PostgreSQL
Das Offizielle Handbuch

Peter Eisentraut

PostgreSQL

Das Offizielle Handbuch

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

ISBN 3-8266-1337-6
1. Auflage 2003

Printed in Germany
© Copyright 2003 by mitp-Verlag GmbH/Bonn,
ein Geschäftsbereich der verlag moderne industrie Buch AG&Co.KG/Landsberg

Lektorat: Sabine Schulz
Sprachkorrektorat: Friederike Daenecke
Satz und Layout: G&U e.Publishing Services GmbH, Flensburg
Druck: Media-Print, Paderborn

Inhaltsverzeichnis

Einleitung

Dieses Buch ist die offizielle Dokumentation von PostgreSQL. Es wird parallel zur Entwicklung der PostgreSQL-Software von den Entwicklern und anderen Freiwilligen geschrieben und aktualisiert. Es beschreibt sämtliche Funktionalität, die die vorliegende PostgreSQL-Version offiziell unterstützt.

Um bei diesem umfassenden Anspruch übersichtlich zu bleiben, ist dieses Buch in mehrere Teile aufgeteilt, die der Leser zu unterschiedlichen Zeiten zurate ziehen wird:

- **Teil I** ist eine informelle Einführung für neue Anwender.
- **Teil II** beschreibt die Sprache SQL, Datentypen, Funktionen und Tipps zur Leistungsverbesserung auf Benutzerebene. Jeder PostgreSQL-Benutzer sollte diesen Teil lesen.
- **Teil III** beschreibt die Installation und Verwaltung des Servers. Jeder, der einen PostgreSQL-Server betreibt, egal ob nur für sich selbst oder für andere, sollte diesen Teil lesen.
- **Teil IV** beschreibt die Programmierschnittstellen für PostgreSQL-Clientprogramme.
- **Teil V** enthält Informationen für fortgeschrittene Benutzer über die Erweiterungsfähigkeiten des Servers. Themen sind zum Beispiel benutzerdefinierte Datentypen und Funktionen.
- **Teil VI** enthält Informationen über die Syntax von SQL-Befehlen, Client- und Serverprogrammen. Dieser Teil unterstützt die anderen Teile mit nach Befehl oder Programm sortierten strukturierten Informationen.

Was ist PostgreSQL?

PostgreSQL ist ein objektrelationales Datenbankverwaltungssystem auf Basis von POSTGRES, Version 4.2, welches in der Informatikfakultät der Universität von Kalifornien in Berkeley entwickelt wurde. POSTGRES hat viele Konzepte eingeführt, die erst später in einigen kommerziellen Datenbanken zur Verfügung standen.

PostgreSQL ist ein Open-Source-Nachfolger dieses Codes von Berkeley. Es unterstützt SQL92 und SQL99 und bietet viele moderne Fähigkeiten:

- komplexe Anfragen
- Fremdschlüssel
- Trigger
- Sichten
- transaktionale Integrität
- Multiversionskontrolle

Außerdem ist PostgreSQL auf einzigartige Weise vom Benutzer erweiterbar, zum Beispiel mit

- Datentypen
- Funktionen
- Operatoren
- Aggregatfunktionen
- Indexmethoden
- prozeduralen Sprachen

Und wegen der liberalen Lizenz kann PostgreSQL von allen kostenlos für alle Zwecke, ob privat, geschäftlich oder akademisch, benutzt, verändert und weitergegeben werden.

Die kurze Geschichte von PostgreSQL

Das heutzutage als PostgreSQL bekannte objektrelationale Datenbankverwaltungssystem stammt vom POSTGRES-Paket der Universität von Kalifornien in Berkeley ab. Nach mehr als einem Jahrzehnt Entwicklungsarbeit ist es jetzt das fortschrittlichste Open-Source-Datenbanksystem.

Das POSTGRES-Projekt in Berkeley

Das POSTGRES-Projekt, unter Anleitung von Professor Michael Stonebraker, wurde gesponsert von der Defense Advanced Research Projects Agency (DARPA), dem Army Research Office (ARO), der National Science Foundation (NSF) und ESL, Inc. Die Implementierung von POSTGRES begann 1986. Die anfänglichen Konzepte wurden in *Stonebraker & Rowe 1986* vorgestellt und die Definition des Datenmodells erschien in *Rowe & Stonebraker 1987*. Der Entwurf des Regelsystems zu dieser Zeit wurde in *Stonebraker, Hanson, Hong 1987* beschrieben. Die Architektur des Storagemanagers wurde in *Stonebraker 1987* beschrieben.

Seitdem hat POSTGRES mehrere Hauptversionen erlebt. Das erste „Demoware“-System war 1987 einsatzbereit und wurde 1988 auf der ACM-SIGMOD-Konferenz gezeigt. Version 1, beschrieben in *Stonebraker, Rowe, Hirohama 1990*, wurde im Juni 1989 einigen externen Benutzern zugänglich gemacht. Als Reaktion auf eine Kritik des ersten Regelsystems (*Stonebraker, Hearst, Potamianos 1989*) wurde das Regelsystem neu entworfen (*Stonebraker, Jhingran, Goh 1990*) und im Juni 1990 wurde Version 2 mit dem neuen Regelsystem freigegeben. Version 3 erschien 1991 mit Unterstützung für mehrere Storagemanager, einem verbesserten Anfrage-Executor und einem neu geschriebenen Umschreiberegelsystem. Die folgenden Versionen bis Postgres95 (siehe unten) haben sich dann hauptsächlich auf Portierbarkeit und Zuverlässigkeit konzentriert.

POSTGRES wurde für die Implementierung von vielen verschiedenen Forschungs- und Produktionsanwendungen verwendet. Unter anderem: ein System zur Analyse von finanziellen Daten, ein Paket zur Überwachung der Leistung von Düsentriebwerken, eine Datenbank zur Verfolgung von Asteroiden und eine medizinische Informationsdatenbank. POSTGRES wurde auch als Lehrwerkzeug an mehreren Universitäten verwendet. Schließlich hat Illustra Information Technologies (später fusioniert mit *Informix*, was jetzt zu *IBM* gehört) den Code genommen und kommerzialisiert. Im Jahr 1992 wurde POSTGRES der primäre Datenmanager des wissenschaftlichen Computing-Projekts *Sequoia 2000*.

Im Verlauf des Jahres 1993 hatte sich die Größe der externen Benutzergemeinde nahezu verdoppelt. Es wurde zunehmend klar, dass die Wartung des Prototypcodes und der Support eine Menge Zeit in Anspruch nahmen, die eigentlich der Datenbankforschung gewidmet werden sollte. Um diese Last zu verringern wurde das POSTGRES-Projekt in Berkeley mit Version 4.2 offiziell eingestellt.

Postgres95

Im Jahr 1994 bauten Andrew Yu und Jolly Chen einen SQL-Interpreter in POSTGRES ein. Unter neuem Namen wurde Postgres95 im Web freigegeben, um seinen eigenen Weg als Open-Source-Nachfolger des ursprünglichen POSTGRES-Codes zu finden.

Der Postgres95-Code war komplett in ANSI C und die Größe wurde um 25 % reduziert. Viele interne Veränderungen haben die Leistung und die Wartungsfreundlichkeit des Codes verbessert. Postgres95 Version 1.0.x war in der Wisconsin Benchmark etwa 30–50% schneller als POSTGRES, Version 4.2. Neben berichtigten Fehlern waren dies die bedeutenden Verbesserungen:

- Die Anfragesprache PostQUEL wurde durch SQL ersetzt (im Server implementiert). Unteranfragen wurden erst in PostgreSQL (siehe unten) unterstützt, aber sie konnten in Postgres95 mit benutzerdefinierten SQL-Funktionen nachgeahmt werden. Aggregatfunktionen wurden neu implementiert. Unterstützung für die Anfrageklausel GROUP BY wurde ebenfalls hinzugefügt.
- Neben dem „Monitor“-Programm wurde ein neues Programm (psql) für interaktive SQL-Anfragen angeboten, welches GNU Readline verwendete.
- Eine neue Clientbibliothek, libpgtcl, unterstützte Clients auf Tcl-Basis. Eine Beispiel-Shell, pgtclsh, bot neue Tcl-Befehle, um von Tcl-Programmen aus auf den Postgres95-Server zugreifen zu können.
- Die Large-Object-Schnittstelle wurde überholt. Die Inversion-Large-Objects waren jetzt der einzige Mechanismus für Large Objects. (Das Inversion-Dateisystem wurde entfernt.)
- Das Regelnsystem auf Instanzebene wurde entfernt. Regeln waren immer noch als Umschreiberegeln verfügbar.
- Ein kurzes Tutorial mit einer Einführung in SQL und Postgres95 wurde mit dem Quellcode verteilt.
- GNU Make (anstatt BSD Make) wurde für die Compilierung verwendet. Außerdem konnte Postgres95 mit einem ungepatchten GCC kompiliert werden (Datenausrichtung von double-Werten wurde repariert).

PostgreSQL

Im Jahre 1996 wurde klar, dass der Name „Postgres95“ dem Test der Zeit nicht standhalten würde. Also haben wir einen neuen Namen, PostgreSQL, ausgewählt, der das Verhältnis zwischen dem ursprünglichen POSTGRES und den neueren Versionen mit SQL-Unterstützung widerspiegelt. Gleichzeitig wurde die Versionsnummer auf 6.0 gesetzt, wodurch die Nummern die in Berkeley begonnene Reihe fortsetzten.

Während der Entwicklung von Postgres95 lag die Betonung auf dem Verstehen von bestehenden Problemen in Servercode. Mit PostgreSQL verschob sich die Konzentration auf neue Features und Fähigkeiten, obwohl die Arbeit in allen Bereichen weiterging.

Was sich seither in PostgreSQL im Einzelnen getan hat, können Sie in *Anhang D* sehen.

Über diese Ausgabe

Diese Buchausgabe hat die Dokumentation von PostgreSQL Version zur Grundlage. Allerdings ist die Dokumentation für diese Buchausgabe vollständig überarbeitet worden. Insbesondere wurde die alte Aufteilung der PostgreSQL-Dokumentation in mehrere Bände aufgegeben. Die Kapitel wurden methodisch sinnvoller angeordnet und ganze Abschnitte umgeschrieben. Das Stichwortverzeichnis wurde erheblich erweitert. Und natürlich wurden sämtliche Fakten noch einmal überprüft. Sämtliche Änderungen sind auch in die Originalversion der PostgreSQL-Dokumentation eingeflossen und werden in der nächsten öffentlichen PostgreSQL-Version enthalten sein. (Dieses Buch hat also diesbezüglich gewissermaßen Weltpremierencharakter.)

Weggelassen wurden in dieser Ausgabe nur einige Kapitel aus dem Interna-Teil, weil diese Informationen nur für Entwickler von PostgreSQL selbst sinnvoll sind, und dann auch nur, wenn die Informationen aus der aktuellsten Entwicklerversion verwendet werden. Außerdem wurde aus Platzgründen die Versionsgeschichte bei den älteren Versionen etwas gekürzt.

Konventionen

Dieses Buch verwendet folgende typografische Konventionen, um bestimmte Textteile hervorzuheben: Neue Begriffe, Fremdwörter (die nicht als Fachwörter übernommen wurden) und sonstige wichtige Stellen werden *kursiv* hervorgehoben. Alles, was eine Eingabe in oder eine Ausgabe aus dem Computer darstellt, insbesondere Befehle, Programmcode und Bildschirmausgaben, wird in einer nicht proportionalen Schrift dargestellt (`Beispiel`). Innerhalb solcher Textteile bedeutet kursive Schrift (*`Beispiel`*), dass Sie an dieser Stelle einen konkreten Wert anstelle des Platzhalters einsetzen müssen. Gelegentlich werden Stellen in Programmcode durch fette Schrift (**`Beispiel`**) hervorgehoben, wenn sie gegenüber dem vorangegangenen Beispiel hinzugefügt oder verändert worden sind.

In der Synopsis eines Befehls werden folgende Konventionen verwendet: Eckige Klammern (`[` und `]`) zeigen optionale Teile an. (In der Synopsis eines Tcl-Befehls werden stattdessen Fragezeichen (`?`) verwendet, wie es bei Tcl üblich ist.) Geschweifte Klammern (`{` und `}`) und senkrechte Striche (`|`) bedeuten, dass Sie eine Alternative auswählen müssen. Punkte (`...`) bedeuten, dass das vorherige Element wiederholt werden kann.

Wenn es der Klarheit dient, dann steht vor SQL-Befehlen die Eingabeaufforderung `=>` und vor Shell-Befehlen `$`. In der Regel werden die Eingabeaufforderungen aber nicht mit angegeben.

Im Übrigen ist in diesem Buch jeder ein *Benutzer*, der irgendeinen Teil des PostgreSQL-Systems verwendet oder verwenden möchte. Ein *Administrator* ist im Allgemeinen die Person, die für die Installierung, Einrichtung und Überwachung des Servers verantwortlich ist. Diese Begriffe sollten aber nicht zu eng gesehen werden, denn dieses Buch macht keine Annahmen über die Systemverwaltungsprozeduren.

Weitere Informationen

Neben der Dokumentation, das heißt diesem Buch, gibt es weitere Informationsquellen über PostgreSQL:

FAQ

Die FAQ-Liste enthält laufend aktualisierte Antworten auf häufig gestellte Fragen.

