
> EXCEPTION
>   WHEN no_data_found THEN
>     RAISE NOTICE 'Datensatz nicht gefunden,';
> END;
> $$;
DO
```

8.2.2 Trigger

Trigger können in den verfügbaren prozeduralen Sprachen PL/pgSQL, PL/Tcl, PL/Perl, PL/Python oder auch in C geschrieben werden. Darin unterscheidet sich PostgreSQL von anderen Datenbank-Systemen, bei denen die Sprachauswahl stark eingeschränkt ist. Sicher ist es zu empfehlen, einen Standard-Trigger in PL/pgSQL zu schreiben. Allerdings gibt es Situationen, bei denen die Performance eine große Rolle spielt. Dann bietet die Möglichkeit, zum Beispiel mit „C“ zu arbeiten, eine gute Lösung für das Problem.

Betrachten wir zunächst das Vorgehen zum Schreiben eines Triggers mit PL/pgSQL. Im ersten Schritt muss eine Trigger-Funktion geschrieben werden. Dies erfolgt mit dem CREATE FUNCTION-Befehl wobei der Rückgabotyp „TRIGGER“ angegeben werden muss. Ist die Triggerfunktion in der Datenbank gespeichert, können Sie den Trigger mit dem CREATE TRIGGER-Befehl unter Bezugnahme auf die Funktion erstellen. Für eine Trigger-Funktion hält PostgreSQL eine Reihe von vordefinierten Standardvariablen bereit, die Sie in Tabelle 8.7 finden.

Eine Triggerfunktion muss entweder den Typ RECORD, der der Satzstruktur der Tabelle entspricht, oder den Wert „NULL“ zurückliefern. Wenn man Trigger mit der Option „BEFORE“ feuert, werden Änderungen von Werten durch den Trigger berücksichtigt und die geänderten Werte in die Tabelle eingetragen. Liefert der Trigger den Rückgabewert „NULL“, wird der Satz unverändert eingetragen. Ein Trigger mit der Option „AFTER“ kann

keine Änderungen des Satzes mehr vornehmen. Allerdings ist er in der Lage, die Operation durch das Hervorrufen einer Fehlerbedingung abubrechen (RAISE-Befehl).

Tabelle 8.7 Vordefinierte Standardvariablen

Variable	Typ	Bedeutung
NEW	RECORD	Datensatz von Typ RECORD. Enthält die neuen Werte bei INSERT- oder UPDATE-Anweisungen. Für DELETE-Anweisung ist der Wert „NULL“.
OLD	RECORD	Datensatz von Typ RECORD. Enthält die alten Werte bei INSERT- oder UPDATE-Anweisungen. Für DELETE-Anweisung ist der Wert „NULL“.
TG_NAME	NAME	Name des gefeuerten Triggers.
TG_WHEN	TEXT	Enthält „BEFORE“ oder „AFTER“ je nach Verwendung des Triggers.
TG_LEVEL	TEXT	Enthält entweder „ROW“ oder „STATEMENT“, je nach Definition des Triggers.
TG_OP	TEXT	Enthält die gewählte Operation. Mögliche Werte sind INSERT, UPDATE, DELETE oder TRUNCATE.
TG_RELID	OID	Die Objekt-ID der betroffenen Tabelle.
TG_TABLE_NAME	NAME	Name der betroffenen Tabelle.
TG_TABLE_SCHEMA	NAME	Name des zugehörigen Schemas.
TG_NARGS	INT	Anzahl der Argumente in der CREATE TRIGGER-Anweisung.
TG_ARGV	Array of Text	Argumente der CREATE TRIGGER-Anweisung.

Ein Einsatzgebiet für Trigger ist die Überwachung von Operationen an ausgewählten Tabellen. Der Trigger in Listing 8.58 ist ein entsprechendes Beispiel. Zuerst wird die Audit-Tabelle zum Speichern der Operationen, danach die Trigger-Funktion sowie der Trigger selbst erstellt. Da keine Veränderungen an den Werten der Tabelle vorgenommen werden sollen, wird ein Trigger mit der Option „AFTER“ erstellt.

Listing 8.58 Trigger zur Überwachung von Operationen an einer Tabelle

```
((postgres@[local]:5432) [hanser] > CREATE TABLE audit_employees(
> op char(1),
> datum timestamp,
> userid text,
> sal numeric(8,2));
CREATE TABLE
(postgres@[local]:5432) [hanser] > CREATE OR REPLACE FUNCTION
trig_employees() RETURNS TRIGGER
> AS $$
> BEGIN
> IF (TG_OP = 'INSERT') THEN
> INSERT INTO audit_employees SELECT 'I',now(),user,NEW.salary;
> RETURN NEW;
> ELSIF (TG_OP = 'UPDATE') THEN
> INSERT INTO audit_employees SELECT 'U',now(),user,NEW.salary;
```



```

> ELSIF (TG_OP = 'DELETE') THEN
> INSERT INTO audit_employees SELECT 'U',now(),user,OLD.salary;
> END IF;
> RETURN NULL;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
> AFTER INSERT OR UPDATE OR DELETE ON employees
> FOR EACH ROW EXECUTE PROCEDURE trig_employees();
CREATE TRIGGER
(postgres@[local]:5432) [hanser] > UPDATE employees SET salary=3000 WHERE
employee_id = 131;
UPDATE 1
(postgres@[local]:5432) [hanser] > SELECT * FROM audit_employees;
  op |                datum                | userid |  sal
-----+-----+-----+-----
  U  | 2011-06-04 14:16:15.899627 | postgres | 3000.00

```

Ein weiteres Einsatzgebiet für Trigger sind Summentabellen. Sie reduzieren den Ressourcenverbrauch für spezielle Abfragen.



Praxistipp

Beachten Sie, dass die im Trigger definierten Operationen ausgeführt werden müssen, bevor die Datenbank die Steuerung an den Client zurückgibt. Ein falscher oder übermäßiger Einsatz von Triggern kann negative Auswirkungen auf die Performance haben.

8.2.3 SQL erweitern

PostgreSQL speichert, wie andere relationale Datenbanksysteme auch, Informationen über Datenbank-Objekte im Data Dictionary, einer Sammlung von Metadaten des Clusters. Die Pflege erfolgt automatisch, d.h. wenn Sie eine Tabelle mit einem „CREATE TABLE“-Befehl erstellen, erscheint sie direkt im Data Dictionary in der Tabelle „pg_tables“. Im Unterschied zu anderen Datenbank-Systemen speichert PostgreSQL jedoch wesentlich mehr Informationen, zum Beispiel über Datentypen, Funktionen oder Zugriffsmethoden. Darüber hinaus können die Tabellen im Data Dictionary verändert werden. Da PostgreSQL alle Operationen basierend auf den Informationen des Data Dictionary ausführt, besteht die Möglichkeit einer SQL-Erweiterung durch den Anwender. Eine solche SQL-Erweiterung ist zum Beispiel in einem Oracle-Datenbanksystem nicht möglich. Dort muss eine zusätzliche Funktionalität über PL/SQL erreicht werden, was gewisse Nachteile beinhaltet.

Die Erweiterung der SQL-Funktionalität ist ein mächtiges Feature, das in dieser Form in keinem anderen relationalen Datenbanksystem zu finden ist. Um dies zu verdeutlichen, schauen Sie sich am besten die Möglichkeiten der Erweiterung an den folgenden Beispielen an.

8.2.3.1 SQL-Funktionen

Die Möglichkeit der Einbindung von Funktionen haben Sie bereits bei der Betrachtung anderer Programmiersprachen, wie zum Beispiel PL/pgSQL oder C kennengelernt. Bisher haben wir nicht erwähnt, dass eine dieser Sprachen „SQL“ ist. Diese Sprache ist natürlich immer standardmäßig vorgeladen, und es bedarf keines Nachladens, wie wir es für „Perl“ getan haben.

Listing 8.59 „SQL“ als Standardsprache im Cluster

```
(postgres@:5432) [hanser] > SELECT lanname, lanpltrusted, lanispl
> FROM pg_language;
lanname | lanpltrusted | lanispl
-----+-----+-----
internal | f             | f
c        | f             | f
sql      | t             | f
plpgsql  | t             | t
```

Der Körper einer SQL-Funktion besteht aus einer durch Semikolon separierten Liste von SQL-Anweisungen. Dabei muss die letzte SQL-Anweisung stets eine RETURNING-Klausel enthalten und eine SELECT-, INSERT-, UPDATE- oder DELETE-Anweisung sein. Anweisungen zur Transaktionskontrolle, wie zum Beispiel „COMMIT“ oder „VACUUM“, sind in SQL-Funktionen nicht zulässig.



Hinweis

Das Semikolon hinter der letzten SQL-Anweisung einer SQL-Funktion ist optional.

Für SQL-Funktionen, die keine Werte zurückliefern sollen, kann der Rückgabewert „void“ definiert werden. Eine einfache Funktion finden Sie in Listing 8.60. Ohne die Verwendung einer speziellen Programmiersprache ist es also möglich, SQL-Funktionen zu erstellen. Diese sind Bestandteil des Datenbank-Clusters und zeichnen sich durch einfache und transparente Bearbeitung sowie sehr gute Performancewerte aus. SQL-Funktionen finden zum Beispiel einen breiten Einsatz bei PostGIS (siehe Kapitel 11).

Listing 8.60 Eine SQL-Funktion mit Rückgabewert „void“

```
(postgres@:5432) [postgres] > CREATE FUNCTION cl_prod()
> RETURNS void
> AS $$
> DELETE FROM products WHERE price < 1.0;
> $$
> LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT cl_prod();
cl_prod
-----
(1 Zeile)
```

So wie bei anderen Sprachen muss der Funktionskörper als String-Konstante dargestellt werden. Man verwendet häufig das Zeichen „\$\$“. Es ist aber auch möglich, den String in ein einfaches Hochkomma einzuschließen. In diesem Fall müssen einfache Hochkommas und Backslash innerhalb des Strings verdoppelt werden. Im Funktionskörper können von den Platzhaltern „\$1“, „\$2“ bis „\$n“ dargestellte Argumente verwendet werden. Die folgenden Beispiele verdeutlichen, welche Möglichkeiten beim Einsatz von SQL-Funktionen bestehen. In Listing 8.61 finden Sie eine Funktion, die einen einzelnen Rückgabewert definiert.

Listing 8.61 SQL-Funktion mit Rückgabe eines einzelnen Wertes

```
(postgres@:5432) [postgres] > CREATE FUNCTION max_prod_price()
> RETURNS numeric
> AS
> $$
> SELECT MAX(price) FROM products;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT max_prod_price();
 max_prod_price
-----
                29.99
(1 Zeile)
```

Die SQL-Funktion in Listing 8.62 verwendet Parameter für eine UPDATE-Anweisung. Die Typen werden im Definitionsteil festgelegt. In den SQL-Anweisung werden die Parameter durch die Platzhalter „\$1“, „\$2“, usw. ersetzt. Zusätzlich wird noch eine SELECT-Anweisung angefügt, die das Ergebnis der UPDATE-Anweisung zurückliefert.

Listing 8.62 SQL-Funktion mit Parametern und Ergebnistrückgabe

```
(postgres@:5432) [postgres] > CREATE FUNCTION fix_products(integer, numeric)
> RETURNS numeric
> AS $$
> UPDATE products SET price = price - $2
> WHERE prod_id = $1;
> SELECT price FROM products WHERE prod_id = $1;
> $$ LANGUAGE SQL;
(postgres@:5432) [postgres] > SELECT fix_products(1, 5.0);
 fix_products
-----
                20.99
(1 Zeile)
```

Derselbe Effekt lässt sich mit einer einzelnen UPDATE-Anweisung erzielen. Mit der Option „RETURNING“ wird ebenfalls das Ergebnis zurückgeliefert.

Listing 8.63 SQL-Funktion mit einer RETURNING-Option

```
(postgres@:5432) [postgres] > CREATE FUNCTION fix_products_2(integer, numeric)
> RETURNS numeric
> AS $$
> UPDATE products SET price = price - $2
> WHERE prod_id = $1
> RETURNING price;
> $$ LANGUAGE SQL;
```



```
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT fix_products_2(1, 3.0);
fix_products_2
-----
          17.99
(1 Zeile)
```

In SQL-Funktionen können anstelle von einfachen Datentypen auch Sätze verwendet werden. Die Funktion in Listing 8.64 liefert einen Record als Rückgabewert. In der ersten Abfrage erhalten Sie einen einzelnen, zusammengesetzten Wert zurück, mit der zweiten Abfrage einen Record. Die Funktion kann man in dieser Form auch für eine INSERT-Anweisung verwenden.

Listing 8.64 SQL-Funktion mit Record als Rückgabewert

```
(postgres@:5432) [postgres] > CREATE FUNCTION add_product()
> RETURNS products
> AS $$
> SELECT ROW(999999,0,'Leer','Leer',20.0,0,0)::products;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT add_product();
add_product
-----
(999999,0,Leer,Leer,20.00,0,0)
(1 Zeile)
(postgres@:5432) [postgres] > SELECT * FROM add_product();
prod_id | category | title | actor | price | special | common_prod_id
-----+-----+-----+-----+-----+-----+-----
  999999 |         0 | Leer  | Leer  | 20.00 |        0 |              0
(1 Zeile)
(postgres@:5432) [postgres] > INSERT INTO products
> SELECT * FROM add_product();
INSERT 0 1
(postgres@:5432) [postgres] > SELECT * FROM products
> WHERE prod_id = 999999;
prod_id | category | title | actor | price | special | common_prod_id
-----+-----+-----+-----+-----+-----+-----
  999999 |         0 | Leer  | Leer  | 20.00 |        0 |              0
(1 Zeile)
```

Für Argumente (Parameter) von SQL-Funktionen können Standardwerte deklariert werden. Die Werte lassen sich mit Hilfe der Option „DEFAULT“ deklarieren.



Hinweis

Die Syntax lässt ausschließlich zu, dass Argumente nur vom Ende der Liste weggelassen werden können. Aus diesem Grund müssen alle Parameter, die einem Parameter mit Standardwert folgen, ebenfalls mit einem Standardwert erklärt werden.

Listing 8.65 SQL-Funktion mit Standardwerten für Parameter

```
(postgres@:5432) [postgres] > CREATE FUNCTION fix_products_3(category integer
DEFAULT 0)
> RETURNS numeric
> AS $$
> UPDATE products SET price = price - 1.0
> WHERE category = $1
> RETURNING price;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT fix_products_3();
fix_products_3
-----
19.00
```

In Listing 8.64 wurde eine SQL-Funktion in der FROM-Klausel einer SELECT-Anweisung verwendet. Diese Option gilt für alle SQL-Funktionen. Insbesondere ist die Verwendung einer SQL-Funktion in der FROM-Klausel hilfreich, wenn ein zusammengesetzter Wert oder ein Record zurückgegeben wird. In diesem Fall erzeugt die SQL-Funktion eine Spalte für jedes Attribut. Ein Beispiel hierzu finden Sie in Listing 8.66.

Listing 8.66 Eine SQL-Funktion als Quelle für eine Tabellenabfrage verwenden

```
(postgres@:5432) [postgres] > CREATE FUNCTION get_products(integer)
> RETURNS products
> AS $$
> SELECT * FROM products WHERE prod_id = $1;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT SUBSTR(title,1,12),LOWER(actor)
> FROM get_products(4711);
substr | lower
-----+-----
AFRICAN RAGI | matthew damon
(1 Zeile)
```

Die bislang erörterten SQL-Funktionen haben jeweils eine Zeile als Ergebnis zurückgeliefert. Das liegt an den Datentypen, die wir in der Option „RETURNING“ erläuterten. Deklariert man als Rückgabebetyp „SETOF <typ>“, wird jeder Ausgabesatz als Rückgabewert interpretiert, und es werden mehrere Sätze zurückgeliefert.

Listing 8.67 Mehrere Zeilen als Ergebnis einer SQL-Funktion zurückliefern

```
(postgres@:5432) [postgres] > CREATE FUNCTION get_products_2(integer)
> RETURNS SETOF products
> AS $$
> SELECT * FROM products WHERE prod_id < $1;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT prod_id,title,price
> FROM get_products_2(5);
prod_id | title | price
-----+-----+-----
1 | ACADEMY ACADEMY | 17.99
2 | ACADEMY ACE | 20.99
```



```

3 | ACADEMY ADAPTATION | 28.99
4 | ACADEMY AFFAIR      | 14.99
(4 Zeilen)

```

An dieser Stelle stellt sich die Frage: Was geschieht mit DML-Anweisungen in einer SQL-Funktion, die mehrere Sätze selektieren, wobei der Rückgabewert nicht mit „SETOF“ erklärt ist? Hier tut PostgreSQL das einzig Richtige. DML-Anweisungen (INSERT, UPDATE, DELETE) werden grundsätzlich komplett abgearbeitet, d.h. es werden alle betroffenen Sätze verarbeitet.

Eine weitere Möglichkeit, mehrere Spalten und Zeilen in einer SQL-Funktion zurückzugeben, bietet der Rückgabotyp „TABLE“ – gewissermaßen eine Kombination aus „SETOF“ und der Rückgabe mehrerer Spalten. Beachten Sie jedoch:



Hinweis

Die Syntax erlaubt es nicht, Ausgabeparameter vom Typ „OUT“ gemeinsam mit dem Rückgabotyp „TABLE“ zu verwenden. Alle Spalten müssen nach dem Schlüsselwort „TABLE“ aufgelistet werden (Beispiel siehe Listing 8.68).

Listing 8.68 SQL-Funktion mit Rückgabotyp „TABLE“

```

(postgres@:5432) [postgres] > CREATE FUNCTION price_and_tax(prod_id integer)
> RETURNS TABLE(prod_id int,price numeric,tax numeric)
> AS $$
> SELECT prod_id,price,price*0.19 FROM products WHERE prod_id < $1;
> $$ LANGUAGE SQL;
CREATE FUNCTION
(postgres@:5432) [postgres] > SELECT * FROM price_and_tax(5);
 prod_id | price | tax
-----+-----+-----
1 | 17.99 | 3.4181
2 | 20.99 | 3.9881
3 | 28.99 | 5.5081
4 | 14.99 | 2.8481
(4 Zeilen)

```

Die Sprache „SQL“ gestattet ein Überladen von Funktionen (Function Overloading), so wie man es von vielen Programmiersprachen her kennt. Wenn sich zwei Funktionen mit demselben Namen in Anzahl oder Typ der Parameter unterscheiden, werden sie wie unterschiedliche Funktionen behandelt. Schwierig wird es für das System, wenn aus den übergebenen Parametern nicht zu erkennen ist, welche überladenen Funktionen verwendet werden soll. Denken Sie zum Beispiel an Datentypen wie „integer“ und „small integer“. Obwohl für solche Fälle Regeln existieren, wird aus der Sicht eines guten und transparenten Programmierstils empfohlen, solche Situationen zu vermeiden.

Wie Sie sehen konnten, bietet die SQL-Erweiterung vielfältige Möglichkeiten, zusätzliche Funktionen allein mit SQL-Sprachmitteln zu erstellen. SQL-Funktionen sind direkt in die Datenbank integriert und liefern somit eine sehr gute Performance.

Hinter dem Regelsystem von PostgreSQL verbirgt sich das Neuschreiben von SQL, was in anderen Datenbanksystemen auch als „Query Rewrite“ bezeichnet wird. SQL-Anweisungen werden unter Berücksichtigung der definierten Regeln modifiziert und an den Ausführungsplaner weitergegeben. Das Regelsystem ist sehr flexibel und lässt sich für vielfältige Zwecke einsetzen.

Um das Regelsystem von PostgreSQL und seine Einsatzmöglichkeiten zu durchschauen, muss man die einzelnen Schritte des Gesamtdurchlaufs einer SQL-Anweisung verstehen. Das Regelsystem ist zwischen dem Parser und dem Ausführungsplaner platziert. Der Parser liefert einen Ausführungsplan, den das Regelsystem zusammen mit den definierten Regeln verarbeitet. Nach der Verarbeitung liefert das Regelsystem einen oder mehrere Ausführungspläne und gibt diese an den Ausführungsplaner weiter. Im Sonderfall kann das Ergebnis leer sein, sodass kein Ausführungsplan an den Planer geliefert wird.

Auch wenn Sie das Regelsystem noch nicht kennen, haben Sie es möglicherweise bereits implizit angewandt. In PostgreSQL werden Views mit Hilfe des Regelsystems implementiert. Das ist äußerst konsequent, da bei näherer Betrachtung ein View nichts anderes als eine Regel auf eine SELECT-Anweisung ist. Eine Regel wird mit dem CREATE RULE-Befehl erstellt.

Die Regeln des Regelsystems sind beliebig erweiterbar, da im Aktionsteil des Befehls beliebig viele Befehle angegeben werden können. Wenn Sie mehrere Befehle verwenden, dann sind diese in runde Klammern einzuschließen. Die Abarbeitung erfolgt in der vorgegebenen Reihenfolge.



Hinweis

Beachten Sie: Wenn man mehrere Regeln für dieselbe Tabelle anlegt, lässt sich die Reihenfolge der Ausführung nicht steuern.

Wenn Sie als Aktion das Schlüsselwort „NOTHING“ angeben, wird die ursprüngliche Abfrage nicht ausgeführt. Dies ist eine gern genutzte Methode, um zum Beispiel das teilweise oder vollständige Löschen einer Tabelle zu verhindern.

Listing 9.1 Das Löschen einer Tabelle durch eine Regel verhindern

```
(postgres@[local]:5432) [postgres] > CREATE RULE no_delete_orders AS
> ON DELETE TO orders
> DO INSTEAD NOTHING;
CREATE RULE
(postgres@[local]:5432) [postgres] > DELETE FROM orders;
DELETE 0
(postgres@[local]:5432) [postgres] > SELECT count(*) FROM orders;
count
-----
3000000
```

■ 9.1 Das Regelsystem für SELECT-Anweisungen

Regeln für SELECT-Anweisungen werden als letzter Schritt implementiert. Beachten Sie außerdem, dass sich die Syntax im Detail vom Regelsystem für DML-Anweisungen unterscheidet. Die Syntax finden Sie in Listing 9.2.

Listing 9.2 Die CREATE RULE-Syntax für SELECT-Anweisungen

```
CREATE RULE <regelname> AS
ON [SELECT | INSERT | UPDATE | DELETE]
TO <tabelle> [WHERE <bedingung>]
DO [INSTEAD] <aktion>;
```

Der Name der Regel ist frei wählbar. Die Tabelle muss bereits vorhanden sein, sonst lässt sich den Befehl nicht erfolgreich ausführen. Die Option „INSTEAD“ entscheidet, ob die ursprüngliche Abfrage komplett ersetzt oder die Aktion zusätzlich ausgeführt werden soll. Wenn Sie die Option „INSTEAD“ nicht verwenden, werden die in „<aktion>“ angegebenen Befehle **vor** der Ausführung der ursprünglichen Anweisung abgearbeitet. Im Platzhalter „<aktion>“ können folgende Kommandos stehen:

```
<SQL-Abfrage> | (<SQL-Abfrage>; <SQL-Abfrage>; . . .) | NOTHING
```

**Hinweis**

Aktuell ist für SELECT-Regeln nur eine Aktion erlaubt. Diese darf keine Bedingung enthalten, wenn die Option „INSTEAD“ verwendet wird.

Wie bereits erwähnt, werden Views mit Hilfe des Regelsystems gebildet. Letztendlich ist der CREATE VIEW-Befehl in Listing 9.3 identisch mit dem CREATE RULE-Befehl.

Listing 9.3 Eine View über eine Regel anlegen

```

(postgres@[local]:5432) [postgres] > CREATE VIEW v_orders
> AS SELECT * FROM orders;
CREATE VIEW
(postgres@[local]:5432) [postgres] > CREATE TABLE v_orders_2
> (
>   orderid int NOT NULL,
>   orderdate date NOT NULL,
>   customerid integer,
>   netamount numeric(12,2) NOT NULL,
>   tax numeric(12,2) NOT NULL,
>   totalamount numeric(12,2) NOT NULL);
CREATE TABLE
(postgres@[local]:5432) [postgres] > CREATE RULE "_RETURN" AS
> ON SELECT TO v_orders_2
> DO INSTEAD
> SELECT * FROM orders;
CREATE RULE

```

■ 9.2 Das Regelsystem für DML-Anweisungen

Regeln, die auf INSERT-, UPDATE- und DELETE-Anweisungen definiert werden, unterscheiden sich von Regeln auf SELECT-Anweisungen in den folgenden Punkten:

- Es ist mehr als eine Aktion erlaubt.
- Sie können keine Aktion haben.
- Als Optionen sind „INSTEAD“ und „ALSO“ erlaubt.
- Sie können Pseudo-Relationen „NEW“ und „OLD“ verwenden.

**Hinweis**

Regeln für DML-Anweisungen verändern, im Gegensatz zu Regeln für SELECT-Anweisungen, den Ausführungsplan nicht. Sie erstellen die Pläne neu. Im Sonderfall wird kein Plan erstellt.

Die Syntax für DML-Anweisungen finden Sie in Listing 9.4.

Listing 9.4 Die CREATE RULE-Syntax für DML-Anweisungen

```

CREATE [ OR REPLACE ] RULE <name> AS ON <event>
TO <table> [ WHERE condition ]
DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }

```

Die Regelqualifikation wird über die Schlüsselworte „ALSO“ und „INSTEAD“ gesteuert. Im Fall von „ALSO“ wird der Ausführungsplan ergänzt, im Fall von „INSTEAD“ ersetzt.

Bei Regeln auf INSERT-Anweisungen wird der originale Plan ausgeführt und danach erst die in der Regel definierten Aktionen. Für Regeln auf UPDATE- und DELETE-Anweisungen gilt, dass die Original-Anweisung nach den definierten Regeln ausgeführt wird.

Das folgende Beispiel lässt erahnen, wie mächtig das Regelsystem in PostgreSQL ist. Sie werden erkennen, dass Regeln fester Bestandteil der Business-Funktionalität der Applikation sind.

Betrachten Sie die Tabelle „product_data“ in Listing 9.5, die zur Pflege eines Produkts dient. Eine Anforderung an die Applikation besagt, dass Änderungen in der Tabelle, insbesondere beim Lagerbestand (Spalte p_avail), in einer separaten Log-Tabelle aufgezeichnet werden sollen. Der Name der Log-Tabelle ist „product_log“.

Listing 9.5 Die Produkt-Tabelle und die Log-Tabelle

```
(postgres@[local]:5432) [postgres] > CREATE TABLE product_data(
> p_name text,
> p_avail int,
> p_color text,
> p_size int,
> p_unit text);
CREATE TABLE
(postgres@[local]:5432) [postgres] > CREATE TABLE product_log(
> p_name text,
> p_avail int,
> log_who text,
> log_when timestamp);
CREATE TABLE
```

Erstellen wir nun die Regel. Wir verwenden die Bedingung so, dass nur dann ein Eintrag erfolgt, wenn sich der Lagerbestand ändert.

Listing 9.6 Eine UPDATE-Regel erstellen

```
(postgres@[local]:5432) [postgres] > CREATE RULE log_product AS
> ON UPDATE TO product_data
> WHERE NEW.p_avail <> OLD.p_avail
> DO INSERT INTO product_log VALUES(
> NEW.p_name,
> NEW.p_avail,
> current_user,
> current_timestamp);
CREATE RULE
```

Testen wir nun die Regel, indem wir den Lagerbestand verändern. Ein entsprechender Eintrag wird in der Tabelle „product_log“ vorgenommen.

Listing 9.7 Die Regel testen

```
(postgres@[local]:5432) [postgres] > UPDATE product_data
> SET p_avail = 23 WHERE p_name = 'keyboard';
UPDATE 1
(postgres@[local]:5432) [postgres] > SELECT * FROM product_log;
 p_name | p_avail | log_who | log_when
-----+-----+-----+-----
keyboard | 23 | postgres | 2012-04-09 13:57:33.677265
```



Praxistipp

Beachten Sie, dass das Regelsystem Bestandteil der Business-Funktionalität ist. Regeln gehören deshalb zur Applikation wie Funktionen oder sonstiger Programmcode.

Regeln werden in der Tabellenbeschreibung mit angezeigt. Die Anzeige erfolgt sowohl auf der Kommandozeile als auch in „pgAdmin III“.

Listing 9.8 Eine Regel in der Tabellendefinition anzeigen

```
(postgres@[local]:5432) [postgres] > \d product_data
Table "public.product_data"
Column | Type      | Modifiers
-----+-----+-----
p_name | text      |
p_avail | integer   |
p_color | text      |
p_size | integer   |
p_unit | text      |
Rules:
  log_product AS
  ON UPDATE TO product_data
  WHERE new.p_avail <> old.p_avail DO INSERT INTO product_log (p_name,
p_avail, log_who, log_when)
  VALUES (new.p_name, new.p_avail, "current_user"(), now())
```

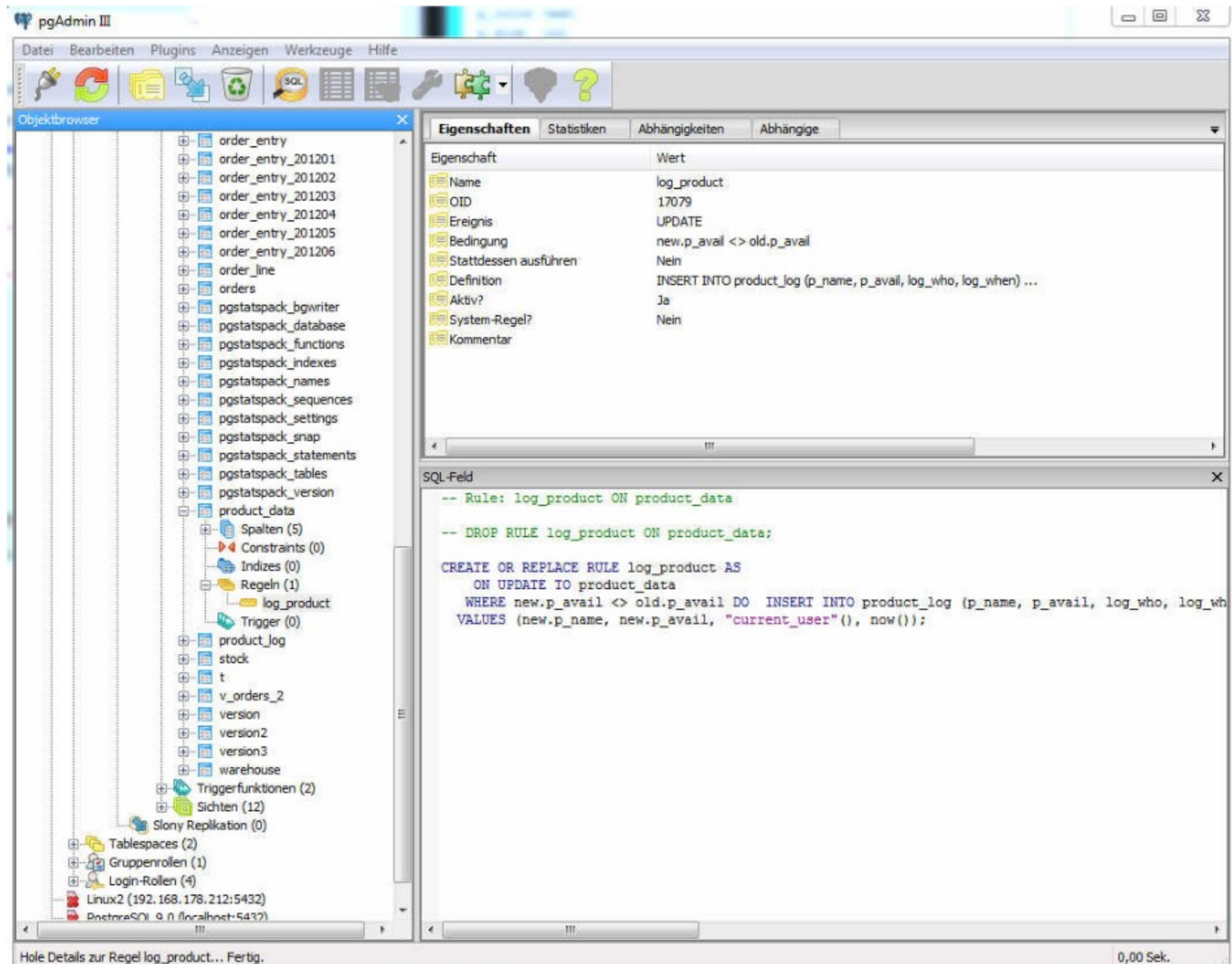


Bild 9.1 Regeln in „pgAdmin III“ anzeigen

Alle Regeln der Datenbank können zusätzlich mit Hilfe des Views „pg_rules“ abgefragt werden.

Listing 9.9 Regeln mit dem View „pg_rules“ abfragen

```
(postgres@[local]:5432) [postgres] > SELECT * FROM pg_rules
> WHERE tablename = 'product_data';
 schemaname | tablename | rulename | definition
-----+-----+-----+-----
 public     | product_data | log_product | CREATE RULE log_product AS ON
 UPDATE TO product_data WHERE (new.p_avail <> old.p_avail) DO INSERT INTO
 product_log (p_name, p_avail, log_who, log_when) VALUES (new.p_name,
 new.p_avail, "current_user"(), now());
```

■ 9.3 Regeln und Views

Mit Hilfe von Regeln können Sie Views vor Änderungen schützen. Die Klausel „DO INSTEAD NOTHING“ haben Sie bereits bei Einführung der Regel zum Verhindern von Löschvorgängen kennengelernt. Wir verwenden sie jetzt wieder.

Listing 9.10 Regel auf eine View anwenden

```
(postgres@:5432) [postgres] > CREATE RULE r_insert
> AS ON INSERT TO v_orders
> DO INSTEAD NOTHING;
CREATE RULE
(postgres@:5432) [postgres] > INSERT INTO v_orders
VALUES(1,null,3,4.0,5.0,6.0);
INSERT 0 0
```

Der Versuch einer INSERT-Anweisung führt nun nicht mehr zu einem Fehler. Die Regel wird angewandt, und es wird nichts getan.

Ein weiterer Anwendungsfall ist das Umleiten von Daten in eine andere Tabelle. Die Applikation funktioniert weiter unter Verwendung des originalen Tabellennamens in den SQL-Anweisungen und muss nicht verändert werden. Dies ist nützlich im Falle einer Migration, bei Wartungsarbeiten an einer Tabelle oder beim Einspielen eines neuen Releases ohne Downtime. Um die Umleitung zu beenden, wird die Regel einfach gelöscht. Ähnliche Regeln können Sie für UPDATE- und DELETE-Anweisungen erstellen. Das Ändern oder Löschen der originalen Datensätze kann bei der Zusammenführung der Tabellen erfolgen.

Listing 9.11 Daten in eine andere Tabelle temporär umleiten

```
(postgres@:5432) [postgres] > CREATE TABLE orders_delta
> AS SELECT * FROM orders WHERE 1=2;
SELECT 0
(postgres@:5432) [postgres] > CREATE RULE migr_orders AS
> ON INSERT TO orders
> DO INSTEAD
> INSERT INTO orders_delta VALUES(
> NEW.orderid, NEW.orderdate, NEW.customerid,
> NEW.netamount, NEW.tax, NEW.totalamount);
CREATE RULE
```




Hinweis

Beachten Sie, dass bei einer Umleitung, so wie in Listing 9.11 vorgenommen, die Primary Key Constraints außer Kraft gesetzt sind.

Wie Sie bereits feststellen konnten, ist das Regelsystem sehr mächtig. Mit Hilfe einer Regel ist es möglich, in andere Tabellen zu schreiben. Regeln verfügen über keinen eigenen Besitzer. Der Eigentümer der Tabelle ist automatisch der Eigentümer der Regel. In einer Regel verwendete Aktivitäten auf Tabellen und Views werden mit den Rechten des Eigentümers der Regel und nicht denen des Erstellers ausgeführt. In der Datenbanksprache heißt das: Regeln werden grundsätzlich mit Owner-Rechten und nicht mit Invoker-Rechten angelegt.

Regeln für INSERT, UPDATE oder DELETE gestatten die Verarbeitung der ankommenden Daten. Letztere können zum Beispiel im Falle einer INSERT-Anweisung mit „NEW“ referenziert werden. Dies erinnert auffallend an die Verarbeitungslogik von Triggern. In der Tat lassen sich viele mit Hilfe von Regeln implementierte Aktionen per Trigger abwickeln. Auch wenn aus logischer Sicht kein Unterschied besteht, unterscheiden sich die technischen Abläufe dennoch voneinander. Während eine Regel die SQL-Anweisung verändert oder zusätzliche Anweisungen generiert, wird ein Trigger für jeden relevanten Datensatz gefeuert. Wenn mehrere Sätze verarbeitet werden, ist eine Regel wahrscheinlich effektiver als ein Trigger. Für Trigger spricht, dass sie durchschaubarer und einfacher zu verstehen sind.

Listing 9.12 Ausführungsplan bei Anwendung einer Regel

```
(postgres@:5432) [postgres] > CREATE RULE del_orders
> AS ON DELETE TO orders
> DO DELETE FROM orderlines WHERE orderid = OLD.orderid;
CREATE RULE
(postgres@:5432) [postgres] > DELETE FROM orders WHERE orderid=1;
DELETE 1
(postgres@:5432) [postgres] > EXPLAIN ANALYZE
> DELETE FROM orders WHERE orderid=10;
               QUERY PLAN
-----
Delete on orderlines  (cost=0.00..23.66 rows=5 width=12) (actual
time=0.328..0.328 rows=0 loops=1)
  -> Nested Loop  (cost=0.00..23.66 rows=5 width=12) (actual
time=0.207..0.214 rows=2 loops=1)
    -> Index Scan using orders_pkey on orders  (cost=0.00..8.27 rows=1
width=10) (actual time=0.181..0.181 rows=1 loops=1)
          Index Cond: (orderid = 10)
    -> Index Scan using ix_orderlines_orderid on orderlines
(cost=0.00..15.34 rows=5 width=10) (actual time=0.017..0.021 rows=2 loops=1)
          Index Cond: (orderid = 10)
Total runtime: 0.483 ms
Delete on orders  (cost=0.00..8.27 rows=1 width=6) (actual time=0.052..0.052
rows=0 loops=1)
  -> Index Scan using orders_pkey on orders  (cost=0.00..8.27 rows=1
width=6) (actual time=0.033..0.035 rows=1 loops=1)
        Index Cond: (orderid = 10)
Total runtime: 0.120 ms
(12 Zeilen)
```


Die Erfahrung aus der Praxis zeigt, dass man mit der Optimierung von SQL-Anweisungen eine wesentlich höhere Performance-Verbesserung der Datenbank-Applikation als mit der Optimierung des PostgreSQL-Servers erzielt. Auch wenn es zur Server-Optimierung keine Alternative gibt, sollten Sie beachten, dass in der SQL-Optimierung das größere Potenzial liegt. Außerdem hat die Anzahl nicht-optimierter SQL-Anweisungen in den vergangenen Jahren stetig zugenommen. Im Fokus der Software-Entwicklung stehen heute eher Kosten-Effizienz und Funktionalität, wobei das Thema SQL-Performance häufig auf der Strecke bleibt. In diesem Kapitel beschäftigen wir uns mit dem Thema der SQL-Optimierung, wobei wir voraussetzen, dass der PostgreSQL-Server bereits weitgehend optimiert ist.

■ 10.1 Der Query Optimizer

Der Query Optimizer ist verantwortlich für die Auswahl des Ausführungsplans. Es gibt eine Reihe von Server-Parametern, die den Optimizer beeinflussen können. Beachten Sie dabei, dass eine Veränderung dieser Parameter Einfluss auf die Ausführungspläne aller Statements aller Datenbanken des Servers haben kann. Parameteränderungen sollten deshalb stets mit einem Performancetest der Applikation verbunden sein.

Die folgenden Parameter haben Einfluss auf die Wahl des Ausführungsplans durch den Optimizer:

- *seq_page_cost*: Bestimmt die Kosten für das Lesen einer Page von der Disk für sequenzielle Lesevorgänge. Der Standardwert ist „1.0“. Eine Erhöhung macht das sequenzielle Lesen von Tabellen und Indexen teurer.
- *random_page_cost*: Legt die Kosten für nicht-sequenzielles Lesen einer Page von der Disk fest. Der Standardwert ist 4.0. Eine Reduzierung des Wertes verringert die Kosten für die Verwendung von Indexen. Der Wert für „random_page_cost“ sollte den Wert von „seq_page_cost“ nicht unterschreiten.
- *effective_cache_size*: Setzt den Wert für den Disk Cache, der in die Planung des Optimizers eingeht. Ein größerer Wert favorisiert die Verwendung von Indexen, ein niedrigerer

die Ausführung sequenzieller Tabellen- und Index-Scans. Der Standardwert ist 128 MByte.

- *cpu_operator_cost*: Kosten, die der Optimizer für die Verarbeitung eines Operators im Ausführungsplan annimmt. Der Standardwert ist 0,0025.
- *cpu_tuple_cost*: Kosten für den Optimizer zur Verarbeitung einer Zeile. Der Standardwert ist 0,01.
- *cpu_index_tuple_cost*: Kosten, die der Optimizer für die Verarbeitung eines einzelnen Index-Eintrags ansetzt. Der Standardwert ist 0,005.



Praxistipp

Verändern Sie die beschriebenen Parameter vorsichtig und stets nur einen. Die Auswirkungen können zu einem veränderten Performance-Verhalten der Applikation führen. Ein nachfolgender Performance-Test sollte zwingend stattfinden.

Die folgenden Parameter nehmen Einfluss auf den Genetic Query Optimizer. Der Genetic Optimizer benutzt eine heuristische Suchmethode und reduziert damit die Verarbeitungszeit bei der Erstellung der Ausführungspläne:

- *geqo*: Schaltet den Genetic Query Optimizer ein und aus. Letzterer ist standardmäßig eingeschaltet und sollte nur in Ausnahmefällen ausgeschaltet werden.
- *geqo_pool_size*: Kontrolliert die durch den Genetic Optimizer verwendete Pool-Größe. Der Standardwert ist „0“. Damit wird der Wert basierend auf dem Wert des Parameters „geqo_effort“ sowie der Anzahl der eingebundenen Tabellen berechnet.
- *geqo_effort*: Bestimmt die maximalen Aufwände des Optimizer. Der Wert legt den Kompromiss zwischen Planungszeit und Qualität des Plans fest. Die Werte können sich auf einer Skala von „1“ bis „10“ bewegen, der Standardwert ist „5“. Eine Vergrößerung des Wertes erhöht die Wahrscheinlichkeit, dass ein besserer Plan gefunden wird, aber gleichzeitig auch die zum Parsen einer SQL-Anweisung erforderliche Zeit.

Folgende Parameter beeinflussen die Wahl der Ausführungsmethoden.

- *enable_index_scan*: Erlaubt dem Query Planner die Verwendung von Index-Scans; Standard ist „on“.
- *enable_seq_scan*: Dient zum Ein- oder Ausschalten sequenzieller Scans. Ein komplettes Ausschalten ist nicht möglich, der Planner versucht, sequenzielle Scans möglichst zu vermeiden. Der Standardwert ist „on“.
- *enable_nestloop*: Der Query Planner darf Nested-Loop-Vereinigungen verwenden. Der Standard ist „on“. Setzt man den Parameter auf off, werden Nested Loops nicht komplett abgeschaltet. Der Query Planner versucht, diese durch andere Methoden zu ersetzen, wo immer es möglich ist.
- *enable_hash_join*: Ein- und Ausschalten von Hash Joins. Der Standardwert ist „on“.
- *enable_bitmapscan*: Die Verwendung von Bitmap Scans kann ein- und ausgeschaltet werden. Der Standard ist „on“.

■ 10.2 Indexe effektiv einsetzen

Der Index ist eines der wichtigsten Performance-Instrumente in relationalen Datenbanken. Die Anzahl der Daten, die gelesen werden müssen, um das Ergebnis zu erhalten, lässt sich auf diese Weise signifikant reduzieren. Um festzustellen, wie viele Datenblöcke von der Disk (oder aus dem Buffer Cache) gelesen werden müssen, bemühen wir die Statistik-Views des PostgreSQL-Servers. Detaillierte Informationen dazu finden Sie in Kapitel 6 („Performance Tuning“).

Listing 10.1 Eine View für Lesestatistik

```
(postgres@[local]:5432) [postgres] > CREATE OR REPLACE VIEW io_stats AS
> SELECT t.relname, seq_scan, seq_tup_read, idx_scan,
> heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
> FROM pg_stat_user_tables t, pg_statio_user_tables i
> WHERE i.relid = t.relid;
CREATE VIEW
```

Zweifellos sind Abfragen, die einen Full-Table-Scan auf größeren Tabellen verursachen, eine schlechte Voraussetzung für eine performante Applikation. Mit Hilfe der View in Listing 10.2 lässt sich herausfinden, auf welcher Tabelle Full-Table-Scans stattfinden.

Listing 10.2 Objekte mit Full-Table-Scan herausfiltern

```
(postgres@[local]:5432) [postgres] > SELECT * FROM io_stats WHERE seq_tup_
read > 1000000;
-[ RECORD 1 ]--+-+-----
relname          | kb
seq_scan         | 4
seq_tup_read     | 40000000
idx_scan         |
heap_blks_read   | 123492
heap_blks_hit    | 11047654
idx_blks_read    |
idx_blks_hit     |
```

Schauen wir uns nun den Ausführungsplan für ein SELECT auf die Tabelle „kb“ an. Die Tabelle enthält ca. 10 Millionen Sätze.

Listing 10.3 Ausführungsplan SELECT-Anweisung auf Tabelle „kb“

```
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM kb WHERE id = 4711;
-----
Seq Scan on kb (cost=0.00..248457.00 rows=1 width=71) (actual
time=0.698..1243.371 rows=1 loops=1)
  Filter: (id = 4711)
  Total runtime: 1243.403 ms
```

Legen wir nun einen Index an, um einen besseren Ausführungsplan zu erreichen und weniger Ressourcen auf dem Datenbankserver zu verbrauchen:

```
(postgres@[local]:5432) [postgres] > CREATE INDEX i_kb ON kb(id);
CREATE INDEX
```


Listing 10.4 zeigt, dass sich nicht nur die Ausführungszeit, sondern auch die Kosten deutlich reduziert haben. Damit reduziert sich der Ressourcen-Verbrauch von CPU- und I/O-Kapazitäten erheblich. Das schlägt insbesondere dann zu Buche, wenn die Anzahl der konkurrierenden Sitzungen zunimmt.

Listing 10.4 Der Ausführungsplan mit Verwendung eines Index

```
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM kb WHERE id = 4711;
                                QUERY PLAN
-----
Index Scan using i_kb on kb  (cost=0.00..9.36 rows=1 width=71) (actual
time=0.018..0.020 rows=1 loops=1)
  Index Cond: (id = 4711)
Total runtime: 0.049 ms
(postgres@[local]:5432) [postgres] > SELECT * FROM io_stats WHERE relname
like '%kb';
-[ RECORD 1 ]--+-+-----
relname          | kb
. . .
idx_scan          | 3
. . .
idx_blks_read     | 4
idx_blks_hit      | 6
```

Betrachten Sie den Ausführungsplan in Listing 10.5. Obwohl ein Index vorhanden ist, bevorzugt der Optimizer einen Plan, der einen Full-Table-Scan durchführt und den Index nicht verwendet.

Listing 10.5 Verwendung des Full-Table-Scans trotz Index

```
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM kb WHERE id > 10000;
                                QUERY PLAN
-----
Seq Scan on kb  (cost=0.00..248457.00 rows=9990132 width=71) (actual
time=0.008..14838.193 rows=9990000 loops=1)
  Filter: (id > 10000)
```

Auch in diesem Fall hat der Optimizer richtig entschieden. Weil die SQL-Abfrage ein großes Resultset (fast die gesamte Tabelle) zurückliefert, wäre die Verwendung eines zusätzlichen Index kontra-produktiv. Ein sequenzieller Scan der gesamten Tabelle ist der optimale Plan. Lange Verkettungen von AND und OR stellen den Optimizer einer Datenbank häufig vor Probleme. Wichtig dabei ist vor allem die Abschätzung der Ergebnismenge. PostgreSQL hält Bitmap-Indexe bereit, die es ermöglichen, Indexe miteinander zu verbinden. Betrachten Sie die Tabelle „kb2“ in Listing 10.6 mit 10 Millionen Sätzen.

Listing 10.6 Tabelle „kb2“ mit doppeltem Index

```
(postgres@[local]:5432) [postgres] > \d kb2
      Table "public.kb2"
  Column |  Type  | Modifiers
-----+-----+-----
 id1     | bigint |
 id2     | bigint |
```

```

Indexes:
    "i1_kb2" btree (id1)
    "i2_kb2" btree (id2)
id1      |      id2
-----+-----
     1    | 10000000
     2    |  9999999
     3    |  9999998
     4    |  9999997
     5    |  9999996
. . .
9999995  |      6
9999996  |      5
9999997  |      4
9999998  |      3
9999999  |      2
10000000 |      1

```

Schauen Sie sich den optimalen Plan in Listing 10.7 an. PostgreSQL verwendet zwei Indexe und verknüpft diese mit Hilfe einer Bitmap-Or-Operation. Dieser Plan ist sehr effektiv in Ausführungszeit und Ressourcenverbrauch.

Listing 10.7 Bitmap-Operation von Indexen

```

(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM kb2 WHERE id1=2 OR id2=5;
               QUERY PLAN
-----
Bitmap Heap Scan on kb2  (cost=10.71..18.70 rows=2 width=16) (actual
time=0.159..0.165 rows=2 loops=1)
  Recheck Cond: ((id1 = 2) OR (id2 = 5))
    -> BitmapOr  (cost=10.71..10.71 rows=2 width=0) (actual time=0.146..0.146
rows=0 loops=1)
      -> Bitmap Index Scan on i1_kb2  (cost=0.00..5.35 rows=1 width=0)
      (actual time=0.083..0.083 rows=1 loops=1)
        Index Cond: (id1 = 2)
      -> Bitmap Index Scan on i2_kb2  (cost=0.00..5.35 rows=1 width=0)
      (actual time=0.058..0.058 rows=1 loops=1)
        Index Cond: (id2 = 5)
    Total runtime: 0.198 ms

```

Was geschieht, wenn wir Bitmap-Index-Scans explizit unterbinden? Wie Sie in Abschnitt 10.1 erfahren haben, können einzelne Operationen auf Tabellen und Indexen erlaubt und zugelassen werden. In Listing 10.8 finden Sie den Ausführungsplan. Der Optimizer verzichtet komplett auf Indexe und führt einen Full-Table-Scan durch.

Listing 10.8 Bitmap-Operationen unterbinden

```

(postgres@[local]:5432) [postgres] > SET enable_bitmapscan TO off;
SET
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM kb2 WHERE id1=2 OR id2=5;
               QUERY PLAN
-----
Seq Scan on kb2  (cost=0.00..204055.00 rows=2 width=16) (actual
time=426.164..1960.906 rows=2 loops=1)
  Filter: ((id1 = 2) OR (id2 = 5))
    Total runtime: 1960.939 ms

```


■ 10.3 Optimierung von SQL-Anweisungen

Bevor wir mit der Analyse von Ausführungsplänen beginnen, untersuchen wir, wie die einzelnen Vereinigungsmethoden (Joins) funktionieren. PostgreSQL verwendet die folgenden drei Algorithmen:

- Hash Join
- Nested Loop
- Merge Join

Hash Joins

Ein Hash Join ist dann interessant, wenn eine größere mit einer kleineren Tabelle vereinigt werden soll. Idealerweise passt die kleinere Tabelle in den Buffer Cache. Damit wird im Vergleich zu einer Nested-Loop-Operation die Anzahl von I/O-Operationen deutlich reduziert.

Die kleinere Tabelle wird gescannt und in eine Hash-Tabelle geladen. Aus den Join-Attributen wird ein Schlüssel (Hash Key) gebildet. Danach wird die größere Tabelle gescannt (Full-Table-Scan, i.d.R. von der Disk) und der gefundene Schlüssel in jedem Satz mit dem Hash Key der Hash-Tabelle im Memory auf Übereinstimmung geprüft. Über die Hash-Tabelle wird der Datensatz der kleineren Tabelle gefunden, und die Vereinigung der beiden Sätze kann stattfinden. Entscheidend für eine bessere Performance, im Vergleich zu einer Nested-Loop-Operation, ist, dass der Vergleich der Schlüssel über die Hash-Tabelle erfolgt, die sich permanent im Memory befindet. Passt die Hash-Tabelle nicht mehr komplett in den Cache, leidet die Performance des Hash Joins.

Nested Loops

Die Vereinigungsoperation ist stark I/O-lastig. Die linke Tabelle wird einmal komplett durchgelesen (Table Scan). Jeder Satz der linken Tabelle wird mit der rechten Tabelle verglichen. Im Extremfall produziert jeder Satz der linken Tabelle einen Full-Table-Scan der rechten. Sowohl für die linke als auch für die rechte Tabelle kann alternativ ein Index-Scan erfolgen.

Nested-Loop-Operations mit Einbindung von Indexen werden auch dann ineffizient, wenn das Resultset relativ groß ist und einen hohen Prozentsatz der Tabellen zurückliefert. Der ständige Wechsel von Index-Scan zu Table Access by Rowid beansprucht das I/O-Subsystem erheblich und macht die Ausführungszeit sehr abhängig von der Latenzzeit des Systems. Ideal sind Nested-Loop-Operations, wenn der Zugriff über Indexe erfolgt und nur wenige Sätze als Ergebnis zurückkommen. Nested-Loop-Operations ohne Indexe sind sehr kostenintensiv. In solchen Fällen erweisen sich Hash Joins häufig als effektiver.

Merge Joins

Vor Ausführung der Vereinigung wird jede Tabelle nach den Vereinigungsattributen sortiert. Danach werden beide Tabellen parallel gelesen und nach den Vereinigungsschlüsseln zusammengeführt. Obwohl die Sortierung am Anfang der Operation als recht aufwendig zu

betrachten ist, hat die Methode den Vorteil, dass jede Tabelle danach nur einmal durchlaufen werden muss. Die vorherige Sortierung der Tabellen lässt sich durch einen vorhandenen Index ersetzen.

Sind die Eingaben nicht bereits nach den Vereinigungsschlüsseln sortiert (zum Beispiel aufgrund einer früheren Merge-Join-Operation oder eines vorhandenen Index), fügt der Planner einen Sortierschritt hinzu. Diese Kosten gehen separat in den Merge Join ein.

Betrachten wir nun konkrete Fälle von Ausführungsplänen. Für die Beispiele in diesem Abschnitt verwenden wir die Dellstore 2-Datenbank. Sie finden diese auf der pgFoundry-Webseite unter <http://pgfoundry.org/projects/dbsamples> und die Installationsanleitung in Listing 10.9.

Listing 10.9 Die Beispiel-Datenbank „Dellstore 2“ installieren

```
$ createdb dellstore2
$ createlang plpgsql dellstore2
$ psql -f dellstore2-normal-1.0.sql dellstore2
. . .
$ psql -d dellstore2 -c VACUUM
VACUUM
$ psql -d dellstore2 -c ANALYZE
ANALYZE
```

10.3.1 Der EXPLAIN-Befehl

Der EXPLAIN-Befehl zeigt den Query-Plan an, der bei Ausführung der SQL-Anweisung verwendet wird. Wenn Sie den Befehl „EXPLAIN ANALYZE“ verwenden, erhalten Sie sowohl den erwarteten Ausführungsplan als auch reale Informationen über das, was geschieht, wenn die Anweisung tatsächlich ausgeführt wird.



Hinweis

Beachten Sie, dass der Befehl „EXPLAIN ANALYZE“ die SQL-Anweisung im Hintergrund tatsächlich ausführt. Für DML-Anweisungen (INSERT, UPDATE, DELETE) bedeutet dies, dass Daten verändert werden.

```
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> DELETE FROM kb2;
. . .
(postgres@[local]:5432) [postgres] > SELECT count(*) FROM kb2;
count
-----
0
```

Der Befehl „EXPLAIN ANALYZE“ schreibt auch die zu erwartende Laufzeit aus. In Listing 10.10 sehen Sie einen Vergleich zwischen der berechneten und der tatsächlichen Laufzeit durch. Wie Sie sehen, weicht die geschätzte Laufzeit aus dem „EXPLAIN ANALYZE“-Befehl stark von der tatsächlichen Laufzeit ab.

Listing 10.10 Vergleich der geschätzten mit der tatsächlichen Laufzeit

```
> SELECT count(*) FROM customer;
                                QUERY PLAN
-----
Aggregate  (cost=394135.40..394135.41 rows=1 width=0) (actual
time=9531.387..9531.389 rows=1 loops=1)
  -> Seq Scan on customer  (cost=0.00..380308.32 rows=5530832 width=0)
(actual time=812.943..5321.357 rows=3000000 loops=1)
Total runtime: 9531.434 ms
(postgres@[local]:5432) [postgres] > \timing
Timing is on.
(postgres@[local]:5432) [postgres] > SELECT count(*) FROM customer;
 count
-----
3000000
```

**Hinweis**

Führen Sie für die Ermittlung der tatsächlichen Laufzeit den SQL-Befehl aus. Der Befehl „EXPLAIN ANALYZE“ liefert ungenaue Werte, die unter Umständen stark von der Realität abweichen.

Die Laufzeiten einer SQL-Anweisung sind stark davon abhängig, ob sich die Datenblöcke bereits im Cache befinden oder nicht. Das betrifft sowohl den Daten-Cache des PostgreSQL-Servers als auch den Filesystem-Cache des Betriebssystems. Im folgenden Beispiel demonstrieren wir die erheblichen Unterschiede zwischen beiden Zuständen. Dazu werden die Laufzeit gemessen und der I/O-Durchsatz berechnet. Die Tabelle „customer“ weist eine Größe von 2539 MB auf.

```
(postgres@[local]:5432) [postgres] > SELECT pg_size_pretty(pg_relation_
size('customer'));
 pg_size_pretty
-----
2539 MB
```

Die Ausführungszeit beträgt 1369,959 ms. Insofern ergibt sich ein I/O-Durchsatz von 1853 Mbyte/sec. Damit ist bereits klar, dass PostgreSQL-Cache und Filesystem-Cache im Spiel sind.

Jetzt löschen wir beide Caches. Der Cache des PostgreSQL-Servers lässt sich am einfachsten durch einen Neustart leeren. Danach messen wir die Laufzeit erneut.

**Praxistipp**

Führen Sie die folgenden Schritte durch, um den Cache des Datenbankservers und den Filesystemcache zu löschen:

```
$ pg_ctl stop
waiting for server to shut down.... done
server stopped
[root@tt ~]# sync
echo 3 > /proc/sys/vm/drop_caches
$ pg_ctl start
server starting
```

Nach Löschen des Cache vergrößert sich die Ausführungszeit von ca. 1 Sekunde auf ca. 46 Sekunden (genau: 46 497 290 ms). Der I/O-Durchsatz liegt somit bei 54,6 Mbyte/sec. Dies bestätigt, dass die Daten von der Disk und nicht aus dem Cache kommen.

**Praxistipp**

Verwenden Sie für Performance-Analysen stets die Testwerte, die sich aus einem leeren Cache ergeben. Im produktiven Betrieb greifen mehrere Benutzer parallel auf das System zu, und es werden SQL-Anweisungen in Konkurrenz ausgeführt, sodass die Caches normalerweise nicht mit der Tabelle des aktuellen States gefüllt sind.

In der Ausgabe des Befehls „EXPLAIN ANALYZE“ finden sich eine Reihe von Werten wieder, die wir kurz erklären wollen. Betrachten Sie eine einfache Abfrage:

```
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM customer;

                                QUERY PLAN
-----
Seq Scan on customer (cost=0.00..380308.32 rows=5530832 width=167) (actual
time=855.352..5398.205 rows=3000000 loops=1)
  Total runtime: 9627.875 ms
(2 rows)
Time: 9628.249 ms
```

Die erste Zeile „cost=“ beinhaltet die Werte, die dem Optimizer vor der Ausführung zur Verfügung stehen. Im Einzelnen sind dies:

- *cost=0.00..380308.32*: Die erste Zahl beschreibt die Startup-Kosten. Dabei handelt es sich um Kosten, die notwendig sind, um den ersten Output zu erzeugen. Der Wert ist in diesem Fall „0“, da ein sequenzieller Scan sofort die erste Zeile zurückliefert. Der zweite Wert beschreibt die anfallenden Kosten, um den gesamten Zweig abzuarbeiten.
- *rows=5530832*: Anzahl der nach Ausführung des Zweiges erwarteten Sätze.
- *width=167*: Geschätzte durchschnittliche Satzlänge für alle zurückgelieferten Sätze.

Die Zeile „(actual=“ enthält die ermittelten Werte, die entstanden sind, nachdem die SQL-Anweisung tatsächlich gelaufen ist. Dabei bedeutet:

- *loops=1*: Anzahl von Durchläufen des aktuellen Knotens. Bei Vereinigungen über mehrere Tabellen ist es zum Beispiel möglich, dass Scans mehrfach durchlaufen werden.

Mit der Option „VERBOSE“ können Sie sich zusätzlich die Spalten anzeigen lassen, die in einem Zweig zurückkommen.

Listing 10.11 Die Option „VERBOSE“ in der EXPLAIN-Anweisung

```
(postgres@[local]:5432) [postgres] > EXPLAIN VERBOSE
> SELECT * FROM customer;

                                QUERY PLAN
-----
Seq Scan on public.customer (cost=0.00..380308.32 rows=5530832 width=167)
  Output: c_id, c_d_id, c_w_id, c_first, c_middle, c_last, c_street_1,
c_street_2, c_city, c_state, c_zip, c_phone,
c_since, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment,
c_payment_cnt, c_delivery_cnt, c_data
(2 rows)
Time: 0.369 ms
```

Ein neues Feature in der Version 9 ist die Möglichkeit, zwischen verschiedenen Ausgabeformaten im EXPLAIN-Befehl zu wählen. Neben einem XML-Format ist es möglich, ein sogenanntes Maschinen-lesbares Format zu erstellen, das als Eingabe für Analysis-Tools dient, von denen es in Zukunft hoffentlich mehr geben wird.

Listing 10.12 Mit dem EXPLAIN-Befehl eine Maschinen-lesbare Ausgabe erstellen

```
(postgres@[local]:5432) [postgres] > EXPLAIN (FORMAT YAML)
> SELECT count(*) FROM customer;

                                QUERY PLAN
-----
- Plan:                                                                    +
  Node Type: "Aggregate"                                                    +
  Strategy: "Plain"                                                         +
  Startup Cost: 394135.40                                                    +
  Total Cost: 394135.41                                                      +
  Plan Rows: 1                                                              +
  Plan Width: 0                                                             +
  Plans:                                                                    +
    - Node Type: "Seq Scan"                                                  +
      Parent Relationship: "Outer"                                          +
      Relation Name: "customer"                                             +
      Alias: "customer"                                                     +
      Startup Cost: 0.00                                                     +
      Total Cost: 380308.32                                                  +
      Plan Rows: 5530832                                                    +
      Plan Width: 0
```

10.3.2 Ausführungspläne verstehen

Der einfachste Plan ist der sequenzielle Scan einer Tabelle. Dieser wird durchgeführt, wenn kein Index an der WHERE-Bedingung vorhanden ist oder ein großer Teil der Tabelle in das Resultset eingeht. Im Falle einer hohen Selektivität in der WHERE-Bedingung ist der Index-Scan wesentlich Ressourcen-sparender und verkürzt die Antwortzeit erheblich.

Subqueries werden relativ häufig in Applikationen verwendet und vom Planner letztendlich als Vereinigungen dargestellt. Eine typische SQL-Abfrage mit Subquery finden Sie in Listing 10.13. Die Anweisung wurde mit einem Hash Semi Join geplant.

Listing 10.13 Ausführungsplan für ein Subquery

```
(postgres@[local]:5432) [dellstore2] > EXPLAIN ANALYZE
> SELECT * FROM orders WHERE customerid IN
> (SELECT customerid FROM customers WHERE state = 'NY');
          QUERY PLAN
-----
Hash Semi Join  (cost=740.58..995.13 rows=275 width=30) (actual
time=5.612..41.313 rows=124 loops=1)
  Hash Cond: (orders.customerid = customers.customerid)
    -> Seq Scan on orders  (cost=0.00..220.00 rows=12000 width=30) (actual
time=0.006..17.181 rows=12000 loops=1)
    -> Hash  (cost=738.00..738.00 rows=206 width=4) (actual time=5.127..5.127
rows=206 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 8kB
          -> Seq Scan on customers  (cost=0.00..738.00 rows=206 width=4)
(actual time=0.012..4.771 rows=206 loops=1)
                Filter: ((state)::text = 'NY'::text)
Total runtime: 41.521 ms
```

Die Entscheidung des Optimizer, einen Hash Join zu verwenden, war sicher richtig, da wir ein relativ großes Resultset erhalten. Doch wie sieht der Plan aus, wenn die Selektivität erhöht und nur ein Satz zurückgeliefert wird?

Listing 10.14 Ausführungsplan eines Subquery mit erhöhter Selektivität

```
(postgres@[local]:5432) [dellstore2] > EXPLAIN ANALYZE
> SELECT * FROM orders WHERE customerid IN
> (SELECT customerid FROM customers WHERE lastname = 'ITHOMQJNYX');
          QUERY PLAN
-----
Nested Loop  (cost=738.00..746.29 rows=1 width=30) (actual time=5.212..5.212
rows=0 loops=1)
  -> HashAggregate  (cost=738.00..738.01 rows=1 width=4) (actual
time=5.188..5.190 rows=1 loops=1)
        -> Seq Scan on customers  (cost=0.00..738.00 rows=1 width=4)
(actual time=0.007..5.178 rows=1 loops=1)
              Filter: ((lastname)::text = 'ITHOMQJNYX'::text)
        -> Index Scan using ix_order_custid on orders  (cost=0.00..8.27 rows=1
width=30) (actual time=0.011..0.011 rows=0 loops=1)
              Index Cond: (customerid = customers.customerid)
Total runtime: 5.265 ms
```

In diesem Fall greift der Optimizer richtigerweise auf eine Nested-Loop-Operation für die Vereinigung zurück. Wir sehen also ein ähnliches Verhalten, wie wir es von Anweisungen mit nur einer Tabelle kennen.

**Fazit**

Subqueries stellen in PostgreSQL kein Problem dar. Der Designer der SQL-Anweisung muss sich um die Performance keine Gedanken machen, da Planner und Optimizer eine Umwandlung in eine Vereinigung vornehmen und die optimale Vereinigungsmethode auswählen.

Die Funktionsweise von Nested-Loop-Operationen haben Sie bereits kennengelernt. Planner und Optimizer unterscheiden zwischen „Inner Loop“ und „Outer Loop“ je nachdem, welche Tabelle (oder welcher Output eines anderen Joins) die treibende Tabelle ist. Die Auswahl der treibenden Tabelle nimmt der Optimizer vor. Da Nested-Loop-Operationen in der Regel mit Indexen arbeiten, spricht man auch von „Inner Index Scan“ und „Outer Index Scan“.

Ein solcher Plan kommt sehr häufig vor, wenn der Optimizer in der Lage ist, die Anzahl von Sätzen in einem inneren Loop über einen Index signifikant zu reduzieren. Ein Beispiel dafür finden Sie in Listing 10.15.

Listing 10.15 „Nested Loop“ mit „Inner Index Scan“

```
(postgres@[local]:5432) [dellstore2] > EXPLAIN ANALYZE
> SELECT * FROM orders o, orderlines l
> WHERE o.orderid = l.orderid
> AND o.netamount = 383.46;

                                QUERY PLAN
-----
Nested Loop  (cost=0.00..265.41 rows=5 width=48) (actual time=0.027..3.048
rows=9 loops=1)
  -> Seq Scan on orders o  (cost=0.00..250.00 rows=1 width=30) (actual
time=0.013..2.990 rows=1 loops=1)
      Filter: (netamount = 383.46)
  -> Index Scan using ix_orderlines_orderid on orderlines l
      (cost=0.00..15.34 rows=5 width=18) (actual time=0.006..0.023 rows=9 loops=1)
      Index Cond: (orderid = o.orderid)
Total runtime: 3.097 ms
```

Ein Merge Join setzt voraus, dass beide Eingaben vorsortiert sind. Dabei kann die innere Tabelle mehr als einmal gescannt werden, wenn die äußere doppelte Werte ausweist. Merge Joins kommen recht häufig vor, wenn die Vereinigungsbedingung auf Gleichheit basiert (siehe Listing 10.16).

Listing 10.16 Beispiel für einen Merge Join

```
> SELECT c.customerid, SUM(totalamount)
> FROM customers c, orders o
> WHERE c.customerid = o.customerid
> GROUP BY c.customerid;

                                QUERY PLAN
-----
GroupAggregate  (cost=0.05..2119.49 rows=12000 width=10) (actual
time=0.053..187.167 rows=8996 loops=1)
  -> Merge Join  (cost=0.05..1939.49 rows=12000 width=10) (actual
time=0.034..150.700 rows=12000 loops=1)
      Merge Cond: (c.customerid = o.customerid)
  -> Index Scan using customers_pkey on customers c
```



```
(cost=0.00..1019.25 rows=20000 width=4) (actual time=0.011..51.521 rows=20000
loops=1)
-> Index Scan using ix_order_custid on orders o (cost=0.00..720.24
rows=12000 width=10) (actual time=0.008..31.139 rows=12000 loops=1)
Total runtime: 200.073 ms
```

Bei einem Hash Join erfolgt keine Vorsortierung der Eingaben. Aus der kleineren Tabelle wird eine Hash-Tabelle gebildet. Diese wird in den Cache geladen. Jede Zeile der äußeren Tabelle wird mit der Hash-Tabelle verglichen. Der Performance-Vorteil gegenüber einer Nested-Loop-Operation liegt vor allem darin, dass der Vergleich mit der inneren Tabelle über einen Hash im Memory sehr effektiv ist und keine Disk-Operationen verursacht.

Listing 10.17 Ein Hash Join für ein Subquery

```
(postgres@[local]:5432) [dellstore2] > EXPLAIN ANALYZE
> SELECT p.title, p.price
> FROM products p
> WHERE EXISTS (SELECT 'x' FROM orderlines l
> WHERE l.prod_id = p.prod_id);
               QUERY PLAN
-----
Hash Join (cost=1358.12..2300.12 rows=9722 width=22) (actual
time=237.174..283.195 rows=9973 loops=1)
  Hash Cond: (p.prod_id = l.prod_id)
    -> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=26)
(actual time=0.005..14.307 rows=10000 loops=1)
    -> Hash (cost=1236.60..1236.60 rows=9722 width=4) (actual
time=237.151..237.151 rows=9973 loops=1)
      Buckets: 1024 Batches: 1 Memory Usage: 351kB
      -> HashAggregate (cost=1139.38..1236.60 rows=9722 width=4)
(actual time=206.061..221.393 rows=9973 loops=1)
        -> Seq Scan on orderlines l (cost=0.00..988.50 rows=60350
width=4) (actual time=0.006..102.625 rows=60350 loops=1)
Total runtime: 297.333 ms
```

Sind mehr als zwei Tabellen in die Vereinigung eingebunden, spielt die Reihenfolge eine wichtige Rolle. Das optimale Ergebnis erhält man, wenn beim ersten Join bereits so viele Sätze wie möglich eliminiert werden. Anders ausgedrückt: SQL-Anweisungen sind sehr zeintensiv und verbrauchen viele Ressourcen, wenn dieses Prinzip nicht berücksichtigt wird. Auch wenn das Endergebnis nur aus einem kleinen Resultset besteht, können solche Anweisung erhebliche Probleme verursachen.

Normalerweise trifft der Optimizer die richtige Entscheidung und wählt die günstigste Reihenfolge für die Vereinigungen. Dies ist allerdings nicht immer der Fall. Wenn Sie sich über die optimale Reihenfolge im Klaren sind, können Sie den Optimizer zwingen, diese zu wählen und verkürzen gleichzeitig die Zeit für das Parsen der SQL-Anweisung. Dies erreicht man, indem man den Parameter „join_collapse_limit“ auf den Wert „1“ setzt. Insofern berücksichtigt man keine alternativen Pläne.

Listing 10.18 Die Reihenfolge von Joins vorgeben

```
(postgres@[local]:5432) [dellstore2] > SET join_collapse_limit = 1;
SET
(postgres@[local]:5432) [postgres] > EXPLAIN ANALYZE
> SELECT * FROM cust_hist h, products p, customers c
```



```

> WHERE h.prod_id = p.prod_id
> AND h.customerid = c.customerid;
          QUERY PLAN
-----
Hash Join (cost=326.05..6539.84 rows=60350 width=329) (actual
time=33.219..595.108 rows=60350 loops=1)
  Hash Cond: (h.prod_id = p.prod_id)
    -> Merge Join (cost=0.05..4705.09 rows=60350 width=280) (actual
time=0.038..359.348 rows=60350 loops=1)
      Merge Cond: (c.customerid = h.customerid)
      -> Index Scan using customers_pkey on customers c
(cost=0.00..1019.25 rows=20000 width=268) (actual time=0.012..35.562
rows=20000 loops=1)
      -> Index Scan using ix_cust_hist_customerid on cust_hist h
(cost=0.00..2881.46 rows=60350 width=12) (actual time=0.009..102.504
rows=60350 loops=1)
    -> Hash (cost=201.00..201.00 rows=10000 width=49) (actual
time=33.139..33.139 rows=10000 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 814kB
    -> Seq Scan on products p (cost=0.00..201.00 rows=10000 width=49)
(actual time=0.005..15.147 rows=10000 loops=1)
Total runtime: 679.980 ms

```

Neben der Ausgabe des Ausführungsplans auf der Kommandozeile ermöglicht das Tool „pgAdmin III“ eine grafische Darstellung. Dies ist insbesondere bei komplexen Plänen sehr hilfreich.

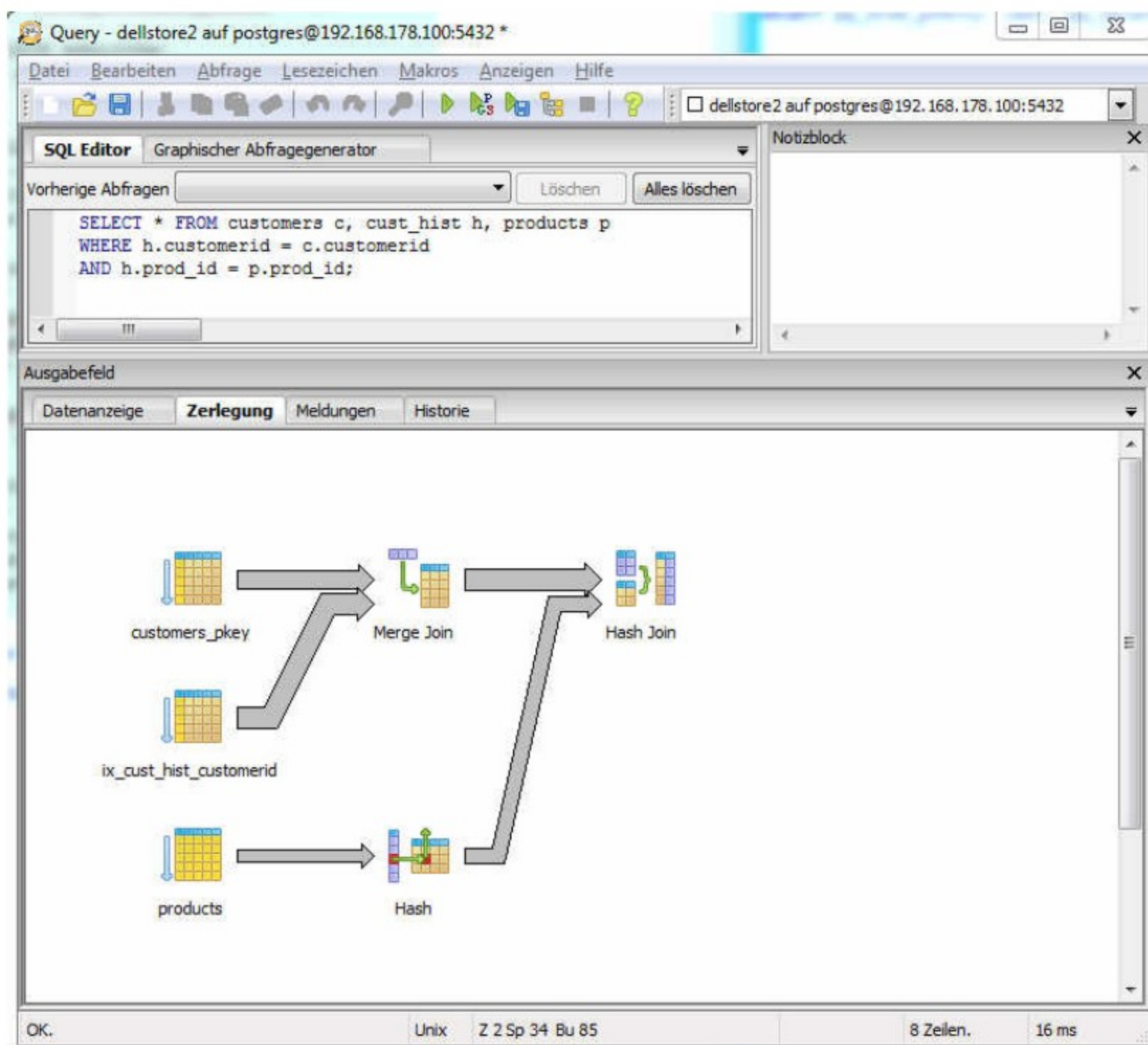


Bild 10.1 Grafische Darstellung von Ausführungsplänen in „pgAdmin III“

„GIS“ in PostGIS steht für „Geographic Information System“, in Fachkreisen auch „Spatial System“ genannt. Das Wort „Spatial“ leitet sich vom lateinischen Wort „spatium“ (Raum) her. In den letzten Jahren hat die Anzahl von Anwendungen, die geografische Informationen speichern und verarbeiten, signifikant zugenommen. Anwendungen reichen von Routenplanern bis zum Einsatz in wissenschaftlichen und militärischen Bereichen.

Wozu dienen Spatial-Datenbanken überhaupt? Eine Spatial-Datenbank ermöglicht es, geometrische Datentypen für geometrische Objekte zu verwenden. Diese Daten sind in der Regel geografischer Natur. Darüber hinaus stellt eine Spatial-Datenbank spezielle Indexe und Funktionen zur Verfügung, mit deren Hilfe die Daten sich effizient und performant verarbeiten lassen. Außerdem vereinfachen die Funktionen den Umgang mit geometrischen Daten und machen die zugehörige Applikation transparent.

Eine Spatial-Datenbank ist also zugleich ein Werkzeug zur Speicherung und zur Analyse von geometrischen (geografischen) Daten. Auf die unterschiedliche Betrachtungsweise von geometrischen und geografischen Daten gehen wir noch detailliert ein. Eine weitere wichtige Funktion der Spatial-Datenbank ist die visuelle Präsentation. Schließlich will man den eigenen Standort, die nächste Tankstelle oder die Entfernung einer Immobilie vom Strand entsprechend auf einer Karte oder Fotografie darstellen können.

■ 11.1 PostGIS und PostgreSQL

PostGIS wurde speziell und exklusiv für PostgreSQL entwickelt und ist ebenfalls ein Open-Source-Produkt. PostgreSQL wurde von PostGIS als relationales Datenbanksystem aufgrund seiner einfachen Erweiterbarkeit beim Erstellen neuer Datentypen und Operatoren und der Möglichkeit der Kontrolle von Index-Operatoren ausgewählt (nicht umgekehrt). Denken Sie hier an die Möglichkeit der Erweiterung der SQL-Sprache, die wir im Entwicklungsteil detailliert dargestellt haben! Aber auch die große Anzahl von Programmier- und Skriptsprachen, die sich bei PostgreSQL anbinden lassen, hat zu dieser Entscheidung beigetragen. PostGIS 2.0 verfügt aktuell über mehr als 300 Operationen, Funktionen und Datentypen, und die Entwicklung ist noch nicht abgeschlossen. Für künftige Versionen erwartet man Erweiterungen und Verbesserungen. All diese Objekte sind in der PostgreSQL-Datenbank gespeichert; Web- und Frontendapplikationen können sie direkt verwenden.

■ 11.2 PostGIS installieren

Zwei Wege führen zu einer PostGIS-Datenbank. Wir zeigen im Folgenden sowohl die einfachere Paket-orientierte Methode für ein Windows-Betriebssystem als auch die Installation aus dem Quellcode unter Linux. Als dieses Buch geschrieben wurde, war PostGIS 2.0 die aktuellste freigegebene Version. Dazu passend empfehlen wir die neueste Version 9.1.3 von PostgreSQL. Verwenden Sie mit PostGIS 2.0 keine Datenbankversion kleiner als 9.1.

11.2.1 Paketorientierte Installation

Installieren Sie zunächst die PostgreSQL-Software. Das Paket finden Sie auf der Webseite <http://enterprisedb.com>. Detaillierte Informationen zur Paketinstallation in Kapitel 2 „Installation und Quellen“. Erstellen Sie einen Standard-Server mit einer Standard-Datenbank „postgresql“, wie in diesem Kapitel beschrieben. Im vorliegenden Beispiel wurde die Version 9.1.3 installiert. Sie können jede Version verwenden, die mit der Version der geplanten PostGIS-Software kompatibel ist.

Listing 11.1 Die Version des PostgreSQL-Server abfragen