README*s*

Für einige Extrapakete gibt es README-Dateien im Quellcodebaum.

Website

Die *PostgreSQL-Website* enthält Einzelheiten über neue Versionen und andere Informationen, die Ihre Arbeit oder Ihr Vergnügen mit PostgreSQL produktiver machen können.

Mailinglisten

Die Mailinglisten sind ein guter Ort, wo Sie Antworten auf Ihre Fragen erhalten, Ihre Erfahrungen mit anderen teilen oder mit den Entwicklern in Verbindung treten können. Einzelheiten dazu erfahren Sie auf der PostgreSQL-Website.

Sie selbst!

PostgreSQL ist ein Open-Source-Projekt. Daher baut es darauf, dass sich die Benutzer gegenseitig unterstützen. Wenn Sie anfangen, PostgreSQL zu verwenden, werden Sie die Hilfe anderer benötigen, entweder durch die Dokumentation oder durch Mailinglisten. Wenn Sie Erfahrungen gesammelt haben, dann sollten Sie diese wiederum mit anderen teilen. Lesen Sie die Mailinglisten und beantworten Sie Fragen. Wenn Sie etwas erfahren, das in der Dokumentation unzureichend erklärt wird, dann schreiben Sie darüber und schicken es ein. Wenn Sie den Code erweitern, dann schicken Sie Patches an die Entwickler.

Richtlinien für Fehlerberichte

Wenn Sie einen Fehler in PostgreSQL finden, dann wollen wir das wissen. Ihre Fehlerberichte spielen eine wichtige Rolle bei der Entwicklung von PostgreSQL, weil selbst die größte Sorgfalt nicht garantieren kann, dass jeder Teil von PostgreSQL unter allen Umständen und auf jeder Plattform funktioniert.

Die folgenden Vorschläge sollen Ihnen dabei helfen, Fehlerberichte zusammenzustellen, die effektiv bearbeitet werden können. Niemand ist verpflichtet, sich daran zu halten, aber es ist meist für alle Seiten von Vorteil.

Wir können Ihnen nicht versprechen, jeden Fehler sofort zu reparieren. Wenn der Fehler offensichtlich oder gefährlich ist oder viele Benutzer betrifft, dann stehen die Chancen gut, dass sich jemand die Sache anschauen wird. Es könnte auch sein, dass wir Ihnen sagen werden, eine neuere Version auszuprobieren und festzustellen, ob der Fehler dort auch auftritt. Oder wir könnten zu dem Schluss kommen, dass der Fehler erst repariert werden kann, wenn ein größeres, geplantes Codeprojekt abgeschlossen ist. Oder es ist einfach zu kompliziert und es gibt wichtigere Dinge auf der Tagesordnung. Wenn Sie sofort Hilfe benötigen, dann sollten Sie kommerzielle Unterstützung suchen.

Fehler identifizieren

Bevor Sie über einen Fehler berichten, lesen Sie bitte die Dokumentation gründlich, um zu überprüfen, ob das, was Sie versucht haben, wirklich zulässig ist. Wenn es aus der Dokumentation nicht klar hervorgeht, ob etwas zulässig ist, dann sollten Sie das auch berichten; das ist nämlich ein Fehler in der Dokumentation. Wenn es sich herausstellt, dass ein Programm etwas anderes tut, als die Dokumentation aussagt, dann ist das ein Fehler. Die folgende Liste enthält mögliche Fehlerumstände, ist aber nicht abschließend.

- Ein Programm wird von einem Signal oder mit einer Fehlermeldung aus dem Betriebssystem, die auf ein Problem im Programm hindeutet, abgebrochen. (Ein Gegenbeispiel wäre eine Meldung, dass die Festplatte voll ist, weil Sie das Problem selber lösen müssen.)
- Ein Programm erzeugt die falsche Ausgabe für irgendeine Eingabe.
- Ein Programm weigert sich, gültige Eingaben anzunehmen (wobei gültige Eingaben in der Dokumentation definiert sind).
- Ein Programm akzeptiert ungültige Eingaben ohne Warn- oder Fehlermeldung. Aber bedenken Sie, dass Ihre Vorstellung von ungültiger Eingabe unsere Vorstellung von einer Erweiterung oder Kompatibilität mit alten Gewohnheiten sein könnte.
- PostgreSQL kann nicht entsprechend den Installationsanweisungen auf einer unterstützten Plattform kompiliert, gebaut oder installiert werden.

Hier bezieht sich “Programm” auf jedes Programm, nicht nur auf den Server.

Wenn ein Programm zu langsam ist oder zu viele Ressourcen verbraucht, dann ist das nicht unbedingt ein Fehler. Lesen Sie die Dokumentation oder fragen Sie auf einer der Mailinglisten nach, um Unterstützung bei der Feinabstimmung Ihrer Anwendungen zu erhalten. Mangelhafte Konformität mit dem SQL-Standard ist auch kein Fehler, es sei denn, die Konformität wurde in einem bestimmten Bereich ausdrücklich angegeben.

Bevor Sie weitermachen, prüfen Sie die “TODO”-Liste und die FAQ, um zu sehen, ob Ihr Fehler schon bekannt ist. Wenn Sie die Informationen auf der TODO-Liste nicht entziffern können, dann berichten Sie Ihr Problem trotzdem. Zumindest könnten wir dann die TODO-Liste etwas klarer machen.

Was berichtet werden sollte

Am wichtigsten ist es bei einem Fehlerbericht, dass Sie alle Fakten und nur Fakten angeben. Fangen Sie nicht an, Vermutungen darüber anzustellen, was wohl schiefgegangen ist oder was es “anscheinend”

gemacht hat oder welcher Teil des Programms einen Fehler hat. Wenn Sie sich in der Implementierung nicht auskennen, dann wird Ihre Vermutung wahrscheinlich falsch sein und uns kein bisschen weiterhelfen. Aber selbst wenn Sie sich auskennen, dann sind wohl begründete Erklärungen eine großartige Ergänzung zu, aber kein Ersatz für Fakten. Wenn wir den Fehler berichtigen sollen, dann müssen wir ihn erst selbst auftreten sehen. Die reinen Fakten zu berichten, ist ziemlich einfach (Sie können sie wahrscheinlich direkt vom Bildschirm kopieren), aber viel zu oft werden wichtige Einzelheiten weggelassen, weil jemand dachte, dass sie nicht wichtig sind oder dass der Bericht trotzdem verstanden werden würde.

Die folgenden Elemente sollten in jedem Fehlerbericht enthalten sein:

- Die genaue Reihenfolge der Schritte *vom Programmstart an*, mit denen das Problem reproduziert werden kann. Diese Informationen sollten vollständig sein; ein einzelner `SELECT`-Befehl reicht nicht aus, wenn die Ausgabe von Daten in einer Tabelle abhängig sein soll; dann müssen Sie auch die vorangegangenen `CREATE TABLE`- und `INSERT`-Befehle angeben. Wir haben keine Zeit, Ihr Datenbankschema zu erraten, und wenn wir unsere eigenen Daten erfinden sollen, dann werden wir das Problem wahrscheinlich nicht entdecken.

  Das beste Format für ein Testbeispiel eines Problems, das mit SQL zu tun hat, ist eine Datei, die mit dem Clientprogramm `psql` ausgeführt werden kann und so das Problem aufzeigt. (Achten Sie darauf, dass Sie nichts in der Startdatei `~/.psqlrc` haben.) Eine einfache Möglichkeit, so eine Datei zusammenzustellen, ist mit `pg_dump` die Tabellendeklarationen und Daten in einer Datei zu speichern und dann die problematische Anfrage hinzuzufügen. Sie sollten versuchen, die Größe Ihres Beispiels auf ein Minimum zu reduzieren, aber das ist nicht unbedingt nötig. Wenn der Fehler reproduzierbar ist, dann werden wir ihn so oder so finden.

  Wenn Ihre Anwendung eine andere Clientschnittstelle verwendet, etwa `PHP`, dann versuchen Sie bitte, die problematischen Anfragen herauszufinden. Wir werden wahrscheinlich nicht erst einen Webserver einrichten, um Ihr Problem zu entdecken. Denken Sie auf jeden Fall daran, die genauen Eingabedateien mitzuschicken. Stellen Sie keine ungenauen Vermutungen an, dass das Problem bei “großen Dateien” oder “mittelgroßen Datenbanken” auftritt, weil solche Informationen nutzlos sind.

- Die Ausgabe, die Sie erhalten haben. Schreiben Sie bitte nicht, dass etwas “nicht funktioniert” oder “abgestürzt” ist. Wenn es eine Fehlermeldung gibt, dann zeigen Sie sie, selbst wenn Sie sie nicht verstehen. Wenn das Programm mit einem Betriebssystemfehler abbricht, dann zeigen Sie den ebenfalls. Wenn gar nichts passiert, dann sagen Sie das. Selbst wenn das Ergebnis Ihres Testbeispiels ein Programmabsturz oder anderweitig offensichtlich ist, könnte es sein, dass es auf unseren Plattformen anders ausgeht. Am einfachsten ist es, wenn möglich, die Ausgabe vom Terminal zu kopieren.

  Anmerkung

  Bei schweren Fehlern, die die Datenbanksitzung abbrechen, enthält die im Client sichtbare Fehlermeldung möglicherweise nicht alle verfügbaren Informationen. Bitte schauen Sie auch in die Logausgabe des Datenbankservers. Wenn Sie die Logausgabe des Servers nicht speichern, dann wäre dies eine gute Gelegenheit, damit anzufangen.

- Welche Ausgabe Sie erwartet haben, ist ganz wichtig. Wenn Sie einfach schreiben: “Dieser Befehl erzeugt diese Ausgabe.” oder “Dieses Ergebnis habe ich nicht erwartet.”, dann werden wir es vielleicht selbst probieren, die Ausgabe kurz prüfen, und denken, dass alles in Ordnung ist und dass wir es genau so erwartet hätten. Sie sollten nicht erwarten, dass wir die Bedeutung Ihrer Befehle entziffern werden. Sie sollten auch davon absehen, einfach nur zu sagen, “Das stimmt nicht mit dem SQL-Standard überein.” oder “Oracle macht es anders.” Das richtige Verhalten aus dem SQL-Standard herauszusuchen, ist nicht besonders unterhaltsam, und wir wissen auch nicht, wie all die anderen relationalen Datenbanken dieser Welt funktionieren. (Wenn Ihr Problem ein Programmabsturz ist, dann müssen Sie hierzu logischerweise nichts angeben.)
- Welche Kommandozeilenoptionen und andere Startparameter, einschließlich Umgebungsvariablen und Konfigurationsdateien, Sie gegenüber der Standardeinstellung verändert haben. Seien Sie wiederum genau. Wenn Sie eine Paketdistribution verwenden, die den Datenbankserver beim Booten startet, dann sollten Sie herausfinden, wie das geschieht.

- Jegliche Abweichungen von den Installationsanweisungen.
- Die PostgreSQL-Version. Mit dem Befehl `SELECT version();` können Sie die Version des Servers herausfinden, mit dem Sie verbunden sind. Die meisten Programme unterstützen auch eine Option `--version`; zumindest sollte `postmaster --version` und `psql --version` funktionieren. Wenn die Funktion oder die Optionen nicht vorhanden sind, dann ist Ihre Version mehr als alt und sollte unbedingt aktualisiert werden. Wenn Sie eine Paketdistribution verwenden, zum Beispiel ein RPM-Paket, dann sollten Sie das sagen und auch eine eventuelle Unterversion des Pakets angeben. Wenn Sie einen CVS-Schnappschuss verwenden, dann sollten Sie Datum und Zeit dazu angeben.

  Wenn Ihre Version älter ist als , dann werden wir Ihnen höchstwahrscheinlich raten, eine neuere Version zu verwenden. In jeder neuen Version werden eine Menge Fehler berichtigt, deswegen bringen wir neue Versionen heraus.
- Plattforminformationen. Das bedeutet Kernelname und -version, C-Bibliothek, Prozessor und Speicherinformationen. In den meisten Fällen reicht es aus, den Hersteller und die Version anzugeben, aber Sie sollten nicht davon ausgehen, dass jeder weiß, was in “Debian” enthalten ist oder dass jeder Pentiums verwendet. Wenn Sie Installationsprobleme haben, dann benötigen wir auch Informationen über den Compiler, das Make-Programm usw.

Sie sollten sich nicht scheuen, wenn Ihr Fehlerbericht ziemlich lang wird. Das ist halt so. Es ist besser, wenn Sie alles beim ersten Mal mit angeben, als dass wir die Fakten aus Ihnen herausquetschen müssen. Wenn Ihre Eingabedateien allerdings riesig sind, dann ist es vertretbar, dass Sie erstmal nachfragen, ob sich jemand die Sache ansehen will.

Verschwenden Sie Ihre Zeit nicht damit, herauszufinden, durch welche Änderungen in der Eingabe das Problem verschwindet. Das wird bei der Lösung wahrscheinlich nicht helfen. Wenn es sich herausstellt, dass der Fehler nicht sofort berichtigt werden kann, dann werden Sie immer noch Zeit haben, einen vorübergehenden Ausweg zu finden und mitzuteilen. Und noch einmal sei gesagt: Verschwenden Sie Ihre Zeit nicht damit, zu raten, warum der Fehler auftritt. Das werden wir schon früh genug herausfinden.

Wenn Sie einen Fehlerbericht schreiben, dann verwenden Sie bitte klare Terminologie. Das Softwarepaket selbst heißt “PostgreSQL”, manchmal auch kurz “Postgres”. Wenn Sie konkret vom Serverprogramm sprechen, dann sagen Sie das und nicht einfach “PostgreSQL stürzt ab”. Ein Absturz eines einzelnen Serverprozesses ist etwas ganz anderes als ein Absturz des “postmaster”-Prozesses; bitte schreiben Sie nicht “Der Postmaster ist abgestürzt”, wenn nur ein einzelner Serverprozess abgestürzt ist, und auch nicht umgekehrt. Außerdem sind Clientprogramme wie “`psql`” vollkommen getrennt vom Server. Bitte geben Sie genau an, ob das Problem auf der Client- oder der Serverseite liegt.

Wo Fehler berichtet werden können

Prinzipiell sollten Sie Fehlerberichte an die dafür gedachte Mailingliste `pgsql-bugs@postgresql.org` senden. Bitte wählen Sie eine beschreibende Betreffzeile, vielleicht einen Teil der Fehlermeldung.

Eine weitere Möglichkeit ist das Ausfüllen des entsprechenden Web-Formulars auf der Projektwebsite . Wenn ein Bericht dort eingegeben wird, dann wird er automatisch an die Mailingliste `pgsql-bugs@postgresql.org` geschickt.

Bitte senden Sie keinen Fehlerbericht an Benutzer-Mailinglisten wie `pgsql-sql@postgresql.org` oder `pgsql-general@postgresql.org`. Diese Mailinglisten sind für die Beantwortung von Benutzerfragen, und die Leser wollen dort normalerweise keine Fehlerberichte erhalten. Und höchstwahrscheinlich werden die Leser die Fehler auch nicht berichtigen können.

Senden Sie bitte auch *keine* Fehlerberichte an die Entwickler-Mailingliste pgsql-hackers@postgresql.org. Diese Liste ist für Diskussionen über die Entwicklung von PostgreSQL, und es wäre nett, wenn wir die Fehlerberichte getrennt halten könnten. Wenn Ihr Problem einer ausführlicheren Prüfung bedarf, dann könnten wir eine Diskussion über das Problem auf `pgsql-hackers` aufnehmen.

Wenn Sie ein Problem mit der Dokumentation haben, dann schreiben Sie am besten an die Dokumentations-Mailingliste pgsql-docs@postgresql.org. Bitte geben Sie den genauen Teil der Dokumenation an, mit dem Sie ein Problem haben.

Wenn Ihr Problem ein Portierungsproblem auf einer nicht unterstützten Plattform ist, dann schreiben Sie an pgsql-ports@postgresql.org, damit wir (und Sie) an der Unterstützung für Ihre Plattform arbeiten können.

Anmerkung

Weil leider viel zu viel Spam umhergeht, sind alle oben genannten E-Mail-Adressen geschlossene Mailinglisten. Das heißt, Sie müssen sich erst bei der Liste anmelden, um dort posten zu dürfen. (Für das Web-Formular müssen Sie sich aber nicht irgendwo anmelden.) Wenn Sie an die Listen schreiben wollen, aber selbst keine Post bekommen wollen, dann können Sie sich anmelden und dann Ihre Anmeldung auf nomail setzen. Um weitere Informationen zu erhalten, schreiben Sie eine E-Mail an majordomo@postgresql.org mit nur dem Wort help im Text.

Teil I

Tutorial

Willkommen zum PostgreSQL-Tutorial. Die folgenden Kapitel sollen Ihnen eine Einführung in PostgreSQL, relationale Datenbanken und die Sprache SQL geben, wenn Sie mit einem dieser Bereiche noch nicht vertraut sind. Sie benötigen nur etwas allgemeines Wissen, wie man Computer verwendet. Keine besonderen Erfahrung mit Unix oder mit Programmieren ist vonnöten. Dieser Teil soll Ihnen vor allem praktische Erfahrung mit den wichtigen Teilen des PostgreSQL-Systems geben. Sie sollten nicht davon ausgehen, dass die Behandlung der Themen vollständig und tiefgründig ist.

Wenn Sie dieses Tutorial durchgearbeitet haben, können Sie ***Teil II*** lesen, um die Sprache SQL genauer kennen zu lernen, oder ***Teil IV*** um zu erfahren, wie Sie Anwendungen für PostgreSQL entwickeln können. Wenn Sie Ihren eigenen Server einrichten und verwalten wollen, dann sollten Sie auch ***Teil III*** lesen.

1 Ein Einstieg

1.1 Installation

Bevor Sie PostgreSQL verwenden können, müssen Sie es natürlich installieren. Möglicherweise ist PostgreSQL bei Ihnen schon installiert, entweder, weil es in Ihrer Betriebssystem-Distribution enthalten war, oder weil der Systemadministrator es schon installiert hat. In diesem Fall sollten Sie Ihren Systemadministrator um Informationen bitten, wie Sie auf PostgreSQL zugreifen können.

Wenn Sie nicht sicher sind, ob PostgreSQL bereits verfügbar ist oder ob Sie es zum Ausprobieren verwenden können, dann können Sie es selbst installieren. Das ist nicht schwer und kann eine gute Übung sein. PostgreSQL kann von jedem unprivilegierten Benutzer installiert werden; Superuser-Rechte (`root`) sind nicht erforderlich.

Wenn Sie PostgreSQL selbst installieren, finden Sie die Installationsanweisungen in Kapitel 14. Kehren Sie zu diesem Tutorial zurück, wenn die Installation abgeschlossen ist. Mit Sorgfalt sollten Sie den Abschnitt lesen, der beschreibt, wie man die passenden Umgebungsvariablen einrichtet.

Wenn der Systemadministrator Einrichtungen vorgenommen hat, die nicht den Standardvorgaben entsprechen, haben Sie eventuell noch etwas Arbeit vor sich. Wenn zum Beispiel der Datenbankserver ein anderer Computer ist, dann müssen Sie die Umgebungsvariable `PGHOST` auf den Namen der Datenbankservermaschine setzen. Die Umgebungsvariable `PGPORT` muss möglicherweise auch gesetzt werden. Letztendlich gilt: Wenn Sie ein Anwendungsprogramm starten und es beschwert, sich dass es nicht zur Datenbank verbinden kann, sollten Sie sich mit Ihrem Systemadministrator in Verbindung setzen, oder wenn Sie das selbst sind, dann sollten Sie die Dokumentation zurate ziehen um zu prüfen, ob Ihre Umgebungsvariablen richtig gesetzt sind. Wenn Sie den vorangegangenen Absatz nicht verstanden haben, lesen Sie den nächsten Abschnitt.

1.2 Grundlagen der Architektur

Ehe wir fortfahren, sollten Sie ein grundlegendes Verständnis der Systemarchitektur von PostgreSQL haben. Wenn Sie verstehen, wie die Teile von PostgreSQL zusammenarbeiten, wird dieses Kapitel etwas klarer.

In der Sprache der Datenbanktechnik verwendet PostgreSQL ein Client/Server-Modell. Eine PostgreSQL-Sitzung besteht aus den folgenden kooperierenden Prozessen (Programmen):

- Ein Server-Prozess, der die Datenbankdateien überwacht, Verbindungen von Clientanwendungen akzeptiert und Aktionen in der Datenbank für den Client ausführt. Das Datenbankserverprogramm heißt `postmaster`.
- Die Clientanwendung des Benutzers, die Datenbankoperationen vornehmen will. Clientanwendungen können sehr verschiedener Natur sein: Ein Client könnte ein textorientiertes Tool sein, eine grafische Anwendung, ein Webserver, der auf die Datenbank zugreift um Webseiten darzustellen, oder eine spezialisiertes Datenbankverwaltungswerkzeug. Einige Clientanwendungen sind in der PostgreSQL-Distribution enthalten, aber die meisten werden von Benutzern entwickelt.

Wie es bei Client/Server-Anwendungen typisch ist, können sich Client und Server auf verschiedenen Rechnern befinden. In diesem Fall kommunizieren sie über eine TCP/IP-Netzwerkverbindung. Sie sollten das im Hinterkopf behalten, denn Dateien, die auf der Clientmaschine zur Verfügung stehen, sind möglicherweise auf der Datenbankserver-Maschine nicht zugänglich (oder nur unter einem anderen Namen).

Der PostgreSQL-Server kann mehrere gleichzeitige Verbindungen von Clients handhaben. Dazu startet er einen neuen Prozess für jede Verbindung (mit dem Systemaufruf `fork`). Danach kommunizieren der Client und der neue Serverprozess ohne das Eingreifen des ursprünglichen `postmaster`-Prozesses. Folglich läuft der `postmaster` immer und wartet auf Verbindungen, während Client- und die zugehörigen Serverprozesse entstehen und wieder verschwinden. (AlPostgreSQLl das ist natürlich für Benutzer unsichtbar. Wir erwähnen es hier nur der Vollständigkeit halber.)

1.3 Eine Datenbank erzeugen

Der erste Test, um zu sehen, ob Sie auf den Datenbankserver zugreifen können, ist, eine Datenbank zu erzeugen. Ein laufender PostgreSQL-Server kann mehrere Datenbanken verwalten. Normalerweise verwendet man eine separate Datenbank für jedes Projekt oder für jeden Benutzer.

Möglicherweise hat der Systemadministrator schon eine Datenbank für Sie erstellt. Er sollte Ihnen dann mitgeteilt haben, was der Name dieser Datenbank ist. In diesem Fall können Sie diesen Schritt auslassen und zum nächsten Abschnitt springen.

Um eine neue Datenbank, in diesem Beispiel mit Namen `meinedb`, zu erstellen, verwenden Sie den folgenden Befehl:

```
$ createdb meinedb
```

Das sollte folgende Antwort erzeugen:

```
CREATE DATABASE
```

Wenn ja, dann war dieser Schritt erfolgreich und Sie können den Rest dieses Abschnitts überspringen.

Wenn Sie eine Mitteilung ähnlich wie

```
createdb: command not found
```

sehen, dann ist PostgreSQL nicht ordnungsgemäß installiert worden. Entweder wurde es überhaupt nicht installiert oder der Suchpfad wurde nicht richtig gesetzt. Versuchen Sie, den Befehl mit einem absoluten Pfad aufzurufen:

```
$ /usr/local/pgsql/bin/createdb meinedb
```

Der Pfad mag bei Ihnen ein anderer sein. Setzen Sie sich mit dem Systemadministrator in Verbindung oder prüfen Sie noch einmal die Installationsanweisungen, um die Situation zu bereinigen.

Eine andere Antwort könnte diese sein:

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
createdb: database creation failed
```

Das bedeutet, dass der Server nicht gestartet wurde oder dass er nicht dort gestartet wurde, wo `createdb` ihn erwartete. Auch in diesem Fall gilt, schauen Sie nochmal in den Installationsanweisungen nach oder fragen Sie den Systemadministrator.

Wenn Sie die Rechte nicht haben, die zum Erzeugen einer Datenbank erforderlich sind, dann werden Sie Folgendes sehen:

```
ERROR:  CREATE DATABASE: permission denied
createdb: database creation failed
```

Nicht jeder Benutzer hat die Autorisierung, neue Datenbanken zu erzeugen. Wenn PostgreSQL sich weigert, für Sie neue Datenbanken zu erzeugen, dann muss Ihnen Ihr Systemadministrator die Erlaubnis erteilen, Datenbanken erzeugen zu dürfen. Wenden Sie sich an den Systemadministrator, falls das passiert. Wenn Sie PostgreSQL selbst installiert haben, sollten Sie sich für dieses Tutorial als der Benutzer anmelden, als der Sie den Server gestartet haben. [1]

Sie können auch Datenbanken mit anderen Namen erzeugen. PostgreSQL erlaubt die Erzeugung von beliebig vielen Datenbanken in einer Installation. Die Namen von Datenbanken müssen einen Buchstaben als erstes Zeichen haben und sind auf maximal 63 Zeichen Länge begrenzt. Eine günstige Wahl ist eine Datenbank mit dem gleichen Namen wie Ihr Benutzername. Viele Programme nehmen diesen Namen als Vorgabe an und können Ihnen somit etwas Tipparbeit ersparen. Um eine solche Datenbank zu erzeugen, geben Sie einfach ein:

```
$ createdb
```

Wenn Sie Ihre Datenbank nicht mehr verwenden wollen, können Sie sie entfernen. Wenn Sie zum Beispiel der Eigentümer (Erzeuger) der Datenbank `meinedb` sind, dann können Sie sie mit dem folgenden Befehl zerstören:

```
$ dropdb meinedb
```

(Bei diesem Befehl ist der Vorgabewert für den Datenbanknamen nicht der Benutzername. Der Datenbankname muss immer angegeben werden.) Diese Aktion entfernt alle Dateien, die zu dieser Datenbank gehören, und kann nicht rückgängig gemacht werden. Er sollte daher nur nach sorgfältiger Überlegung angPostgreSQLewendet werden.

1. Eine Erklärung warum das funktioniert: PostgreSQL-Benutzernamen sind getrennt von den Benutzernamen des Betriebssystems. Wenn Sie zu einer Datenbank verbinden, können Sie bestimmen, als welcher PostgreSQL-Benutzer Sie verbinden wollen; wenn Sie das nicht tun, dann wird der Benutzername des Betriebssystems als Vorgabe verwendet. Passenderweise gibt es immer einen PostgreSQL-Benutzer der den gleichen Namen hat wie der Betriebssystembenutzer der den Server gestartet hat, und passenderweise hat dieser Benutzer auch immer das Recht, Datenbanken zu erzeugen. Anstelle sich als dieser Benutzer anzumelden, können Sie auch überall die Option `-U` verwenden, um einen PostgreSQL-Benutzer zu bestimmen, als der Sie verbinden wollen.

1.4 Auf eine Datenbank zugreifen

Wenn Sie eine Datenbank erstellt haben, können Sie darauf zugreifen, indem Sie

- das interaktive PostgreSQL-Terminalprogramm, genannt *psql*, verwenden, welches Ihnen erlaubt, SQL-Befehle interaktiv einzugeben, zu bearbeiten und auszuführen,
- eine bestehende grafische Anwendung wie `PgAccess` oder ein Office-Paket mit ODBC-Unterstützung verwenden, um Datenbanken zu erstellen und zu verändern. Diese Möglichkeiten werden in diesem Tutorial nicht besprochen,
- eine neue Anwendung schreiben unter Anwendung der verschiedenen verfügbaren Sprachanbindungen. Diese Möglichkeiten werden ausführlicher in Teil IV besprochen.

Sie möchten wahrscheinlich `psql` starten, um die Beispiele in diesem Tutorial auszuprobieren. Um es für die Datenbank `meinedb` zu aktivieren, geben Sie diesen Befehl ein:

```
$ psql meinedb
```

Wenn Sie den Datenbanknamen auslassen, wird Ihr Benutzername als Vorgabe verwendet. Dieses System wurde im vorigen Abschnitt besprochen.

Das `psql`-Programm begrüßt Sie mit der folgenden Mitteilung:

```
Welcome to psql , the PostgreSQL interactive terminal.

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

meinedb=>
```

Die letzte Zeile könnte auch

```
meinedb=#
```

sein. Das würde bedeuten, dass Sie ein Datenbank-Superuser sind, was wahrscheinlich der Fall ist, wenn Sie PostgreSQL selbst installiert haben. Wenn Sie ein Superuser sind, dann umgehen Sie alle Kontrollen der Zugriffsrechte. Für dieses Tutorial ist das nicht erheblich.

Wenn Sie Probleme damit haben, `psql` zu starten, gehen Sie zurück zum vorigen Abschnitt. Die Mitteilungen von `psql` und `createdb` sind ähnlich, und wenn Ersteres funktioniert hat, sollte Letzteres das auch tun.

Die letzte Zeile, die von `psql` ausgegeben wurde, ist der Prompt, und der zeigt an, dass `psql` darauf wartet, dass Sie SQL-Befehle in den Arbeitsbereich von `psql` eingeben. Probieren Sie diese Befehle aus:

```
meinedb=> SELECT version();
                           version
-----------------------------------------------------------------
 PostgreSQL 7.3devel on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)
```

```
meinedb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

meinedb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

Das `psql`-Programm hat eine Anzahl von internen Befehlen, die keine SQL-Befehle sind. Sie fangen mit dem Backslash-Zeichen (»\«) an. Einige dieser Befehle wurden in der Willkommensmeldung aufgelistet. Zum Beispiel können Sie Hilfe zur Syntax der verschiedenen SQL-Befehle in PostgreSQL erhalten, indem Sie eingeben:

```
meinedb=> \h
```

Um `psql` zu verlassen, geben Sie ein:

```
meinedb=> \q
```

und `psql` wird beendet und Sie kehren zu Ihrer Shell zurück. (Um weitere interne Kommandos zu sehen, geben Sie am `psql`-Prompt `\?` ein.) Die gesamten Fähigkeiten von `psql` sind in Teil VI beschrieben. Wenn PostgreSQL richtig installiert ist, können Sie auch am Shell-Prompt des Betriebssystems `man psql` eingeben um die Anleitung zu sehen. In diesem Tutorial werden wir diese Fähigkeiten nicht ausdrücklich nutzen, aber Sie können Sie anwenden, wenn Sie es passend finden.

2

Die SQL-Sprache

2.1 Einführung

Dieses Kapitel gibt Ihnen einen Überblick, wie Sie SQL verwenden, um einfache Operationen auszuführen. Dieses Tutorial ist nur eine Einführung und in keiner Weise ein kompletter SQL-Lehrgang. Sie sollten sich auch darüber im Klaren sein, dass manche PostgreSQL-Features Erweiterungen gegenüber dem SQL-Standard sind.

In den folgenden Beispielen nehmen wir an, dass Sie eine Datenbank namens `meinedb` erzeugt haben, wie im vorigen Kapitel beschrieben, und dass sie `psql` gestartet haben.

Die Bespiele in diesem Tutorial können Sie auch im PostgreSQL-Quelltextpaket im Verzeichnis `src/tutorial/` finden. Die Datei `README` in diesem Verzeichnis enthält weitere Anweisungen, wie die Dateien zu verwenden sind. Um das Tutorial zu starten, geben Sie Folgendes ein:

```
$ cd ..../src/tutorial
$ psql -s meinedb
...

meinedb=> \i basics.sql
```

Der Befehl `\i` liest Befehle von der angegebenen Datei. Die Option `-s` schaltet den Einzelschrittmodus ein, welcher jedes Mal eine Pause verursacht, bevor ein Befehl an den Server geschickt wird. Die Befehle, die in diesem Kapitel verwendet werden, befinden sich in der Datei `basics.sql`.

2.2 Konzepte

PostgreSQL ist ein *relationales Datenbankverwaltungssystem* (englisch *relational database management system*, RDBMS). Das bedeutet, dass es ein System ist, welches Daten verwaltet, die in *Relationen* gespeichert sind. Relation ist im Prinzip ein mathematischer Begriff für eine *Tabelle*. Die Idee, Daten in Tabellen zu speichern, ist heutzutage so gewöhnlich, dass sie eigentlich offensichtlich erscheint, aber es gibt eine Anzahl anderer Möglichkeiten, Datenbanken zu organisieren. Dateien und Verzeichnisse in Unix-ähnlichen Betriebssystemen bilden ein Beispiel einer hierarchischen Datenbank. Eine modernere Entwicklung ist die objektorientierte Datenbank.

Jede Tabelle ist eine mit einem Namen versehene Sammlung von *Zeilen*. Jede Zeile einer Tabelle hat dieselbe Menge von benannten *Spalten* und jede Spalte hat einen bestimmten Datentyp. Während Spalten in jeder Zeile eine feste Ordnung haben, sollten Sie sich merken, dass SQL die Reihenfolge der Zeilen in einer Tabelle in keiner Weise garantiert. (Sie können aber für die Ausgabe ausdrücklich sortiert werden.)

Tabellen werden in Datenbanken zusammengefasst und eine Sammlung von Datenbanken, die von einem einzigen PostgreSQL-Server verwaltet werden, ergeben einen *Datenbank-Cluster* (*cluster*, englisch für Sammlung oder Traube).

2.3 Eine neue Tabelle erzeugen

Sie können eine neue Tabelle erzeugen, indem Sie den Namen der Tabelle sowie die Namen und Datentypen aller Spalten angeben:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- Tiefsttemperatur
    temp_hi         int,           -- Höchsttemperatur
    prcp            real,          -- Niederschlag (precipitation)
    date            date
);
```

Sie können das in `psql` mit den Zeilenumbrüchen eingeben. `psql` erkennt, dass die Befehle erst mit dem Semikolon abgeschlossen sind.

Leerzeichen, Tabs und Zeilenumbrüche können in SQL-Befehlen beliebig verwendet werden. Das bedeutet, dass sie den Befehl auch mit einer anderen Formatierung oder gar auf einer einzigen Zeile eingeben könnten. Zwei Minuszeichen (“`--`”) zeigen den Start eines Kommentars an. Alles was danach folgt, bis zum Zeilenende, wird ignoriert. Groß- und Kleinschreibung von Schlüsselwörtern und Namen ist in SQL unerheblich, es sei denn, Namen werden in doppelte Anführungszeichen gesetzt, um die Groß- und Kleinschreibung zu erhalten (oben nicht der Fall).

varchar(80) ist ein Datentyp, der beliebige Zeichenketten bis 80 Zeichen Länge aufnehmen kann. int ist der normale Typ für ganze Zahlen. real ist ein Typ, der Fließkommazahlen mit einfacher Präzision aufnimmt. date ist natürlich für Datumsangaben. (Die Spalte vom Typ date heißt selbst auch `date`. Das mag bequem oder verwirrend sein – Ihre Wahl.)

PostgreSQL unterstützt die gewöhnlichen SQL-Datentypen int, smallint, real, double precision, char(*N*), varchar(*N*), date, time, timestamp und interval sowie weitere allgemein anwendbare und eine große Sammlung von geometrischen Typen. PostgreSQL kann auch mit einer beliebigen Zahl benutzerdefinierter Typen angepasst werden. Deswegen sind Typnamen keine syntaktischen Schlüsselwörter, außer wenn das erforderlich is,t um Sonderfälle im SQL-Standard zu unterstützen.

Das zweite Beispiel wird Städte und ihre geographische Position speichern:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

Der Typ point ist ein Beispiel eines besonderen Typs in PostgreSQL.

Schließlich sollten wir erwähnen, dass, wenn Sie eine Tabelle nicht mehr benötigen oder Sie mit anderen Details neu erzeugen wollen, Sie sie dann mit dem folgenden Befehl entfernen können:

```
DROP TABLE tabellenname;
```

2.4 Eine Tabelle mit Zeilen füllen

Der Befehl `INSERT` wird verwendet, um eine Tabelle mit Zeilen zu füllen:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Alle Datentypen haben einigermaßen klare Eingabeformate. Für numerische Werte muss ein Punkt anstelle des Kommas verwendet werden. Konstanten, die keine einfachen numerischen Werte sind, müssen von einfachen Anführungszeichen (') eingeschlossen werden, wie im Beispiel. Der Typ date ist sehr flexibel bezüglich der Werte, die er akzeptiert, aber in diesem Tutorial werden wir das hier gezeigte unzweideutige Format verwenden.

Der Typ point erfordert als Eingabewert ein Koordinatenpaar, wie hier:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

Das bisher verwendete Format des `INSERT`-Befehls erfordert, dass man sich die Reihenfolge der Spalten gemerkt hat. Ein alternatives Format ermöglicht, dass man die Spalten ausdrücklich aufzählt:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Sie können die Spalten auch in einer anderen Reihenfolge aufzählen oder sogar einige Spalten weglassen, wenn zum Beispiel der Niederschlag unbekannt ist:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Viele Entwickler finden, dass es stilistisch besser ist, die Spalten immer aufzuzählen, anstatt sich auf die Ordnung zu verlassen.

Bitte geben Sie alle oben gezeigten Befehle ein, damit Sie einige Daten haben, mit denen wir in den folgenden Abschnitten arbeiten können.

Sie könnten auch `COPY` verwenden, um große Mengen Daten von einfachen Textdateien zu laden. Das ist normalerweise schneller, weil der `COPY`-Befehl für diese Anwendung optimiert ist, aber weniger Flexibilität als `INSERT` erlaubt. Ein Beispiel wäre:

```
COPY weather FROM '/home/user/weather.txt';
```

Der Dateiname der Quelldatei muss auf der Maschine des Datenbankservers zur Verfügung stehen, nicht auf der Clientmaschine, weil der Server die Datei direkt liest. Mehr über `COPY` können Sie auf Seite 646 lesen.

2.5 Eine Tabelle abfragen

Um Daten aus einer Tabelle zu lesen, wird eine **Anfrage** ausgeführt. Dazu wird der SQL-Befehl `SELECT` verwendet. Der Befehl teilt sich auf in eine Select-Liste (der Teil, der die auszugebenden Spalten auflistet), eine Tabellenliste (der Teil, der die Tabelle auflistet, von denen die Daten zu entnehmen sind) und eine wahlfreie Qualifizierung (der Teil, der etwaige Einschränkungen angibt). Um zum Beispiel alle Zeilen der Tabelle `weather` abzurufen, geben Sie ein:

```
SELECT * FROM weather;
```

(* bedeutet hier “alle Spalten”). Das Ergebnis sollte sein:

```
      city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

Sie können beliebige Ausdrücke in der Select-Liste angeben. Zum Beispiel können Sie Folgendes machen:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Das Ergebnis sollte sein:

```
      city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Die AS-Klausel wird verwendet, um der Ergebnisspalte einen Namen zu geben. (Sie ist optional.)

Beliebige Boole'sche Operationen (AND, OR, NOT) können in der Qualifikation einer Anfrage verwendet werden. Zum Beispiel ergibt die folgende Anfrage das Wetter in San Francisco an regnerischen Tagen:

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
    AND prcp > 0.0;
```

Ergebnis:

```
      city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Schließlich können Sie auch verlangen, dass die Ergebnisse einer Anfrage sortiert und doppelte Zeilen entfernt werden:

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

DISTINCT und ORDER BY können natürlich auch getrennt verwendet werden.

2.6 Verbunde zwischen Tabellen

Bisher haben unsere Anfragen immer nur auf eine Tabelle zugegriffen. Anfragen können auch auf mehrere Tabellen auf einmal zugreifen oder können auf eine Tabelle so zugreifen, dass mehrere Zeilen der Tabelle auf einmal verarbeitet werden. Eine Anfrage, die auf mehrere Zeilen derselben oder verschiedener Tabellen zugreift heißt, *Verbund*-Anfrage (englisch *join*). Sagen wir zum Beispiel, Sie wollen alle Wetterdatensätze zusammen mit den Koordinaten der zugehörigen Stadt sehen. Dazu müssen wir die `city`-Spalte in jeder Zeile der Tabelle `weather` mit der `name`-Spalte in allen Zeilen der Tabelle `cities` vergleichen und diejenigen Paare auswählen, bei denen die Werte gleich sind.

Anmerkung

Das ist nur das konzeptionelle Modell. In Wirklichkeit können viele Verbundoperationen effektiver ausgeführt werden, aber das ist für den Benutzer nicht sichtbar.

Dies können wir mit der folgenden Anfrage erreichen:

```
SELECT *
    FROM weather, cities
    WHERE city = name;
     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+---------
--
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Sie werden zwei Dinge in diesem Ergebnis bemerken:

- Es gibt keine Ergebniszeile für die Stadt Hayward. Der Grund ist, dass es für Hayward keinen passenden Eintrag in der Tabelle `cities` gibt und der Verbund die Zeilen in `weather` ignoriert, für die es keine Paarung gibt. Wir werden gleich sehen, wie das verbessert werden kann.
- Es gibt zwei Spalten mit dem Namen der Stadt. Das ist korrekt, weil die Spaltenlisten von `weather` und `cities` aneinandergehängt werden. In der Praxis ist das unerwünscht, sodass Sie die Spalten wahrscheinlich ausdrücklich aufzählen wollen, anstatt * zu verwenden.