```
(postgres@:5432) [postgres] > select version();
               version
-----
PostgreSQL 9.1.3, compiled by Visual C++ build 1500, 64-bit
```

Laden Sie nun das Windows-Paket von der PostGIS-Webseite <http://www.postgis.org> herunter. Das im Beispiel verwendete Paket heißt „postgis-pg91x64-setup-2.0.0-4.exe“. Die Setup-Datei kann zur Installation direkt aufgerufen werden. Im nächsten Fenster können Sie die Optionen für die Installation wählen. Markieren Sie die beiden Optionen „PostGIS“ und „Create spacial database“.

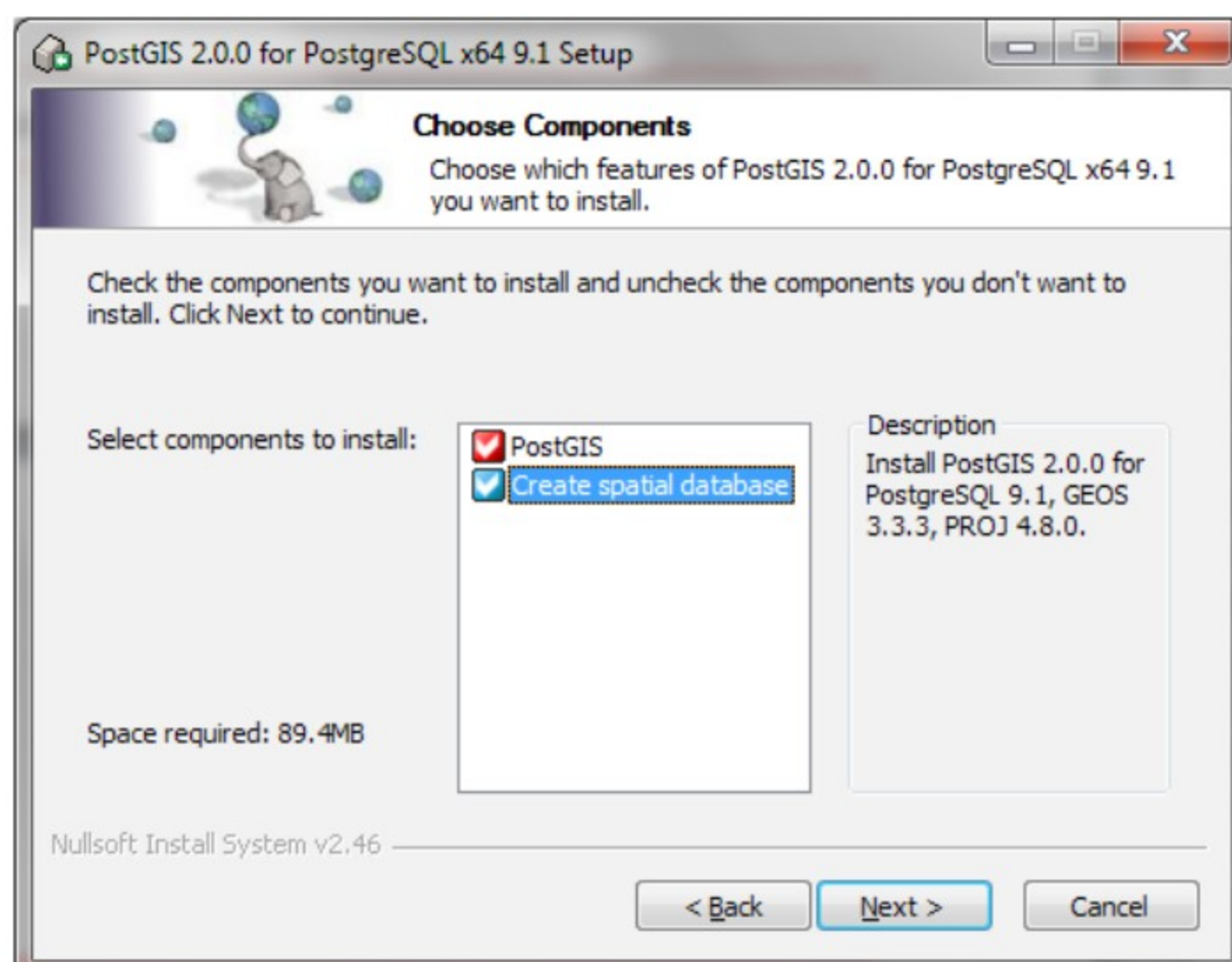


Bild 11.1

Die Optionen für die PostGIS-Installation wählen

Mit der Option „PostGIS“ wird PostGIS 2.0.0 für PostgreSQL 9.1, GEOS 3.3.3 sowie PROJ 4.8.0 installiert. Auf die einzelnen Pakete gehen wir im Abschnitt 11.2.2 „Installation aus dem Quellcode“ näher ein. Mit der Option „Create spacial database“ legen Sie eine neue Spatial-Datenbank direkt mit an.

Im folgenden Schritt wird der Speicherort der Installation des PostgreSQL-Server abgefragt. Danach müssen Sie die Verbindungsinformationen zum Datenbankserver eingeben. Verwenden Sie hier am besten der User „postgres“, der Superadmin-Rechte besitzt.

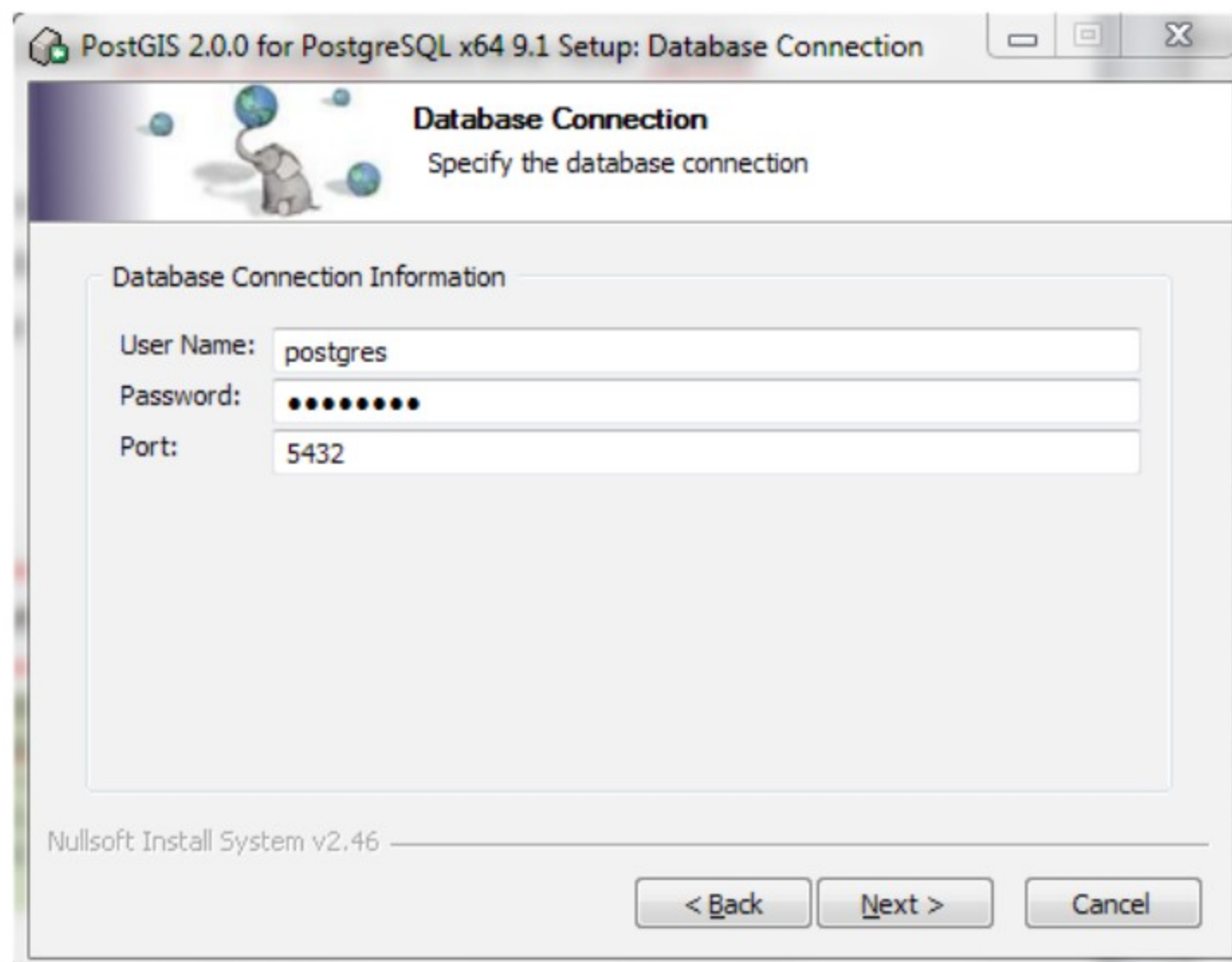


Bild 11.2
Credentials für die
Datenbank-Ver-
bindung eingeben

Schließlich wird noch der Name für die Spatial-Datenbank abgefragt. Danach beginnt die Installation. Der Installer übernimmt die Installation der Software sowie das Erstellen einer neuen Spatial-Datenbank. Die Installation ist nun bereits abgeschlossen.

Wenn Sie jetzt auf den Server schauen, finden Sie die neue Datenbank und können die PostGIS-Version abfragen.

Listing 11.2 Die PostGIS-Version abfragen

```
(postgres@:5432) [postgres] > \l
                          Liste der Datenbanken
  Name | Eigentümer | Kodierung | Sortierfolge
-----+-----+-----+-----
 postgres20 | postgres | UTF8 | German_Germany.1252
 postgres | postgres | UTF8 | German_Germany.1252
 template0 | postgres | UTF8 | German_Germany.1252
 template1 | postgres | UTF8 | German_Germany.1252
 template_postgis_20 | postgres | UTF8 | German_Germany.1252
(postgres@:5432) [postgres] > \c postgres20
(postgres@:5432) [postgres20] > SELECT postgis_full_version();
                                postgis_full_version
-----
 POSTGIS="2.0.0 r9605" GEOS="3.3.3-CAPI-1.7.4" PROJ="Rel. 4.8.0, 6 March
 2012" GDAL="GDAL 1.9.0, released 2011/12/29" LIBXML="2.7.8"
 LIBJSON="UNKNOWN" RASTER
```


Wie Sie sehen, wurde ein Template mit dem Namen „template_postgis_20“ angelegt. Dieses lässt sich zum Anlegen weiterer Spatial-Datenbanken verwenden.

11.2.2 Installation aus dem Quellcode

Die Installation aus dem Quellcode setzt voraus, dass folgende Systeme installiert sind:

- PostgreSQL Server 9.0 oder höher
- Proj4 reprojection library
- GEOS geometry library
- LibXML2

Führen Sie nun die folgenden Schritte aus, um PostGIS zu installieren:

1. Entpacken Sie den Quellcode.

```
# tar xvfz postgis-2.0.0.tar.gz
```

2. Konfigurieren Sie die Installation für die aktuelle Umgebung.

```
# cd postgis-2.0.0
# ./configure
```

3. Erstellen Sie die Binaries, und installieren Sie diese.

```
# make
# make install
```

4. Erstellen Sie eine PostGIS-Datenbank.

```
$ createdb postgis20
$ createlang plpgsql postgis20
$ psql -d postgis20 -f postgis.sql
$ psql -d postgis20 -f postgis_comments.sql
$ psql -f postgis20 -f spatial_ref_sys.sql
```

■ 11.3 Erste Schritte mit PostGIS

Wie bereits erwähnt, unterscheiden Spatial-Datenbanken geometrische und geografische Datentypen und Funktionen. Geometrische Daten lassen sich einfach mit geometrischen Objekten, z.B. Punkten, Geraden, Rechtecken oder Polygonen, wie wir sie aus dem Schulunterricht kennen, assoziieren. Im ersten Beispiel wollen wir einen Punkt darstellen. Die Abfrage in Listing 11.3 liefert eine Zahl, die einen Punkt im kartesischen Koordinatensystem widerspiegelt.

Listing 11.3 Einen Punkt als geografisches Objekt darstellen

```
(postgres@:5432) [postgis20] > SELECT ST_Point(2,3) AS myPoint;
      mypoint
-----
010100000000000000000000004000000000000000840
```

Weder für die Applikation noch für eine Verarbeitung ist es von Interesse, wie die Lange Zahl zustande kommt. Interessant ist die Tatsache, dass es sich um einen Punkt im Koordinatensystem mit entsprechenden Koordinaten handelt. Ein einzelner Punkt ist noch gut vorstellbar, doch es gibt auch komplexere Objekte. Da wäre es gut, diese visualisieren zu können. Auch dafür gibt es natürlich Unterstützung. Eines dieser Produkte ist „OpenJUMP“, ein Open Source-Programm, das sich kostenlos herunterladen lässt.

Mit der folgenden Funktion können Sie eine Linie über drei Punkte in Form eines Daches darstellen:

[illegible]

Aus dem langen Zahlenwert der geometrischen Spalte lässt sich nicht einfach ableiten, um welche geometrische Form es sich handelt. Mit Hilfe vom OpenJUMP ist eine grafische (visuelle) Darstellung möglich.

Starten Sie OpenJUMP, und wählen Sie unter dem Datei-Menü den Punkt „Datenbankabfrage ausführen“. Stellen Sie eine Verbindung zur Datenbank her, und geben Sie die folgende SQL-Anweisung ein:

```
SELECT ST_GeomFromText('LINESTRING(0 0,20 20,40 0)') AS myDach
```

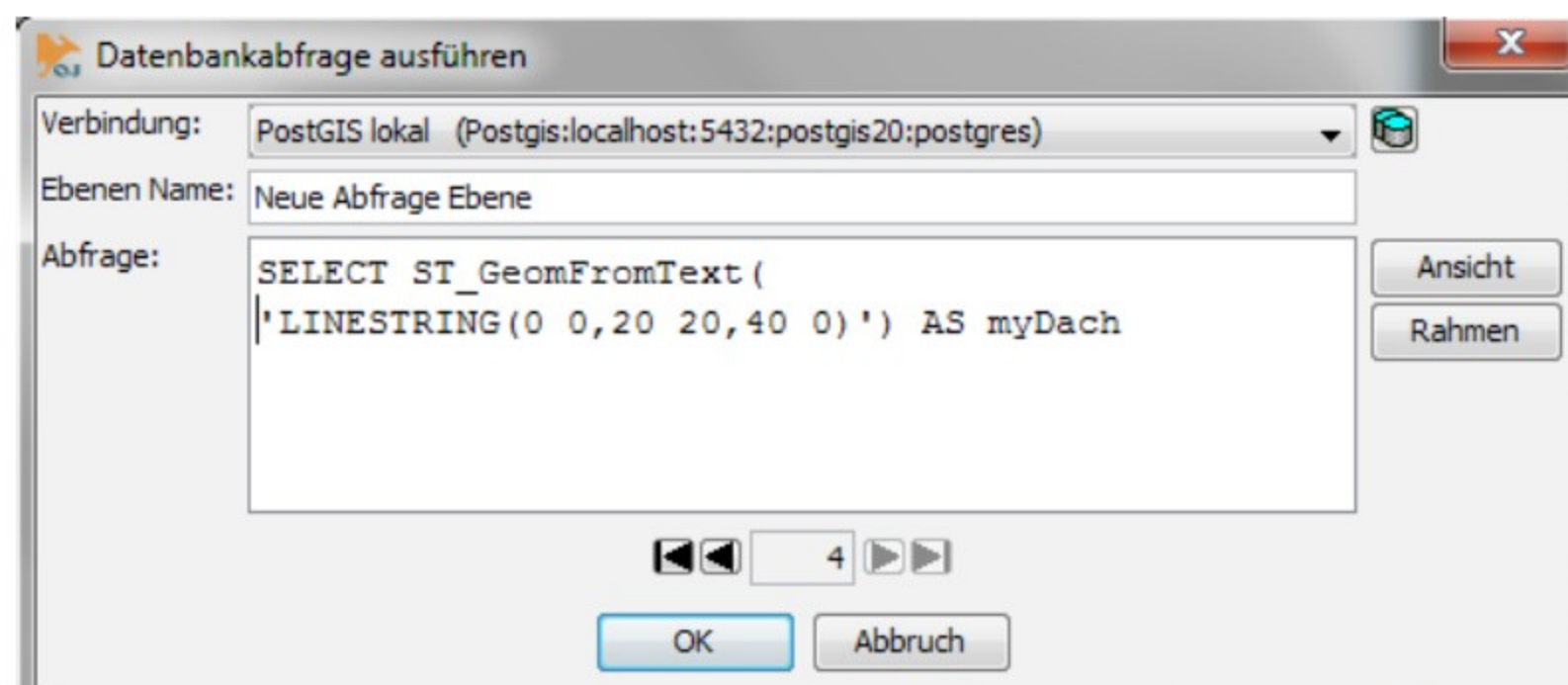


Bild 11.3 Eine Datenbankabfrage in OpenJUMP ausführen

Die Geometrie ist eine Linienverbindung in Form eines Daches. OpenJUMP liefert eine visualisierte Anzeige, die Sie in Bild 11.4 sehen können.

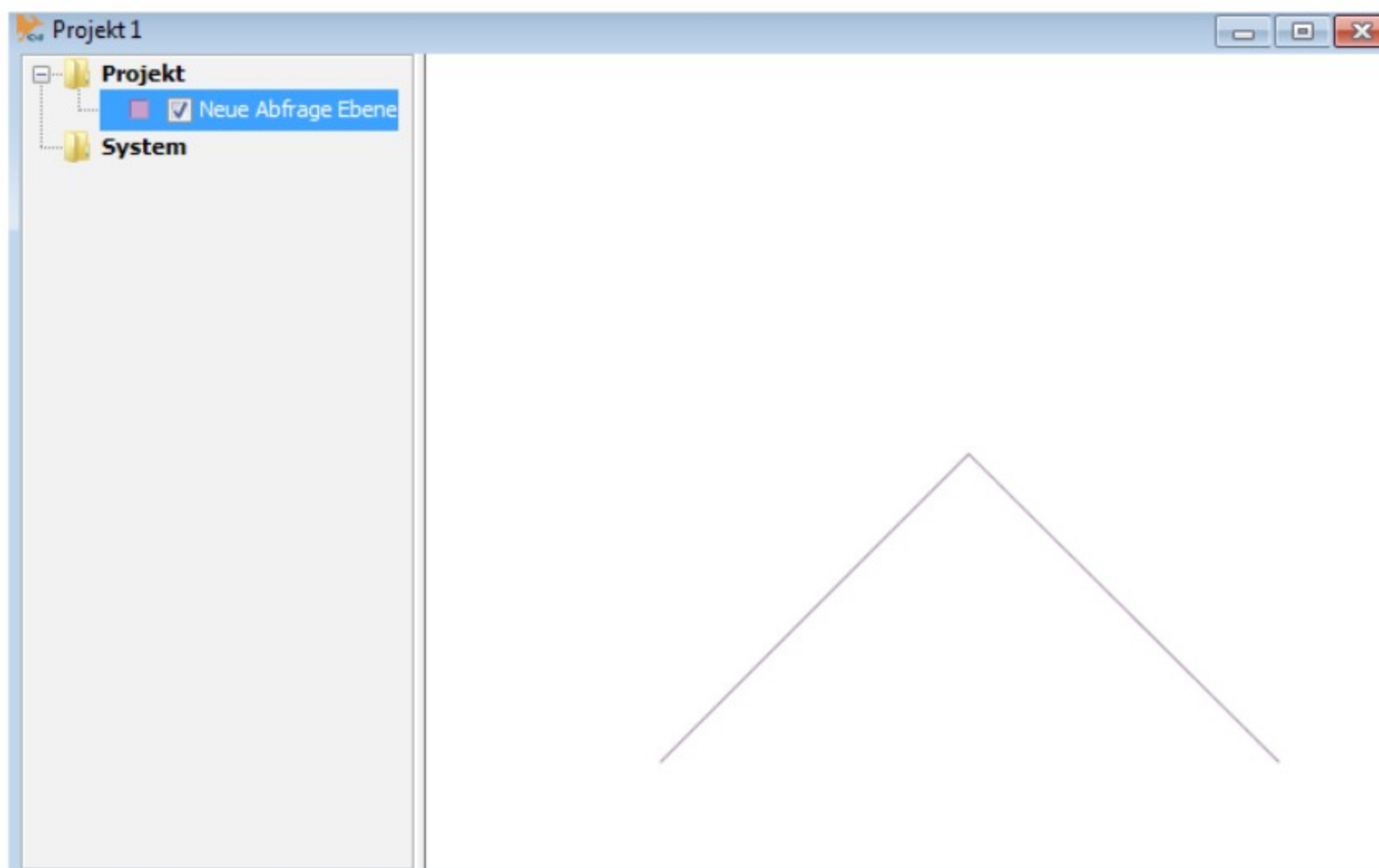


Bild 11.4 Visualisierung einer Datenbank-Abfrage mit OpenJUMP

OpenJUMP bietet noch einiges mehr an Funktionalität und ist für uns an dieser Stelle ein hervorragendes Werkzeug, um geometrische Datenbank-Abfrage zu visualisieren.

Sehen Sie bereits die ersten Vorteile einer Spatial-Datenbank? Geometrische Objekte mit herkömmlichen Datentypen und Funktionen zu speichern und zu verarbeiten dürfte wesentlich aufwendiger sein.

Bisher haben wir nur mit geometrischen Daten gearbeitet, also mit Werten in einem zweidimensionalen Koordinatensystem. Bei einfacher Betrachtung erweisen sich geografische Daten als nichts anderes als geometrische Daten, die sich über Längen- und Breitengrad definieren. Lassen wir die Genauigkeit einmal außen vor, kann damit jeder beliebige Punkt auf der Erde dargestellt werden.

Geografische Datentypen müssen als Längen- und Breitengrade im WGS 84-Format gespeichert werden. Darüber hinaus ist die Maßeinheit in Metern vorgegeben. Wenn die Datenquelle jedoch andere Formate oder Maßeinheiten verwendet, können diese nicht direkt in geografische Datentypen geladen werden. Hier kommt der Vorteil geometrischer Datentypen zum Tragen. Diese lassen sich unabhängig vom Format der Quelle und der Maßeinheit speichern und verarbeiten.

Wann also sollte man geografische anstelle von geometrischen Datentypen verwenden? Die folgenden Punkte erleichtern Ihnen die Entscheidung:

- Befindet sich Ihr Datenbestand in einem relativ kleinen Gebiet, sind geografische Datentypen in der Regel unter dem Blickwinkel verfügbarer Funktionen und Performance die bessere Lösung.
- Falls Sie mit globalen Daten oder größeren Regionen arbeiten, bieten geografische Datentypen den Vorteil, dass man sich nicht um Umwandlungen und Projektionen kümmern muss. Beachten Sie jedoch, dass aktuell für geografische Datentypen weniger Funktionen zur Verfügung stehen als für geometrische.

- Geometrische Datentypen bedingen fast immer Projektion. Wenn Sie sich mit dem Thema Projektionen nicht beschäftigen oder diese nicht anwenden wollen, sind geografische Datentypen die bessere Wahl. Sie sind einfacher zu benutzen.
- Ein Vorteil geometrischer Datentypen besteht darin, dass sie in einem beliebigen Spatial-System gespeichert werden können. Maßeinheit ist immer die Einheit des entsprechenden Systems. Eine Übernahme oder Migration in ein anderes System wird damit vereinfacht. Bei der Verwendung geografischer Datentypen ist zum Beispiel zu beachten, dass der Microsoft SQL-Server verschiedene Referenzsysteme unterstützt, während PostGIS ausschließlich das System „EPSG:4326“ (auch bekannt als „WGS 84 Lon Lat“) unterstützt.

Im Folgenden betrachten wir eine Beispiel-Tabelle „tankstellen“. Diese enthält zunächst neben dem Tankstellen-Identifizier nur den Längen- und den Breitengrad.

Listing 11.4 Tabelle „tankstellen“ mit Längen- und Breitengrad

```
(postgres@:5432) [postgis20] > \d tankstellen
          Tabelle public.tankstellen
Spalte |          Typ          |          Attribute
-----+-----+-----
ts_id  | integer              | Vorgabewert nextval('ts'::regclass)
lat    | double precision     |
lon    | double precision     |
(postgres@:5432) [postgis20] > SELECT * FROM tankstellen LIMIT 10;
ts_id | lat    | lon
-----+-----+-----
49990 | 25.8092 | -80.24
49991 | 25.8587 | -80.2094
49992 | 25.4849 | -80.461
49993 | 25.8471 | -80.2415
49994 | 25.6849 | -80.3125
49995 | 25.7502 | -80.2429
49996 | 25.6003 | -80.3537
49997 | 25.8306 | -80.2661
49998 | 25.926  | -80.1617
49999 | 25.9466 | -80.2056
(10 Zeilen)
```

Da wir PostGIS benutzen können, wollen wir natürlich nicht mit Längen- und Breitengrad arbeiten und betrachten zunächst einen geometrischen Datentyp. PostGIS hält die Funktion *AddGeometryColumn* bereit, um eine Spalte vom geometrischen Datentyp an die Tabelle „tankstellen“ anzuhängen. Sie besitzt folgende Syntax:

```
AddGeometryColumn(<schema_name>,<table_name>,<column_name>,<srid>,<type>,<dimension>)
```

Die Parameter haben folgende Bedeutung:

<schema_name>... Name des Schemas, unter dem die Tabelle gespeichert ist

<table_name>... Name der Tabelle

<column_name>... Name der hinzuzufügenden Spalte

<srid>... Spatial Reference Identifier für die Projektion

<type>... Objekttyp, zum Beispiel „POINT“ oder „LINESTRING“

<dimension>... Dimension des Koordinatensystems, in der Regel „2“

An dieser Stelle müssen wir die Abkürzung „SRID“ erklären. „SRID“ steht für Spatial Reference Identifier und repräsentiert das Spatial-Referenzsystem. Letzteres haben wir bereits erwähnt: das EPSG:4326 (WGS 84). Alle dem System bekannten Referenzsysteme sind in der Tabelle „spatial_ref_sys“ gespeichert. Sie enthält den Namen des Systems sowie die erforderlichen Parameter zur Projektion auf andere Systeme. In Listing 11.5 finden Sie den Eintrag für EPSG:4326.

Listing 11.5 Eintrag für EPSG:4326 (WGS84) in der Tabelle „spatial_ref_sys“

```
(postgres@:5432) [postgis20] > SELECT * FROM spatial_ref_sys
WHERE srid=4326;
-[ RECORD 1 ]-----
srid      | 4326
auth_name | EPSG
auth_srid | 4326
srttext   | GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS
84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY
["EPSG","6326"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],
AUTHORITY["EPSG","4326"]]
proj4text | +proj=longlat +datum=WGS84 +no_defs
```

An die Tabelle „tankstellen“ sollen nun eine geometrische Spalte angehängt und die Werte für Längen- und Breitengrad entsprechend projiziert und eingetragen werden. Für die Projektion verwenden wir EPSG:2163 (USA-Kontinent). Mit Hilfe der Funktion „ST_Transform“ wird die geometrische Spalte für die bereits vorhandenen Daten gefüllt.

Listing 11.6 Eine Spalte vom geometrischen Datentyp anhängen

```
(postgres@:5432) [postgis20] > SELECT AddGeometryColumn('', 'tankstellen',
> 'geom_col', '2163', 'POINT', 2);
          addgeometrycolumn
-----
public.tankstellen.geom_col SRID:2163 TYPE:POINT DIMS:2
(1 Zeile)
(postgres@:5432) [postgis20] > \d tankstellen
      Tabelle 'public.tankstellen'
  Spalte |          Typ          | Attribute
-----+-----+-----
ts_id    | integer               |
lat      | double precision      |
lon      | double precision      |
geom_col | geometry(Point,2163) |
```

Schließlich muss die Spalte noch gefüllt werden. Dies erfolgt mit Hilfe einer Transformationsfunktion. Sie projiziert Längen- und Breitengrad in einen geometrischen Wert.

Listing 11.7 Die geometrische Spalte füllen

```
(postgres@:5432) [postgis20] > UPDATE tankstellen
> SET geom_col = ST_Transform(
> ST_GeomFromText('POINT('||lon||' '||lat||')', 4326), 2163);
(postgres@:5432) [postgis20] > SELECT * FROM tankstellen LIMIT 10;
ts_id | lat    | lon    | geom_col
-----+-----+-----+-----
49990 | 25.8092 | -80.24 | 010100002073080000AABD80B50C4D3E4132F12FAA87FE3CC1
```



```

49991 | 25.8587 | -80.20 | 010100002073080000518AF5F5F4543E412AA9D6D2D7E63CC1
49992 | 25.4849 | -80.46 | 0101000020730800008AA81DEEE7103E416481C7FD6B9B3DC1
49993 | 25.8471 | -80.24 | 010100002073080000938E7B1B86493E41E4C76F6D99EE3CC1
49994 | 25.6849 | -80.31 | 010100002073080000CC1EAB6DCA3A3E416060E88E9C393DC1
49995 | 25.7502 | -80.24 | 01010000207308000009FC5E4985503E41716E1B9DCA173DC1
49996 | 25.6003 | -80.35 | 010100002073080000ECCCC3F971313E417966EC3518613DC1
49997 | 25.8306 | -80.26 | 01010000207308000042679BC159413E41F749F91BC4F73CC1
49998 | 25.926   | -80.16 | 010100002073080000B7BD026E05623E418551888217C63CC1
49999 | 25.9466  | -80.20 | 010100002073080000F3AF411A8E4F3E412565AE3740C13CC1
(10 Zeilen)

```

Sie erkennen bereits die Vorzüge, aber auch die Problematiken geometrischer Datentypen. Einerseits lassen sich die Daten aufgrund der Vielzahl von Funktionen recht flexibel umwandeln (projizieren). Da andererseits die Daten im projizierten Format gespeichert sind, ist man an diese Projektion gebunden oder muss erneut projizieren.

Die geometrische Spalte enthält nun die Informationen über die Lage der Tankstellen im Koordinatensystem. Von der Applikation können diese Daten verwendet werden, um beispielsweise die in einem Umkreis des aktuellen Standpunkts am nächsten gelegenen Tankstellen zu finden.

Die Tabelle „tankstellen“ enthält ca. 50 000 Sätze. Um gute Antwortzeiten zu garantieren, wird auf der geometrischen Spalte ein Spatial-Index angelegt.

Listing 11.8 Einen Index auf der geometrischen Spalte anlegen

```

(postgres@:5432) [postgis20] > CREATE INDEX i_geom_tankstellen
> ON tankstellen USING gist(geom_col);
CREATE INDEX

```

Nun fügen wir eine Spalte mit geografischem Datentyp hinzu. Dies erfolgt mit Hilfe der Funktion „GEOGRAPHY“.

Listing 11.9 Eine Spalte mit geografischem Datentyp hinzufügen

```

(postgres@:5432) [postgis20] > ALTER TABLE tankstellen
> ADD COLUMN geog_col GEOGRAPHY(POINT);'
ALTER TABLE
(postgres@:5432) [postgis20] > \d tankstellen

```

Spalte	Typ	Attribute
ts_id	integer	Vorgabewert nextval('ts'::regclass)
lat	double precision	
lon	double precision	
geom_col	geometry(Point,2163)	
geog_col	geography(Point,4326)	

Indexe:

```

    "i_geom_tankstellen" gist (geom_col)

```


**Hinweis**

Beim Hinzufügen der geografischen Spalte wurde keine SRID angegeben. In der aktuellen Version 2.0 ist „4326“ der Standard und gleichzeitig die einzige Option. Für zukünftige Versionen ist die Unterstützung für weitere SRID zu erwarten.

Analog zur geometrischen Spalte wird die geografische Spalte gefüllt, und es wird ein Spatial-Index angelegt.

Listing 11.10 Füllen der geografischen Spalte

```
(postgres@:5432) [postgis20] > UPDATE tankstellen
> SET geog_col=ST_GeogFromText('SRID=4326;POINT('||lon||' '||lat||')');
(postgres@:5432) [postgis20] > CREATE INDEX i_geog_tankstellen
> ON tankstellen USING gist(geog_col);
CREATE INDEX
```

■ 11.4 SQL-Abfragen in PostGIS

Die häufigsten Abfragen werden in Zusammenhang mit dem „Nearest Neighbor“-Problem gestellt. Solche Abfragen werden auch als Umgebungsanalyse (Proximity Analysis) bezeichnet. Es geht also um die Entfernung von Objekten zueinander. Für unser Beispiel wäre eine typische Abfrage: Welche Tankstellen befinden sich in einem Umkreis von 10 km vom aktuellen Standpunkt. Um den aktuellen Standpunkt in geometrische Daten umzuwandeln, verwenden wir eine Tabelle „strassen“. Diese enthält verbale Informationen über den aktuellen Standort sowie eine Spalte, mit der dieser Standort geometrisch dargestellt wird. Die Entfernung wird mit Hilfe der Funktion „ST_DWithin“ ermittelt. Die Maßeinheit ist Meter.

Listing 11.11 Tankstellen in einer Umgebung von 10 km abfragen

```
(postgres@:5432) [postgis20] > SELECT td.name, td.ort, td.strasse
> FROM tankstellen AS ts
> INNER JOIN ts_desc AS td
> ON ts.ts_id = td.ts_id
> INNER JOIN strassen AS st
> ON ST_DWithin(ts.geom_col, st.geom, 10000)
> WHERE st.name = 'Autobahnkreuz Heidelberg';
name | ort | strasse
-----+-----+-----
ESS0 | Mannheim | Autobahn 5
Total | Heidelberg | Orteingang
Shell | Mannheim | Am Tor
```



Hinweis

Die Abfrage wurde in einer Zeit von 3,8 ms ausgeführt. Das ist ein hervorragendes Ergebnis. Die kurze Antwortzeit wird durch den Spatial-Index ermöglicht. Für die Applikationen, die hinter solchen Abfragen stecken, sind kurze Antwortzeiten sehr wichtig. Zusätzlich muss eingeplant werden, dass Tausende solcher Anfragen parallel gestellt werden. Mit Hilfe des Spatial-Index wird nicht nur die Antwortzeit reduziert, sondern auch die Systembelastung klein gehalten.

Wie hätten Sie das Problem ohne PostGIS gelöst? Das ist sicher machbar, aber wesentlich aufwendiger, und man würde sich die Abfragen mit einer höheren Ressourcenbelastung und schlechteren Performance erkaufen.

Nun sollten Sie noch einige Abfragen ausführen, um folgende Ergebnisse zu erzielen:

- Darstellung des Verlaufs der Autobahn
- Markieren der Tankstellen, die sich in einer maximalen Entfernung von der Autobahn befinden
- Darstellung des betrachteten Radius

Die Abfrage nach dem Verlauf der Autobahn ist einfach. Die Funktion `ST_AsBinary` liefert die Objekte zurück.

Listing 11.12 Abfrage des Straßenverlaufs