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

Da alle Spalten unterschiedliche Namen hatten, hat der Parser automatisch herausgefunden, zu welcher Tabelle sie gehörten, aber es ist stilistisch besser, in Verbundanfragen die Spaltennamen um die Tabellennamen zu ergänzen:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

Die Verbundanfragen, die wir bisher gesehen haben, können auch in dieser Alternativform geschrieben werden:

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

Diese Form ist nicht so gebräuchlich wie die weiter oben verwendeten, aber wir zeigen sie hier, um Ihnen zu helfen, die folgenden Themen zu verstehen.

Jetzt werden wir herausfinden, wie wir die Datensätze der Stadt Hayward wieder sehen können. Was wir wollen, ist eine Anfrage, die die Tabelle `weather` durchgeht und für jede Zeile die passende `cities`-Zeile findet. Wenn keine passende Zeile gefunden wurde, sollten irgendwelche "leeren Werte" als Ersatz genommen werden. Diese Art Anfrage heißt **äußerer Verbund** (englisch *outer join*). (Die Verbunde, die wir bisher gesehen haben, heißen innerer Verbund (englisch *inner join*.) Der Befehl sieht so aus:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

Diese Anfrage heißt **linker äußerer Verbund** (*left outer join*), weil die Tabelle, die links vom Verbundoperator steht, jede ihrer Zeilen mindestens einmal in der Ergebnismenge haben wird, während die Tabelle auf der rechten Seite nur die Zeilen im Ergebnis haben wird, die zu einer Zeile der linken Tabelle passen. Wenn eine Zeile der linken Tabelle ausgegeben wird, für die es keine Entsprechung in der rechten Tabelle gibt, dann werden leere Werte (NULL) in die Spalten der rechten Tabelle eingesetzt.

Übung

Es gibt auch rechte äußere Verbunde (*right outer join*) und volle äußere Verbunde (*full outer join*). Versuchen Sie herauszufinden, was diese machen.

Wir können auch einen Verbund einer Tabelle mit sich selbst ausführen. Das ist ein **Selbstverbund**. Zum Beispiel nehmen wir an, dass wir alle Wetterdatensätze finden wollen, die im Temperaturbereich anderer Sätze sind. Also müssen wir die Spalten `temp_lo` und `temp_hi` von jeder Zeile in der Tabelle `weather` mit den `temp_lo`- und `temp_hi`-Spalten aller anderen `weather`-Zeilen vergleichen. Wir können das mit der folgenden Anfrage erreichen:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Wir haben die Tabelle `weather` hier als `W1` und `W2` bezeichnet, um die linke und rechte Seite des Verbunds unterscheiden zu können. Diese Art von Alias können Sie auch in anderen Anfragen verwenden, um Tipparbeit zu sparen. Zum Beispiel:

```
SELECT * FROM weather w, cities c WHERE w.city = c.name;
```

Diesen Abkürzungsstil werden Sie häufig antreffen.

2.7 Aggregatfunktionen

Wie die meisten anderen relationalen Datenbanken, unterstützt PostgreSQL Aggregatfunktionen. Eine Aggregatfunktion berechnet ein einzelnes Ergebnis aus mehreren Eingabezeilen. Zum Beispiel gibt es Aggregatfunktionen, die die Anzahl der Zeilen (`count`), die Summe der Eingabewerte (`sum`), den Durchschnitt (`avg`), die Minimal- (`min`) oder den Maximalwert (`max`) jeweils aus mehreren Zeilen berechnen.

Wir können zum Beispiel die höchste Tiefsttemperatur in unseren Daten mit folgender Anfrage ermitteln:

```
SELECT max(temp_lo) FROM weather;
 max
-----
  46
(1 row)
```

Wenn wir wissen wollen, in welcher Stadt (oder in welchen Städten) dieser Wert auftrat, würden wir vielleicht dies versuchen:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);       FALSCH
```

Aber das funktioniert nicht, weil die Aggregatfunktion `max` nicht in der `WHERE`-Klausel verwendet werden kann. (Die Einschränkung besteht, weil die `WHERE`-Klausel bestimmt, welche Zeilen in die Aggregatphase einfließen und damit ausgewertet werden muss, bevor die Aggregatfunktionen berechnet werden.) Wie es so oft der Fall ist, kann diese Anfrage aber umgeschrieben werden, um das beabsichtigte Ergebnis zu erzielen, hier mit einer **Unteranfrage**:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
     city
---------------
 San Francisco
(1 row)
```

Das ist in Ordnung, weil die Unteranfrage eine unabhängige Berechnung darstellt, die ihre Aggregatfunktionen getrennt von der äußeren Anfrage auswertet.

Aggregatfunktionen sind auch sehr nützlich in Kombination mit den `GROUP BY`- und `HAVING`-Klauseln. Wir können zum Beispiel die maximale Tiefsttemperatur in jeder Stadt ermitteln mit:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
     city      | max
```

```
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

Dies gibt uns eine Ergebniszeile pro Stadt. Jedes Aggregatergebnis wird aus den Tabellenzeilen für diese Stadt berechnet. Wir können diese gruppierten Zeilen mit `HAVING` filtern:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
  city   | max
---------+-----
 Hayward |  37
(1 row)
```

Das gibt uns das gleiche Ergebnis, aber nur mit Städten, die alle `temp_lo`-Werte unter 40 haben. Wenn wir schließlich nur an Städten interessiert sind, deren Namen mit “S” beginnt, könnten wir Folgendes machen:

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'(1)
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

Die LIKE-Operation führt einen Mustervergleich aus und wird in Abschnitt 9.6 erklärt.

Es ist wichtig, die Wechselwirkung zwischen Aggregatfunktionen und den SQL-Klauseln `WHERE` und `HAVING` zu verstehen. Der grundlegende Unterschied zwischen `WHERE` und `HAVING` ist folgender: `WHERE` wählt Zeilen aus, bevor Gruppen und Aggregate berechnet werden (es kontrolliert also, welche Zeilen in die Aggregatberechnung einfließen), während `HAVING` gruppierte Zeilen auswählt, nachdem Gruppen und Aggregate berechnet worden sind. Die `WHERE`-Klausel darf also keine Aggregatfunktionen enthalten, weil es keinen Sinn hat, eine Aggregatfunktion zu verwenden, um zu bestimmen, welchen Zeilen Eingabe für die Aggregatfunktionen sein werden. Die `HAVING`-Klausel hingegen enthält immer Aggregatfunktionen. (Genau genommen können Sie `HAVING`-Klauseln schreiben, die keine Aggregatfunktionen enthalten, aber das ist Verschwendung: Dieselben Bedingungen könnten in der `WHERE`-Klausel-Phase effizienter verarbeitet werden.)

Beachten Sie, dass wir die Einschränkung des Städtenamens in der `WHERE`-Klausel anwenden können, weil sie keine Aggregatfunktion braucht. Das ist effektiver, als die Bedingung zur `HAVING`-Klausel hinzuzufügen, weil wir dadurch die Gruppierung und Aggregatberechnungen der Zeilen, die durch die `WHERE`-Klausel herausfiltert werden, vermeiden.

2.8 Daten aktualisieren

Bestehende Zeilen können mit dem Befehl `UPDATE` aktualisiert werden. Nehmen wir an, Sie entdecken, dass alle Temperaturmessungen seit sdem 28. November zwei Grad zu hoch waren. Dann können Sie Ihre Daten folgendermaßen aktualisieren:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Schauen Sie sich den neuen Stand der Daten an:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9 Daten löschen

Nehmen wir an, dass Sie das Wetter in Hayward nicht mehr interessiert. Dann können Sie diese Zeilen wie folgt aus der Tabelle löschen. Löschoperationen werden mit dem Befehl `DELETE` durchgeführt:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Alle Datensätze in der Tabelle `weather`, die zu Hayward gehören, werden entfernt.

```
SELECT * FROM weather;
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Man sollte vorsichtig sein mit Befehlen der Form

```
DELETE FROM tabellenname;
```

Wenn keine Bedingungen angegeben sind, löscht `DELETE` *alle* Datensätze der angegebenen Tabelle und lässt die Tabelle damit leer. Das System verlangt keine Bestätigung, bevor es dies macht!

3

Fortgeschrittene Funktionen

3.1 Einführung

Im vorigen Kapitel ging es um die Grundlagen von SQL zum Speichern und Abrufen von Daten in PostgreSQL Jetzt werden wir einige fortgeschrittene Funktionen von PostgreSQL besprechen, die dazu dienen, die Verwaltung der Daten zu vereinfachen oder den Verlust oder die Verfälschung von Daten zu verhindern. Am Ende werden wir einige PostgreSQL-Erweiterungen anschauen.

Dieses Kapitel wird sich gelegentlich auf die Beispiele in Kapitel 2 beziehen, um sie zu ändern oder zu verbessern. Es ist also von Vorteil, wenn Sie dieses Kapitel gelesen haben. Einige Beispiele in diesem Kapitel können Sie auch in der Datei `advanced.sql` im Tutorial-Verzeichnis finden. Diese Datei enthält auch einige Beispieldaten zum Laden, die hier nicht wiederholt werden. (Sehen Sie unter Abschnitt 2.1 nach, wie Sie diese Datei verwenden können.)

3.2 Sichten

Schauen Sie sich noch einmal die Anfragen in Abschnitt 2.6 an. Nehmen wir an, dass die kombinierte Liste aus Wetterdatensätzen und Städtekoordinaten für Ihre Anwendung besonders interessant ist, aber Sie die Anfrage nicht jedes Mal eingeben wollen. Sie können eine **Sicht** (englisch *View*) für die Anfrage erzeugen, wodurch Sie der Anfrage einen Namen geben, den Sie wie eine normale Tabelle verwenden können.

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Die großzügige Verwendung von Sichten ist ein wichtiger Aspekt eines guten SQL-Datenbankentwurfs. Mit Sichten können die Strukturdetails Ihrer Tabellen, welche sich verändern könnten, während Ihre Anwendung wächst, hinter konsistenten Schnittstellen verbergen.

Sichten können fast überall verwendet werden, wo eine richtige Tabelle verwendet werden kann. Auch das Erzeugen von Sichten basierend auf anderen Sichten ist nicht unüblich.

3.3 Fremdschlüssel

Erinnern Sie sich an die Tabellen `weather` und `cities` aus Kapitel 2. Betrachten Sie folgendes Problem: Sie wollen sich versichern, dass niemand eine Zeile in die Tabelle `weather` einfügen kann, die keinen passenden Eintrag in der Tabelle `cities` hat. Dies bezeichnet man als das Erhalten der **referenziellen Integrität** der Daten. In einfachen Datenbanksystemen würde man das, wenn überhaupt, implementieren, indem man in die Tabelle `cities` schaut, um zu sehen, ob ein passender Datensatz existiert, und danach den neuen `weather`-Datensatz einfügt oder ablehnt. Diese Lösung hat eine Reihe von Problemen und ist ziemlich unbequem, aber PostgreSQL kann das für Sie übernehmen.

Die neue Deklaration der Tabellen sieht so aus:

```
CREATE TABLE cities (
   city     varchar(80) primary key,
    location point
);

CREATE TABLE weather (
   city      varchar(80) references cities,
   temp_lo   int,
   temp_hi   int,
   prcp      real,
   date      date
);
```

Nun versuchen Sie, einen ungültigen Datensatz einzufügen:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  <unnamed> referential integrity violation - key referenced from weather
not found in cities
```

Das Verhalten von Fremdschlüsseln kann genau auf Ihre Anwendung abgestimmt werden. Wir werden es in diesem Tutorial bei diesem einfachen Beispiel belassen und verweisen Sie auf Kapitel 5 für weitere Informationen. Die korrekte Verwendung von Fremdschlüsseln wird die Qualität Ihrer Datenbankanwendungen ganz bestimmt verbessern, und daher sollten Sie sich unbedingt mit ihnen vertraut machen.

3.4 Transaktionen

Transaktionen sind ein grundlegendes Konzept aller Datenbanksysteme. Die wichtigste Eigenschaft einer Transaktion ist, dass sie mehrere Schritte zu einer einzigen Alles-oder-nichts-Operation zusammenfasst. Die Stadien zwischen den Schritten sind für andere, gleichzeitig ausgeführte Transaktionen nicht sichtbar, und falls irgendein Fehler passiert, der verhindert, dass die Transaktion beendet wird, dann haben die Schritte der Transaktion überhaupt keine Auswirkung auf die Datenbank.

Betrachten Sie zum Beispiel die Datenbank einer Bank, die die Kontostände verschiedener Kundenkonten enthält, sowie die Summen der Kontostände in den Zweigstellen. Nehmen wir an, wir möchten eine Überweisung von 100 Euro von Annes auf Bernds Konto tätigen. Stark vereinfacht könnten die SQL-Befehle dafür etwa so aussehen:

```
UPDATE konten SET kontostand = kontostand - 100.00
    WHERE name = 'Anne';
UPDATE zweigstellen SET kontostand = kontostand - 100.00
    WHERE name = (SELECT zweig_name FROM konten WHERE name = 'Anne');
UPDATE konten SET kontostand = kontostand + 100.00
    WHERE name = 'Bernd';
UPDATE zweigstellen SET kontostand = kontostand + 100.00
    WHERE name = (SELECT zweig_name FROM konten WHERE name = 'Bernd');
```

Die Einzelheiten dieser Befehle sind hier nicht wichtig; das Wichtige ist, dass mehrere getrennte Aktualisierungen benötigt werden, um diese relativ einfache Operation auszuführen. Wir wollen sicher sein, dass entweder alle diese Aktualisierungen oder keine von ihnen stattfinden. Es wäre sicherlich nicht akzeptabel, wenn nach einem Systemausfall Bernd 100 Euro hätte, die von Annes Konto nicht abzogen wurden. Und Anne wäre sicherlich nicht lange ein zufriedener Kunde, wenn das Geld von ihrem Konto abgezogen würde, ohne dass Bernd es erhalten hätte. Wir brauchen eine Garantie, dass, wenn etwas mitten in der Operation fehlschlägt, die bis dahin ausgeführten Schritte nicht ausgeführt werden. Wenn wir die Aktualisierungen in eine **Transaktion** zusammenfassen, dann haben wir diese Garantie. Transaktionen sind **atomar**: Vom Standpunkt anderer Transaktionen aus gesehen, passiert entweder die ganze Transaktion oder gar nichts.

Außerdem möchten wir die Garantie haben, dass, wenn die Transaktion einmal abgeschlossen und vom Datenbanksystem bestätigt wurde, sie dann wirklich aufgezeichnet wurde und nicht durch einen Systemausfall kurze Zeit später verloren gehen kann. Wenn zum Beispiel Bernd Bargeld von seinem Konto abhebt, wollen wir sicherstellen, dass die Belastung seines Kontos nicht in einem Systemausfall verloren geht, während er gerade die Bank verlässt. Eine Datenbank, die Transaktionen unterstützt, garantiert dass alle Aktualisierungen auf eine dauerhaftes Speichermedium (z.B. Festplatte) geschrieben worden sind, bevor die Transaktion als abgeschlossen bestätigt wird.

Eine weitere wichtige Eigenschaft einer transaktionsfähigen Datenbank ist eng mit der Idee der atomaren Akualisierungen verwandt: Wenn mehrere Transaktionen gleichzeitig ablaufen, sollte es ihnen nicht möglich sein, die unvollständigen Änderungen der anderen zu sehen. Wenn zum Beispiel eine Transaktion damit beschäftigt ist, die Kontostände aller Zweigstellen zusammenzuzählen, dann wäre es nicht akzeptabel, wenn sie die Belastung von Annes Konto mitzählt, aber nicht die Gutschrift auf Bernds Konto, oder umgekehrt. Transaktionen müssen also nicht nur in Bezug auf ihre dauerhafte Auswirkung auf die Datenbank alles-oder-nichts sein, sondern auch in Bezug auf ihre Sichtbarkeit, während sie ablaufen. Die Aktualisierungen, die von einer noch nicht abgeschlossenen Transaktion gemacht wurden, sind für andere Transaktionen unsichtbar, bis erstere abgeschlossen wird, und zu diesem Zeitpunkt werden alle Aktualisierungen gleichzeitig sichtbar.

In PostgreSQL verwendet man eine Transaktion, indem man die SQL-Befehle der Transaktion mit den Befehlen `BEGIN` und `COMMIT` umschließt. Unsere Banktransaktion würde also etwa so aussehen:

```
BEGIN;
UPDATE konten SET kontostand = kontostand - 100.00
    WHERE name = 'Anne';
-- usw. usw.
COMMIT;
```

Wenn wir mitten in der Transaktion entscheiden, dass wir die Transaktion nicht abschließen wollen (vielleicht haben wir festgestellt, dass das Konto von Anne jetzt im Minus ist), dann können wir den Befehl `ROLLBACK` statt `COMMIT` ausführen und alle unsere Aktualisierungen werden verworfen.

PostgreSQL führt in Wirklichkeit jeden Befehl in einer Transaktion aus. Wenn Sie keinen `BEGIN`-Befehl ausführen, dann wird vor jedem Befehl ein automatisches `BEGIN` und, falls der Befehl erfolgreich war, danach ein automatisches `COMMIT` ausgeführt. Eine Gruppe von Befehlen zwischen einem ausdrücklichen `BEGIN` und `COMMIT` wird manchmal **Transaktionsblock** genannt.

Anmerkung

Manche Clientbibliotheken führen `BEGIN` und `COMMIT` automatisch aus und erzeugen daher möglicherweise den Effekt eines Transaktionsblocks, ohne dass Sie dazu aufgefordert werden müssen. Prüfen Sie die Anleitung der Schnittstelle, die Sie verwenden möchten.

3.5 Vererbung

Vererbung ist ein Konzept der objektorientierten Datenbanken. Es eröffnet interessante neue Möglichkeiten beim Entwerfen einer Datenbank.

Erzeugen wir zwei Tabellen: Eine Tabelle `cities` für Städte und eine Tabelle `capitals` für Hauptstädte. Natürlich sind Hauptstädte auch Städte, also möchten Sie vielleicht eine Möglichkeit haben, alle Hauptstädte automatisch anzuzeigen, wenn Sie alle Städte auflisten. Wenn Sie sehr raffiniert sind, würden Sie vielleicht ein System wie dieses erfinden:

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (in Fuß)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int     -- (in Fuß)
);

CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

Das funktioniert auch gut für Anfragen, aber es wird sehr umständlich, wenn Sie mehrere Zeilen aktualisieren wollen, um nur ein Problem zu nennen.

Hier ist eine bessere Lösung:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (in Fuß)
);
```

```
CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

In diesem System **erbt** (englisch *to inherit*) eine Zeile der Tabelle `capitals` alle Spalten (`name`, `population` und `altitude`) der **Elterntabelle** `cities`. Der Typ der Spalte `name` ist text, ein eingebauter PostgreSQL-Typ für Zeichenketten mit beliebiger Länge. Die Hauptstädte der Bundesstaaten haben eine zusätzliche Spalte, `state`, die anzeigt, zu welchem Bundesstaat die Hauptstadt gehört. In PostgreSQL- kann eine Tabelle von ein oder mehreren Tabellen erben, oder natürlich von gar keiner.

Die folgende Anfrage findet zum Beispiel die Namen aller Städte, einschließlich der Hauptstädte, die höher als 500 Fuß gelegen sind:

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

Das Ergebnis:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

Die folgende Anfrage andererseits findet alle Städte, die nicht Hauptstädte und 500 Fuß oder höher gelegen sind:

```
SELECT name, altitude FROM ONLY cities WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Das `ONLY` vor `cities` zeigt an, dass die Anfrage nur die `cities`-Tabelle lesen soll und nicht die Tabelle unterhalb von `cities` in der Vererbungshierarchie. Viele der bisher besprochenen Befehle – `SELECT`, `UPDATE` und `DELETE` – unterstützen diese `ONLY`-Option.

3.6 Schlussfolgerung

Die Einführung war für neue Benutzer bestimmt und PostgreSQL hat viele Features, die hier nicht besprochen wurden. Diese werden in allen Einzelheiten im Rest dieses Buchs besprochen.

Wenn Sie denken, dass Sie noch mehr Material zum Lernen benötigen, besuchen Sie bitte die PostgreSQL Website, wo Sie Links zu weiteren Quellen finden.

Teil II

Die SQL-Sprache

Dieser Teil beschreibt, wie die Sprache SQL in PostgreSQL verwendet wird. Zuerst erklären wir die allgemeine Syntax von SQL, dann behandeln wir, wie Datenstrukturen erzeugt werden, die Datenbank gefüllt wird und wie Anfragen geschrieben werden. Der Mittelteil listet die Datentypen und Funktionen, die in SQL-Datenbefehlen verwendet werden können. Der Rest des Teils behandelt verschiedene Aspekte, die wichtig sind, um eine Datenbank auf optimale Leistung abzustimmen.

Die Informationen in diesem Teil sind so angeordnet, dass ein neuer Anwender sie von vorne nach hinten durcharbeiten kann und dabei ein vollständiges Verständnis der Themen erhält, ohne zu oft nach vorne blättern zu müssen. Andererseits sind die Kapitel unabhängig voneinander, sodass fortgeschrittene Anwender die Kapitel einzeln auswählen und lesen können. In diesem Teil sind die Informationen erzählerisch in Themenbereichen aufbereitet. Leser, die eine vollständige Beschreibung eines bestimmten Befehls suchen, sollten in ***Teil VI*** nachschauen.

Leser dieses Teils sollten wissen, wie man mit einer PostgreSQL-Datenbank verbindet und SQL-Befehle ausführt. Leser, die damit noch nicht vertraut sind, sollten zuerst ***Teil I*** lesen. SQL-Befehlew werden normalerweise mit dem interaktiven PostgreSQL-Terminalprogramm psql eingegeben, aber andere Programme mit ähnlicher Funktionalität können ebenso verwendet werden.

4

SQL-Syntax

Dieses Kapitel beschreibt die Syntax der Sprache SQL. Es bildet die Grundlage, die zum Verstehen der folgenden Kapitel notwendig ist, welche beschreiben, wie man SQL-Befehle verwendet, um Daten zu definieren oder zu verändern.

Wir raten auch Benutzern, die sich mit SQL schon auskennen, dieses Kapitel sorgfältig zu lesen, denn mehrere Regeln und Konzepte sind in verschiedenen SQL-Datenbanken unterschiedlich umgesetzt oder sind nur in PostgreSQL vorhanden.

4.1 Lexikalische Struktur

Eine SQL-Eingabe besteht aus einer Reihe von **Befehlen**. Ein Befehl besteht aus einer Reihe von **Tokens** und wird mit einem Semikolon (»;«) abgeschlossen. Das Ende des Eingabestroms zeigt auch das Ende des Befehls an. Welche Tokens gültig sind, hängt von der Syntax des jeweiligen Befehls ab.

Ein Token kann ein **Schlüsselwort**, ein **Name**, ein Name in Anführungszeichen, eine `Konstante` oder ein Sonderzeichen sein. Tokens sind normalerweise durch Leerzeichen, Tabs oder Zeilenenden getrennt, aber das muss nicht sein, wenn dadurch keine Zweideutigkeit entsteht (was in der Regel nur der Fall ist, wenn ein Sonderzeichen neben einem anderen Tokentyp steht).

Zusätzlich können im SQL-Text **Kommentare** auftreten. Sie sind keine Tokens und werden gewissermaßen durch Leerzeichen ersetzt.

Folgender Text ist zum Beispiel (syntaktisch) gültiger SQL-Text:

```
SELECT * FROM MEINE_TABELLE;
UPDATE MEINE_TABELLE SET A = 5;
INSERT INTO MEINE_TABELLE VALUES (3, 'Hallo daheim!');
```

Das ist eine Reihe von drei Befehlen, einem pro Zeile (obwohl das nicht notwendig ist; eine Zeile kann mehrere Befehle enthalten und ein Befehl kann auch sinnvoll über mehrere Zeilen verteilt werden).

Welche Tokens Befehle anzeigen und welche Tokens Operanden oder Parameter sind, ist in der SQL-Syntax nicht klar geregelt. Die ersten paar Tokens stehen in der Regel für den Namen des Befehls, sodass man in dem Beispiel oben gewöhnlich von einem »SELECT«-, einem »UPDATE«- und einem »INSERT«-Befehl spricht. Aber der `UPDATE`-Befehl zum Beispiel erfordert immer auch ein SET-Token in einer bestimmten Position und diese bestimmte Variante von `INSERT` erfordert zur Vollständigkeit immer auch ein VALUES. Die genauen Syntaxregeln für jeden Befehl sind in Teil VI beschrieben.