```
(postgres@:5432) [postgis20] > SELECT st.gid, st.name, ST_AsBinary(st.geom)
> AS verlauf
> FROM strassen AS st
> WHERE r.name = 'Autobahn 3';
```

Nun gilt es, die Tankstellen abzufragen, die sich in einer maximalen Entfernung von 10 km von der Autobahn befinden.

Listing 11.13 Abfrage der Tankstellen in einer maximalen Entfernung von 10 km

```
(postgres@:5432) [postgis20] > SELECT
> ST_AsBinary(ts.geom_col) AS entfernung
> FROM tankstellen AS ts
> WHERE EXISTS(SELECT st.gid
> FROM strassen AS st
> WHERE ST_DWithin(ts.geom_col, st.geom, 10000)
> AND st.name = 'Autobahn 3'
> );
```

Schließlich soll zur optischen Verbesserung die betrachtete Umgebung als Fläche dargestellt werden. Die zugehörige SQL-Anweisung finden Sie in Listing 11.14.

Listing 11.14 Betrachtetes Gebiet als Fläche darstellen

```
(postgres@:5432) [postgis20] > SELECT
> ST_AsBinary(ST_Union(ST_Buffer(st.geom,10000))) AS flaeche
> FROM strassen As st
> WHERE st.name = 'Autobahn 3';
```


Da wir an dieser Stelle über keine Applikation verfügen, um die Ergebnisse grafisch anzuzeigen, verwenden wir „OpenJUMP“ für die Darstellung. Die Ergebnisse der drei SQL-Abfragen werden übereinandergelegt und ergeben die Abbildung in Bild 11.5.

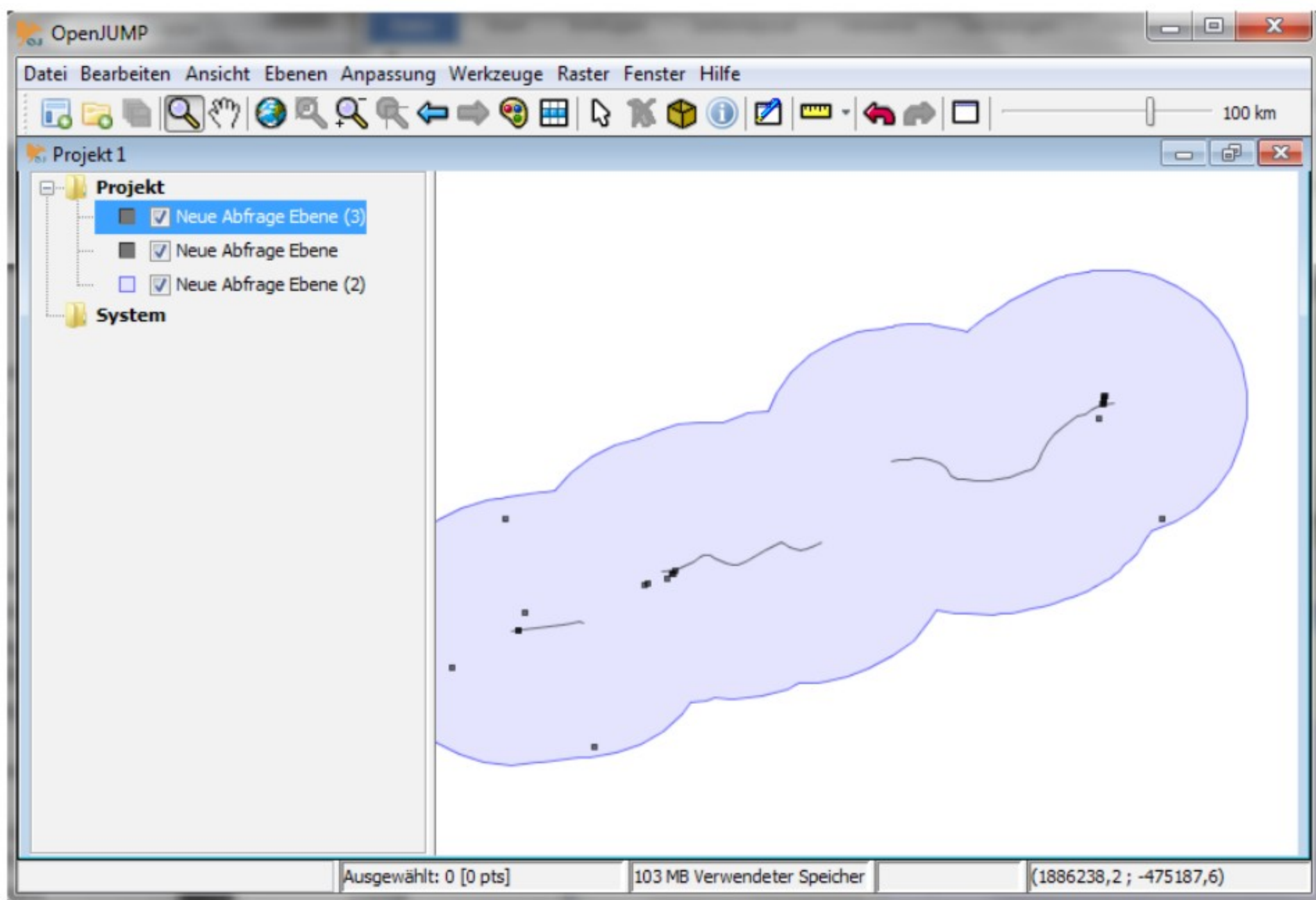


Bild 11.5 Darstellen der Spatial-Abfragen mit OpenJUMP

Wir waren in der Lage, mit relativ wenig Aufwand sowie Unterstützung durch PostGIS eine typische Aufgabenstellung zu implementieren und universelle Abfragen zu bilden. Das Kapitel erhebt keineswegs Anspruch auf eine vollständige Darstellung von PostGIS. Es ist eine Einführung und dient als Appetitanreger, um PostgreSQL als Spatial-Datenbank zu verwenden.

In den vorhergehenden Kapiteln haben wir die einzelnen Features von PostgreSQL auch immer unter dem Blickwinkel der Betriebstauglichkeit durchleuchtet. Was nützt ein noch so gutes Datenbankbetriebssystem, wenn es nicht den Anforderungen für einen professionellen Betrieb genügt? Obwohl im Fall von PostgreSQL 9 viele dieser Fragen bereits positiv beantwortet werden konnten, soll die Datenbank in diesem Kapitel noch einmal gezielt auf dieses eine Kriterium untersucht werden. Somit können Sie sich ein Bild davon machen, ob PostgreSQL für den betrieblichen Einsatz in Ihrem Umfeld geeignet ist oder nicht. In diesem Sinn: Setzen Sie nun bitte Ihre „Betriebsbrille“ auf!

Welche Voraussetzungen müssen erfüllt sein, um ein Datenbank-Betriebssystem erfolgreich in den Betrieb zu integrieren? Obwohl sich Betriebsumgebungen im Detail unterscheiden, lassen sich dennoch allgemeingültige Kriterien für einen erfolgreichen Betrieb in den folgenden Kategorien festlegen:

- Datensicherung und Wiederherstellung
- Disaster Recovery
- Überwachung
- Administrierbarkeit
- Verfügbarkeit
- Datensicherheit und Auditing
- Performance und Skalierbarkeit
- Schnittstellen und Kommunikation
- Support

■ 12.1 Funktionen und Features

Der Umfang der Funktionen und Features von PostgreSQL ist wirklich beeindruckend und im Open-Source-Umfeld insofern auch führend. Darüber hinaus ist der Funktionsumfang durchaus vergleichbar mit den großen kommerziellen Datenbanken wie Oracle, DB/2, Sybase, MS SQLServer oder Informix. Dabei ist es den Entwicklern von PostgreSQL stets gelungen, sich auf das Kernprodukt, die Datenbank-Engine, zu beschränken.

In puncto Erweiterbarkeit oder Einbindung von Skripten und Sprachen übertrifft PostgreSQL teilweise sogar das eine oder andere kommerzielle Produkt. Denken Sie nur an die Möglichkeit der SQL-Erweiterung. Von Vorteil ist dabei das Konzept, verschiedene Programmier- und Skriptsprachen in die Engine einzubinden. Das bietet Performance-Vorteile gegenüber anderen Lösungen und gestattet es, prinzipiell jede Sprache einzubinden.

PostgreSQL bietet eine solide Unterstützung für den Einsatz als Data Warehouse-Datenbank. Insbesondere die Partitionierung, das Neuschreiben der Abfrage (Query Rewrite) und die analytischen Funktionen sind ein wesentlicher Baustein. Dabei wurde auch in diesem Bereich das Grundprinzip eingehalten: Fokus auf die Datenbank-Engine. Schließlich findet man im Data-Warehouse-Umfeld genügend Werkzeuge und Toolsets für den Aufbau und den Betrieb.

Mit PostGIS bietet PostgreSQL Unterstützung für GIS-Applikationen. PostGIS verfügt über eine starke Integration in die PostgreSQL Engine und wird von GIS-Spezialisten als das führende Produkt im Bereich der Spatial-Datenbanken eingestuft. Es handelt sich um ein modernes Produkt, das aktuell auch noch weiterentwickelt wird und gegen die Konkurrenzprodukte „OracleSpatial“ und „SQL Server Spatial“ mehr als mithalten kann.

Als Fazit lässt sich festhalten, dass PostgreSQL eine moderne Datenbank ist, deren Funktionen und Features als Datenspeicher den Anforderungen von Applikationen und Projekten aus den verschiedensten Branchen gerecht wird.

■ 12.2 Datensicherung und Wiederherstellung

Zu den wichtigsten Anforderungen gehört die Möglichkeit, eine Sicherung im laufenden Betrieb durchzuführen sowie die Wiederherstellung zu einem beliebigen Zeitpunkt zu garantieren (Point-in-time Recovery).

Mit Aktivierung der WAL-Archivierung sind sowohl eine Online-Sicherung als auch die Wiederherstellung der Datenbank zu einem beliebigen Zeitpunkt möglich. Die Sicherung der archivierten WAL Records auf ein externes Medium garantiert die Wiederherstellung auch bei einem Totalverlust des Datenbank-Servers bis zum letzten gesicherten Satz.

Damit genügt PostgreSQL den Mindestanforderungen für produktive Systeme. Eine zusätzliche Sicherung mit SQL-Dump kann eine sinnvolle Ergänzung der Backup-Strategie sein. SQL-Dump erstellt eine Textdatei mit SQL-Befehlen, mit deren Hilfe die Datenbank ganz oder teilweise wiederhergestellt werden kann. Die Wiederherstellung ist allerdings nur für den Zeitpunkt der Sicherung möglich. Mit der Option „schema_only“ können die Objekte der Datenbank ohne Dateninhalt ausgegeben werden. Ein solcher Dump ist schnell erstellt, verbraucht wenig Platz und kann bei Verlust einzelner Objekte sehr hilfreich sein.

Mit diesen Backup-Optionen unterscheidet sich PostgreSQL nicht von anderen Datenbank-Systemen. Auch hier besteht das Problem, dass eine Datenbank-Sicherung nicht den Verlust einzelner Objekte abdeckt. Es gibt die Option, das Backup auf eine getrennte Hardware zurück zu speichern und daraus das gelöschte Objekt zu exportieren. Bei größeren Datenbanken wird häufig darauf verzichtet, den Speicherplatz für eine zusätzliche Umgebung vorzuhalten.

Häufig werden im Umfeld von produktiven Datenbanken User Acceptance-Datenbanken (UAT) eingesetzt – Kopien von Produktionsdatenbanken oder produktionsnahe Kopien. Auf UAT-Datenbanken kann ein neues Release der Applikation getestet werden, bevor es in die Produktion geht, oder es können Infrastruktur-Patches ausprobiert werden. Da es nur eine Frage der Zeit ist, bis die UAT-Datenbank von der Produktionsdatenbank abweicht, besteht die Notwendigkeit eines regelmäßigen Abgleichs. Die regelmäßige Wiederherstellung der UAT-Datenbank aus einem produktiven Backup besitzt den Charme, dass einerseits das produktive System für das Erstellen der Kopie nicht belastet oder gestoppt werden muss und andererseits der Nachweis erbracht wird, dass die Sicherung der Produktions-Datenbanken im Bedarfsfall auch funktioniert. Man spart sich also regelmäßige Restore-Tests. Aus Sicherheitsgründen sollten die Daten nach der Wiederherstellung anonymisiert werden. Alles in allem erfüllt PostgreSQL damit die Anforderungen an Backup und Recovery im produktiven Umfeld. Wünschenswert für zukünftige Versionen wäre die Möglichkeit einer inkrementellen Sicherung. Realistisch gesehen ist dies allerdings nicht einfach umsetzbar.

■ 12.3 Disaster Recovery (DR)

Disaster Recovery (DR) ist erforderlich, wenn es zum Totalausfall des Primärsystems oder auch der gesamten Infrastruktur in der Umgebung des Primärsystems kommt. Die Ausfallszenarien reichen vom Verlust des Datenbank-Servers bis zum Verlust des gesamten Data Center. Auch beim einfacheren Szenario, dem Verlust des Datenbank-Servers, stellt sich die Forderung nach möglichst geringen Recovery-Zeiten, um den Datenbank-Server möglichst schnell wieder zur Verfügung stellen zu können.

Konzepte für DR-Lösungen sehen daher häufig den Einsatz von Standby-Datenbanken vor. Diese können sich im gleichen Data Center oder einer entfernten Lokation befinden. Sie stellen eine kostengünstige und sichere Lösung dar und funktionieren auch über größere Entfernungen.

PostgreSQL hat in der Version 9 diese Funktionalität mit den Features „Streaming Replication“ und „Hot Standby“ noch einmal gestärkt. „Streaming Replication“ garantiert eine zeitnahe Übertragung von WAL Records bis hin zur synchronen Replikation. Damit ist der Datenverlust im DR-Fall minimal oder nicht vorhanden.

Das Feature „Hot Standby“ ermöglicht eine Nutzung der Standby-Datenbank im Normalbetrieb. Die Verlagerung von Reports und Ad-hoc-Abfragen auf den Standby-Server ermöglicht eine sinnvolle Nutzung der Standby-Datenbanken im Normalbetrieb und entlastet gleichzeitig das Primärsystem.

Damit erfüllt PostgreSQL die wesentlichen Anforderungen an die DR-Fähigkeit der Datenbanken. Für zukünftige Versionen würde man sich noch Kleinigkeiten wie ein verbessertes und vereinfachtes Switchover und Failover oder die Möglichkeit einer verzögerten Einarbeitung der WAL Records wünschen. Das ist sicher nur eine Frage der Zeit und steht bei der Developer Group bereits auf der Agenda.

■ 12.4 Überwachung

Monitoring ist ein wichtiger Baustein, um die Verfügbarkeit des Datenbankservers zu gewährleisten. Dabei ist die Möglichkeit der Einbindung in eine vorhandene betriebliche Überwachungsstruktur ein entscheidendes Kriterium für den Einsatz.

PostgreSQL bietet eine Reihe von Statistik-Views, deren Daten direkt in das Monitoring einfließen können. Diese reichen von Informationen über verbundene Benutzer, Sperren auf Objekten bis hin zu Performance-Daten.

Auch wenn der PostgreSQL-Server kein eigenes Framework für das Monitoring mitbringt, lässt er sich recht einfach in eine vorhandene Infrastruktur einbinden. Da wir uns im Open Source-Bereich bewegen, ist die Überwachung mit „Nagios“ zu empfehlen. Dafür stehen vorgefertigte Module zur Verfügung.

Viele Monitoring-Frameworks verfügen über eine SNMP-Schnittstelle. Mit Hilfe von Agents, die auf den Zielsystemen laufen, können Informationen als SNMP-Traps in das zentrale Monitoring geladen werden. Das Projekt „pgsnmp“ als Teil des pgFoundry-Frameworks liefert ein gutes Beispiel dafür, wie man eine solche Infrastruktur realisiert.

■ 12.5 Administrierbarkeit

Im Bereich Administrierbarkeit ist PostgreSQL mit kommerziellen Datenbanksystemen durchaus vergleichbar. Neben den üblichen Kommandozeilen-Werkzeugen, die für einen Batch-Betrieb unabdingbar sind, steht mit „pgAdmin III“ ein grafisches Tool zur Verfügung, das die Administration wesentlich erleichtert. Neben einem Objekt-Browser enthält es eine SQL- und eine PL/pgSQL-Konsole sowie viele weitere nützliche Features.

Aufgrund seiner architektonischen Nähe zu Oracle sowie des hohen Kompatibilitätsgrades zwischen PL/SQL und PL/pgSQL sind erfahrene Oracle-DBA's in der Lage, in sehr kurzer Zeit die grundlegenden Aufgaben zu lernen und die Administration zu übernehmen.

Allgemein ist festzustellen, dass PostgreSQL ein sehr Administrator-freundliches System ist. Dies liegt sicher auch an den vielen kleinen Features, wie das Recycling von Log- und Tracedateien, die die tägliche Administration vereinfachen.

In diesem Zusammenhang lässt sich auch festhalten, dass PostgreSQL ein wartungsarmes Datenbanksystem ist. Die wichtigsten Wartungsarbeiten, wie „VACUUM“ oder „ANALYZE“, können weitgehend automatisiert werden. PostgreSQL verfügt über eine sehr gute Out-of-the-box-Funktionalität, d.h. es sind nur wenige Anpassungen vorzunehmen, um einen Datenbank-Server in den Betrieb zu integrieren. Dies betrifft sowohl Anpassungen, die aus rein technischer Sicht als auch aus Sicherheitsgründen notwendig sind.

Mit dem grafischen Tool „pgAdmin III“ ist eine zentrale Verwaltung aller eingesetzten PostgreSQL-Server möglich. Die Funktionalität ist vergleichbar mit den Basisfunktionen des Enterprise Manager von MS SQL oder Oracle. Inzwischen gibt es Erweiterungen von Drittanbietern, mit denen die Funktionalität mit den erwähnten Produkten vergleichbar ist. Dazu werden Agenten auf den Zielsystemen installiert, die eine umfangreiche Enterprise-Funktionalität gewährleisten.

■ 12.6 Verfügbarkeit

Im Bereich Verfügbarkeit wurden im Laufe der vergangenen Jahre immer höhere Anforderungen an Datenbanken gestellt. Für viele Datenbanken ist ein 24x7-Betrieb inzwischen der Normalfall, und es ist für die Administratoren schwierig, eine Downtime für Wartungsaufgaben zu erhalten. Man erwartet von einem Datenbankserver, dass der überwiegende Anteil der notwendigen Wartung im laufenden Betrieb erfolgt.

Mit der Möglichkeit des Online-Backups erfüllt PostgreSQL schon mal eine wesentliche Anforderung. Mit einer Downtime zum Zwecke der Datensicherung lässt sich heutzutage nicht mehr argumentieren. Auch die Möglichkeit, Server-Parameter im laufenden Betrieb zu ändern, trägt wesentlich zur Erhöhung der Verfügbarkeit bei. Hier hat PostgreSQL einen guten Status, natürlich können nicht alle Parameter online geändert werden.

Mit der Einführung von „Streaming Replication“ ist es einfacher geworden, zwischen Primär- und Standby-Datenbank hin- und herzuschalten. Diesen Umstand kann man sich für Patching und Upgrades zunutze machen und damit die geplante Downtime für Wartungsarbeiten reduzieren.

Die Tatsache, dass PostgreSQL ein wartungsarmes Datenbank-System ist, hat positiven Einfluss auf die Verfügbarkeit. Die meisten Wartungsarbeiten wie „VACUUM“ oder „ANALYZE“ erfolgen im laufenden Betrieb. Datenbank-Sicherungen können im laufenden Betrieb vorgenommen werden.

Auch wenn PostgreSQL ohne eigene Software für Hochverfügbarkeit ist, lässt sich der Server problemlos in vorhandene Cluster-Lösungen einbinden. Auf Linux-Ebene bietet sich ein Linux-Cluster mit DRBD-Replikation an. Die Anbindung an ein Windows-Cluster (Aktiv/Passiv) funktioniert problemlos und kann ohne Zusatzsoftware vorgenommen werden. Ein Vergleich mit Oracle Real Application Cluster ist an dieser Stelle nicht sinnvoll, da Oracle die einzige Shared-Everything-Lösung überhaupt anbietet. Andere Datenbanken wie MS SQL oder DB/2 verwenden eine Shared-Nothing-Architektur. Hier sind die Features und Failoverzeiten mit einem Aktiv/Passiv-Cluster vergleichbar. Nicht vergessen werden sollte an dieser Stelle die Standby-Datenbank mit Streaming Replication und der Möglichkeit der synchronen Replikation.

Die Möglichkeiten einer hohen Verfügbarkeit sind also gegeben. Damit ist der PostgreSQL-Server in keinem Fall ein Außenseiter in der Betriebsstruktur.

■ 12.7 Datensicherheit und Auditing

PostgreSQL bietet ein geschlossenes Benutzer- und Rollenkonzept im Server und den einzelnen Datenbanken. Die wenigen erweiterten Privilegien können auf Administratoren begrenzt werden. Mit Hilfe des Systems lässt sich eine Rollentrennung zwischen Administration und Applikation vornehmen. Die Rechteverwaltung ist granular und kann bis auf einzelne Objekte beschnitten werden.

Passwörter sind stark verschlüsselt, und bei der Authentifizierung erfolgt die Übertragung über das Netzwerk ausschließlich in verschlüsselter Form. Mit Hilfe des Moduls „password-check“ ist es möglich, schwache Passwörter nicht zuzulassen und die Passwortregeln der Unternehmenspolicy anzupassen.

Eine flexible Zugangskontrolle (Datei „pg_hba.conf“) ermöglicht eine Einschränkung des Zugangs nach den Anforderungen in der betrieblichen Umgebung, unabhängig von Rechtevergabe über Privilegien und Rollen. Für sicherheitsrelevante Datenbanken kann eine Datenübertragung über SSL-Verschlüsselung eingerichtet werden.

Das Privilegien- und Rollenkonzept ermöglicht eine strikte Trennung zwischen Eigentümer der Datenbankobjekte und Benutzer der Applikationsschemata. Benutzer sind damit nicht in der Lage, Veränderungen im Datenbankschema vorzunehmen. Die Umsetzung von Änderungen über ein Change-Request-Verfahren wird auf diese Weise technisch sichergestellt.

Ein integriertes Auditing bietet die Möglichkeit, Sicherheitsverletzungen a posteriori nachzuweisen sowie Attacken zeitnah zu erkennen.

Als Fazit lässt sich festhalten, dass PostgreSQL in Sachen Sicherheit die Anforderungen für den Einsatz in einem betrieblichen Umfeld erfüllt.

■ 12.8 Performance und Skalierbarkeit

Die Benchmarks haben ergeben, dass PostgreSQL in vielen Bereichen mit traditionellen Datenbanksystemen mithalten kann oder sie sogar übertrifft. PostgreSQL besticht durch eine schlanke Architektur. Features wie zum Beispiel die Partitionierung von Tabellen ermöglichen seinen Einsatz auch für größere Datenbanken. Eine von Beginn an konsequent gestaltete Transaction Engine garantiert sowohl Transaktionssicherheit als auch Lesekonsistenz.

Durch den VACUUM-Mechanismus erfolgt permanent eine Defragmentierung von Tabellen und Indexen. Das ist nicht für alle Datenbanksysteme selbstverständlich. Dadurch ist die Skalierbarkeit auch bei Vergrößerungen der Datenbank weitgehend gewährleistet.

Performance-Probleme auf Grund nicht-optimierter SQL-Anweisungen können wie in allen Datenbanksystemen zum Problem werden. Hierfür bietet PostgreSQL sehr gute Unterstützung für Problemanalyse und -lösung.

Auch die Skalierbarkeit im Multi-User-Bereich ist sehr gut. Benchmarks haben gezeigt, dass das System mit wachsender Anzahl von konkurrierenden Benutzern sehr gut skaliert und die Antwortzeiten nur langsam steigen.

Der PostgreSQL-Server bietet darüber hinaus eine breite Auswahl an Optimierungsmöglichkeiten, um die Datenbanken den Anforderungen der Applikation anzupassen. Diese reichen von der Vergrößerung des Shared Buffer Pool bis zur Anpassung der Checkpoints und des Transaktions-Log-Verhaltens.

■ 12.9 Schnittstellen und Kommunikation

Für den Bereich Schnittstellen lässt sich festhalten, dass PostgreSQL ein sehr offenes System ist und den Anforderungen an ein modernes Datenbank-Betriebssystem genügt. Die Möglichkeiten der Anbindung an andere Datenbank-Systeme lernen Sie in Kapitel 14 kennen. Über die ODBC-Schnittstelle ist es möglich, mit Datenbank-Links zu arbeiten sowie asynchrone Replikationsmechanismen anderer Systeme anzubinden. Dies betrifft insbesondere die Datenübertragung nach PostgreSQL. Für die Replikation nach außen ist die Unterstützung seitens PostgreSQL noch etwas dünn, und man ist auf Werkzeuge von Drittherstellern angewiesen.

Das Export-Utility ist eine weitgehend offene Schnittstelle, da es mit einem Textformat arbeitet. Die Export-Dateien sind somit „human readable“ und können gleichzeitig als Basis für eine Übertragung in andere Datenbank-Systeme benutzt werden. Aber auch die Übertragung von PostgreSQL nach PostgreSQL für unterschiedliche Betriebssysteme ist damit denkbar einfach.

Darüber hinaus stehen Standard-Schnittstellen wie ODBC und JDBC zur Verfügung. Der PostgreSQL-Server selbst stellt Module für die Verwendung der verbreiteten Programmier- und Skriptsprachen wie C/C++, Java, Perl, Python, Tcl. Wohl einzigartig unter Datenbanksystemen sind die Erweiterungsmöglichkeiten der SQL-Sprache. Die Anbindungsmöglichkeit von Skriptsprachen ist natürlich aus betrieblicher Sicht sehr wichtig.

Für eine sichere Kommunikation zwischen Datenbank-Server und Client sowie zwischen Datenbank-Servern bietet PostgreSQL den Anschluss von „OpenSSL“ an. Wie in vielen anderen Bereichen orientiert sich PostgreSQL richtigerweise an Open Source-Produkten. Der PostgreSQL-Server ist auf die SSL-Anbindung vorbereitet und die Konfiguration ist einfach vorzunehmen.

Im Bereich Schnittstellen und Kommunikation erfüllt PostgreSQL die Anforderungen an ein modernes Datenbank-Betriebssystem. Einer Eingliederung in den Betrieb steht insofern nichts im Wege.

■ 12.10 Support

Das Thema des Vendor-Supports ist ein wichtiger Punkt für die Entscheidung, ob ein Produkt seinen Weg auf die Roadmap eines Unternehmens findet oder nicht. Es wird bei Open-Source-Produkten stets etwas kritisch gesehen – ohne nachvollziehbaren Grund. Viele Unternehmen setzen zum Beispiel Linux erfolgreich ein.

Auch für PostgreSQL gibt es professionellen Support. Die bekannteste Firma ist „EnterpriseDB“. Weitere Informationen finden Sie auf der Webseite „enterprisedb.com“. Positiv zu verzeichnen ist die Nachricht, dass die Support-Kosten weit unter denen kommerzieller Datenbank-Vendoren liegen. Dieser Kostenfaktor ist einer der häufigsten Gründe, weshalb Unternehmen PostgreSQL einsetzen oder auf PostgreSQL umsteigen.

■ 12.11 Fazit

PostgreSQL erfüllt als Datenbank-Betriebssystem alle wichtigen Anforderungen für die Einbindung in den Betrieb. Es ist als Open-Source-Produkt transparent und ermöglicht erhebliche Einsparungen bei den Lizenz- und Supportkosten.

PostgreSQL ist ein stabiles und modernes Datenbanksystem, das nur wenige Wünsche offen lässt. Seine schlanke Architektur und der Fokus auf die Kernfunktionalität machen das System zum Konkurrenten alt eingesessener kommerzieller Datenbanksysteme. Ein Einsatz im produktiven Betriebsumfeld kann bedenkenlos empfohlen werden.

13

Migration von MySQL nach PostgreSQL

In den letzten Jahren war ein vermehrter Anstieg der Migrationen von MySQL nach PostgreSQL zu verzeichnen. Der Grund dafür liegt hauptsächlich in den erweiterten technischen Möglichkeiten und Features von PostgreSQL. Dazu gehören die stabile Transaction Engine, partitionierte Tabellen, Stored Functions, aber auch die Features für den betrieblichen Einsatz. Die Fusion von Sun und Oracle führte dazu, dass über einen längeren Zeitraum Unklarheit herrschte, wie sich die Weiterentwicklung von MySQL gestalten würde.

Trotz einiger Unterschiede zwischen beiden Datenbanksystemen ist die Migration nicht problematisch. Es gibt eine Reihe von Toolkits und Skripten zur Migrationsunterstützung. Im folgenden Abschnitt stellen wir Ihnen die wichtigsten Unterschiede zwischen MySQL und PostgreSQL vor.

■ 13.1 Unterschiede zwischen MySQL und PostgreSQL

Am wichtigsten für eine Migration sind die Unterschiede bei den Datentypen. In Tabelle 13.1 finden Sie eine Richtlinie zur Konvertierung.

Tabelle 13.1 Datentypen MySQL und PostgreSQL

MySQL	PostgreSQL
TINYINT	SMALLINT
SMALLINT	SMALLINT
MEDIUMINT	INTEGER
BIGINT	BIGINT
FLOAT	REAL
FLOAT UNSIGNED	REAL
DOUBLE	DOUBLE PRECISION
BOOLEAN	BOOLEAN

Tabelle 13.1 Datentypen MySQL und PostgreSQL (Fortsetzung)

MySQL	PostgreSQL
TINYTEXT	TEXT
TEXT	TEXT
MEDIUMTEXT	TEXT
LONGTEXT	TEXT
BINARY	BYTEA
BLOB	BYTEA
MEDIUMLOB	BYTEA
LONGLOB	BYTEA
DATE	DATE
TIME	TIME
DATETIME	TIMESTAMP
TIMESTAMP	TIMESTAMP

In Tabelle 13.2 finden Sie außerdem die wichtigsten Syntax-Unterschiede.

Tabelle 13.2 Syntax-Unterschiede MySQL und PostgreSQL

Syntax	MySQL	PostgreSQL
Kommentar	#	—
Begrenzer für Zeichenketten	' oder „	Ausschließlich '
Case-Sensitivität	Nein	Ja
Operatoren	bedeutet „ODER“	String-Konkatinaton

In MySQL werden NULL-Werte standardmäßig am Anfang, in PostgreSQL am Ende angezeigt. Mit der Klausel „NULLS FIRST“ ist die Reihenfolge wieder identisch mit MySQL.

Listing 13.1 Die Sortierreihenfolge für NULL-Werte

```
(postgres@[local]:5432) [postgres] > SELECT * FROM kb3
> ORDER BY text;
 id | text
----+-----
  1 | A
  2 | B
  3 |
(postgres@[local]:5432) [postgres] > SELECT * FROM kb3
> ORDER BY text NULLS FIRST;
 id | text
----+-----
  3 |
  1 | A
  2 | B
```

Für eine MySQL-Datenbank bestimmt das Betriebssystem, ob Identifier (Tabellennamen, Spaltennamen) groß oder klein geschrieben werden. Während unter Windows die Schreibweise keine Rolle spielt, muss sie unter Unix/Linux exakt erfolgen. Der Zusammenhang ergibt sich dadurch, dass Tabellen als Unterverzeichnisse abgelegt werden.

In einer PostgreSQL-Datenbank erfolgt die Speicherung unabhängig vom Betriebssystem. Dabei werden Identifier generell in Kleinbuchstaben hinterlegt. Möchte man die Groß- und Kleinschreibung behalten, können Identifier in doppelte Hochkommas eingeschlossen werden.

Listing 13.2 Speicherung der Identifier in PostgreSQL

```
(postgres@[local]:5432) [postgres] > CREATE TABLE c_test(
> ID      INT,
> "Text"  varchar(100));
CREATE TABLE
(postgres@[local]:5432) [postgres] > \d c_test
          Table "public.c_test"
  Column |          Type          | Modifiers
-----+-----+-----
  id     | integer                |
  Text   | character varying(100) |
```

In MySQL wird häufig der Datentyp SMALLINT(1) für „true“ oder „false“ verwendet. Das Problem ist, dass nicht nur die Werte „0“ oder „1“ angenommen werden können, sondern alle einstelligen Zahlen.

■ 13.2 Eine Migration durchführen

Die einfachste Methode ist die Verwendung eines MySQL-Dumps. Der Dump ist eine Textdatei. Diese Datei lässt sich mit Hilfe eines Skripts in eine SQL-Datei umwandeln, die PostgreSQL als Importdatei akzeptiert. Es gibt fertige Skripte (vorwiegend in Perl oder Python geschrieben), die eine Umwandlung des Dumps auf Basis regulärer Ausdrücke vornehmen. Das Skript übernimmt somit die Überarbeitung der wichtigsten bekannten Unterschiede, sodass am Ende nur noch wenige manuelle Nacharbeiten erforderlich sind. Eine komplette automatische Umstellung ist nur selten möglich, da erfahrungsgemäß immer gewisse Besonderheiten berücksichtigt werden müssen. Auch beim Austausch der Datentypen funktionieren die Skripte nach einem einfachen Mechanismus. Daher ist eine manuelle Nachkontrolle der Datentypen zu empfehlen.

Die Firma „EnterpriseDB“ bietet einen Migration Wizzard im Rahmen des Migration Studios an. Voraussetzung ist allerdings, dass ein PostgreSQL-Server in der Ausprägung „EnterpriseDB Advanced Server“ eingesetzt wird. Der Migration Wizzard bietet drei Funktionen an: Migration einer gesamten Datenbank, eines einzelnen Schemas oder ausschließlich die Übernahme von Daten.

Listing 13.3 Einen Dump für eine MySQL-Datenbank erstellen

```
C:\Program Files\MySQL\MySQL Server 5.5\bin>mysqldump -u root -p
classicmodels > migration.sql
Enter password: *****
```


Den Kopf der Dump-Datei finden Sie in Listing 13.4. Darin befinden sich die SQL-Anweisungen zum Neuaufbau der Datenbank im MySQL-Format. Enthalten sind sowohl DDL-Anweisungen zum Anlegen der Tabellen als auch INSERT-Anweisungen zum Laden der Daten. Für kleinere Schemas ist es sinnvoll, die Daten direkt mit zu übernehmen.

Listing 13.4 Kopf der Dump-Datei

```
-- MySQL dump 10.13  Distrib 5.5.22, for Win64 (x86)
--
-- Host: localhost    Database: classicmodels
-- -----
-- Server version    5.5.22
--
-- Table structure for table `customers`
--
DROP TABLE IF EXISTS `customers`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `customers` (
  `customerNumber` int(11) NOT NULL,
  `customerName` varchar(50) NOT NULL,
  `contactLastName` varchar(50) NOT NULL,
  `contactFirstName` varchar(50) NOT NULL,
  `phone` varchar(50) NOT NULL,
  `addressLine1` varchar(50) NOT NULL,
  `addressLine2` varchar(50) DEFAULT NULL,
  `city` varchar(50) NOT NULL,
  `state` varchar(50) DEFAULT NULL,
  `postalCode` varchar(15) DEFAULT NULL,
  `country` varchar(50) NOT NULL,
  `salesRepEmployeeNumber` int(11) DEFAULT NULL,
  `creditLimit` double DEFAULT NULL,
  PRIMARY KEY (`customerNumber`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;
--
-- Dumping data for table `customers`
--
LOCK TABLES `customers` WRITE;
/*!40000 ALTER TABLE `customers` DISABLE KEYS */;
INSERT INTO `customers` VALUES (103,'Atelier graphique','Schmitt','Carine',
', '40.32.2555','54, rue Royale',NULL,'Nantes',NULL,'44000','France',
1370,21000),
. . .
/*!40000 ALTER TABLE `customers` ENABLE KEYS */;
UNLOCK TABLES;
. . .
```

Nach Anwendung des Perl-Programms auf die Dump-Datei erfolgt die Konvertierung in ein SQL-Format, das von PostgreSQL verstanden wird. Den Kopf der konvertierten Datei finden Sie in Listing 13.5. Die Hochkommas für die Identifier (Tabellen- und Spaltennamen) wurden durch ein doppeltes Hochkomma ersetzt. In PostgreSQL bedeutet dies, dass die Groß- und Kleinschreibung berücksichtigt wird. Im vorliegenden Beispiel werden die Identifier in Kleinbuchstaben gespeichert. Dies kommt der Applikation entgegen und unterstützt den PostgreSQL-Standard. In den INSERT-Anweisungen wurde bei den Strings ein Escape-Zeichen „E“ vorangestellt. Damit können Strings mit Escape-Zeichen „\“ ohne Fehler ge-

laden werden. Wie Sie außerdem erkennen können, wurden die Datentypen wie in Listing 13.5 beschrieben angepasst. Aus „int(11)“ wurde „int“, aus „double“ wurde „double precision“.

Listing 13.5 Der MySQL-Dump nach der Konvertierung

```
DROP TABLE "customers" CASCADE\g
CREATE TABLE "customers" (
  "customernumber" int NOT NULL,
  "customername" varchar(50) NOT NULL,
  "contactlastname" varchar(50) NOT NULL,
  "contactfirstname" varchar(50) NOT NULL,
  "phone" varchar(50) NOT NULL,
  "addressline1" varchar(50) NOT NULL,
  "addressline2" varchar(50) DEFAULT NULL,
  "city" varchar(50) NOT NULL,
  "state" varchar(50) DEFAULT NULL,
  "postalcode" varchar(15) DEFAULT NULL,
  "country" varchar(50) NOT NULL,
  "salesrepemployeenumber" int DEFAULT NULL,
  "creditlimit" double precision DEFAULT NULL,
  primary key ("customernumber")
) ;
INSERT INTO "customers" VALUES (103,E'Atelier graphique',E'Schmitt',E'Carine
',E'40.32.2555',E'54, rue Royale',NULL,E'Nantes',NULL,E'44000',E'France',
1370,21000);
. . .
```

Für das vorliegende Beispiel konnte die Dump-Datei ohne Probleme geladen werden.

Listing 13.6 Laden des konvertierten Dumps in PostgreSQL

```
(postgres@[local]:5432) [postgres] > DROP TABLE "customers" CASCADE\g
ERROR: table "customers" does not exist
(postgres@[local]:5432) [postgres] > CREATE TABLE "customers" (
>   "customernumber" int NOT NULL,
>   "customername" varchar(50) NOT NULL,
>   "contactlastname" varchar(50) NOT NULL,
>   "contactfirstname" varchar(50) NOT NULL,
>   "phone" varchar(50) NOT NULL,
>   "addressline1" varchar(50) NOT NULL,
>   "addressline2" varchar(50) DEFAULT NULL,
>   "city" varchar(50) NOT NULL,
>   "state" varchar(50) DEFAULT NULL,
>   "postalcode" varchar(15) DEFAULT NULL,
>   "country" varchar(50) NOT NULL,
>   "salesrepemployeenumber" int DEFAULT NULL,
>   "creditlimit" double precision DEFAULT NULL,
>   primary key ("customernumber")
> ) ;
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "customers_
pkey" for table "customers"
CREATE TABLE
(postgres@[local]:5432) [postgres] > INSERT INTO "customers" VALUES
(103,E'Atelier graphique',E'Schmitt',E'Carine ',E'40.32.2555',E'54, rue Royal
e',NULL,E'Nantes',NULL,E'44000',E'France',1370,21000);
INSERT 0 1
. . .
```


Auch wenn möglicherweise nicht alle Dumps nach der Konvertierung ohne Fehler geladen werden können, halten sich die auftretenden Fehler in Grenzen. Das Skript ist wiederholbar. Fehler können nachgebessert werden, bis die Ausführung des Skripts keine Fehler mehr liefert.

Für größere Schemata empfiehlt es sich aus Performance-Gründen, die Daten separat im CSV-Format zu laden. Verwenden Sie die Option „--no-data“ im Programm mysqldump. Damit werden die INSERT-Anweisungen ausgelassen.

Listing 13.7 Erstellen eines MySQL-Dumps ohne Daten

```
C:\Program Files\MySQL\MySQL Server 5.5\bin>mysqldump -u root -p --no-data
classicmodels > migration.sql
Enter password: *****
```

Nachdem die Dump-Datei konvertiert und in PostgreSQL geladen wurde, kann die Datenübernahme durchgeführt werden. Die CSV-Datei lässt sich in MySQL mithilfe einer einfachen SELECT-Anweisung erstellen.

Listing 13.8 Eine CSV-Datei in MySQL erstellen

```
mysql> SELECT * INTO OUTFILE 'employees.csv'
-> FIELDS TERMINATED BY '|'
-> OPTIONALLY ENCLOSED BY ''
-> ESCAPED BY '\\'
-> LINES TERMINATED BY '\n'
-> FROM employees;
Query OK, 23 rows affected (0.01 sec)
```

Die so entstandene Datei hat das folgende Format:

```
1002|'Murphy'|'Diane'|'x5800'|'dmurphy@classicmodelcars.
com'|'1'|\N|'President'
1056|'Patterson'|'Mary'|'x4611'|'mpatterso@classicmodelcars.com'|'1'|1002|'VP
Sales'
1076|'Firrelli'|'Jeff'|'x9273'|'jfirrelli@classicmodelcars.com'|'1'|1002|'VP
Marketing'
. . .
```

Als Spaltenbegrenzer wurde das Zeichen '|' gewählt. Strings sind wie vorgegeben in Hochkommata eingeschlossen. Null-Werte werden durch „\N“ repräsentiert.

Listing 13.9 Laden der CSV-Datei mit dem COPY-Befehl

```
(postgres@[local]:5432) [postgres] > COPY employees
> FROM '/home/postgres/employees.csv'
> WITH DELIMITER '|';
COPY 23
(postgres@[local]:5432) [postgres] > SELECT * FROM employees;
 employeeenum | lastname | firstname | extension | email
| officecode | reportsto | jobtitle
-----+-----+-----+-----+-----
1002 | Murphy | Diane | x5800 | dmurphy@classic-
modelcars.com | '1' | President
1056 | Patterson | Mary | x4611 | mpatterso@classic-
modelcars.com | '1' | 1002 | VP Sales
. . .
```

**Praxistipp**

Obwohl das Laden mit dem COPY-Befehl bereits eine sehr gute Performance aufweist, gibt es Möglichkeiten, sie weiter zu steigern. Detaillierte Informationen finden Sie im „pgFoundry“ unter den Projekten „pgloader“ und „pg_bulkload“.

14

Von Oracle auf PostgreSQL umsteigen

Wenn Sie sich bereits ein wenig mit PostgreSQL beschäftigt haben, dürfte Ihnen die Nähe zum Oracle RDBMS nicht entgangen sein. Das betrifft nicht nur die Features und das Verhalten des Datenbank-Betriebssystems, sondern auch die prozedurale Programmiersprache PLpg/SQL mit einem hohen Kompatibilitätsgrad zu Oracles PL/SQL.

Aus diesem Blickwinkel ist PostgreSQL ein guter Kandidat für einen Umstieg. Wenn man einen Wechsel in Erwägung zieht, spielen jedoch weitere Faktoren eine Rolle, und es stellt sich die Frage, wie es um die Einbindung in eine bestehende betriebliche IT-Infrastruktur steht. Auf diesen Punkt sind wir in Kapitel 12 detailliert eingegangen. Manch einen mag es überraschen: PostgreSQL erfüllt viele Kriterien für einen stabilen und performanten Datenbankbetrieb. Dazu haben nicht zuletzt die neuen Features der Version 9, wie zum Beispiel „Hot Standby“ oder „Streaming Replication“, beigetragen. Auch die Performance-Werte im Ergebnis des Benchmarks in Kapitel 6 überzeugen. Wenn Sie keinen kompletten Umstieg für alle Datenbanken planen, stellt ein Mischbetrieb kein Problem dar. Kapitel 14 erläutert die Möglichkeiten des Datenaustauschs zwischen PostgreSQL und Oracle. Für alle Fälle: Auch für das Open-Source-Produkt PostgreSQL können Sie professionellen Support zu einem attraktiven Preis einkaufen.

Andererseits sind Sie mit einem geplanten Umstieg nicht allein. In dem Bereich gibt es viel Erfahrung, die Sie einkaufen und nutzen können. Wenn Sie sich selbst Fähigkeiten in diesem Umfeld aneignen wollen, hilft Ihnen dieses Kapitel mit wertvollen Hinweisen und Erfahrungen.

■ 14.1 Den Umstieg planen

Zur Planung des Umstiegs gehören neben der technischen Durchführung die Bereitstellung der notwendigen Hard- und Software. Überlegen Sie, über welchen Zeitraum ein Parallelbetrieb sinnvoll und realisierbar ist. Ein umfangreicher Test der Applikationen ist unbedingt erforderlich. Im Ergebnis des Tests sollte sowohl die fachliche Richtigkeit als auch eine akzeptable Performance nachgewiesen werden. Planen Sie auch das Zeitfenster für die Downtime der Applikationen während der produktiven Umstellung.

Die Migration sollte mehrmals testweise durchgeführt werden, bevor die eigentliche Umstellung der produktiven Datenbank erfolgt. Idealerweise können Sie mit einer Testdatenbank vorher die Umstellung üben. Häufig verfügen Testdatenbanken nicht über denselben Datenbestand wie produktive Datenbanken. Eine Aussage zur Performance lässt sich in diesen Fällen nicht treffen. Planen Sie die folgenden Schritte für eine Umstellung:

1. Bereitstellung der Hard- und Softwarekomponenten für die PostgreSQL-Datenbanken.
2. Entwicklung der Programme und Skripte sowie Konfiguration der Tools für den technischen Migrationsvorgang.
3. Durchführung von Testmigrationen mit Hilfe der Test-Datenbanken.
4. Anpassung der Client-Applikationen.
5. Test der Applikationen auf den Testdatenbanken in der neuen Zielumgebung.
6. Durchführung mindestens einer Testmigration mit den produktiven Datenbanken.
7. Fachliche Abnahme der Applikationen auf der neuen Zielumgebung.
8. Last- und Performancetest auf der neuen produktiven Zielumgebung.
9. Durchführung der produktiven Migration und Go Live.

Die Schritte erscheinen recht aufwendig, sind jedoch erforderlich, um eine sichere Migration zu garantieren sowie eine fachlich korrekte und performante Zielumgebung bereitstellen zu können.



Praxistipp

Eine Beschneidung der oben genannten Maßnahmen kann zwar zu einer Verkürzung der Migrationszeit und Senkung der Kosten führen, erhöht jedoch das Risiko für einen stabilen Betrieb in der neuen Umgebung. Bedenken Sie auch, dass es häufig schwierig ist, auf die alte Plattform zurückzukehren, sobald produktive Daten in die neue Umgebung geflossen sind.

Um eine Abschätzung des Umstellungsaufwands zu liefern, sollte man sich die Objekte in der Oracle-Datenbank ansehen. Bei einem hohen Anteil von PL/SQL Code ist von einem erhöhten Aufwand auszugehen, da hier mehr manuelle Aspekte einfließen. Die Umstellung von Tabellen und Indexen ist in der Regel weniger aufwendig, wobei auch hier zu prüfen ist, inwieweit selbst definierte Datentypen vorhanden sind. Beachten Sie auch den folgenden Hinweis.



Wichtig!

Oracle-Datenbankapplikationen, die spezielle Oracle-Produkte und Features wie zum Beispiel „Oracle Text“, „Oracle Multimedia“ usw. verwenden, sind für eine Umstellung auf PostgreSQL nicht geeignet, da analoge Produkte dort nicht vorhanden sind.

■ 14.2 Unterschiede zwischen Oracle und PostgreSQL

Oracle-Spaltentypen findet man in PostgreSQL wieder, doch unter anderen Namen. Somit ist eine Umstellung der „CREATE TABLE“-Anweisungen nötig. Auch die Syntax für das Anlegen anderer Objekte oder Constraints unterscheidet sich, sodass eine Anpassung der extrahierten Skripte erforderlich ist.

Bei der Migration von Tabellen ist eine Anpassung der Datentypen erforderlich. In Tabelle 14.1 finden Sie einen Vergleich der wichtigsten Datentypen in Oracle und PostgreSQL.

14.2.1 Unterschiede bei Datentypen

Tabelle 14.1 Vergleich der wichtigsten Datentypen

Oracle	PostgreSQL	Bemerkung
NUMBER	NUMERIC	Konvertierung nach FLOAT8 kann erforderlich sein.
CHAR	CHAR	Oracle: Limit 2000 Bytes, PostgreSQL: Limit 8104 Bytes
NCHAR	CHAR	PostgreSQL besitzt keinen National Characterset.
VARCHAR, VARCHAR2, NVARCHAR, NVARCHAR2	VARCHAR	VARCHAR ist ein Relikt aus Oracle 7 und 8 mit einer maximalen Länge von 4000 Byte. VARCHAR2 hat ebenfalls eine maximale Länge von 4000. VARCHAR in PostgreSQL kann max. 8104 Bytes lang sein.
DATE	TIMESTAMP	Oracle DATE speichert Datum und Uhrzeit bis auf die Sekunde.
LONG	TEXT	LONG kann bis zu 2 GB groß werden.
CLOB, NCLOB	TEXT	CLOB kann bis zu 4 GB groß werden. Zusätzliche Portierung erforderlich.
BLOB, BFILE	BYTEA	BLOB kann bis zu 4 GB groß werden. Zusätzliche Portierung erforderlich.
XMLTYPE	XML	Spezielle Unterstützung mit PL/SQL-Paketen

Ein immer wiederkehrendes Problem bei der Konvertierung ist die Groß- und Kleinschreibung. Während bei Oracle ein Identifier standardmäßig in Großbuchstaben gespeichert wird (das liegt wohl in der weit zurückreichenden Historie), speichert PostgreSQL diese in Kleinbuchstaben. Dieses Problem wird uns auch beschäftigen, wenn wir in Kapitel 15 das Thema „Datenbanklinks“ erörtern. In Listing 14.1 finden Sie ein Beispiel, das die Unterschiede verdeutlicht. Dabei sind sowohl der Tabellename als auch der Spaltenname betroffen.

Listing 14.1 Unterschiede in Groß- und Kleinschreibung für Identifier

```
(postgres@[local]:5432) [postgres] > CREATE TABLE test(
> id INT);
CREATE TABLE
```



```
(postgres@[local]:5432) [postgres] > \d test
      Table "public.test"
  Column | Type   | Modifiers
  -----+-----+-----
   id    | integer|
(postgres@[local]:5432) [postgres] > \d "test"
      Table "public.test"
  Column | Type   | Modifiers
  -----+-----+-----
   id    | integer|
(postgres@[local]:5432) [postgres] > \d "TEST"
Did not find any relation named ""TEST"".
```

```
SQL> CREATE TABLE test(
  2 id NUMBER);
Table created.
SQL> DESC "TEST"
      Name                                     Null?    Type
      ----
      ID                                     NUMBER
SQL> DESC "test"
ERROR:
ORA-04043: object "test" does not exist
```



Praxistipp

Eine einfache Methode bei der Migration, um die Unterschiede in Groß- und Kleinschreibung in den Griff zu bekommen, besteht darin, anzunehmen, dass auf der Oracle-Seite alle Identifier und Großbuchstaben hinterlegt sind. Die meisten Entwickler machen sich keine Gedanken um die Schreibweise, sodass eine automatische Konvertierung in Großbuchstaben erfolgt. Wenn Sie die Möglichkeit haben, bei der Übernahme der Tabellendefinition die Großschreibung auszublenden, legt PostgreSQL die Identifier in Kleinbuchstaben an, was viele Vorteile für den Einsatz der Applikation unter PostgreSQL mit sich bringt. Das Anlegen der Tabelle in PostgreSQL sollte dann in der Schreibweise „CREATE TABLE TEST. . .“ und nicht in der Schreibweise „CREATE TABLE „TEST“. . . „, erfolgen.

Zu beachten ist auch das unterschiedliche Verhalten eines leeren Strings. Betrachten wir eine Tabelle mit einem VARCHAR/VARCHAR2-Feld und fügen einen leeren String ein. Während PostgreSQL diesen bei anschließender Abfrage weiterhin als leeren String behandelt, ist unter Oracle daraus ein Null-Wert entstanden. Ein Beispiel finden Sie in Listing 14.2. Dieser Umstand wird häufig übersehen, da er bei der Migration selbst erst einmal nicht auffällt. Die Fehler entstehen jedoch beim Testen der Applikation und führen zu einer verfälschten Businessfunktionalität.

Listing 14.2 Unterschiedliche Behandlung von Leer-Strings

```
(postgres@[local]:5432) [postgres] > INSERT INTO test values(1,'');
INSERT 0 1
(postgres@[local]:5432) [postgres] > SELECT * FROM test WHERE text='';
```

```

id | text
----+-----
 1 |
(1 row)
(postgres@[local]:5432) [postgres] > SELECT * FROM test WHERE text IS NULL;
id | text
----+-----
(0 rows)

-----
SQL> INSERT INTO test VALUES(1,'');
1 row created.
SQL> SELECT * FROM test WHERE text='';
no rows selected
SQL> SELECT * FROM test WHERE text IS NULL;
   ID TEXT
-----
    1

```

Das Systemdatum „sysdate“ unter Oracle ist zu ersetzen durch „current_timestamp“ in PostgreSQL.

Die Funktion „DECODE“ ist bei Entwicklern sehr beliebt und wird in Oracle-Applikationen häufig verwendet. Da es unter PostgreSQL keine adäquate Funktion gibt, muss sie durch eine CASE-WHEN-Anweisung ersetzt werden:

```
CASE WHEN <ausdruck> THEN . . . ELSE <default> END
```

Auch bei Sequenzen gibt es Unterschiede in der Syntax. Hier empfiehlt sich ein Austausch der Syntax im Programmcode. Ersetzen Sie:

```

SELECT seq.nextval FROM dual;
      NEXTVAL
-----
       1

```

durch:

```

(postgres@[local]:5432) [postgres] > SELECT nextval('seq');
      nextval
-----
       1

```

Die Oracle-Datenbank hat über einen längeren Zeitraum an einer eigenen SQL-Syntax festgehalten und erst mit der Version 9i die ANSI-Syntax eingeführt. Die eigene Syntax ist aus Kompatibilitätsgründen immer noch gültig und kann in Applikationen auftauchen. Ein Beispiel hierfür ist der Outer Join, der unter Oracle mit einem „(+)-Symbol auf der rechten Seite gekennzeichnet wird. Weil PostgreSQL diese Syntax nicht kennt, empfiehlt sich eine Umwandlung in ANSI-Syntax. Ersetzen Sie:

```
SELECT * FROM tab1, tab2 WHERE tab1.x = tab2.y (+);
```

durch:

```
SELECT * FROM tab1 LEFT JOIN tab2 ON x = y;
```


Unter Oracle sind Optimizer Hints zur Optimierung von SQL-Anweisungen zugelassen. Diese sind in die Begrenzer „/*+“ und „*/“ eingeschlossen. Eine SQL-Anweisung sieht dann zum Beispiel so aus:

```
SELECT /*+ index(tab1,i1) */ * FROM test WHERE id = 4711;
```

Oracle interpretiert die Zeichen „/*“ als Beginn und „*/“ als Ende eines Kommentars. Diese Hints gibt es unter PostgreSQL nicht, sie führen zum Syntaxfehler und sollten deshalb entfernt werden.

Oracle gestattet eine Arithmetik mit Datumsfeldern. So kann, wie im folgenden Beispiel auf Variablen vom Typ „DATE“, direkt addiert oder subtrahiert werden:

```
SQL> SELECT sysdate + 1 FROM dual;
SYSDATE+1
-----
11.04.2012 11:40:07
```

Dies ist unter PostgreSQL nicht möglich. Eine solche Arithmetik führt zum direkten Fehler und muss deshalb ersetzt werden:

```
(postgres@[local]:5432) [postgres] > SELECT current_timestamp + 1;
ERROR:  operator does not exist: timestamp with time zone + integer
LINE 1: SELECT current_timestamp + 1;
                        ^
HINT:  No operator matches the given name and argument type(s). You might
need to add explicit type casts.
```

PostgreSQL stellt umfangreiche Formatierungsmöglichkeiten für Datum und Uhrzeit zur Verfügung. Eine einfache Lösung ist die folgende Transformation:

```
(postgres@[local]:5432) [postgres] > SELECT DATE(current_timestamp) +
INTEGER '1' AS morgen;
      morgen
-----
2012-04-11
```

Einige Oracle-Applikationen arbeiten mit der Option „ROWNUM“, um ein Stopp-Limit zu definieren. Diese Option muss unter PostgreSQL durch „LIMIT“ ersetzt werden. Ersetzen Sie:

```
SQL> SELECT * FROM test WHERE ROWNUM < 2;
```

durch:

```
SELECT * FROM test LIMIT 1;
```

In früheren Jahren wurde von Datenbank-Programmierern die unsichtbare Spalte ROWID in der Applikation verwendet. Das Argument war, dass der Zugriff über die ROWID sehr schnell ist. Oracle empfiehlt seit Längerem, diesen Programmierstil nicht mehr zu verwenden, da einerseits die Alternativen adäquat schnell sind und es andererseits bei Migrationen oder Upgrades immer wieder zu Problemen kommt. Auch für die Migration nach PostgreSQL gibt es keine Ersatzlösung.



Praxistipp

Für den Fall, dass die zu migrierende Applikation unter Oracle die Spalte „ROWID“ verwendet, gibt es keine adäquate Lösung auf der PostgreSQL-Seite. Da dieser Programmierstil auch nicht mehr empfohlen wird, sollte eine Anpassung auf der Oracle-Seite vor der Migration erfolgen und der Zugriff über „ROWID“ durch alternative Methoden ersetzt werden.

Weitere Anpassungen sind vorzunehmen, wenn die Applikation Abfragen auf das Oracle Dictionary, also den Datenbank-Katalog verwendet. Der PostgreSQL-Katalog unterscheidet sich vom Oracle-Katalog.

Damit haben wir bereits die wichtigsten Unterschiede zwischen Oracle und PostgreSQL identifiziert. Informationen zur Kompatibilität und zur Umwandlung von PL/SQL- und PL/pgSQL-Code finden Sie in Abschnitt 14.3.

Haben Sie mehr oder weniger Unterschiede erwartet? Die Unterschiede sind doch überschaubar, und man kann behaupten, dass es wohl kein Datenbanksystem gibt, das in der Kompatibilität so nahe an Oracle dran ist. Wie groß der Migrationsaufwand sein wird, ist stark von der Applikation abhängig und wie weit spezielle Oracle-Features benutzt wurden oder wie streng man versucht hat, sich an Standard-SQL zu halten.

Ein weiterer, sehr interessanter Ansatzpunkt besteht darin, die Kompatibilität der PostgreSQL-Datenbank generell zu erhöhen, um damit den Migrationsaufwand zu senken.

14.2.2 Steigerung der Kompatibilität von PostgreSQL

Die Überlegung besteht darin, die Kompatibilität von PostgreSQL-Datenbanken an sich zu erhöhen, indem Objekte und Funktionen, so wie sie standardmäßig in einer Oracle-Datenbank funktionieren, definiert werden. Damit wird der Migrationsaufwand von vornherein und ohne Kenntnis des zu migrierenden Schemas verringert. Als berühmtes Beispiel dient die Tabelle „DUAL“, die gerne von Oracle-Entwicklern verwendet wird und in PostgreSQL nicht existiert.

Listing 14.3 Ein View als Ersatz für die Tabelle DUAL anlegen

```
(postgres@[local]:5432) [hanser] > CREATE VIEW public.dual AS SELECT
'X'::varchar AS dummy;
CREATE VIEW
(postgres@[local]:5432) [hanser] > GRANT SELECT, REFERENCES ON public.dual
TO PUBLIC;
GRANT
```

Wenn Sie zusätzlich eine Funktion „SYSDATE“ anlegen, sind Sie schon sehr nahe an der Oracle-Syntax dran und sparen Migrationsschritte und -Probleme.

Listing 14.4 Eine Funktion „SYSDATE“ in PostgreSQL anlegen

```
(postgres@[local]:5432) [postgres] > CREATE FUNCTION sysdate()
> RETURNS DATE
> AS $$
> BEGIN
> RETURN DATE(current_timestamp);
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Es funktioniert nicht nur die berühmte und weit verbreitete Oracle-Anweisung „SELECT sysdate FROM dual“, sondern auch die Datumsarithmetik, die im vorhergehenden Abschnitt noch als Migrationsproblem beschrieben wurde.

Listing 14.5 Datums-Arithmetik mit der neuen Funktion „SYSDATE“

```
(postgres@[local]:5432) [postgres] > SELECT sysdate() + 1 FROM dual;
?column?
-----
2012-04-11
```

Wie Sie sehen, spart eine solche Erhöhung der Kompatibilität eine Menge von Problemen für den Migrationsaufwand. Die Firma „EnterpriseDB“ geht diesen Weg und bietet eine PostgreSQL-Datenbank an, die ein ganzes Stück Oracle-kompatibler ist. Das geht bis zum Anlegen eines Oracle-kompatiblen Datenbank-Katalogs, so dass zum Beispiel Abfragen auf das View „ALL_TABLES“ funktionieren und solche SQL-Anweisungen bei der Migration nicht berücksichtigt werden müssen. Weitere Details finden Sie auf der Webseite <http://enterprisedb.com>.

■ 14.3 Portierung von Oracle PL/SQL

Die Syntax von Oracle PL/SQL und PostgreSQL PL/pgSQL ist auf den ersten Blick fast identisch, allerdings gibt es auch hier eine Reihe von Besonderheiten. PostgreSQL kennt nur Funktionen, keine Prozeduren oder Pakete. Eine Prozedur in Oracle entspricht einer Funktion in PostgreSQL mit dem Rückgabewert „void“. Auch die Syntax zum Anlegen von Funktionen unterscheidet sich:

- Oracle: CREATE FUNCTION ... RETURN <ret_type> ...
- PostgreSQL: CREATE FUNCTION ... RETURNS <ret_type> ...

In PostgreSQL sollte der Funktionsrumpf in doppelte Dollarzeichen „\$\$“ eingefasst werden. Auch benötigt PostgreSQL explizit eine DECLARE-Anweisung im Rumpf. Oracle PL/SQL kann zwischen Namen und Variablen unterscheiden, PL/pgSQL kann das nicht und interpretiert alle Token als Variable.

Da PL/pgSQL keine Pakete kennt, müssen Schemas verwendet werden, um Funktionen zu gruppieren. In PostgreSQL können Sie keine globalen Variablen verwenden.

Variablen in FOR-Schleifen kann man in Oracle implizit verwenden, wohingegen sie in PostgreSQL definiert werden müssen:

- FOR var IN ... LOOP...: Die Variable „var“ muss definiert werden.

In PL/SQL können ROWTYP-Variablen am Cursor definiert werden. In PostgreSQL ist das nicht möglich, es muss eine Variable vom Typ „RECORD“ erklärt werden:

- Oracle:

```
CURSOR c1 IS SELECT . . . FROM . . . ;
var c1%ROWTYPE;
BEGIN
  FOR var IN c1 LOOP . . .
```

- PostgreSQL:

```
DECLARE
  CURSOR c1 IS SELECT . . . FROM . . . ;
  var RECORD;
BEGIN
  FOR var IN c1 LOOP . . .
```

Aufrufe von Prozeduren erfolgen in PostgreSQL mit dem PERFORM-Kommando:

```
PERFORM hr.fill_employees(1000);
```

DDL-Anweisungen in PostgreSQL können ein „EXECUTE“ benötigen:

```
EXECUTE 'DROP TABLE ' || table_name || ' . . .
```

In PostgreSQL wird die Transaktion im Falle einer Exception automatisch zurückgerollt. Beachten Sie auch, dass sich die Fehlercodes unterscheiden.

Betrachten Sie die einfache PL/SQL-Funktion in Listing 14.6. Um diese nach PostgreSQL zu portieren, sind wenige Anpassungen erforderlich. Die PL/pgSQL-Funktion finden Sie in Listing 14.7.

Listing 14.6 PL/SQL-Funktion in Oracle

```
CREATE OR REPLACE FUNCTION get_nv(v_str1 VARCHAR2, v_str2 VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
  IF v_str2 IS NULL THEN
    RETURN v_str1;
  END IF;
  RETURN v_str1 || ':' || v_str2;
END;
/
```

Listing 14.7 Die nach PL/pgSQL konvertierte Funktion aus Listing 14.6

```
CREATE OR REPLACE FUNCTION get_nv(v_str1 VARCHAR, v_str2 VARCHAR)
RETURNS VARCHAR AS $$
BEGIN
  IF v_str2 IS NULL THEN
    RETURN v_str1;
  END IF;
  RETURN v_str1 || ':' || v_str2;
END;
$$ LANGUAGE plpgsql;
```


Cursor-Variablen können in der Oracle-Syntax nicht verwendet werden. Wandeln Sie deshalb den folgenden Code:

```
CURSOR c1 IS SELECT . . .;
rec table%ROWTYPE;
BEGIN
  FOR rec IN c1 LOOP
    . . .
```

in den nachfolgenden Code für PostgreSQL um:

```
DECLARE
CURSOR c1 IS SELECT . . .;
rec RECORD;
BEGIN
  FOR rec IN c1 LOOP
    . . .
```

Während in Oracle PL/SQL eine Prozedur direkt aufgerufen werden kann, muss ein Funktionsaufruf innerhalb eines PL/pgSQL-Blocks mit „PERFORM“ erfolgen.

Die Variable „SQLERRM“ steht für die Fehlerbehandlung in PL/pgSQL nicht zur Verfügung.

Um eine Exception zu erzeugen, benötigt man eine andere Syntax. Eine Umwandlung von

```
RAISE <exception_name>;
```

in

```
RAISE EXCEPTION '<exception_name>;'
```

ist erforderlich.

Beachten Sie außerdem, dass die Exception „NO_DATA_FOUND“ unter PostgreSQL nur für SELECT-Anweisungen funktioniert. Verwenden Sie „IF NOT FOUND“ für INSERT, UPDATE und DELETE.

Fängt man in PL/pgSQL eine Exception in einer Exception-Klausel ein, werden alle Änderungen im Block mit einer BEGIN-Anweisung am Anfang automatisch zurückgerollt. Dies entspricht nicht dem Verhalten eines baugleichen PL/SQL-Blocks. In PL/SQL müsste der Block wie folgt aussehen:

```
BEGIN
SAVEPOINT p1;
. . .
EXCEPTION
  WHEN <exception> THEN
    ROLLBACK TO p1;
    . . .
END;
```



Hinweis

Beachten Sie, dass Pakete in PL/pgSQL nicht unterstützt werden. Paket-Definitionen und Aufrufe müssen deshalb in reguläre Funktionsaufrufe umgeändert werden.

■ 14.4 Tools zur Unterstützung der Migration

Wenn Sie im Internet nach Werkzeugen für eine automatische Migration suchen, werden Sie feststellen, dass es Insellösungen gibt, die an der einen oder anderen Stelle die Migration unterstützen. Allerdings gibt es kein Werkzeug, das Sie von „A bis Z“ bei der Migration begleitet. Dies wird umso deutlicher, wenn Sie sich das technische Vorgehen im folgenden Abschnitt vor Augen führen.

Das wohl bekannteste Tool ist „Ora2Pg“. Es besteht aus einem Perl-Skript und bietet folgende Funktionalität:

- Export eines Datenbank-Schemas der Oracle-Datenbank
- Entladen des PL/SQL-Quellcodes.
- Konvertierung der Schema-Definition in ein PostgreSQL-Format.
- Basisunterstützung für PL/SQL nach PL/pgSQL-Konvertierung. Manuelle Vervollständigung ist erforderlich.

Ein weiteres Tool ist „orafce“. Es ist Bestandteil des pgFoundry-Projekts und implementiert einige Funktionen zur Erhöhung der Kompatibilität zu Oracle, unterstützt also den Ansatz, den wir in Abschnitt 14.2.2 diskutiert haben. Neben Datumsfunktionen werden Module wie „DBMS_OUTPUT“ oder „UTL_FILE“ installiert.

■ 14.5 Technisches Vorgehen

Für das technische Vorgehen haben sich zwei Schritte als sinnvoll erwiesen. Der erste besteht in der Übernahme und Anpassung der Schema-Objekte, angefangen von Tabellen und Indexen über Privilegien bis hin zum Programmcode in PL/SQL. Im zweiten Schritt übernimmt man die Daten. Beide Schritte unterscheiden sich insofern, als der erste den größten Vorbereitungs- und manuellen Anpassungsaufwand erfordert und in der Ausführung recht schnell über die Bühne geht. Die Datenübernahme erfordert, vorausgesetzt, technische Hilfsmittel und Tools sind vorhanden, wenig manuellen Vorbereitungs- und Anpassungsaufwand, verschlingt jedoch den Großteil der Laufzeit in der Migration. Insbesondere bei großen Datenbanken gilt es, die zur Verfügung stehenden technischen Möglichkeiten auszureizen, damit die Migration in das vorgesehene Zeitfenster passt.

Im ersten Schritt ist eine Bestandsaufnahme des oder der zu migrierenden Schemas in der Oracle-Datenbank sinnvoll. Dazu gehört es, die Anzahl von Objekten zu ermitteln sowie die verwendeten Datentypen aufzulisten.

Es folgt das Extrahieren des DDL-Codes aus dem Oracle-Schema. Dafür kann man das Paket „DBMS_METADATA“ verwenden, aber auch viele Entwicklertools, wie zum Beispiel der kostenlose „SQL Developer“ von Oracle, weisen diese Funktionalität auf.

Der Code muss nun PostgreSQL angepasst werden, um dort die Objekte anlegen zu können. Für kleine Schemas mit wenigen Objekten kann die Anpassung manuell erfolgen. Wenn Sie größere Schemas migrieren, sollten Sie entsprechende Werkzeuge mit einer automatischen Anpassung verwenden.

Im nächsten Schritt erfolgt die Übernahme der Daten. Die Vorgehensweise hängt von der Größe des Schemas ab. Für kleinere Schemas sollte man sich wenig Gedanken machen. Man kann sich zum Beispiel von Tools INSERT-Anweisungen generieren lassen, was immer mit wenig Aufwand zum Ziel führt. Es gibt auch Tools, die eine Übernahme über JDBC- oder ODBC-Schnittstelle anbieten.

Für größere Schemas muss auf Performance sowohl beim Entladen als auch beim Laden geachtet werden. Hier empfiehlt sich das Entladen in CSV-Dateien sowie das Laden derselben in PostgreSQL.



Praxistipp

Erstellen Sie für große Schemas die Indexe und Primary Keys nach dem Laden der Daten. Das Laden mit angehängten Indexen führt zu einem signifikanten Performance-Verlust.

■ 14.6 Eine Beispiel-Migration

Die Beispiel-Migration zeigt auf, dass zwar einiges beim Umstieg von Oracle nach PostgreSQL zu beachten ist, andererseits viel automatisiert werden kann. Auch wenn die Migration des Beispielschemas nicht typisch für alle Datenbanken und Migrationen ist, werden Sie feststellen, dass der Umstieg einfacher ist, als man im Vorfeld vermuten würde. Aus praktischer Erfahrung lässt sich sagen, dass doch sehr viele Applikationen nur die Standard-Funktionalitäten der Oracle-Datenbank verwenden und damit für einen Umstieg sehr gut geeignet sind.

Für die Beispielmigration in diesem Abschnitt verwenden wir ein Beispielschema, das standardmäßig zusammen mit der Oracle-Datenbank ausgeliefert wird. Die Beispielschemas befinden sich auf der DVD Oracle 11gR2 und müssen, falls noch nicht geschehen, mit dem Universal Installer und dem DBCA nachinstalliert werden. Wir migrieren das Schema „HR“. Für einen ersten Überblick schauen wir uns die Objekttypen an, die dem Schema „HR“ angehören.

Listing 14.8 Objekttypen im Schema „HR“

```
SQL> select object_type,count(*)
2   from dba_objects
3  where owner = 'HR'
4  group by object_type;
```

OBJECT_TYPE	COUNT(*)
SEQUENCE	3
PROCEDURE	2
TRIGGER	2
TABLE	7
INDEX	19
VIEW	1

In Listing 14.9 finden Sie eine Funktion zum Erstellen der DDL-Befehle für die Tabellen im HR-Schema.

Listing 14.9 Funktion zum Extrahieren von DDL-Skripten aus Oracle

```
CREATE OR REPLACE FUNCTION get_table_md(p_table VARCHAR2) RETURN CLOB IS
h NUMBER;
th NUMBER;
doc CLOB;
BEGIN
DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,
'STORAGE',FALSE);
h := DBMS_METADATA.OPEN('TABLE');
DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
DBMS_METADATA.SET_FILTER(h,'NAME',p_table);
th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SEGMENT_ATTRIBUTES',false);
DBMS_METADATA.SET_TRANSFORM_PARAM(th,'SQLTERMINATOR',true);
doc := DBMS_METADATA.FETCH_CLOB(h);
DBMS_METADATA.CLOSE(h);
RETURN doc;
END;
/
```



Praxistipp

Beachten Sie bei der Verwendung von DBMS_METADATA, dass die DDL-Anweisungen sehr groß werden und die maximale Zeichenzahl einer VARCHAR2-Variablen von 4000 übersteigen können. In der Funktion in Listing 14.9 wurde deshalb bewusst der Datentyp „CLOB“ gewählt, der bis zu 4 GByte an Daten aufnehmen kann.

Wir benutzen die Funktion „get_table_md“ für den kompletten DDL-Code aller Tabellen im Schema „HR“. Im ersten Schritt werden die Befehle zum Aufruf der Funktion für alle Tabellen generiert.

Listing 14.10 Befehle zum Aufruf der Funktion „get_table_md“ generieren

```
SQL> SELECT 'SELECT get_table_md('''||table_name||'') FROM DUAL;'
2 FROM dba_tables WHERE owner = 'HR';
'SELECTGET_TABLE_MD(''||TABLE_NAME||'')FROMDUAL;'
-----
SELECT get_table_md('REGIONS') FROM DUAL;
SELECT get_table_md('LOCATIONS') FROM DUAL;
SELECT get_table_md('DEPARTMENTS') FROM DUAL;
SELECT get_table_md('JOBS') FROM DUAL;
SELECT get_table_md('EMPLOYEES') FROM DUAL;
SELECT get_table_md('JOB_HISTORY') FROM DUAL;
```

Die so gewonnenen Befehle werden nun zur Ausgabe der DDL-Befehle verwendet. Setzen Sie dazu in der Session den Parameter „LONG“ auf einen hohen Wert, damit der DDL-Text komplett aufgelistet wird.

Listing 14.11 Die DDL-Anweisungen für „Tabelle extrahieren“

```

SQL> SET LONG 10000
SQL> SELECT get_table_md('EMPLOYEES') FROM DUAL;
CREATE TABLE "HR"."EMPLOYEES"
(
  "EMPLOYEE_ID" NUMBER(6,0),
  "FIRST_NAME" VARCHAR2(20),
  "LAST_NAME" VARCHAR2(25) CONSTRAINT "EMP_LAST_NAME_NN" NOT NULL ENABLE,
  "EMAIL" VARCHAR2(25) CONSTRAINT "EMP_EMAIL_NN" NOT NULL ENABLE,
  "PHONE_NUMBER" VARCHAR2(20),
  "HIRE_DATE" DATE CONSTRAINT "EMP_HIRE_DATE_NN" NOT NULL ENABLE,
  "JOB_ID" VARCHAR2(10) CONSTRAINT "EMP_JOB_NN" NOT NULL ENABLE,
  "SALARY" NUMBER(8,2),
  "COMMISSION_PCT" NUMBER(2,2),
  "MANAGER_ID" NUMBER(6,0),
  "DEPARTMENT_ID" NUMBER(4,0),
  CONSTRAINT "EMP_SALARY_MIN" CHECK (salary > 0) ENABLE,
  CONSTRAINT "EMP_EMAIL_UK" UNIQUE ("EMAIL") ENABLE,
  CONSTRAINT "EMP_EMP_ID_PK" PRIMARY KEY ("EMPLOYEE_ID") ENABLE,
  CONSTRAINT "EMP_DEPT_FK" FOREIGN KEY ("DEPARTMENT_ID")
    REFERENCES "HR"."DEPARTMENTS" ("DEPARTMENT_ID") ENABLE,
  CONSTRAINT "EMP_JOB_FK" FOREIGN KEY ("JOB_ID")
    REFERENCES "HR"."JOBS" ("JOB_ID") ENABLE,
  CONSTRAINT "EMP_MANAGER_FK" FOREIGN KEY ("MANAGER_ID")
    REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ENABLE
) ;

```

DBMS_METADATA liefert die Identifier in Großbuchstaben und in doppelten Hochkommas eingeschlossen. Wie bereits erwähnt, wird PostgreSQL die Identifier in Großbuchstaben anlegen. Dies ist für das Handling und die Applikation eher ungünstig. Wir entfernen die Hochkommas mit einem einfachen sed-Skript:

```
$ sed 's/"//g' cr_tables.sql > cr_tables2.sql
```

Schließlich müssen Sie noch die Datentypen anpassen. Dies kann man ebenfalls weitgehend automatisieren. Beachten Sie, dass entweder die Tabellen in der richtigen Reihenfolge oder die Foreign Key Constraints komplett im Anschluss erstellt werden.