4.1.1 Namen und Schlüsselwörter

Tokens wie SELECT, UPDATE oder VALUES im obigen Beispiel sind Beispiele für **Schlüsselwörter**, das heißt Wörter, die eine festgelegte Bedeutung in der SQL-Sprache haben. Die Tokens MEINE_TABELLE und A sind Beispiele von **Namen** oder **Bezeichnern**. Sie bezeichnen die Namen von Tabellen, Spalten oder anderen Datenbankobjekten, abhängig davon, in welchem Befehl sie verwendet werden. Schlüsselwörter und Namen haben die gleiche lexikalische Struktur; das heißt, dass man nur wissen kann, welches Token ein Name und welches ein Schlüsselwort ist, wenn man die Sprache kennt. Eine komplette Liste der Schlüsselwörter ist in Anhang B zu finden.

SQL-Namen und -Schlüsselwörter müssen mit einem Buchstaben (`a-z`, aber auch Umlaute und nichtlateinische Buchstaben) oder einem Unterstrich (_) beginnen. Die nachfolgenden Zeichen in einem Namen oder einem Schlüsselwort können Buchstaben, Ziffern (`0-9`) oder Unterstriche sein, obwohl der SQL-Standard kein Schlüsselwort definieren wird, das Ziffern enthält oder mit einem Unterstrich beginnt oder endet.

Das System verwendet nur maximal *NAMEDATALEN*-1 Zeichen eines Namens; längere Namen können verwendet werden, aber sie werden abgeschnitten. Die Vorgabe für *NAMEDATALEN* ist 64, also ist die Maximallänge eines Namens 63. (Wenn PostgreSQL gebaut wird, kann *NAMEDATALEN* in der Datei `src/include/postgres_ext.h` verändert werden.)

Die Groß- oder Kleinschreibung von Namen und Schlüsselwörtern ist unerheblich. Also kann

```
UPDATE MEINE_TABELLE SET A = 5;
```

genauso als

```
uPDaTE mEine_TabeLLE SeT a = 5;
```

geschrieben werden. Eine oft verwendete Konvention ist, dass man Schlüsselwörter groß und Namen klein schreibt, also zum Beispiel

```
UPDATE meine_tabelle SET a = 5;
```

Es gibt noch eine zweite Art von Namen: den Namen in Anführungszeichen. Er wird geschrieben, indem man eine beliebige Zeichenfolge in doppelte Anführungszeichen (") einschließt. Ein Name in Anführungszeichen ist immer ein Name und niemals ein Schlüsselwort. `"select"` könnte also verwendet werden, um auf eine Spalte oder Tabelle namens `select` zu verweisen, wohingegen ein `select` ohne Anführungszeichen als Schlüsselwort interpretiert würde und daher einen Parsefehler verursachen würde, wenn es dort verwendet wird, wo PostgreSQL einen Tabellen- oder Spaltennamen erwartet. Das obige Beispiel kann mit Namen in Anführungszeichen folgendermaßen geschrieben werden:

```
UPDATE "meine_tabelle" SET "a" = 5;
```

Namen in Anführungszeichen können jedes Zeichen enthalten außer dem Anführungszeichen selbst. Um ein Anführungszeichen in einen Namen zu schreiben, schreiben Sie es doppelt. Dadurch können Sie Tabellen- oder Spaltennamen verwenden, die ansonsten nicht möglich wären, zum Beispiel solche mit Leerzeichen oder Punkten. Die Längenbeschränkung gilt aber weiterhin.

Wenn man einen Namen in Anführungszeichen setzt, dann spielt auch die Groß- und Kleinschreibung eine Rolle, wohingegen Namen ohne Anführungszeichen in Kleinbuchstaben umgewandelt werden. So sind zum Beispiel `FOO`, `foo` und `"foo"` derselbe Name, aber `"Foo"` und `"FOO"` bezeichnen Namen, die sich von den ersten drei und voneinander unterscheiden. [1]

1. Die Umwandlung von Namen ohne Anführungszeichen in Kleinbuchstaben in PostgreSQL ist nicht mit dem SQL-Standard kompatibel, welcher aussagt, dass sie in Großbuchstaben umgewandelt werden sollten. Laut dem Standard sollte folglich `foo` gleich mit `"FOO"` sein und nicht mit `"foo"`. Wenn Sie portierbare Anwendungen schreiben wollen dann empfehlen wir, dass sie für jeden Namen entweder immer oder nie Anführungszeichen verwenden.

4.1.2 Konstanten

Es gibt drei Arten von Konstanten mit **implizitem Typ** in PostgreSQL: Zeichenketten, Bitketten und Zahlen. Konstanten können auch mit explizitem Typ angegeben werden, wodurch eine genauere und effizientere Verarbeitung im System erreicht werden kann. Die impliziten Konstanten sind unten beschrieben; die expliziten Konstanten folgen danach.

Zeichenkettenkonstanten

Eine Zeichenkette in SQL ist eine Reihe von beliebigen Zeichen, die von halben Anführungszeichen bzw. Apostrophen (»'«) eingeschlossen sind, zum Beispiel `'Das ist eine Zeichenkette'`. Ein Apostroph kann in eine Zeichenkette eingefügt werden, indem man ihn doppelt schreibt (z.B. `'Dianne''s horse'`). In PostgreSQL kann ein Apostroph auch in eine Zeichenkette eingefügt werden, indem man ihm einen Backslash (»\«) voranstellt (z.B. `'Dianne\'s horse'`).

Backslash-Escape-Folgen im Stile von C sind auch möglich: `\b` ist ein Backspace (vorangegangenes Zeichen löschen), `\f` ist ein Form-Feed, `\n` ist ein Zeilenumbruch, `\r` ist ein Wagenrücklauf (englisch *carriage return*), `\t` ist ein Tab und `\xxx`, wobei *xxx* eine Oktalzahl ist, ist das Zeichen mit dem entsprechenden ASCII-Code. Jedes andere Zeichen nach einem Backslash wird unverändert übernommen. Um also ein Backslash in eine Zeichenkettenkonstante zu schreiben, geben Sie zwei Backslashes hintereinander ein.

Das Zeichen mit dem Code null kann nicht in einer Zeichenkettenkonstante enthalten sein.

Zwei Zeichenkettenkonstanten, die nur durch Leerzeichen, Tabs und *mindestens einen Zeilenumbruch* getrennt sind, werden zusammengehängt und faktisch so behandelt, als ob sie als eine Konstante geschrieben worden wären. Zum Beispiel:

```
SELECT 'foo'
'bar';
```

ist gleichbedeutend mit:

```
SELECT 'foobar';
```

aber

```
SELECT 'foo'    'bar';
```

ist eine ungültige Syntax. (Dieses etwas wundersame Verhalten ist von SQL vorgeschrieben; PostgreSQL folgt dem Standard.)

Bitkettenkonstanten

Bitkettenkonstanten sehen aus wie Zeichenkettenkonstanten mit einem `B` (groß oder klein geschrieben) direkt vor dem Anführungszeichen (ohne Leerzeichen dazwischen), z.B. `B'1001'`. Die einzigen Zeichen, die in Bitkettenkonstanten erlaubt sind, sind `0` und `1`.

Als Alternative können Bitkettenkonstanten auch in hexadezimaler Schreibweise, mit einem `X` (groß oder klein) vor dem Anführungszeichen, geschrieben werden, z.B. `X'1FF'`. Diese Schreibweise ist gleichbedeutend mit einer Bitkettenkonstante mit vier Binärziffern für jede Hexadezimalziffer.

Beide Formen der Bitkettenkonstante können wie normale Zeichenkettenkonstanten über mehrere Zeilen verteilt werden.

Zahlenkonstanten

Zahlenkonstanten können in diesen allgemeinen Formen geschrieben werden:

```
ziffern
ziffern.[ziffern][e[+-]ziffern]
[ziffern].ziffern[e[+-]ziffern]
zifferne[+-]ziffern
```

wobei *ziffern* für eine oder mehrere Dezimalziffern (0 bis 9) steht. Mindestens eine Ziffer muss vor oder nach dem Punkt (der für das Komma steht) stehen, falls ein Punkt verwendet wird. Mindestens eine Ziffer muss nach der Exponentenmarkierung (e) stehen, falls eine verwendet wird. Leerzeichen oder andere Zeichen dürfen in einer Zahlenkonstante nicht vorkommen. Beachten Sie auch, dass ein Plus- oder Minuszeichen am Anfang nicht wirklich Teil der Konstante ist, sondern als vorgestellter Operator verarbeitet wird.

Hier sind einige Beispiele für gültige Zahlenkonstanten:

42
3.5
4.
.001
5e2
1.925e-3

Eine Zahlenkonstante ohne Punkt und Exponent wird anfänglich als Typ `integer` verarbeitet, wenn der Wert in den Typ `integer` passt (32 Bits), ansonsten wird der Typ `bigint` angenommen, wenn der Wert in den Typ `bigint` passt (64 Bits), ansonsten wird der Typ `numeric` verwendet. Konstanten mit Punkt oder Exponent werden anfänglich immer als Typ `numeric` verarbeitet.

Der anfänglich zugewiesene Datentyp einer Zahlenkonstante ist nur ein Ausgangspunkt für den Typauflösungsalgorithmus. In den meisten Fällen wird die Konstante automatisch je nach Zusammenhang in den passenden Typ umgewandelt. Wenn es erforderlich ist, dann können Sie einen Zahlenwert in einen bestimmten Datentyp mit einer expliziten Typumwandlung (englisch *Cast*) zwingen. Sie können zum Beispiel erzwingen, dass ein Zahlenwert als Typ real (float4) behandelt wird, indem Sie schreiben

```
REAL '1.23' -- Zeichenkettenstil
1.23::REAL  -- PostgreSQL-Stil (historisch)
```

Konstanten anderer Typen

Eine Konstante eines *beliebigen* Typs kann in jeder der folgenden Schreibweisen eingegeben werden:

```
typ 'zeichen'
'zeichen'::typ
CAST ( 'zeichen' AS typ )
```

Der Text der Zeichenkette wird der Eingabeumwandlungsroutine des benannten Typs übergeben. Das Ergebnis ist eine Konstante des angegebenen Typs. Die ausdrückliche Umwandlung kann ausgelassen werden, wenn es keine Zweideutigkeit bezüglich des Typs, den die Konstante haben muss, gibt (zum Beispiel weil sie als Argument einer nicht überladenen Funktion verwendet wird); in diesem Fall wird sie automatisch umgewandelt.

Eine Typumwandlung kann auch mit einer Syntax ähnlich eines Funktionsaufrufs durchgeführt werden:

```
typname ( 'zeichen' )
```

Aber nicht alle Typnamen können auf diese Art verwendet werden; siehe Abschnitt 4.2.6 wegen Einzelheiten.

Die Formen `::`, `CAST()` und die funktionsaufrufähnliche Form können auch verwendet werden, um den Typ eines beliebigen Ausdrucks anzugeben, wie in Abschnitt 4.2.6 besprochen. Aber die Form `typ`

'zeichen' kann nur verwendet werden, um den Typ einer Konstante zu bestimmen. Eine weitere Einschränkung bezüglich *typ 'zeichen'* ist, dass sie nicht für Arraytypen funktioniert; verwenden Sie :: oder CAST(), um den Typ einer Arraykonstante festzulegen.

Arraykonstanten

Das allgemeine Format einer Arraykonstante ist das folgende:

```
'{ wert1 trenn wert2 trenn ... }'
```

wobei *trenn* für das Trennzeichen des Typs steht, wie es in seinem pg_type-Eintrag gespeichert ist. (Für alle eingebauten Typen ist das das Komma.) Jedes *wert* ist entweder eine Konstante des Arrayelementtyps oder ein Subarray. Ein Beispiel einer Arraykonstante ist

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Diese Konstante ist ein zweidimensionales 3-mal-3-Array, das aus drei Subarrays mit Ganzzahlen besteht.

Die einzelnen Arrayelemente können zwischen Anführungszeichen (") gesetzt werden, um Unklarheiten über Leerzeichen zu vermeiden. Ohne Anführungszeichen überspringt der Arrayparser vorangestellte Leerzeichen.

(Arraykonstanten sind im Prinzip nur ein besonderer Fall der allgemeinen Typkonstanten, die im vorigen Abschnitt besprochen wurden. Die Konstante wird anfänglich als Textkonstante behandelt und wird dann an die Eingabeumwandlungsroutine für Arrays übergeben. Eine ausdrückliche Typumwandlung kann notwendig sein.)

4.1.3 Operatoren

Ein Operator ist eine Folge von bis zu *NAMEDATALEN*-1 (normalerweise 63) Zeichen aus der folgenden Liste:

```
+ - * / < > = ~ ! @ # % ^ & | ` ? $
```

Es gibt allerdings ein paar Einschränkungen für die Operatornamen:

- $ (Dollarzeichen) kann nicht als einzelnes Zeichen ein Operator sein. Es kann allerdings Teil eines Operatornamens aus mehreren Zeichen sein.
- -- und /* können nirgendwo in einem Operatornamen auftauchen, weil sie als der Anfang eines Kommentars angenommen werden.
- Ein Operatorname, der aus mehreren Zeichen besteht, kann nicht auf + oder – enden, es sei denn, der Name enthält mindestens eins der folgenden Zeichen:

```
~ ! @ # % ^ & | ` ? $
```

  So ist zum Beispiel @- ein zulässiger Operatorname, nicht aber *-. Diese Einschränkung erlaubt es PostgreSQL, Anfragen, die dem SQL-Standard folgen, zu verarbeiten, ohne Leerzeichen zwischen den Tokens zu erfordern.