Listing 14.12 Die Tabelle „employees“ in PostgreSQL anlegen

```

(postgres@[local]:5432) [postgres] > CREATE TABLE EMPLOYEES
> (
  >   EMPLOYEE_ID INT,
  >   LAST_NAME VARCHAR(25) CONSTRAINT EMP_LAST_NAME_NN NOT NULL ,
  >   FIRST_NAME VARCHAR(20),
  >   LAST_NAME VARCHAR(25) CONSTRAINT EMP_LAST_NAME_NN NOT NULL ,
  >   EMAIL VARCHAR(25) CONSTRAINT EMP_EMAIL_NN NOT NULL ,
  >   JOB_ID VARCHAR(10) CONSTRAINT EMP_JOB_NN NOT NULL ,
  >   PHONE_NUMBER VARCHAR(20),
  >   HIRE_DATE DATE CONSTRAINT EMP_HIRE_DATE_NN NOT NULL ,
  >   JOB_ID VARCHAR(10) CONSTRAINT EMP_JOB_NN NOT NULL ,
  >   SALARY NUMERIC(8,2),
  >   COMMISSION_PCT NUMERIC(2,2),
  >   MANAGER_ID NUMERIC(6,0),
  >   DEPARTMENT_ID NUMERIC(4,0),
  >   CONSTRAINT EMP_SALARY_MIN CHECK (salary > 0) ,
  >   CONSTRAINT EMP_EMAIL_UK UNIQUE (EMAIL) ,
  >   CONSTRAINT EMP_EMP_ID_PK PRIMARY KEY (EMPLOYEE_ID)

```



```
'You may only make changes during normal office hours');
END IF;
END secure_dml;
```

Die PL/pgSQL-Funktion nach der Anpassung an PostgreSQL finden Sie in Listing 14.16.

Listing 14.16 PL/pgSQL-Funktion „secure_dml“ nach Anpassung an PostgreSQL

```
(postgres@[local]:5432) [postgres] > CREATE OR REPLACE
> FUNCTION secure_dml()
> RETURNS VOID
> AS $$
> BEGIN
>   IF TO_CHAR (SYSDATE, 'HH24:MI') NOT BETWEEN '08:00' AND '18:00'
>      OR TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN') THEN
> RAISE EXCEPTION 'You may only make changes during normal office hours';
>   END IF;
> END;
> $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Weil das Schema recht klein ist, kann die Datenübernahme einfach durch Erstellen von INSERT-Anweisungen erfolgen. Der SQL-Developer stellt wie viele andere Tools diese Funktionalität zur Verfügung. Beachten Sie, dass Umlaute korrekt übertragen werden.

Damit ist die Migration beendet. Jetzt kann die Applikation gestartet und die Datenbank getestet werden. Vom Applikationstest gemeldete Probleme lassen sich in der Regel problemlos beseitigen.

Die Erfahrung aus der Beispiel-Migration zeigt, dass der Aufwand überschaubar ist. Auch wenn es sich nur um ein relativ kleines Schema handelt, wird der Aufwand für größere Schemas im Rahmen bleiben. Der größte Aufwand ist für die Anpassung der PL/SQL-Programme zu sehen.

14.6.1 Übernahme der Daten

Während Oracle viele Tools zum Laden von Daten anbietet, werden relativ wenig für das Entladen zur Verfügung gestellt. Für kleine Datenmengen können Sie Datenbank-Browser, die sich über JDBC- oder ODBC-Schnittstelle verbinden, einsetzen.

Bei großen Tabellen ist es darüber hinaus wichtig, eine gute Performance beim Entladen zu erreichen. Eine Möglichkeit wäre, eine PL/SQL-Prozedur zu verwenden. PL/SQL hat allerdings den Nachteil, dass Schleifen relativ langsam sind. Es gibt eine Möglichkeit, ohne viel Aufwand eine Tabelle mit SQL*Plus zu entladen.

Listing 14.17 Eine Tabelle im CSV-Format entladen

```
SQL> SET colsep '|'
SET echo off
SET feedback off
SET linesize 1000
SET pagesize 0
SET sqlprompt ''
```

```

SET trimspool on
SET headsep off
SPOOL departments.csv
SELECT * FROM departments;
      10|Administration      |      200|      1700
      20|Marketing              |      201|      1800
      30|Purchasing              |      114|      1700
      40|Human Resources         |      203|      2400
      50|Shipping                |      121|      1500
      60|IT                      |      103|      1400
      70|Public Relations        |      204|      2700
      80|Sales                   |      145|      2500
      90|Executive               |      100|      1700
     100|Finance                 |      108|      1700
. . .

```

Der Import in PostgreSQL kann performant mit dem COPY-Befehl erfolgen.

Listing 14.18 CSV-Datei in PostgreSQL importieren

```

(postgres@[local]:5432) [postgres] > COPY departments
> FROM '/home/postgres/departments.csv'
> DELIMITER '|';
COPY 11
(postgres@[local]:5432) [postgres] > SELECT * FROM departments;
 department_id | department_name | manager_id | location_id
-----+-----+-----+-----
          10 | Administration |         200 |         1700
          20 | Marketing      |         201 |         1800
          30 | Purchasing     |         114 |         1700
          40 | Human Resources|         203 |         2400
          50 | Shipping       |         121 |         1500
. . .

```


Wenn Sie den Einsatz von PostgreSQL planen, stellt sich die Frage nach der Anbindung anderer Datenbanksysteme, die bereits in Betrieb sind. Ein solcher Mischbetrieb kann begrenzt für den Zeitraum der Migration erforderlich sein oder auch permanent geplant werden. Wie Sie bereits feststellen konnten, weist PostgreSQL einen hohen Kompatibilitätsgrad zu Oracle auf. Ein direkter Austausch von Daten zwischen Datenbanken ist ein wichtiges Element für eine stabile Betriebsumgebung sowie einen flexiblen Datenfluss, ohne den kein Unternehmen mehr auskommt.

Für einen Datenaustausch mit anderen Datenbanksystemen bietet Oracle das Heterogeneous Gateway an. Für einige Datenbanksysteme, wie zum Beispiel SQL Server oder Sybase, werden vorgefertigte Treiber mitgeliefert. Für alle anderen besteht die Möglichkeit, dieses Feature über eine ODBC-Schnittstelle zu nutzen. ODBC-Treiber für PostgreSQL stehen sowohl für Windows- als auch für Unix- und Linux-Betriebssysteme zur Verfügung.

In diesem Kapitel erläutern wir, wie man das Heterogeneous Gateway nutzt, um einen Datenbank-Link zwischen Oracle und PostgreSQL zu erstellen sowie eine Replikation auf Basis der Übertragung von Transaktions-Logdaten einzurichten. Diese Form der Replikation basiert auf Oracle Streams, das Sie vielleicht für eine Replikation zwischen Oracle-Datenbanken bereits einsetzen. Oracle Streams bietet die Möglichkeit, sowohl einzelne Tabellen oder Schemas zu replizieren, oder auch die gesamte Datenbank. Der offizielle Name einer solchen Architektur, die nicht-Oracle Datenbanken einbindet, ist „Oracle Streams Heterogeneous Information Sharing“. Als weitere Option stellen wir Oracle XStream für die Anbindung an PostgreSQL vor. XStream basiert auf Oracle Golden Gate und ist seit der Version Oracle 11.2 verfügbar.

■ 15.1 Datenbank-Link zwischen Oracle und PostgreSQL

Ein Datenbank-Link bietet die Möglichkeit des Datenaustauschs in beide Richtungen. Dabei können die Objekte der entfernten Datenbank behandelt werden, als wären sie Bestandteil der lokalen Datenbank. Das Ergebnis ist eine verteilte Transaktion, bei der die Performance

durch die Kapazität des Datenbank-Links, also des darunter liegenden Netzwerks, begrenzt wird.

Um einen Datenbank-Link zwischen Oracle und PostgreSQL erstellen zu können, muss im ersten Schritt das Gateway, also die Heterogeneous Services, eingerichtet werden. Für eine Reihe von Datenbanken gibt es vorgefertigte Module, allerdings nicht für PostgreSQL. Hier erfolgt die Verbindung über einen ODBC-Treiber. Wir verwenden die Versionen 11.2.0.1 für Oracle und 9.1.0 für PostgreSQL in einer Windows-Umgebung.

In Bild 15.1 ist die Architektur des Transparent Gateway abgebildet. Das Produkt wird auch als *Heterogeneous Service* (HS) bezeichnet. Der HS-Agent ist Bestandteil der Oracle-Datenbank (RDBMS) und nicht zusätzlich Lizenz-pflichtig. Diese Aussage gilt auch für die Standard-Edition. Der ODBC Driver stellt die Verbindung zum PostgreSQL-Server her.

Oracle-Datenbank

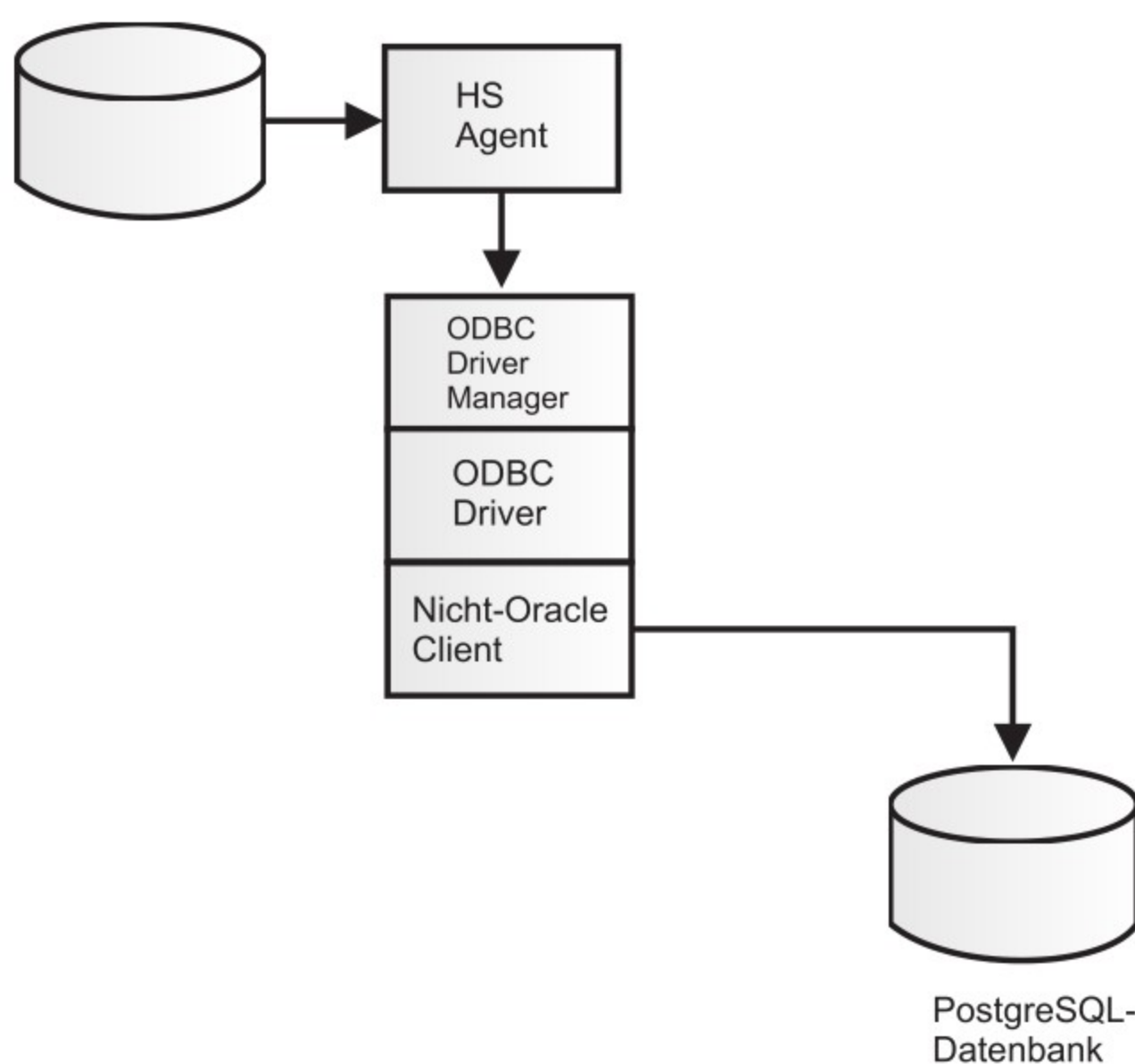


Bild 15.1

Die Architektur von Oracle Transparent Gateway

Die folgenden Schritte beschreiben den Aufbau des Gateway. Führen Sie diese auf dem Server aus, auf dem die Oracle-Datenbank läuft.

1. Laden Sie den ODBC Driver für PostgreSQL herunter. Sie finden den Driver auf der Webseite <http://www.postgresql.org/ftp/odbc/versions/dll/index.html>. Der Download enthält eine Windows DLL-Datei. Es gibt kein Installationsprogramm für den ODBC Driver. Die Installation kann jedoch manuell durchgeführt werden:
 - a) Führen Sie eine Sicherung der Windows Registry durch. Im Notfall können Sie in Folge von Schäden durch manuelle Veränderungen die Sicherung zurückspielen.
 - b) Legen Sie die Datei „pg.reg“ auf dem Windows-Server mit folgendem Inhalt ab:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\PostgreSQL]
"Setup"="C:\\WINDOWS\\SYSTEM\\PSQLODBC.DLL"
"Driver"="C:\\WINDOWS\\SYSTEM\\PSQLODBC.DLL"
```

```

"APILevel"="1"
"ConnectFunctions"="YYN"
"CommLog"="0"
"Optimizer"="0"
"DriverODBCVer"="02.00"
"FileUsage"="0"
"SQLLevel"="1"
"UsageCount"=dword:00000001
"Fetch"="100"
"UniqueIndex"="1"
"ReadOnly"="0"
"UseCursors"="0"
"UnknownSizes"="0"
"TextAsLongVarchar"="0"
"UnknownsAsLongVarchar"="0"
"MaxVarcharSize"="254"
"MaxLongVarcharSize"="4094"
"ConnSettings"=""
"UseDeclareFetch"="0"
"BoolsAsChar"="0"
"ExtraSysTablePrefixes"="dd_;"
"Parse"="1"
"Ksqo"="1"
"CancelAsFreeStmt"="0"

```

- c) Rufen Sie diese Datei unter Windows auf. Die Registry wird aktualisiert.
 - d) Kopieren Sie die Datei „psqlodbc.dll“ in das Verzeichnis „C:\windows\system32“.
2. Starten Sie den ODBC Manager, und wählen Sie das Register „System DSN“ aus. Klicken Sie auf „Hinzufügen“.

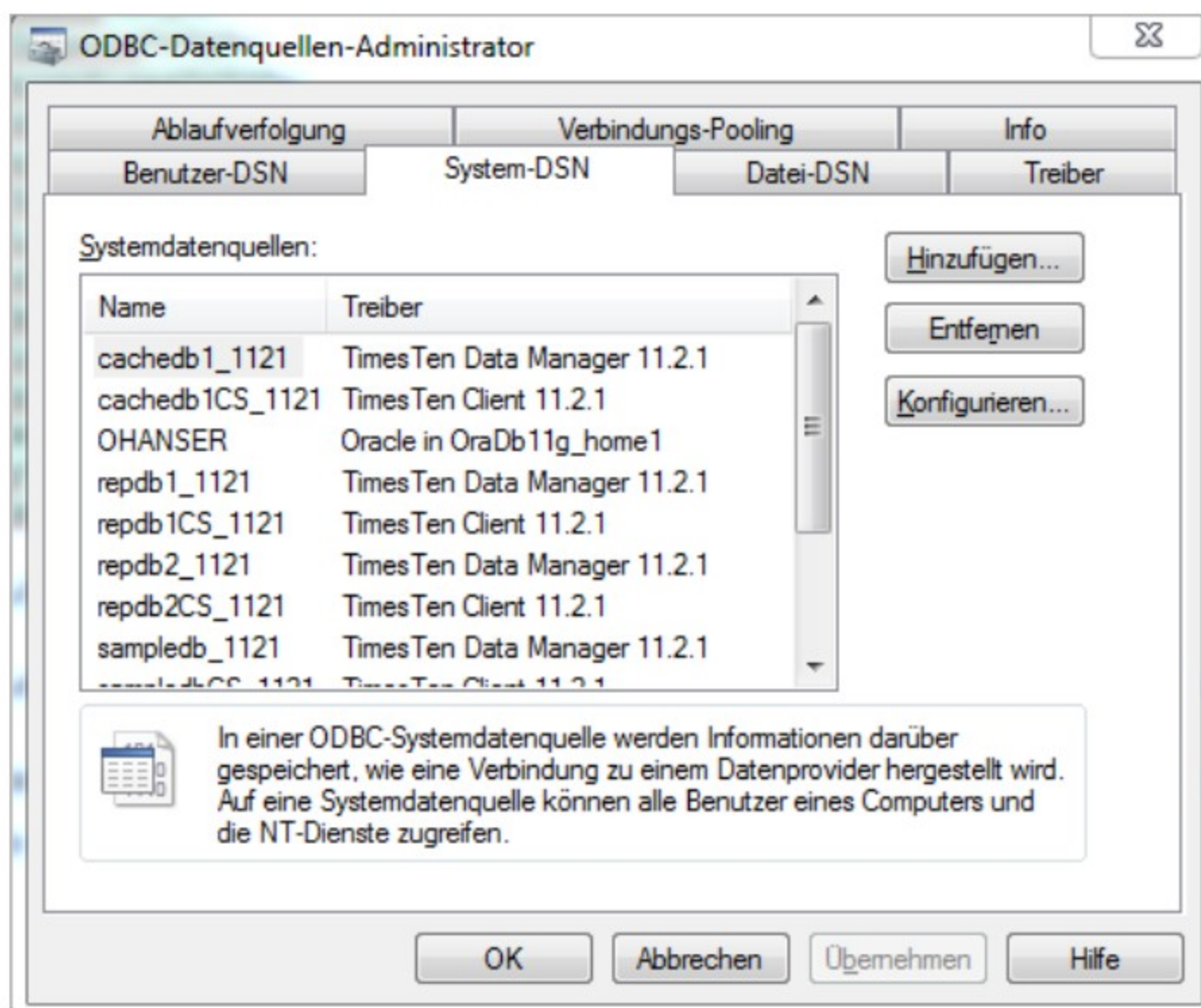


Bild 15.2 Den ODBC-Administrator starten

3. Wählen Sie als neue Datenquelle „PostgreSQL“ aus, und klicken Sie auf „Fertig stellen“.

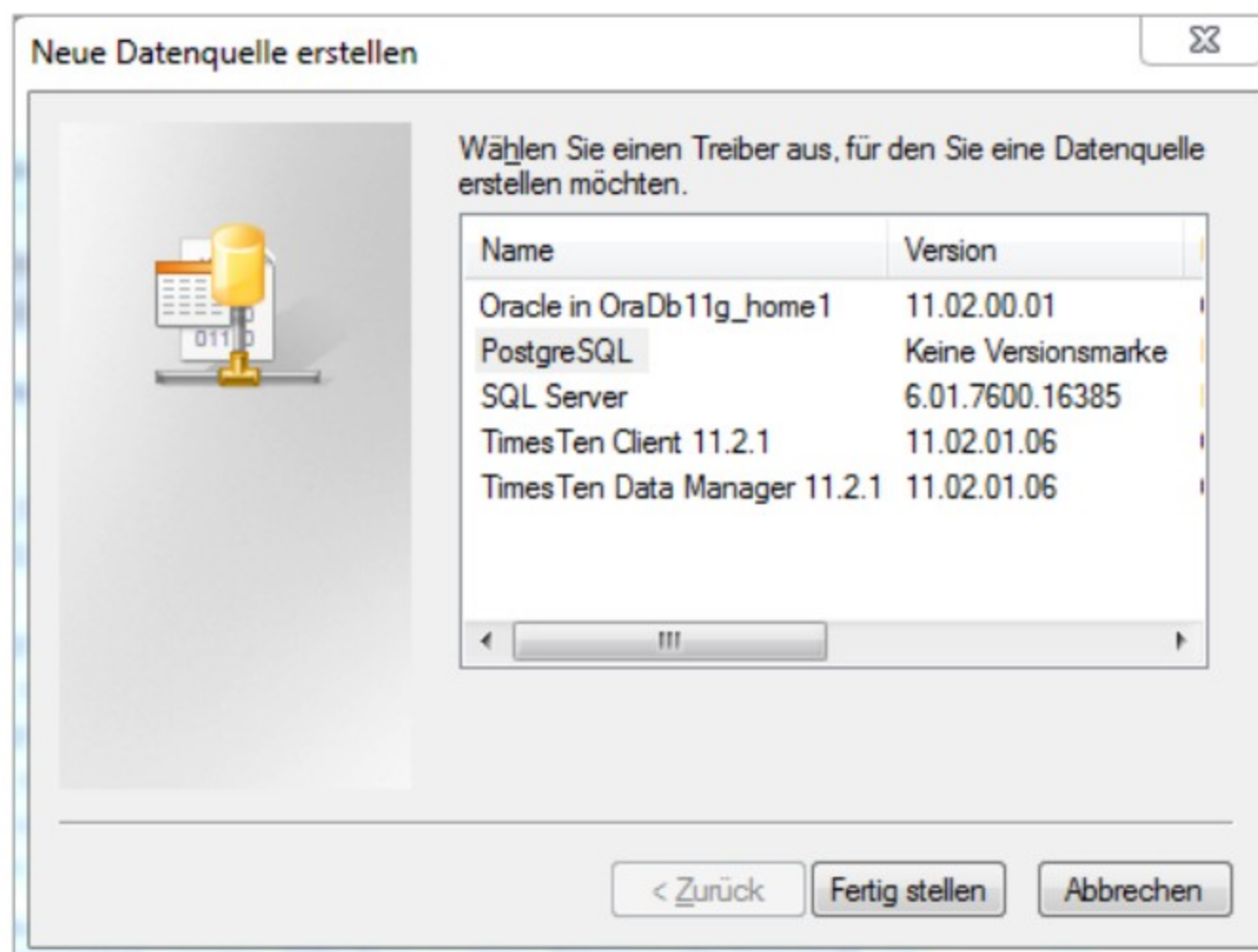


Bild 15.3 PostgreSQL als neue Datenquelle auswählen

4. Füllen Sie die Maske mit den Verbindungsdetails zur PostgreSQL Datenbank aus. Testen Sie die Verbindung zur PostgreSQL-Datenbank.

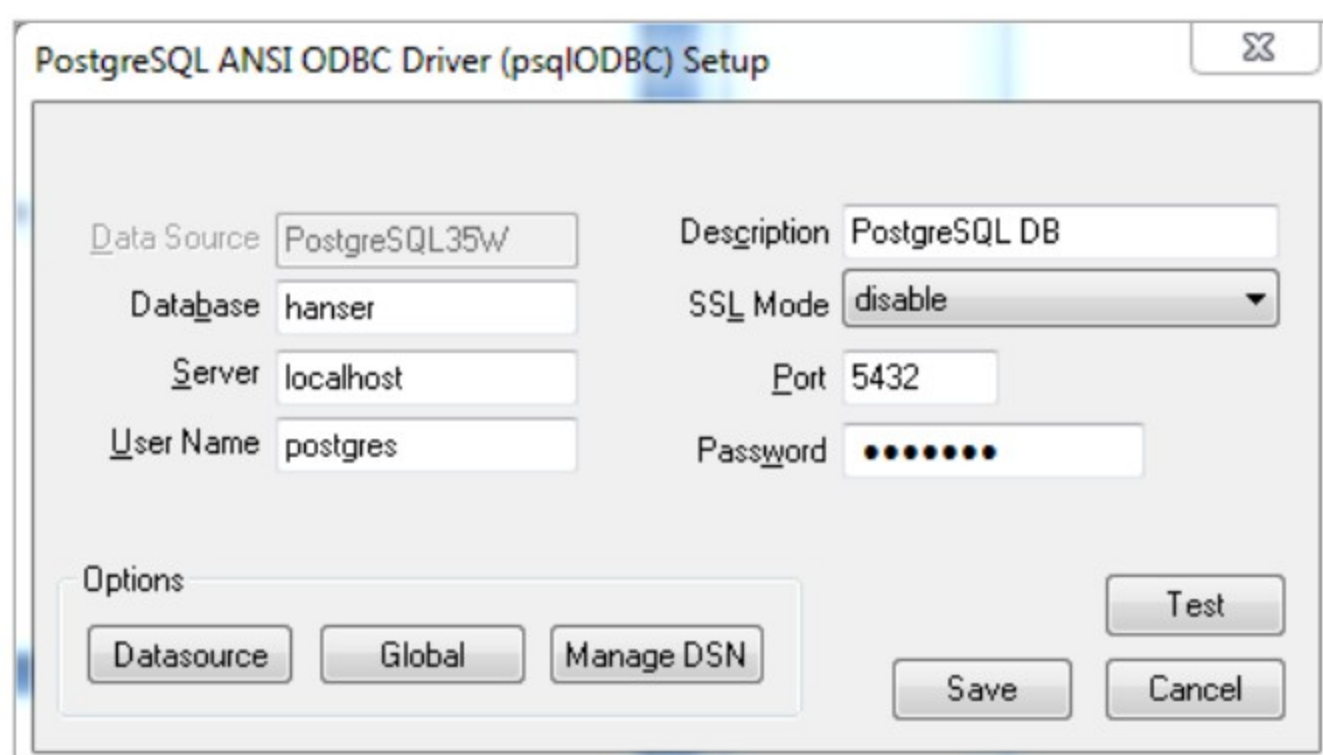


Bild 15.4 Die ODBC-Datenquelle für PostgreSQL konfigurieren

5. Damit ist der Driver installiert und die ODBC-Datenquelle konfiguriert. Im Weiteren erfolgt die Konfiguration des HS-Service der Oracle-Datenbank. Zuerst passen wir die Datei *initdg4odbc.ora* im Verzeichnis „%ORACLE_HOME%\hs\admin“ an. Tragen Sie den Namen der ODBC-Datenquelle im Parameter „HS_FDS_CONNECT_INFO“ ein. Außerdem besteht die Möglichkeit, Umgebungsvariablen für den PostgreSQL-Server mitzugeben.

```
HS_FDS_CONNECT_INFO = PostgreSQL35W          <- ODBC Data Source Name
HS_FDS_TRACE_LEVEL = ON
set PGUSER=postgres
set PGDATABASE=hanser
```

6. Schließlich sind noch Änderungen in den Dateien „tnsnames.ora“ und „listener.ora“ im Verzeichnis „%ORACLE_HOME%\network\admin“ vorzunehmen. Ergänzen Sie den folgenden Eintrag in der Datei „listener.ora“ unter „SID_LIST“:

```

SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = dg4odbc)
      (ORACLE_HOME = C:\app\Lutz\product\11.2.0\dbhome_1)
      (PROGRAM = dg4odbc)
    )
  )

```

7. Fügen Sie den folgenden Eintrag in die Datei „tnsnames.ora“ ein:

```

dg4odbc =
  (DESCRIPTION=
    (ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=1521))
    (CONNECT_DATA=(SID=dg4odbc))
    (HS=OK)
  )

```

Damit ist die Konfiguration des Gateway abgeschlossen. Starten Sie den Oracle Listener jetzt neu. Nun kann ein Datenbanklink von Oracle nach PostgreSQL angelegt werden (siehe Listing 15.1). Eine erste Abfrage aus der PostgreSQL-Datenbank ist erfolgreich.

Listing 15.1 Einen Datenbanklink von Oracle nach PostgreSQL anlegen

```

SQL> CREATE PUBLIC DATABASE LINK "POSTGRES.WORLD"
  2  CONNECT TO "postgres" IDENTIFIED BY 'postgres'
  3  USING 'dg4odbc';

```

Datenbank-Link wurde angelegt.

```

SQL> SELECT "datname","datdba","datlastsysoid"
  2  FROM "pg_database"@postgres.world;

```

datname	datdba	datlastsysoid
template1	10	11866
template0	10	11866
postgres	10	11866
hanser	10	11866

```

SQL> DESC "pg_database"@postgres.world

```

Name	Null?	Typ
datname	NOT NULL	VARCHAR2(64 CHAR)
datdba	NOT NULL	NUMBER(10)
encoding	NOT NULL	NUMBER(10)
datcollate	NOT NULL	VARCHAR2(64 CHAR)
datctype	NOT NULL	VARCHAR2(64 CHAR)
datistemplate	NOT NULL	CHAR(1)
dataallowconn	NOT NULL	CHAR(1)
datconnlimit	NOT NULL	NUMBER(10)
datlastsysoid	NOT NULL	NUMBER(10)
datfrozensid	NOT NULL	NUMBER(10)
dattablespace	NOT NULL	NUMBER(10)
datacl		VARCHAR2(255 CHAR)

Aus dem DESCRIBE-Befehl wird auch ersichtlich, dass eine Konvertierung der Datentypen in das Oracle-Format stattfindet. Bei allen Aktionen, die Sie über das Gateway durchführen, müssen Sie stets beachten, dass Sie mit einer PostgreSQL-Datenbank kommunizieren und es an der einen oder anderen Stelle Unterschiede zu Oracle gibt. Es können nicht alle denkbaren Operationen durchgeführt werden, vieles funktioniert jedoch sehr gut, insbesondere

wenn es um die direkte Abfrage oder den Austausch von Daten geht. Darüber hinaus sind natürlich Spracheinstellungen sowie Zeichensätze zu beachten.

Richtig benutzt, läuft das Gateway stabil. Die Performance ist dabei immer noch gut, obwohl die Kommunikation über den HS-Agent und eine ODBC-Schnittstelle läuft. Dabei bietet die Option, eine PostgreSQL-Datenbank direkt anzuschließen und diese wie eine Oracle-Datenbank zu behandeln, viele Vorteile.

In Listing 15.2 wird eine Tabelle mit 5 Millionen Sätzen von der PostgreSQL-Datenbank zur Oracle-Datenbank kopiert.

Listing 15.2 Tabellen über das Gateway kopieren

```
SQL> CREATE TABLE kb AS SELECT * FROM "kb"@postgres.world;
Tabelle wurde erstellt.
Abgelaufen: 00:01:50.68
```

Das Kopieren in die andere Richtung funktioniert in dieser Form nicht, da DDL-Befehle über diese Schnittstelle nicht zugelassen werden. Auch ein „INSERT INTO ... SELECT FROM“ führt zum Fehler ORA-02025. Abhilfe schafft eine PL/SQL-Prozedur, die Sie in Listing 15.3 finden.

Listing 15.3 Daten von Oracle nach PostgreSQL kopieren

```
SQL> DECLARE
  2  v1 kb."id"%TYPE;
  3  v2 kb."text"%TYPE;
  4  CURSOR c1 IS SELECT * FROM kb;
  5  BEGIN
  6  OPEN c1;
  7  LOOP
  8  FETCH c1 INTO v1, v2;
  9  EXIT WHEN c1%NOTFOUND;
 10  INSERT INTO "public"."kb"@postgres.world VALUES ( v1, v2);
 11  END LOOP;
 12  CLOSE c1;
 13  END;
 14  /
```

Das Wertvolle an einem Datenbank-Link ist die Möglichkeit, mit verteilten Transaktionen zu arbeiten. Die Abfrage in Listing 15.4 vereinigt die Tabelle „customers“ in der Oracle-Datenbank mit der Tabelle „orders“ in der PostgreSQL-Datenbank.

Listing 15.4 Tabellen aus Oracle und PostgreSQL in einer SELECT-Anweisung vereinigen

```
SQL> SELECT c."firstname",c."lastname",c."zip",o."orderid",o."totalamount"
  2  FROM customers c, "orders"@postgres.world o
  3  WHERE c."customerid" = o."customerid"
  4  AND "country" = 'US';
```

firstname	lastname	zip	orderid	totalamount
OPZPWP	PTFPIOWJDJ	21219	3326	69
LVPGXE	AMWSPEATTS	24228	5069	86
LVPGXE	AMWSPEATTS	24228	3857	318
OIQCQR	GBTDGNJHET	45719	11237	105

LNDXIB	PQJPDLSQOW	94144	146	291
ESFKNI	EDHEYQEAWJ	71853	7238	418
WSKLKJ	ZPQIRWKPNL	74402	7918	55

5914 Zeilen ausgewählt.
Abgelaufen: 00:00:03.29



Praxistipp

Das vorliegende Setup ist nicht auf Windows-Betriebssysteme beschränkt. Die gleiche Konfiguration lässt sich auch unter Unix/Linux auf der Oracle-Seite mit Hilfe von Plattform-spezifischen ODBC-Treibern herstellen. Auch ein Mischbetrieb zwischen Windows und Unix/Linux ist möglich.

Wie Sie feststellen mussten, ist eine Ausführung von DDL-Befehlen über den Datenbank-Link nicht möglich. Dabei handelt es sich um eine Beschränkung durch die ODBC-Schnittstelle. Das Heterogeneous Gateway hält ein Feature bereit, um SQL-Befehle durchzuleiten und direkt auf der PostgreSQL-Datenbank auszuführen. Dies erfolgt mit dem Paket *DBMS_HS_PASSTHROUGH*. Durch Auswertung des Return Codes kann man feststellen, ob die Ausführung erfolgreich war. In Listing 15.5 finden Sie ein Beispiel für das Anlegen einer Tabelle in der PostgreSQL Datenbank.

Listing 15.5 Beispiel für das Durchreichen von SQL-Anweisungen zur PostgreSQL-Datenbank

```
SQL> DECLARE
  2  ret NUMBER;
  3  BEGIN
  4  ret := DBMS_HS_PASSTHROUGH.EXECUTE_IMMEDIATE@postgres.world('create
table regions(region_id integer,region_name VARCHAR(50))');
  5  END;
  6  /
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

■ 15.2 Replikation mit Transaktionslog

Moderne Architekturen orientieren sich für die Replikation zwischen Datenbanken an der Aufnahme von Informationen aus dem Transaktionslog. Der Vorteil besteht unter anderem darin, dass keine zusätzliche Belastung der Datenbank stattfindet. Auf dieser Basis funktioniert auch Oracle Streams. Obwohl Streams primär für die Replikation zwischen Oracle-Datenbanken entwickelt wurde, funktioniert es auch für den Datenaustausch mit anderen Datenbanksystemen. Auch hier bedienen wir uns des Heterogeneous Service, so wie wir ihn im vorhergehenden Abschnitt für den Datenbank-Link eingerichtet haben. Oracle Streams ist ein integrierter Bestandteil der Enterprise und der Standard Edition und nicht extra lizenzpflichtig.

Oracle XStream bedient sich aus dem Produkt Oracle Golden Gate und steht seit der Version 11.2. zur Verfügung. Golden Gate stellt einerseits ein Parallelprodukt zu Streams dar, arbeitet jedoch auch mit Streams zusammen. Es ist das strategische Produkt für die Replikation, wohingegen Streams nur begrenzt weiterentwickelt wird. Allerdings ist Golden Gate und somit auch XStream zusätzlich lizenzpflichtig.

15.2.1 Replikation mit Oracle Streams Heterogeneous Information Sharing

Oracle Streams gibt es seit der Version 9i als Standardprodukt für die Replikation zwischen Datenbanken. Es kann für verschiedene Stufen konfiguriert werden: angefangen von einzelnen Datenbanken, Schemas bis hin zur gesamten Datenbank. Streams besteht aus den Komponenten *Capture*, *Propagation* und *Apply*. Der Capture-Prozess liest Informationen aus den Redo-Log-Daten und erstellt Logical Change Records (LCR). Der Propagation-Prozess überträgt die LCR an die Subscriber. Dazu bedient sich Oracle der Advanced Queuing Technologie. Der Apply-Prozess wandelt die LCR in SQL-Anweisungen um und wendet diese in der Zieldatenbank an.

Da es sich um ein Oracle-Feature handelt, kann der Apply-Prozess im Falle einer Replikation von Oracle nach PostgreSQL natürlich nur auf einer Oracle-Datenbank laufen. Um die Daten zu PostgreSQL-Datenbanken zu übertragen, setzt man eine zentrale Oracle-Datenbank auf, die als Gateway dient. Diese Datenbank führt das Apply auf allen angeschlossenen PostgreSQL-Datenbanken durch (siehe Bild 15.5).

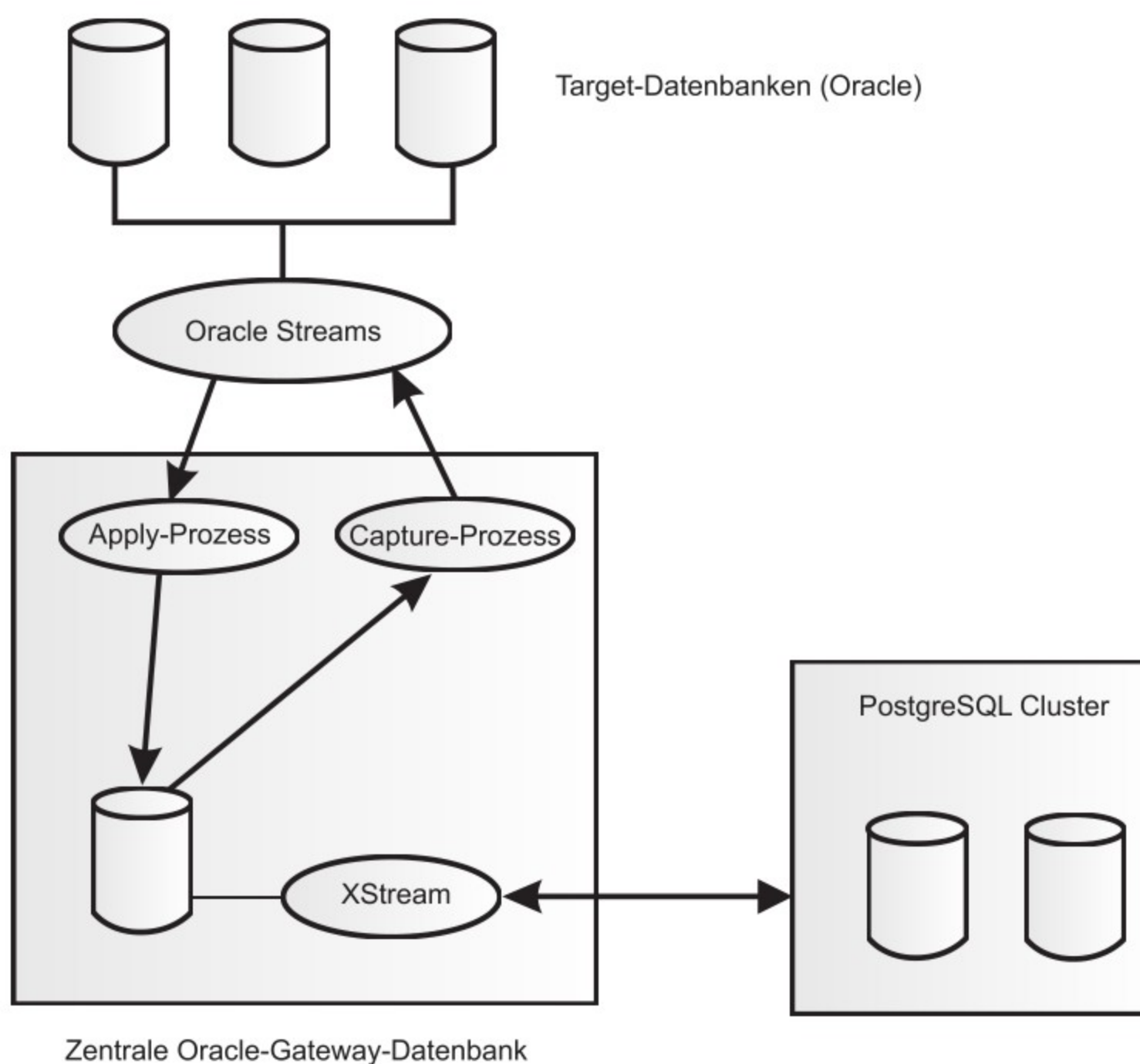


Bild 15.5 Oracle Streams-Konfiguration mit Apply Handler zum PostgreSQL-Cluster

Das folgende Beispiel beschreibt schrittweise das Aufsetzen einer Replikations-Umgebung mit Oracle Streams, einer Oracle-Quell-Datenbank, einer Oracle-Gateway-Datenbank sowie eines PostgreSQL Cluster als Zielsystem. Es besteht natürlich die Möglichkeit, mehrere Quell-Datenbanken und mehrere Ziel-Cluster zu konfigurieren. Wir führen die folgenden Konfigurationsschritte durch:

1. Konfiguration der Oracle-Quell- und Gateway-Datenbank mit Oracle Streams.
2. Konfiguration der Oracle-Gateway-Datenbank mit Oracle Streams und Heterogeneous Service.

15.2.1.1 Konfiguration der Quell-Datenbank mit Oracle Streams

Das Aufsetzen von Oracle-Streams von einer Target-Datenbank zur Oracle-Gateway-Datenbank erfolgt nach Standard. Im verwendeten Beispiel wird eine Tabelle repliziert, die Tabelle „employees“ aus dem Beispiel-Schema „hr“. Die Replikation funktioniert analog für mehrere Tabellen sowie auf Schema- oder Datenbank-Ebene.

Oracle Beispiel-Schemas

Die Oracle-Beispiel-Schemas finden Sie auf der Installations-DVD des entsprechenden Betriebssystems. Das Laden kann mit dem Database Configuration Assistant oder mit SQL*Plus von der Kommandozeile erfolgen. Die Skripte befinden sich im Verzeichnis „\$ORACLE_HOME/demo“.

Führen Sie die folgenden Schritte auf der Quell- und der Gateway-Datenbank aus.

1. Setzen Sie die Datenbank-Parameter laut Voraussetzung für den Einsatz von Oracle-Streams.

Tabelle 15.1 Datenbankparameter für Oracle Streams

Parameter	Wert
GLOBAL_NAMES	True
JOB_QUEUE_PROCESSES	>= 10, abhängig von der Anzahl der replizierten Tabellen
SGA_MAX_SIZE	Erhöhen Sie den Wert, wenn eine große Anzahl von Tabellen und ein hohes Volumen erwartet werden. Beim Einsatz von Automatischem Memory Management sollten die entsprechenden Parameter angepasst werden.
STREAMS_POOL_SIZE	Erhöhen Sie die Größe um 10 MB für jeden Capture-Prozess und um 1 MB für jeden Apply-Prozess. Der Wert des Parameters sollte mindestens 256 MB betragen.

2. Erstellen Sie einen Tablespace für die Objekte von Oracle Streams. Er dient als Default-Tablespace für den Streams-Administrator.

```
SQL> CREATE TABLESPACE streams DATAFILE
  'C:\APP\LUTZ\ORADATA\ORA112\STREAMS_01.DBF' SIZE 25M
  2 AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED;
Tablespace wurde angelegt.
```


3. Legen Sie den Streams-Administrator an, und weisen Sie ihm die erforderlichen Rechte zu.

```
SQL> CREATE USER strmadmin IDENTIFIED BY strmadmin;
Benutzer wurde erstellt.
SQL> ALTER USER strmadmin
  2  DEFAULT TABLESPACE streams
  3  TEMPORARY TABLESPACE TEMP
  4  QUOTA UNLIMITED ON streams;
Benutzer wurde geändert.
SQL> GRANT CONNECT,RESOURCE,AQ_ADMINISTRATOR_ROLE TO strmadmin;
Benutzerzugriff (Grant) wurde erteilt.
SQL> EXEC DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE('STRMADMIN');
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

4. Legen Sie jeweils einen Datenbank-Link für den User „strmadmin“ in beide Richtungen an.

```
SQL> CREATE DATABASE LINK ora112.world
  2  CONNECT TO strmadmin
  3  IDENTIFIED BY strmadmin
  4  USING 'ORA112.WORLD';
Datenbank-Link wurde angelegt.
SQL> CREATE DATABASE LINK target.world
  2  CONNECT TO strmadmin
  3  IDENTIFIED BY strmadmin
  4  USING 'TARGET.WORLD';
Datenbank-Link wurde angelegt.
```

5. Legen Sie jetzt die Streams Queue in beiden Datenbanken an. Dafür stellt Oracle die Prozedur „SETUP_QUEUE“ zur Verfügung.

```
SQL> BEGIN
  2  DBMS_STREAMS_ADM.SET_UP_QUEUE(
  3  queue_name => 'STREAMS_QUEUE',
  4  queue_table => 'STREAMS_QUEUE_TABLE',
  5  queue_user => 'STRMADMIN');
  6  END;
  7  /
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

6. Im nächsten Schritt werden die Capture- und Propagation-Regeln auf der Quell-Datenbank festgelegt. Die Tabelle „hr.employees“ wird für beide Prozesse definiert. Führen Sie die SQL-Befehle unter dem Benutzer „strmadmin“ aus.

```
SQL> BEGIN
  2  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
  3  table_name      => 'HR.EMPLOYEES',
  4  streams_type    => 'CAPTURE',
  5  streams_name    => 'STRMADMIN_CAPTURE',
  6  queue_name      => 'STRMADMIN.STREAMS_QUEUE',
  7  include_dml     => true,
  8  include_ddl     => true,
  9  source_database => 'TARGET.WORLD');
 10  END;
 11  /
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

```

SQL> BEGIN
  2  DBMS_STREAMS_ADM.ADD_TABLE_PROPAGATION_RULES(
  3  table_name => 'HR.EMPLOYEES',
  4  streams_name => 'STRMADMIN_PROP',
  5  source_queue_name => 'STRMADMIN.STREAMS_QUEUE',
  6  destination_queue_name => 'STRMADMIN.STREAMS_QUEUE@ORA112.WORLD',
  7  include_dml => true,
  8  include_ddl => true,
  9  source_database => 'TARGET.WORLD');
 10  END;
 11  /
PL/SQL-Prozedur erfolgreich abgeschlossen.

```

Soweit haben wir die Voraussetzungen für Streams auf beiden Oracle-Datenbanken geschaffen sowie Capture- und den Propagation-Prozess auf der Quell-Datenbank definiert. Jetzt konfigurieren wir die Gateway-Datenbank und vervollständigen die Streams-Konfiguration.

15.2.1.2 Konfiguration der Gateway-Datenbank

Konfigurieren Sie die Oracle Heterogeneous Services, wie in Abschnitt 14.1 beschrieben, und erstellen Sie einen Datenbank-Link.

Wir vervollständigen die Streams-Konfiguration auf der Gateway-Datenbank. Die Streams-Queue wurde bereits angelegt, sodass die replizierten Daten empfangen werden können. Offen ist noch das Einrichten des Apply-Prozesses. Dabei gibt es die Besonderheit, dass das Apply nicht auf die lokale Oracle-Datenbank, sondern auf das PostgreSQL-Cluster über das HS-Gateway erfolgen soll. Die Tabellen-Regeln unterscheiden sich zunächst nicht vom normalen Streams-Setup.

Listing 15.6 Tabellen-Regel auf der Gateway-Datenbank

```

SQL> BEGIN
  2  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
  3  table_name => 'HR.EMPLOYEES',
  4  streams_type => 'APPLY',
  5  streams_name => 'STRMADMIN_APPLY',
  6  queue_name => 'STRMADMIN.STREAMS_QUEUE',
  7  include_dml => true,
  8  include_ddl => true,
  9  source_database => 'TARGET.WORLD');
 10  END;
 11  /
PL/SQL-Prozedur erfolgreich abgeschlossen.

```

Jetzt gilt es, den Apply-Prozess zu definieren. Weil Oracle Streams den HS Service über ODBC nicht unterstützt, muss an dieser Stelle ein normaler Apply-Prozess für eine Oracle-Datenbank konfiguriert werden. Auf den Apply-Prozess kann ein Apply-Handler programmiert werden, der den LCR ausliest und eine SQL-Anweisung erzeugt. Die SQL-Anweisung lässt sich dann auf die PostgreSQL-Datenbank anwenden.

15.2.2 Replikation mit Oracle XStream

Oracle XStream ist ein neues Produkt und erstmalig mit der Version 11.2 erschienen. Es ist offiziell ein API für Clients, um Änderungen von Oracle-Datenbanken zu empfangen und an Oracle-Datenbanken zu senden. Bei genauerer Betrachtung entpuppt es sich auch als Schnittstelle zur Anbindung von Nicht-Oracle-Datenbanken und Nicht-Datenbank-Systemen. Allerdings ist – wie bei einem neuen Produkt nicht anders zu erwarten – der Einsatz von XStream mit Zusatzkosten verbunden.



Hinweis

Der Einsatz von Oracle XStream erfordert den Erwerb einer Golden Gate-Lizenz. Die Kosten lassen sich jedoch minimieren, wenn Sie mit einer zentralen Gateway-Datenbank, wie in Bild 15.6 dargestellt, arbeiten. Damit erfolgt der Datenaustausch über diese eine Datenbank, und nur für diese ist eine Lizenzierung erforderlich.

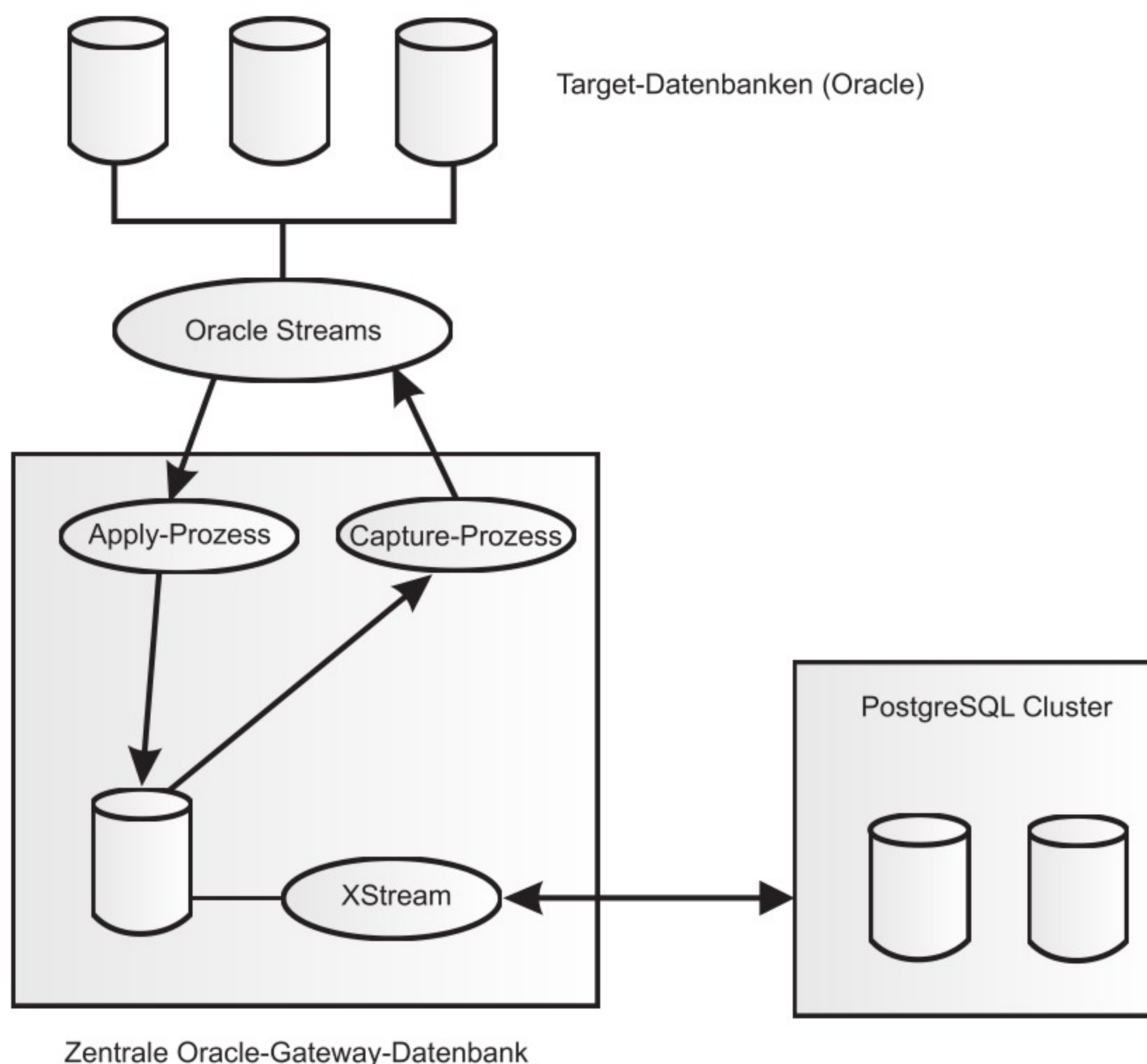


Bild 15.6 XStream-Architektur mit einer zentralen Gateway-Datenbank

Obwohl XStream lizenztechnisch mit Oracle Golden Gate verheiratet ist, ist eine technische Nähe zu Oracle Streams zu erkennen. Die Technologie basiert auf Logical Change Records

(LCR), wie sie von Oracle Streams bekannt ist. Ein Filtern ist auf Datenbank-, Schema-, Tabellen- und sogar auf Spalten- und Zeilenebene möglich. Ebenso können alle von Oracle Streams unterstützten Datentypen, einschließlich LOBs, LONG, LONG RAW und XMLType, auch von XStream verarbeitet werden.

XStream besteht aus den beiden Komponenten „XStream Out“ und „XStream In“. „XStream Out“ liest Änderungen aus den Redo-Log-Dateien der Datenbank und stellt sie in eine Warteschlange. Der XStream Outbound Server liest diese aus und stellt sie der Client-Applikation zur Verfügung. „XStream In“ ermöglicht Client-Applikationen, Informationen aus einem anderen System an die Oracle-Datenbank zu senden. Der Inbound Server ist ein zusätzlicher Hintergrund-Prozess der Oracle-Datenbank, der Logical Change Records empfängt und in die Oracle-Datenbank einarbeitet.

Die XStream API besteht aus einem OCI- und einem Java-Interface, die optional genutzt werden können. Über das Interface werden „XStream Out“ und „XStream In“ direkt angesprochen.

Im folgenden Abschnitt finden Sie ein komplettes Beispiel für eine Konfiguration mit Replikation von Änderungen in einer Oracle-Datenbank zu einem PostgreSQL-Server. Diese Konfiguration kann für eine Gateway-Datenbank, so wie in Bild 15.6 dargestellt, benutzt werden.

15.2.3 Replikation von Oracle zu PostgreSQL mit „XStream Out“

Zunächst ist es erforderlich, die Oracle-Datenbank für die Verwendung von XStream vorzubereiten. Führen Sie die folgenden Schritte aus:

1. Legen Sie einen Tablespace für die Objekte des XStream-Administrators an.

```
SQL> CREATE TABLESPACE xstream
  2  DATAFILE '/opt/oracle/oradata/HANSER/stream01.dbf'
  3  SIZE 25M AUTOEXTEND ON MAXSIZE UNLIMITED;
Tablespace created.
```

2. Erstellen Sie einen Datenbank-Benutzer, der die Rolle des XStream-Administrators übernehmen wird, und geben Sie ihm die notwendigen Privilegien.

```
SQL> CREATE USER xstrmadmin IDENTIFIED BY xstrmadmin
  2  DEFAULT TABLESPACE xstream
  3  QUOTA UNLIMITED ON xstream;
User created.
SQL> GRANT DBA TO xstrmadmin;
Grant succeeded.
SQL> BEGIN
  2  DBMS_XSTREAM_AUTH.GRANT_ADMIN_PRIVILEGE(
  3    grantee          => 'xstrmadmin',
  4    do_grants        => TRUE);
  5  END;
  6  /
PL/SQL procedure successfully completed.
```

3. Im nächsten Schritt wird ein Skript generiert. Dafür muss ein Directory angelegt werden.


```

SQL> CREATE DIRECTORY xstrm_dir AS '/home/oracle';
Directory created.
SQL> BEGIN
  2   DBMS_XSTREAM_AUTH.GRANT_ADMIN_PRIVILEGE(
  3     grantee          => 'xstrmadmin',
  4     do_grants        => FALSE,
  5     file_name         => 'grant_xstrm_privs.sql',
  6     directory_name    => 'xstrm_dir');
  7 END;
  8 /
PL/SQL procedure successfully completed.

```

4. Führen Sie das soeben generierte Skript aus, um dem XStream-Administrator weitere Privilegien zuzuweisen.

```

$ cd /home/oracle
$ sqlplus / as sysdba
SQL> @grant_xstrm_privs.sql
. . .

```

Im folgenden Beispiel soll die Tabelle „hr.employees“ repliziert werden. Analog können mehrere Tabellen oder mehrere Schemas konfiguriert werden. Mit Hilfe der entsprechenden Prozeduren des Pakets „DBMS_XSTREAM_ADM“ können weitere Tabellen und Schemas hinzugenommen oder entfernt werden. Führen Sie die folgende Prozedur als Benutzer „xstrmadmin“ aus.

```

SQL> DECLARE
  2   tables DBMS_UTILITY.UNCL_ARRAY;
  3   schemas DBMS_UTILITY.UNCL_ARRAY;
  4 BEGIN
  5   tables(1) := 'hr.employees';
  6   schemas(1) := '';
  7   DBMS_XSTREAM_ADM.CREATE_OUTBOUND(
  8     server_name      => 'xout',
  9     table_names       => tables,
 10     schema_names     => schemas);
 11 END;
 12 /
PL/SQL procedure successfully completed.

```

Überprüfen Sie, ob der Apply-Prozess gestartet ist:

```

SQL> SELECT apply_name,status FROM dba_apply;
APPLY_NAME                                STATUS
-----
XOUT                                       ENABLED

```

Damit ist die XStream-Konfiguration abgeschlossen, und wir können mit Hilfe der API auf den Outbound-Prozess zugreifen. Alternativ kann die Schnittstelle über Oracle OCI (also in C-Sprache) oder mit Java bedient werden. Im ersten Schritt betrachten wir ein Java-Programm, das sich als XStream-Client mit dem Stream „XOUT“ verbindet und die Logical Change Records ausliest. In Listing 15.7 finden Sie den Aufbau der Hauptprozedur.

Listing 15.7 Aufbau der Hauptprozedur

```

public static void main(String args[]) {
    out_url = "jdbc:oracle:oci:@"+ "localhost" + ":" + "1521" + ":" + "OHANSER"; (1)

```

```

System.out.println(out_url);
out_conn = createConnection(out_url, "xstrmadmin", "xstrmadmin");    (2)
xsOut = attachOutbound(out_conn);                                    (3)
get_lcrs(xsOut);                                                    (4)
detachOutbound(xsOut);
}

```

In Zeile (1) ist die URL für die JDBC-Verbindung zur Datenbank festgelegt. Die Verbindung selbst wird in Zeile (2) hergestellt. Das Java-Programm verbindet sich in Zeile (3) als XStream-Client zum Outbound-Prozess „XOUT“. Die Hauptschleife für das Auslesen der Logical Change Records besorgt die Prozedur „get_lcrs“, die wir uns noch etwas genauer anschauen (4). Zum Schluss meldet sich der Client vom Stream ab.

Listing 15.8 Schleife zum Auslesen eines LCR

```

public static void get_lcrs(XStreamOut xsOut) {
    byte[] ping_pos = null;
    byte[] fetchlwm = null;
    String src_db = null;
    if (null == xsOut) {
        System.out.println("xstreamOut ist nicht aktiv.");
        System.exit(0);
    }
    try {
        while(true) {
            LCR alcr = xsOut.receiveLCR(XStreamOut.DEFAULT_MODE);    (1)
            System.out.println(alcr);                                (2)
            fetchlwm = xsOut.getFetchLowWatermark();                 (3)
            if (null != alcr) {
                src_db = alcr.getSourceDatabaseName();
                if (xsOut.getBatchStatus() == XStreamOut.EXECUTING) {    (4)
                    assert alcr != null;
                    if (alcr instanceof RowLCR) {
                        if (((RowLCR)alcr).hasChunkData()) {            (5)
                            ChunkColumnValue chunk = null;
                            do {
                                chunk = xsOut.receiveChunk(XStreamOut.DEFAULT_MODE);
                            } while (!chunk.isEndOfRow());
                        }
                    }
                    processedLowPosition = alcr.getPosition();          (6)
                    ping_pos = processedLowPosition;
                } else {                                                )
                    assert alcr == null;
                    if (null != processedLowPosition)
                        xsOut.setProcessedLowWatermark(processedLowPosition,
                            XStreamOut.DEFAULT_MODE);
                }
            }
        }
    } catch (Exception e) {
        System.out.println("exception when processing LCRs");
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}

```


In Listing 15.8 finden Sie die Schleife zum Auslesen des LCR. In Zeile (1) wird der LCR vom Outbound-Server empfangen und in Zeile (2) auf dem Bildschirm ausgegeben. Zeile (3) merkt sich den Low Watermark. Falls eine Batchverarbeitung aktiv ist (4), werden in Zeile (5) die Daten gelesen. Anschließend wird die neue Position ermittelt (6).

Beim Auslesen durch den Client wird der LCR aus der Queue entfernt. In Listing 15.9 finden Sie die Ausgabe des Programms nach Absetzen einer INSERT- und einer COMMIT-Anweisung.

Listing 15.9 LCR-Inhalt für eine INSERT- und eine COMMIT-Anweisung

```
DefaultRowLCR:
Header:
source database name: OHANSER.WORLD
command type: INSERT
object owner: HR
object name: EMPLOYEES
tag: []
position: [0 0 0 f ffffffff c4 8 0 0 0 1 0 0 0 1 0 0 0 f ffffffff c3 ffffffff 0 0 0
1 0 0 0 1 1]
transaction id: 5.6.662
source time: 2012-07-10 00:59:17
scn: null
commit scn: null
lcr flags:
Statement LCR: false
IDKeyColumnsOnly LCR: false
Sequence LCR: false
Attribute[2]
(0): ROW_ID : AAEXrAAEAAACEAAD
(1): MESSAGE_TRACKING_LABEL : CAP$_XOUT_4:AUTOTRACK
Old column values (0):
New column values (2):
(0): DefaultColumnValue(csid:0, tde:false, type:0, name:EMPLOYEE_ID,
data:3.0)
(1): DefaultColumnValue(csid:0, tde:false, type:0, name:NAME, data:test4)

DefaultRowLCR:
Header:
source database name: OHANSER.WORLD
command type: COMMIT
object owner:
object name:
tag: []
position: [0 0 0 f ffffffff c4 8 0 0 0 1 0 0 0 1 0 0 0 f ffffffff c4 8 0 0 0 1 0 0
0 1 1]
transaction id: 5.6.662
source time: 2012-07-10 00:59:21
scn: null
commit scn: null
lcr flags:
Statement LCR: false
IDKeyColumnsOnly LCR: false
Sequence LCR: false
Attribute[0]
Old column values (0):
New column values (0):
```

Das komplette Programm zum Auslesen des XStream Outbound Servers „XOUT“ finden Sie in Listing 15.10. Für die Übersetzung und Ausführung müssen folgende Bibliotheken in den Classpath eingebunden werden:

```
$ORACLE_HOME/jdbc/lib/ojdbc6j.jar
$ORACLE_HOME/rdbms/jlib/xstreams.jar
```

Listing 15.10 Java-Programm zum Auslesen eines XStream Outbund-Servers

```
import oracle.streams.*;
import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.*;
import oracle.sql.*;
import java.sql.*;
import java.util.*;

public class xstream_pg {
    public static String out_url = null;
    public static Connection out_conn = null;
    public static XStreamOut xsOut = null;
    public static String xsoutName = "xout";
    public static byte[] lastPosition = null;
    public static byte[] processedLowPosition = null;
    public static Connection createConnection(String url, String username,
String passwd) {
        try {
            DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
            return DriverManager.getConnection(url, username, passwd);
        } catch (Exception e) {
            System.out.println("Datenbankverbindung konnte nicht hergestellt
werden: " +url);
            e.printStackTrace();
            return null;
        }
    }
    public static void printHex(byte[] b) {
        for (int i = 0; i < b.length; ++i) {
            System.out.print(Integer.toHexString((b[i]&0xFF) |
0x100).substring(1,3));
        }
        System.out.println("");
    }
    public static XStreamOut attachOutbound(Connection out_conn) {
        XStreamOut xsOut = null;
        try {
            xsOut = XStreamOut.attach((OracleConnection)out_conn, xsoutName,
lastPosition, XStreamOut.DEFAULT_MODE);
            System.out.println("Mit Outbound Server verbunden: "+xsoutName);
            System.out.print("Letzte Position ist: ");
            if (lastPosition != null) {
                printHex(lastPosition);
            }
            else {
                System.out.println("NULL");
            }
            return xsOut;
        } catch (Exception e) {
            System.out.println("Verbindung zu Outbound Server nicht möglich:
```



```

"+xsoutName);
    System.out.println(e.getMessage());
    e.printStackTrace();
    return null;
}
}
private static boolean samePos(byte[] pos1, byte[] pos2) {
    int cmp_len;
    boolean result;
    if (pos1.length != pos2.length) {
        return false;
    }
    for (int i = 0; i < pos1.length; i++) {
        if (pos1[i] != pos2[i])
            return false;
    }
    return true;
}
public static void get_lcrs(XStreamOut xsOut) {
    byte[] ping_pos = null;
    byte[] fetchlwm = null;
    String src_db = null;
    if (null == xsOut) {
        System.out.println("xstreamOut is null");
        System.exit(0);
    }
    try {
        while(true) {
            LCR alcr = xsOut.receiveLCR(XStreamOut.DEFAULT_MODE);
            System.out.println(alcr);
            fetchlwm = xsOut.getFetchLowWatermark();
            if (null != alcr)
                src_db = alcr.getSourceDatabaseName();
            if (xsOut.getBatchStatus() == XStreamOut.EXECUTING) {
                assert alcr != null;
                if (alcr instanceof RowLCR) {
                    if (((RowLCR)alcr).hasChunkData()) {
                        ChunkColumnValue chunk = null;
                        do {
                            chunk = xsOut.receiveChunk(XStreamOut.DEFAULT_MODE);
                            System.out.println(chunk.toString());
                        } while (!chunk.isEndOfRow());
                    }
                }
                processedLowPosition = alcr.getPosition();
                ping_pos = processedLowPosition;
            } else {
                assert alcr == null;
                if (null != processedLowPosition)
                    xsOut.setProcessedLowWatermark(processedLowPosition, XStreamOut.
DEFAULT_MODE);
            }
        }
    } catch (Exception e) {
        System.out.println("Ausnahmebedingung bei LCR-Verarbeitung");
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}

```

```

    }
    public static void detachOutbound(XStreamOut xsOut) {
        try {
            xsOut.detach(XStreamOut.DEFAULT_MODE);
        } catch (Exception e) {
            System.out.println("Abmeldung vom Outbound Server ist nicht möglich: "
                +xsOutName);
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
    public static void main(String args[]) {
        out_url = "jdbc:oracle:oci:@"+"localhost"+":"+"1521"+":"+"OEM";
        System.out.println(out_url);
        out_conn = createConnection(out_url, "sys as sysdba", "manager");
        xsOut = attachOutbound(out_conn);
        get_lcrs(xsOut);
        detachOutbound(xsOut);
    }
}

```

Der Oracle Enterprise Manager ermöglicht eine komfortable Bearbeitung und Überwachung von XStreams. Neben der Möglichkeit des Startens und Stoppens der einzelnen Komponenten liefert er nützliche Statusinformationen sowie Performance-Werte. Auf die zugehörigen Seiten gelangen Sie über das Register „Verschieben von Daten“ und den Link „Replikation verwalten“.

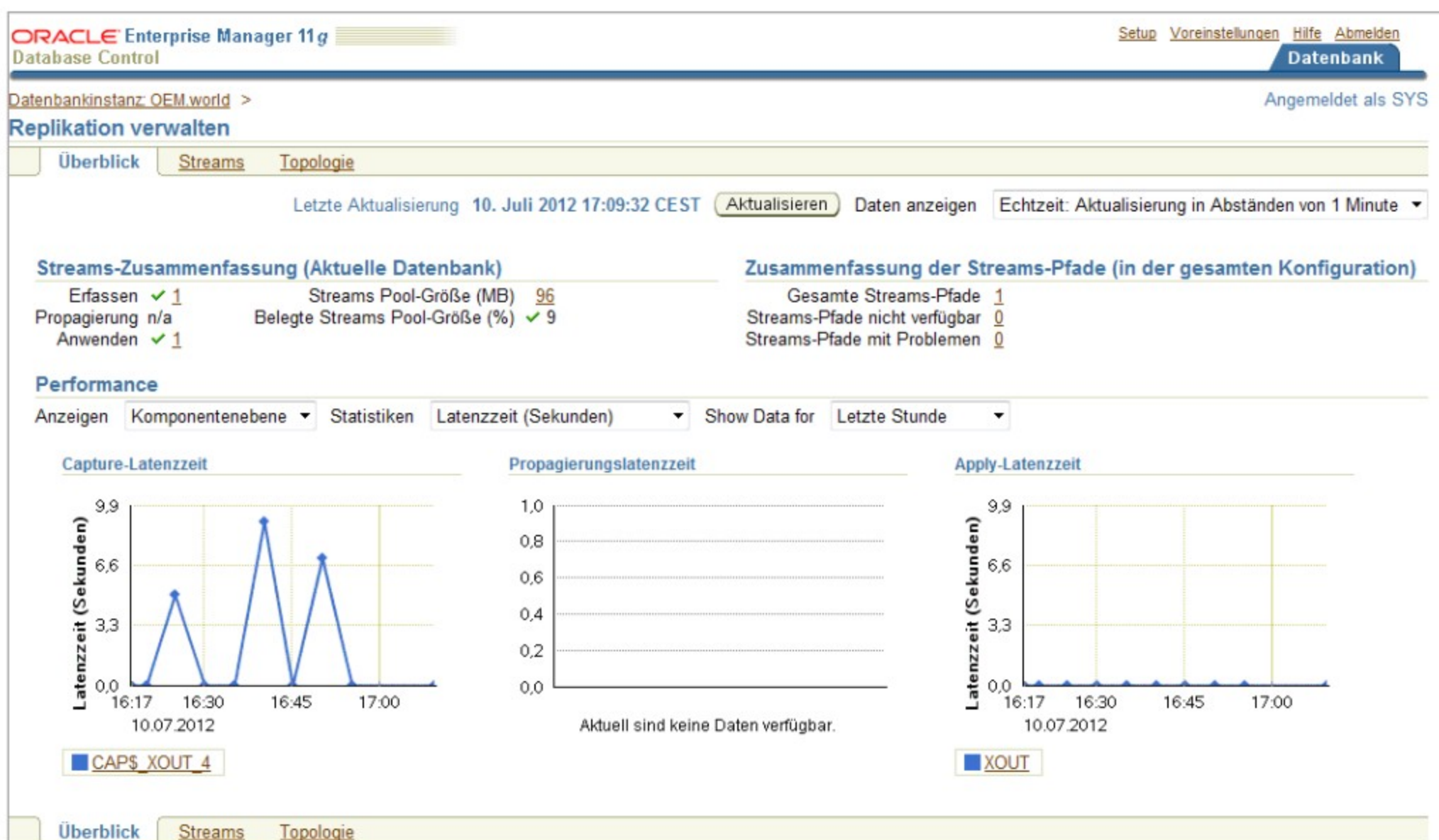


Bild 15.7 Verwaltung der Streams-Replikation im Enterprise Manager

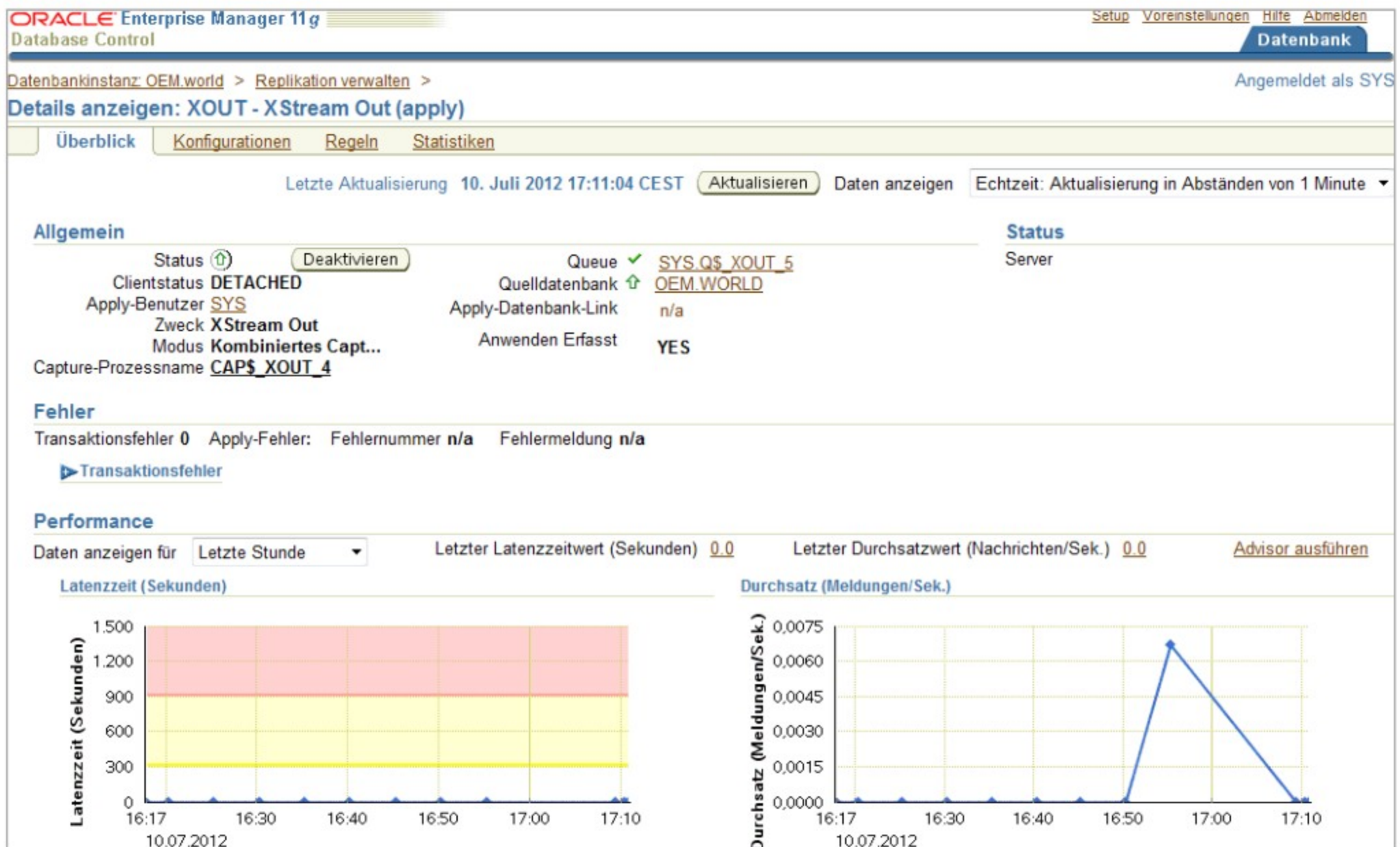


Bild 15.8 Der XStream Outbound-Server im Enterprise Manager



Hinweis

Eine detaillierte Beschreibung des XStream Outbound-Servers finden Sie in der Dokumentation der Oracle-Datenbank. Die Beispielprogramme sollten aufzeigen, wie man mit relativ geringem Programmieraufwand eine Schnittstelle zum Auslesen von Transaktionen aus der Oracle-Datenbank erstellen kann.

Über die Definition des XStream Outbound-Servers können Sie festlegen, welche Objekte aus dem Transaktionslog der Datenbank in die Outbound Queue übertragen werden. Die Outbound Queue besteht aus Logical Change Records (LCR). Mit dem Java-Programm sind wir in der Lage, den LCR auszulesen. Durch eine Modifikation des Programms in Listing 15.10 kann aus dem LCR die zugehörige SQL-Anweisung ausgelesen werden. Dazu wird die Funktion „getStatement“ aus der Java-API verwendet. Das Auslesen sollte nach der „if“-Anweisung erfolgen, in der abgeprüft wird, ob der LCR vom Typ „RowLCR“ ist, also ob es sich um eine DML-Anweisung handelt.

Listing 15.11 Auslesen der SQL-Anweisung aus dem LCR

```
...
if (alcr instanceof RowLCR) {
    System.out.println(" " + ((DefaultRowLCR) alcr).getStatement(false));
}
...
```

Führen Sie nun wie im Beispiel von Listing 15.12 Änderungen in der Oracle-Datenbank durch.

Listing 15.12 Die Tabelle „employees“ ändern

```
SQL> INSERT INTO hr.employees
  2  VALUES(555,'Meier','IT Strasse 1');
1 Zeile wurde erstellt.
SQL> COMMIT;
Transaktion mit COMMIT abgeschlossen.
SQL> UPDATE hr.employees
  2  SET name = 'Maier' WHERE employee_id=555;
1 Zeile wurde aktualisiert.
SQL> COMMIT;
Transaktion mit COMMIT abgeschlossen.
SQL> DELETE FROM hr.employees
  2  WHERE employee_id=555;
1 Zeile wurde gelöscht.
SQL> COMMIT;
Transaktion mit COMMIT abgeschlossen.
```

Das modifizierte Java-Programm verbindet sich zum XStream Out-Interface der Oracle-Datenbank, liest den LCR aus und gibt die zugehörige SQL-Anweisung aus, wenn es sich um eine DML-Anweisung handelt.

Listing 15.13 Ausgabe des Java-Programms

```
jdbc:oracle:oci:@localhost:1521:OHANSER
Mit Outbound Server verbunden: xout
Letzte Position ist: NULL
  INSERT INTO "HR"."EMPLOYEES" ( "EMPLOYEE_ID","NAME","STRASSE" ) VALUES
( 555,'Meier','IT Strasse 1' )
  COMMIT
  UPDATE "HR"."EMPLOYEES" SET "NAME" = 'Maier',"EMPLOYEE_ID" = 555,"STRASSE"
= 'IT Strasse 1' WHERE "EMPLOYEE_ID" = 555 AND "NAME" = 'Meier' AND "STRASSE"
= 'IT Strasse 1'
  COMMIT
  DELETE FROM "HR"."EMPLOYEES" WHERE "EMPLOYEE_ID" = 555 AND "NAME" =
'Maier' AND "STRASSE" = 'IT Strasse 1'
  COMMIT
```

Es ist eine Schnittstelle entstanden, die aus dem LCR von XStream SQL-Anweisung im lesbaren Textformat erzeugt. Obwohl sie noch auf DML-Anweisungen beschränkt ist, können alle Änderungen in den in XStream definierten Objekten ausgelesen werden.

**Hinweis**

Vielleicht haben Sie sich gewundert, dass die SQL-Anweisungen im Fall von „UPDATE“ und „DELETE“ nicht in der Form ausgegeben wurden, wie sie in SQLPlus erfasst wurden. Bedenken Sie dabei, dass XStream, so wie alle Streams-Produkte von Oracle, die Informationen aus dem Redo Log-Stream ausliest. Sie sehen die SQL-Anweisungen genau in der Form, wie sie in das Redo Log der Oracle-Datenbank geschrieben werden (daher die zusätzlichen Bedingungen in der WHERE-Klausel). Wie die WHERE-Klausel letztendlich aussieht, bestimmt die Einstellung für das „Supplemental Logging“. Details zum Thema „Supplemental Logging“ finden Sie in der Oracle-Dokumentation.

Da wir nun in der Lage sind, lesbare SQL-Anweisungen aus der XStream-Schnittstelle zu erzeugen, können wir uns diesen Umstand zunutze machen und die Anweisungen an eine PostgreSQL-Datenbank weiterleiten. Aus Java-Sicht ist das am einfachsten über eine JDBC-Schnittstelle möglich. Und schon verfügen wir über ein einfaches Replikationsprogramm von Oracle nach PostgreSQL. Die Übertragung erfolgt asynchron und die Auswahl der übertragenen Objekte bereits im XStream-Interface der Oracle-Datenbank, so dass nur Änderungen für die ausgewählten Objekte im LCR erscheinen.

Listing 15.14 JDBC-Schnittstelle zum Ausführen der Transaktionen

```
public static void pgTransaction(String sqlstmt, Connection pgconn) {
    Statement stmt;
    int rcount;
    System.out.println(sqlstmt);
    try {
        stmt = pgconn.createStatement();
        rcount = stmt.executeUpdate(sqlstmt);
        System.out.println(rcount + "Sätze wurden geändert.");
    } catch (SQLException sqlex) {
        System.out.println(sqlex.toString());
    }
}

...
if (alcr instanceof RowLCR) {
    sqlstmt = ((DefaultRowLCR) alcr).getStatement(false);
    pgTransaction(sqlstmt, pgconn);
}
...
```

Das Programm ist in dieser Ausbaustufe noch nicht für den produktiven Einsatz geeignet, da natürlich noch einiges mehr zu berücksichtigen ist, wie zum Beispiel Bindevariablen, DDL-Anweisungen oder Konfliktbehandlung. Es zeigt jedoch sehr gut auf, wie man die XStream-Schnittstelle verwendet und mit wenig Aufwand eine Replikation von Oracle nach PostgreSQL erstellen kann. In Listing 15.15 finden Sie das gesamte Programm mit Anwendung der Transaktionen in eine PostgreSQL-Datenbank.

Listing 15.15 Beispielprogramm zur Replikation von PostgreSQL nach Oracle mit XStream

```
import oracle.streams.*;
import oracle.jdbc.internal.OracleConnection;
import oracle.jdbc.*;
import oracle.sql.*;
import java.sql.*;
import java.util.*;

public class xstream_pg2 {
    public static String out_url = null;
    public static Connection out_conn = null;
    public static Connection pgconn = null;
    public static XStreamOut xsOut = null;
    public static String xsoutName = "xout";
    public static byte[] lastPosition = null;
    public static byte[] processedLowPosition = null;
    public static Connection createConnection(String url, String username,
String passwd) {
        try {
            DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

```

        return DriverManager.getConnection(url, username, passwd);
    } catch (Exception e) {
        System.out.println("Datenbankverbindung konnte nicht hergestellt
werden: " + url);
        e.printStackTrace();
        return null;
    }
}

public static Connection createPGConnection(String url, String username,
String passwd) {
    Connection conn = null;
    try {
        Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException ex) {
        System.out.println("Treiber konnte nicht geladen werden.");
        System.out.println(ex.toString());
    }
    try {
        conn = DriverManager.getConnection(url, username, passwd);
    } catch (SQLException cex) {
        System.out.println("Verbindung zur PostgreSQL-Datenbank konnte nicht
hergestellt werden.");
        System.out.println(cex.toString());
    }
    if ( conn != null ) {
        System.out.println("\nVerbindung zur PostgreSQL-Datenbank erfolgreich
hergestellt");
        System.out.println("\n" + url + "\n");
        return conn;
    }
    return null;
}

public static void pgTransaction(String sqlstmt, Connection pgconn) {
    Statement stmt;
    int rcount;
    System.out.println(sqlstmt);
    try {
        stmt = pgconn.createStatement();
        rcount = stmt.executeUpdate(sqlstmt);
        System.out.println(rcount + "Sätze wurden geändert.");
    } catch (SQLException sqllex) {
        System.out.println(sqllex.toString());
    }
}

public static void printHex(byte[] b) {
    for (int i = 0; i < b.length; ++i) {
        System.out.print(Integer.toHexString((b[i]&0xFF) | 0x100).
substring(1,3));
    }
    System.out.println("");
}

public static XStreamOut attachOutbound(Connection out_conn) {
    XStreamOut xsOut = null;
    try {
        xsOut = XStreamOut.attach((OracleConnection)out_conn, xsoutName,
lastPosition, XStreamOut.DEFAULT_MODE);
        System.out.println("Mit Outbound Server verbunden: "+xsoutName);
        System.out.print("Letzte Position ist: ");
    }
}

```



```

        if (lastPosition != null) {
            printHex(lastPosition);
        }
        else {
            System.out.println("NULL");
        }
        return xsOut;
    } catch (Exception e) {
        System.out.println("Verbindung zu Outbound Server nicht möglich:
"+xsoutName);
        System.out.println(e.getMessage());
        e.printStackTrace();
        return null;
    }
}

private static boolean samePos(byte[] pos1, byte[] pos2) {
    int cmp_len;
    boolean result;
    if (pos1.length != pos2.length) {
        return false;
    }
    for (int i = 0; i < pos1.length; i++) {
        if (pos1[i] != pos2[i])
            return false;
    }
    return true;
}

public static void get_lcrs(XStreamOut xsOut, Connection pgconn) {
    byte[] ping_pos = null;
    byte[] fetchlwm = null;
    boolean v_bind = false;
    String src_db = null;
    String sqlstmt = null;
    if (null == xsOut) {
        System.out.println("xstreamOut is null");
        System.exit(0);
    }
    try {
        while(true) {
            LCR alcr = xsOut.receiveLCR(XStreamOut.DEFAULT_MODE);
            fetchlwm = xsOut.getFetchLowWatermark();
            if (null != alcr)
                src_db = alcr.getSourceDatabaseName();
            if (xsOut.getBatchStatus() == XStreamOut.EXECUTING) {
                assert alcr != null;
                if (alcr instanceof RowLCR) {
                    sqlstmt = ((DefaultRowLCR) alcr).getStatement(false);
                    pgTransaction(sqlstmt, pgconn);
                    if (((RowLCR)alcr).hasChunkData()) {
                        ChunkColumnValue chunk = null;
                        do {
                            chunk = xsOut.receiveChunk(XStreamOut.DEFAULT_MODE);
                            System.out.println(chunk.toString());
                        } while (!chunk.isEndOfRow());
                    }
                }
                processedLowPosition = alcr.getPosition();
                ping_pos = processedLowPosition;
            }
        }
    }
}

```

```

        else {
            assert alcr == null;
            if (null != processedLowPosition)
                xsOut.setProcessedLowWatermark(processedLowPosition, XStreamOut.
DEFAULT_MODE);
        }
    }
}
catch(Exception e) {
    System.out.println("Ausnahmebedingung bei LCR-Verarbeitung");
    System.out.println(e.getMessage());
    e.printStackTrace();
}
}
public static void detachOutbound(XStreamOut xsOut) {
    try {
        xsOut.detach(XStreamOut.DEFAULT_MODE);
    } catch(Exception e) {
        System.out.println("Abmeldung vom Outbound Server ist nicht möglich:
"+xsOutName);
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
public static void main(String args[]) {
    out_url = "jdbc:oracle:oci:@"+ "localhost"+ ":"+ "1521"+ ":"+ "OEM";
    System.out.println(out_url);
    out_conn = createConnection(out_url, "sys as sysdba", "manager");
    pgconn = createPGConnection("jdbc:postgresql://localhost:5432/postgres", "
postgres", "post1234");
    if ( pgconn != null ) {
        xsOut = attachOutbound(out_conn);
        get_lcrs(xsOut, pgconn);
        detachOutbound(xsOut);
    }
}
}

```


Index

Symbole

.psqlrc 47

A

Account

- im Betriebssystem 10, 29
- sperren 87

Ad-hoc-Abfragen 132, 227

Administrierbarkeit 228

allow_system_table_mods 27

ANALYZE 44, 228

Analyze-Prozess 42

Anmeldeversuche 86

Anmeldung 74

Anpassung der Datentypen 243

ANSI-Syntax 245

Antwortzeiten 221

Anwendung, asynchrone 148

application_name 25

Apply-Prozess 269

Archiv-Modus 28

archive_command 20, 27

archive_mode 20

archive_timeout 20, 28

Archivlog-Verzeichnis 127

Archivierungskommando 28

Auditing 91, 225

Audit-Tabelle 184

Ausführungsmethoden 200

Ausführungsplan 24, 199, 201, 212

Ausführungsplaner 191

authentication_timeout 22

Authentifizierung 21, 22, 29, 230

- des Clients 29, 83
- Ident-basierte 76

Authentifizierungsmethode 29

Authentifizierungsprozess 74, 88

AUTOCOMMIT 35, 153, 155

autovacuum 23, 43

autovacuum_freeze_max_age 43

autovacuum_max_workers 23, 42

autovacuum_naptime 23, 42

B

Background Writer 19, 108, 122

Backup-Modus 62

Batch-Ladeprozesse 155

Batch-Verarbeitung 156

Benchmarks 118

– TPCC 120

Benchmark Utility 119

Betriebstauglichkeit 225

bgwriter_delay 19

bgwriter_lru_maxpages 19

bgwriter_lru_multiplier 20

Bibliothek 139

Bindevariablen 156

Bitmap-Index 202

Bitmap-Order-Operation 203

Block, dirty 104

Blockgröße 16

Brute-Force-Attacke 85, 86, 87

BSI 67

Buffer Cache 100, 103, 201, 204

Businessfunktionalität 194, 244

C

C-Programm 140

C-Programmstruktur 146

C++ Compiler 142

Capture-Prozess 266

Change-Request-Verfahren 73

Checkpoint 19, 62

Checkpoint-Prozess 108

checkpoint_completion_target 19

checkpoint_segments 19, 100

checkpoint_timeout 19

checkpoint_warning 19

Client, Identität 74

Client-Programme 139

client_min_messages 25

COMMIT 18, 34

Concurrency 33

COPY-Befehl 48, 60, 257

cpu_index_tuple_cost 23

cpu_tuple_cost 23

Crash 18

Crash Recovery 19, 28, 133

CREATE DATABASE 59

CREATE RULE 191

CSV-Datei 238, 252

Cursor 147

Cursor-Variablen 250

D

Data Warehouse-Datenbank 226

Dateisystem-Cache 100

Daten, geometrische 213, 216

Datenaustausch 259

Datenbank

- anlegen 36
- entfernte 259
- kopieren 59
- löschen 36
- wiederherstellen 58

Datenbank-Abfrage, geometrische 218

Datenbank-Auditing 91

Datenbank-Cluster 16

Datenbank-Katalog, Oracle-kompatibler 248

Datenbank-Link 259, 268

Datenbank-Objekte 76

Datenbank-Schema, Export 251

Datenbankwachstum 99

Datenblöcke 201

Daten-Cache 206

Datensicherung 225

Datentypen

- anpassen 254
- geografische 216, 221, 218
- geometrische 219
- Konvertierung 263
- Unterschiede 233
- Vergleich von 243

Datenverlust 43, 227

DBMS_METADATA 254

db_user_namespace 22

DDL-Anweisungen, Überwachung 91

DDL-Befehle 253, 264

Dead Flag 43

Dead Row Flag 43

Deadlock 34

DECLARE-Anweisung 248

default_tablespace 22

Defragmentierung 230

Desaster Recovery 126, 135, 225, 227

Dirty Buffer 19

Disk Cache 24, 199

Downtime 229

Dump-Datei 236

Dumpfile 60

E

effective_cache_size 24, 100

Excel 46, 49

EXCEPTION 182

Exception-Klausel 250

EXPLAIN ANALYZE 205, 207

EXPLAIN-Befehl 205

Export-Utility 231

F

Failover 135

Failover-Situation 129

Features 225

Filesystem-Cache 206

FOR-Schleifen 248

Foreign Key Constraints 254

Form, geometrische 217

FrozenXID 43

fsync 18

Full-Table-Scan 201, 202

Full-Vacuum 42

Funktionsaufruf 250

G

Gateway 263

Gateway-Datenbank 269, 271

geco 24

Genetic Query Optimizer 200

geqo_effort 24

geqo_threshold 24

GRANT-Option 79

Gruppenrollen 72

H

Hacker 86

Hardware-Ressourcen 15

Hash Join 200, 204

Hash Key 204

Hash-Tabelle 211

Hauptspeicher 16

Heterogeneous Gateway 259, 265

HIPPA 67

Hit Ratio 107

Hochverfügbarkeit 125

Host Intrusion Detection-Systeme 86

hot_standby 1, 20, 21, 227

Hot-Standby-Datenbank 132, 134

- Aktivierung 133

Hot-Standby-Modus 133

I

Ident-Server 76

Import in PostgreSQL 257

Indexe 201

- nicht verwendete 49
- Statistiken für 107

Index-Scan 200

Informationen, geografische 213

initdb 16

Inner Loop 210

Installation 5

- aus dem Quellcode 5, 9, 214, 216

INSTEAD, Option 193

I/O-Durchsatz 206

I/O-Operationen 204

I/O-Performance 107

I/O-Statistiken 113

Isolation Level 33, 34

J

Java 152

Java-Applikationen 152

JDBC 139

JDBC-Schnittstelle 152, 153

JDBC-Treiber für PostgreSQL 152

JDBC-Verbindung 273

K

Katalog 35

Kompatibilität 259

- zu Oracle 247

Kompatibilitätsgrad 259

Kontrolldatei 50

Konvertierung 233, 236, 238, 243

- der Schema-Definition 251

Koordinatensystem, kartesisches 216

Kosten 202

- für das Lesen 199
- für den Optimizer 200

L

Längen- und Breitengrad 218, 219

Langläufer 113

Large Objects 149, 157, 159

Large Object-API 160

Laufzeitverbesserung 156

LCR 266

LDAP-Server 75
 Lesekonsistenz 33
 libpq 139, 142
 libpq-Programme 141
 List Partitioning 115
 Locale-Umgebung 39
 Locking 114
 log_autovacuum_min_durations 23
 log_connections 86
 log_destination 25, 48
 log_directory 25
 log_filename 25
 Log-Dateien, Rotation 48
 logging_collector 25
 Logging und Tracing 24
 Logging-Parameter 26
 Logical Change Record 266, 270, 272
 Login-Privileg 71
 Logischer Host 136
 log_min_duration_statement 25
 log_min_error_statement 25
 log_min_messages 25
 log_rotation_age 25
 log_rotation_size 25, 48
 log_statement 26, 91

M

maintenance_work_mem 17
 max_connections 100
 max_fsm_pages 17
 max_fsm_relations 17
 max_prepared_transactions 17
 max_standby_archive_delay 21
 max_standby_streaming_delay 21
 max_wal_senders 21, 130
 md5 31
 Merge Join 205, 210
 Merge-Join-Operation 205
 Metadaten 153
 Microsoft SDK 9
 Microsoft Visual Studio 9
 Migration 233, 242, 244, 248, 256
 Migrationsaufwand 247
 Mischbetrieb 1, 259
 Monitoring 93, 228
 Multi-User-Fähigkeit 124
 MySQL 1, 233, 234

MySQL-Datenbank 234
 MySQL-Dump 235, 238
 MySQL-Format 236

N

Nagios 96
 Nested-Loop 200
 Nested-Loop-Operation 204, 209
 Network Spoofing 88
 Netzwerk-Performance 27
 NULL-Werte 234

O

Objekte, Zugriff auf 70
 Objektprivilegien 73, 78
 ODBC-Datenquelle 262
 ODBC-Schnittstelle 139, 231, 259, 264
 ODBC-Treiber 49
 ODBC-Verbindung 49
 Offline-Sicherung 61
 OID 38, 103, 151
 OLTP-Eigenschaften 122
 Online-Backup 229
 Online-Sicherung 62, 63, 65
 Online Transaction Processing 119
 OpenJUMP 217
 Open-Source 1, 213
 Open-Source-Produkt 241
 Open Source-Programm 217
 OpenSSL 81
 Operator im Ausführungsplan 200
 Optimierung von SQL-Anweisungen 199
 Optimizer 199, 211
 Optimizer Hints 246
 Oracle, umsteigen von 241
 Oracle-Applikationen 246
 Oracle-Datenbank 1, 245, 247, 251, 252, 259
 Oracle-Gateway-Datenbank 267
 Oracle Golden Gate 259, 270
 Oracle Streams 259, 266
 Oracle-Syntax 250
 Oracle XStream 266, 270
 Outbound-Prozess 272
 Outer Join 245

Outer Loop 210
 Out-of-the-box-Sicherheit 85

P

Paging 100
 Paketinstallation 5, 7
 Parameter
 – benutzerspezifische 26
 – dynamische 16
 – für Entwickler 26
 – nicht-Standard 45
 Parsen der SQL-Anweisung 211
 Parser 191
 Partitionen 114, 118
 Partition Key 115
 Partitionierung 114, 115
 password_encryption 22
 Passwort 31, 70
 – auslesen 45
 – MD5 verschlüsseltes 75
 – schwaches 73
 – verschlüsseltes 71, 87
 Passwort-Check 72
 Passwortdatei 45
 Passwort-Komplexität 85
 Passwortkontrolle 84
 Passwortlisten 86
 Performance 124, 225
 Performance-Analyse 105, 106, 109, 113
 Performance-Impact 130
 Performance-Monitoring 93, 94
 Performancetest 199, 242
 Performance Tuning 99
 Performance-Verbesserung 118, 199
 Performance-Vergleich 119
 Performance-Vorteil 211
 Perl 162
 – Active State 9
 – Distribution 162
 pgAdmin III 6, 80, 93, 96, 195, 228
 PGDATA 11, 20, 29, 37, 62, 64, 82, 127
 PGDATA-Verzeichnis 15, 28
 pgFoundry-Projekt 101
 PGPASSFILE 45
 pg_authid 31, 71
 pg_buffercache 102
 pg_class 79
 pg_controldata 50

pg_conversion 40
 pg_ctl 11
 pg_ctl reload 31
 pg_current_xlog_location 65
 pg_default 37
 pg_dump 57
 pg_dumpall 60
 pg_global 37
 pg_hba.conf 12, 29, 74, 84, 130
 pg_is_in_recovery 66
 pg_largeobject 149
 pg_locks 114
 pg_roles 70
 pg_settings 15
 pg_stat_activity 96, 107
 pg_stat_bgwriter 108
 pg_stat_database 94, 112
 pgstatpack 109, 110
 pgstatpack-Bericht 111
 pg_switch_xlog 47
 pg_tablespace 37
 pg_tblspc 38
 pgtune 101
 pg_xlog 21, 28, 62, 127
 pg_xlogfile_name 66
 PL/pgSQL 174, 176, 179, 228, 241, 248
 – anonyme Blöcke 182
 – Datentypen 175
 – Variablen 174
 PL/pgSQL-Block 178, 180, 181
 PL/pgSQL Code 90
 PL/pgSQL-Konvertierung 251
 PL/SQL 241
 – Portierung 248
 PL/SQL Code 242
 – Kompatibilität 247
 PL/SQL-Prozedur 264
 Point-in-time-Recovery 27, 28, 61, 65, 226
 Point-in-time-Wiederherstellung 60
 Pool-Größe 200
 Portnummer 5
 post_auth_delay 27
 PostGIS 213, 226
 PostGIS 2.0.0 215
 PostGIS-Datenbank 214
 PostGIS-Software 214
 Postgres95 2
 postgresql.conf 12, 15, 16
 PostgreSQL-Quellcode 10
 Prepared Statement 156

Prepared Transactions 17
 Primärsystem 126
 primary_conninfo 130
 Private Key 81
 Privilegien 68
 Produktionsdatenbanken 227
 Programmiersprachen 139
 Projektion 218, 220, 221
 psql 11, 78

Q

Quellcode-Distribution 141
 Query Optimizer 23, 199
 Query-Plan 205
 Query-Planer 100

R

RAISE-Befehl 182
 random_page_cost 23
 Range Partitioning 115
 Read Committed 33
 Read-only-Modus 125, 132
 Re-Connect 137
 Recovery 18
 Recovery-Modus 21, 64, 127, 132
 Recovery-Prozess 27
 recovery.conf 64, 128
 Ref Cursor 180
 Regeln für INSERT 197
 Regelsystem 191
 Replikation 129, 259, 265, 267, 271, 280
 – asynchrone 126
 – synchrone 125, 131, 229
 – Überwachung der 131
 Replikationslösungen 125
 Replikationsmechanismen 231
 Replikationsprogramm 280
 Replikations-Umgebung 267
 Reporting 126
 Ressourcenbelastung 223
 Ressourcenverbrauch 95, 105, 203
 restore_command 127
 Resultset 146, 202
 REVOKE-Befehl 80
 Risiko-Minimierung 67
 Rolle anlegen 69
 Rollenkonzept 68, 230
 Rollentausch 126, 135, 137
 rotatelog 49

ROWID, Spalte 246
 ROWTYP-Variable 249

S

Satzbearbeitung, einzelne 146
 Satzlänge, durchschnittliche 207
 SAVEPOINT 33
 – zurückrollen 33
 Scan, sequenzieller 200, 202
 Schema-Eigentümer 73
 search_path 22
 Security-Richtlinien 67
 SELECT-Anweisung 144
 SELECT-Privilegien 70
 Selektivität 209
 seq_page_cost 23
 Server-Crash 63
 Server-Log 25
 Server-Programmierung 161
 Server-Zertifikat 82
 Service Level Agreement 57
 Session killen 45
 SET ROLE 72
 Shared Memory-Konfiguration 17
 shared_buffers 16, 100
 shared_preload_libraries 110
 Sicherheit 67, 84, 230
 Sicherheits-Schwachstelle 85
 Sicherung 57
 Sicherungsmethoden 57
 Single Sign On 76
 Skalierbarkeit 99, 225, 230
 SNMP-Schnittstelle 98, 228
 SOX 67
 Spalte, geometrische 217, 220
 Spatial-Datenbank 213, 215
 Spatial-Index 222
 Spatial System 213
 Speicherbedarf 17
 Sperren 34, 95, 96, 114
 Sprachen, prozedurale 161
 SQL erweitern 185
 SQL-Anweisung 24
 – Laufzeiten 206
 SQL-Dump 57, 60, 65, 226
 SQL-Erweiterung 185
 SQL-Injection 89, 90
 SQL-Optimierung 199
 SQL-Syntax 245
 SRID 220

SSH-Server 83
 SSH-Tunnel 81, 83
 SSL-Verbindung 22, 81, 82, 83
 SSL-Verschlüsselung 81, 230
 sslinfo 83
 Standard-Parameter 99
 Standard-Vacuum 42
 Standardisierungsgrad 68
 Standby-Datenbank 57, 125, 227
 – aufsetzen 126
 Standby-Server 128
 standby_mode 128
 Startup-Kosten 207
 statement_timeout 22
 Statistics Collector 105, 112
 Statistiken 44
 – sammeln 44, 105
 – zurücksetzen 96
 Statistik-Daten 105
 Statistik-Tabellen 93
 Streaming Replication 1, 20,
 125, 127, 130, 227, 229
 Strings, leere 244
 Subqueries 209
 Superuser 31, 62
 Superuser-Privileg 69
 superuser_reserved_
 connections 21
 Switchover 135, 137
 synchronous_commit 18, 132
 Syntax-Fehler 26
 Systemkatalog 22

T

Tabelle
 – innere 210
 – partitionierte 116
 Tabellenpartitionen 37
 Tabellen-Vererbung 115
 Tablespace 22, 37
 – anlegen 37
 – Default- 37
 – Temporary 22
 Target-Datenbank 267
 temp_buffers 17
 temp_tablespace 22
 Template 39
 Testmigration 242
 Totalausfall 227
 Totalverlust 63
 TPCC 119
 trace_locks 27

trace_notify 27
 track_counts 23
 Transaction-ID 42
 Transaction Isolation 33
 Transaktionen
 – implizite 33
 – Anzahl von 119
 – verteilte 259, 264
 Transaktionsaufkommen 19, 101,
 125
 Transaktions-ID 64
 Transaktions-Logdaten 259
 Transaktionslog 18, 265
 Transaktionslog-Shipping 125
 Transaktionsrate 100, 104
 Transaktionssicherheit 18
 Transformationsfunktion 220
 Transparent Gateway 260
 Trigger 183, 197
 Trigger-Funktion 183

U

Übernahme der Daten 252
 Überwachung 67, 68, 90, 225
 Umgebungsanalyse 222
 Umgebungsvariablen 140
 Umstellungsaufwand 242
 UNIX-Format 3
 UTF8 39

V

VACUUM 228
 Vacuum
 – automatisches 41, 42, 43
 – manuelles 42
 Vacuum-Operationen 23
 Vacuum-Prozess 41, 43
 vacuum_freeze_min_age 43
 vacuum_freeze_table_age 43
 Vendor-Support 232
 Verbindungsanfrage 74
 Verbindungsversuch 69
 Vereinigungsbedingung 210
 Vereinigungsmethoden 204
 Verfügbarkeit 65, 93, 229
 Virtuelle IP-Adresse 136

W

WAL 18
 WAL-Archivierung 62, 127, 226

WAL Archiving 27
 WAL-Dateien
 – Archivierung 27
 – rückspeichern 64
 WAL Records 127
 WAL-Segment 20, 28, 47, 62
 WAL Shipping 128
 WAL-Switch manuell auslösen
 47
 WAL-Übertragung 130
 wal_buffers 18, 101, 104
 wal_debug 27
 wal_keep_segments 21
 wal_level 18
 wal_sender_delay 21
 wal_sync_method 18
 wal_writer_delay 19
 Wartungsarbeiten 228
 Wartungsaufgaben 40
 Wiederherstellung 57, 64, 225,
 226
 – des Clusters 60
 Wiederherstellungszeiten 57
 Wiederherstellungszeitpunkt 64
 Windows 3
 Windows 7 10
 Windows-Betriebssystem 5, 25,
 29, 214
 Windows-Dienst 5, 10
 Windows-Eventlog 25
 Windows-Zeichensatz 39
 work_mem 17, 100
 WRITE AHEAD LOG 18, 20

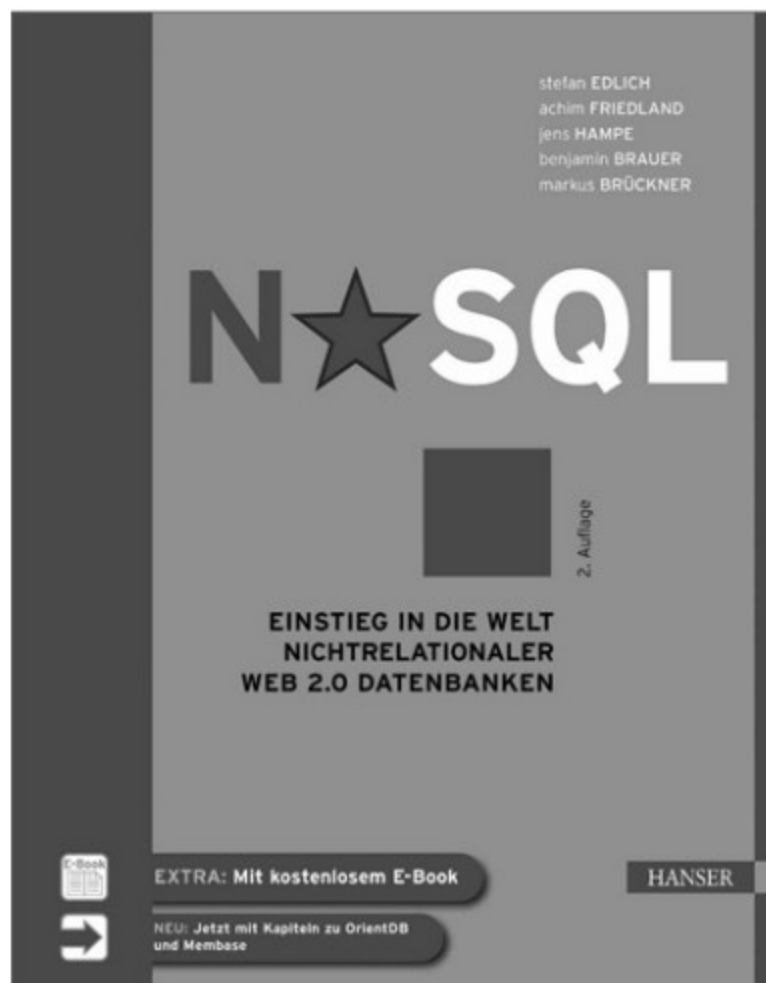
X

XID 42
 XID Wraparound 42
 XML-Format 208
 XStream, Überwachung 277
 XStream-Administrator 271
 XStream API 271
 XStream-Konfiguration 272
 XStream Outbound Server 271

Z

Zeichensätze
 – Multi-byte 39
 – Single-byte 39
 Zeilenstatistiken 95
 Zertifikat 81, 83
 Zugriffsrechte 8

Lieber No statt SQL



Edlich, Friedland, Hampe,
Brauer, Brückner

NoSQL

Einstieg in die Welt nichtrelationaler
Web 2.0 Datenbanken

410 Seiten.

ISBN 978-3-446-42753-2

NoSQL-Datenbanken sind dabei, die Datenbankwelt zu revolutionieren. Laufend hört man von neuen NoSQL-Datenbanken, die sich jeweils für besondere Einsatzbereiche gut eignen. Dieses Buch hilft Datenbank- und Web-Entwicklern, den Überblick zu behalten und das richtige System für ihre Zwecke zu finden.

Sie lernen die Konzepte von NoSQL-Systemen kennen, die sie von den klassischen relationalen Datenbanksystemen unterscheiden. Vorgestellt und erklärt werden die theoretischen Grundlagen Map/Reduce, CAP-Theorem/Eventually Consistent, Consistent Hashing, MVCC-Protokoll, Vector Clocks, Paxos und REST. Detailliert stellen die Autoren die wichtigsten Vertreter der vier großen NoSQL-Datenbanktypen Wide Column Stores, Document Stores, Key/Value-Datenbanken und Graphdatenbanken vor.

Sie erfahren, wie die Systeme aufgesetzt und skaliert werden, welche Vor- und Nachteile sie haben und für welche Anwendungsgebiete Sie sie einsetzen können. So haben Sie eine gute Grundlage für die Entscheidung, welche Datenbank in Ihrem Fall am besten geeignet ist.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser-fachbuch.de/computer

Geballtes Know-how!



Held, Hotzy, Fröhlich, Adar,
Antognini, Häfeli, Steiger, Vetter,
Welker

Der Oracle DBA

Handbuch für die Administration
der Oracle Database 11g R2

802 Seiten

ISBN 978-3-446-42081-6

Als Administrator der Oracle Database finden Sie in diesem Handbuch die ideale Unterstützung für die Herausforderungen Ihres Alltags. Hier hat sich eine Gruppe namhafter und praxiserfahrener Autoren zusammengetan, die gemeinsam 134 Jahre Erfahrung mit dieser Datenbank aufweisen und in den jeweiligen Kapiteln ihre Spezialgebiete darstellen.

Die Autoren vermitteln Ihnen fundiertes Know-how sowie praktische Lösungen zu Themen wie Aufbau und Betrieb eines Datenbankservers, Administration und Monitoring, High Availability, Backup und Recovery, Security, Upgrade einer Datenbank und Optimierung.

Zusätzlich finden Sie in diesem Handbuch neben einer Darstellung der Produkte und Features der Oracle Database 11g R2 viele Beispiele, Praxistipps und Tricks, die Sie direkt in Ihre tägliche Arbeit integrieren können und die über Versionsgrenzen hinweg anwendbar sind.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser-fachbuch.de/computer

Power-Tuning.



Haas

Oracle Tuning in der Praxis

Rezepte und Anleitungen für Datenbankadministratoren und -entwickler
362 Seiten.

3., erweiterte und aktualisierte Auflage
ISBN 978-3-446-41907-0

Damit eine Oracle-Datenbank wirklich gut läuft und die größtmögliche Performance aufweist, reicht die richtige Installation alleine nicht aus. Als Administrator müssen Sie die Methoden der Optimierung und des Tunings kennen und einsetzen, um optimale Ergebnisse zu erzielen.

Dieses Praxisbuch liefert dafür zahlreiche Anleitungen und Skripte. Lernen Sie nicht nur SQL-Abfragen zu optimieren, sondern profitieren Sie auch von Tipps und Informationen für alle anderen Ebenen, auf denen Tuning stattfindet.

Die dritte, erweiterte Auflage enthält viele aktuelle Beispiele und geht auch auf die Neuerungen der Oracle-Version 11g ein, wie z.B. Database Replay, SQL Performance Analyzer und SQL Plan Management.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser-fachbuch.de/computer

PRAXISWISSEN FÜR ENTWICKLER



Testen Sie jetzt

„web & mobile developer“
und sichern Sie sich
2 Ausgaben kostenfrei!

Schwerpunkte:

- HTML5 & CSS3
- PHP & RIA
- iOS
- Android

Jetzt anfordern unter:

webundmobile.de/probelesen

PostgreSQL 9 //

- Administration und Programmierung der Open Source-Datenbank
- Die Version 9 mit vielen neuen Features
- Migration von und Replikation mit anderen Datenbanksystemen
- Einbindung in die IT-Landschaft
- Im Internet: Die Programme und Skripte aus dem Buch unter <http://downloads.hanser.de>

Wenn Sie als Administrator, Entwickler oder IT-/System-Architekt PostgreSQL professionell und erfolgreich in der Praxis einsetzen wollen, finden Sie in diesem Arbeitsbuch die richtige Anleitung. Administratoren bietet es Unterstützung beim täglichen Datenbankbetrieb sowie detaillierte Informationen und Praxistipps zu Themen wie Backup and Recovery, Sicherheit und Überwachung, Performance Tuning oder Hochverfügbarkeit. Einen weiteren Schwerpunkt bildet die Replikation von und nach anderen Datenbanksystemen. IT-Architekten, System-Designer und IT-Manager erfahren, wie PostgreSQL kostengünstig in eine bestehende IT-Infrastruktur platziert werden kann und effektiv mit anderen Datenbanksystemen, wie z.B. Oracle, zusammenarbeitet. Entwickler lernen, wie robuste und performante Datenbankapplikationen erstellt werden und erhalten außerdem einen Leitfaden für den Umstieg von Oracle auf PostgreSQL.

Lutz **FRÖHLICH** ist Diplom-Mathematiker und freiberuflicher IT-Berater. Er beschäftigt sich seit 20 Jahren mit Datenbanken und schöpft seine praktischen Erfahrungen aus langjährigen Consul-

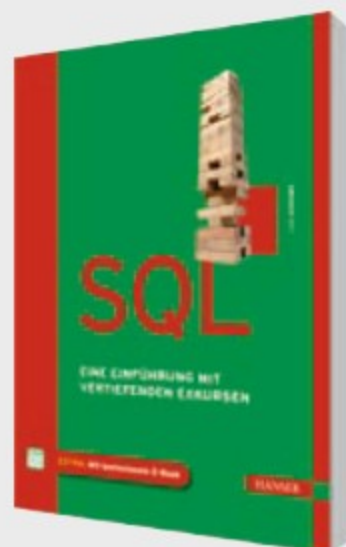


ting-Tätigkeiten in den USA und in Europa. Seine Spezialgebiete sind Performance und Hochverfügbarkeit.

AUS DEM INHALT //

- Konfiguration von Cluster und Datenbank
- Hot Standby und Streaming Replication
- Entwicklung von Applikationen für PostgreSQL
- Replikation zwischen PostgreSQL und Oracle
- PostGIS Spatial Database
- PostgreSQL in die IT-Landschaft einbinden

UNSER BUCHTIPP FÜR SIE //



Adams, SQL
2012. 459 Seiten. FlexCover. € 29,90.
ISBN 978-3-446-43200-0

HANSER

www.hanser-fachbuch.de/computer

€ 39,90 [D] | € 41,10 [A]
ISBN 978-3-446-42239-1



9 783446 422391