Wenn Sie mit Operatornamen arbeiten, die nicht dem SQL-Standard folgen, müssen Sie in der Regel die Operatoren mit Leerzeichen voneinander trennen, um Unklarheiten zu vermeiden. Wenn Sie zum Beispiel einen linken Präfixoperator namens @ definiert haben, können Sie nicht X*@Y schreiben, sondern Sie müssen X* @Y schreiben, um sicherzustellen, dass PostgreSQL den Ausdruck als zwei Operatornamen liest und nicht nur als einen.

4.1.4 Sonderzeichen

Einige Zeichen, die nicht alphanumerisch sind, haben besondere Bedeutung, die aber nicht einem Operatornamen entspricht. Einzelheiten können Sie dort finden, wo das Syntaxelement beschrieben ist. Dieser Abschnitt dient nur dazu, Sie auf das Vorhandensein dieser Zeichen hinzuweisen und ihre Zwecke zusammenzufassen.

- Ein Dollarzeichen (`$`) gefolgt von Ziffern steht für die Positionsparameter im Körper einer Funktionsdefinition. In anderen Zusammenhängen kann das Dollarzeichen Teil eines Operatornamens sein.
- Klammern (`( )`) haben ihre gewöhnliche Bedeutung, Ausdrücke zu gruppieren und die Reihenfolge in der Auswertung der Ausdrücke zu erzwingen. In einigen Fällen sind Klammern fester Bestandteil der Syntax bestimmter SQL-Befehle.
- Eckige Klammern (`[]`) werden verwendet, um Elemente aus Arrays auszuwählen. Sehen Sie in Abschnitt 8.11 nach, um mehr Informationen über Arrays zu erhalten.
- Kommas (`,`) werden in einigen Zusammenhängen verwendet, um Elemente einer Liste zu trennen.
- Das Semikolon (`;`) schließt einen SQL-Befehl ab. Es kann sonst nirgendwo in einem SQL-Befehl auftreten, es sei denn, in einer Zeichenkettenkonstante oder einem Namen in Anführungszeichen.
- Der Doppelpunkt (`:`) wird verwendet, um aus Arrays »Stücke« auszuwählen. (Siehe Abschnitt 8.11.) In einigen SQL-Dialekten (wie zum Beispiel eingebettetes SQL (englisch *Embedded SQL*)), steht der Doppelpunkt vor Variablennamen.
- Der Stern (`*`) hat eine besondere Bedeutung, wenn er im `SELECT`-Befehl oder in der Aggregatfunktion `COUNT` verwendet wird.
- Der Punkt (`.`) wird in Zahlenkonstanten verwendet (anstelle des Kommas) und um Schema, Tabellen- und Spaltennamen zu trennen.

4.1.5 Kommentare

Ein Kommentar ist eine beliebige Reihe von Zeichen, die mit zwei Minuszeichen anfängt und bis zum Zeilenende geht. Zum Beispiel:

```
-- Das ist ein normaler SQL92-Kommentar.
```

Als Alternative können Blockkommentare im C-Stil verwendet werden:

```
/* mehrzeiliger Kommentar
 * mit Verschachtelung: /* verschachtelter Blockkommentar */
 */
```

Der Kommentar beginnt mit `/*` und geht bis zum passenden `*/`. Diese Blockkommentare können verschachtelt werden, wie in SQL99 vorgeschrieben, aber im Gegensatz zu C, sodass man größere Blöcke Code auskommentieren kann, selbst wenn diese selbst Blockkommentare enthalten.

Ein Kommentar wird vom Eingabestrom entfernt, bevor die Syntax weiter analysiert wird, und faktisch durch Leerzeichen ersetzt.

4.1.6 Vorrang

Tabelle 4.1 zeigt den Vorrang und die Anhänglichkeit der Operatoren in PostgreSQL. Die meisten Operatoren haben den gleichen Vorrang und sind links anhänglich. Der Vorrang und die Anhänglichkeit sind im Parser festgeschrieben. Das kann zu einem Verhalten führen, das nicht intuitiv ist; so haben zum Beispiel die Boole'schen Operatoren `<` und `>` einen anderen Vorrang als die Boole'schen Operatoren `<=` und `>=`.

Manchmal werden Sie auch Klammern hinzufügen müssen, wenn Sie Kombinationen von binären und unären Operatoren verwenden. Zum Beispiel:

```
SELECT 5 ! - 6;
```

wird verarbeitet als

```
SELECT 5 ! (- 6);
```

weil der Parser keine Ahnung hat – bevor es zu spät ist -, dass ! als Postfixoperator und nicht als Infixoperator definiert ist. Um das gewünschte Verhalten in diesem Fall zu erreichen, müssen Sie

```
SELECT (5 !) - 6;
```

schreiben. Diesen Preis zahlt man für die Erweiterbarkeit.

Operator/Element	Anhänglichkeit	Beschreibung
`.`	links	Trennung von Tabellen-/Spaltennamen
`::`	links	PostgreSQL -Stil Typumwandlung
`[ ]`	links	Auswahl Arrayelement
`-`	rechts	unäres Minus
`^`	links	Potenzierung
`* / %`	links	Multiplikation, Division, Modulus
`+ -`	links	Addition, Subtraktion
`IS`		`IS TRUE, IS FALSE, IS UNKNOWN, IS NULL`
`ISNULL`		Test für NULL
`NOTNULL`		Test für nicht NULLs
(alle anderen)	links	alle anderen eingebauten und benutzerdefinierten Operatoren
`IN`		Mengenmitgliedschaft
`BETWEEN`		Vergleich
`OVERLAPS`		Zeitintervall überlappt
`LIKE ILIKE SIMILAR`		Mustervergleich von Zeichenketten
`< >`		kleiner als, größer als
`=`	rechts	Gleichheit, Wertzuweisung
`NOT`	rechts	logische Negierung
`AND`	links	logische Konjunktion
`OR`	links	logische Disjunktion

Tabelle 4.1: Operatorvorrang (abnehmend)

Beachten Sie, dass die Operatorvorrangsregeln auch für benutzerdefinierte Operatoren gelten, die die gleichen Namen haben wie die eingebauten Operatoren. Wenn Sie zum Beispiel einen Operator »+« für einen eigenen Datentyp definieren, dann hat er den gleichen Vorrang wie der eingebaute »+«-Operator, ganz gleich, was Ihrer tut.

Wenn ein Operator mit Schemaname in der OPERATOR-Syntax verwendet wird, wie im Beispiel

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

dann hat die OPERATOR-Konstruktion den Standardvorrang, der in Tabelle 4.1 mit »alle anderen« bezeichnet ist. Das gilt unabhängig davon, welcher Operator nun genau innerhalb von OPERATOR steht.

4.2 Wertausdrücke

Wertausdrücke werden in verschiedenen Zusammenhängen verwendet, so zum Beispiel in der Zielliste eines SELECT-Befehls, als neue Spaltenwerte in INSERT- oder UPDATE-Befehlen oder als Suchbedingungen in einer Reihe von Befehlen. Das Ergebnis eines Wertausdrucks wird manchmal als **Skalar** bezeichnet, um es vom Ergebnis eines Tabellenausdrucks (welches eine Tabelle ist) zu unterscheiden. Wertausdrücke werden daher auch als **Skalarausdrücke** (oder einfach **Ausdrücke**) bezeichnet. Die Syntax für Ausdrücke erlaubt die Berechnung von Werten aus einfachen Bestandteilen mit arithmetischen, logischen, mengenarithmetischen und anderen Operationen.

Ein Wertausdruck ist eines der Folgenden:

- eine Konstante; siehe Abschnitt 4.1.2.
- ein Spaltenverweis
- ein Verweis auf einen Positionsparameter, im Körper einer Funktionsdefinition
- ein Operatoraufruf
- ein Funktionsaufruf
- ein Aggregatausdruck
- eine Typumwandlung
- eine skalare Unteranfrage
- ein Wertausdruck in Klammern, um Teilausdrücke zu gruppieren und den Vorrang zu verändern

Über diese Liste hinaus gibt es eine Reihe weiterer Konstruktionen, die als Ausdruck klassifiziert werden können, aber keiner allgemeinen Syntaxregel folgen. Diese ähneln in der Regel einer Funktion oder einem Operator und sind im entsprechenden Abschnitt in Kapitel 9 erläutert. Ein Beispiel ist die IS NULL-Klausel.

Konstanten haben wir schon in Abschnitt 4.1.2 besprochen. Die folgenden Abschnitte besprechen die übrigen Möglichkeiten.

4.2.1 Spaltenverweise

Auf eine Spalte kann man verweisen, indem man schreibt:

```
korrelation.spaltenname
```

oder

```
korrelation.spaltenname[index]
```

(Die eckigen Klammern [ ] sollen hier wortwörtlich erscheinen.)

korrelation ist der Name einer Tabelle (möglicherweise mit Schema) oder ein Alias, das in einer FROM-Klausel definiert wurde, oder die Schlüsselwörter NEW oder OLD. (NEW und OLD können nur in Umschreibungsregeln auftreten, während die anderen Korrelationsnamen in jedem SQL-Befehl verwendet werden

können.) Der Korrelationsname und der trennende Punkt können weggelassen werden, wenn der Spaltenname über alle Tabellen, die in der gegenwärtigen Anfrage verwendet werden, eindeutig ist. (Siehe auch Kapitel 7.)

Wenn *spalte* einen Arraytyp hat, dann kann zusätzlich *index* verwendet werden, um ein bestimmtes Element oder bestimmte Elemente aus dem Array auszuwählen. Wenn kein Index angegeben wird, dann wird das gesamte Array ausgewählt. (Siehe auch unter Abschnitt 8.11 für weitere Informationen über Arrays.)

4.2.2 Positionsparameter

Ein Positionsparameter ist ein Verweis auf einen Wert, der dem SQL-Befehl von einer externen Quelle zur Verfügung gestellt wird. Parameter werden in der Definition von SQL-Funktionen und in vorbereiteten Anfragen verwendet. Die Form eines Parameterverweises ist:

```
$zahl
```

Betrachten Sie zum Beispiel die Definition einer Funktion, `abteilung`, als

```
CREATE FUNCTION abteilung(text) RETURNS abteilung
    AS 'SELECT * FROM abteilung WHERE name = $1'
    LANGUAGE SQL;
```

Wenn die Funktion aufgerufen wird, wird `$1` hier durch das erste Funktionsargument ersetzt.

4.2.3 Operatoraufrufe

Es gibt drei mögliche Syntaxen für einen Operatoraufruf:

ausdruck operator ausdruck (binärer Infix-Operator)

operator ausdruck (unärer Präfix-Operator)

ausdruck operator (unärer Postfix-Operator)

wobei *operator* den in Abschnitt 4.1.3 beschriebenen Syntaxregeln folgt oder eines der Schlüsselwörter AND, OR und NOT oder ein qualifizierter Operatorname

```
OPERATOR(schema.operatorname)
```

ist. Welche Operatoren im Einzelnen vorhanden sind und ob sie unär oder binär sind, hängt davon ab, welche Operatoren vom System oder vom Benutzer definiert worden sind. Kapitel 9 beschreibt die eingebauten Operatoren.

4.2.4 Funktionsaufrufe

Die Syntax eines Funktionsaufrufs ist der Name einer Funktion (möglicherweise mit einem Schemanamen qualifiziert), gefolgt von der Argumentliste in Klammern:

```
funktion ([ausdruck [, ausdruck ... ]] )
```

Das folgende Beispiel berechnet die Quadratwurzel (englisch *square root*) von 2:

```
sqrt(2)
```

Die Liste der eingebauten Funktionen befindet sich in Kapitel 9. Andere Funktionen können vom Benutzer hinzugefügt werden.

4.2.5 Aggregatausdrücke

Ein *Aggregatausdruck* steht für die Anwendung einer Aggregatfunktion über eine Menge von Zeilen, die von einer Anfrage ausgewählt wurden. Eine Aggregatfunktion reduziert mehrere Eingabewerte auf einen einzelnen Wert, wie zum Beispiel die Summe oder den Durchschnitt der Werte. Die Syntax eines Aggregatausdrucks ist eine der folgenden:

```
aggregat_name (ausdruck)
aggregat_name (ALL ausdruck)
aggregat_name (DISTINCT ausdruck)
aggregat_name ( * )
```

wo *aggregat_name* eine vorab definierte Aggregatfunktion (möglicherweise mit Schemaname) und *ausdruck* ein beliebiger Wertausdruck, der nicht selbst einen Aggregatausdruck enthält, ist.

Die erste Form des Aggregatausdrucks führt die Aggregatfunktion für alle Eingabezeilen aus, für die der angegebene Ausdruck keinen NULL-Wert ergibt. (Tatsächlich entscheidet die Aggregatfunktion, ob sie NULL-Werte ignorieren will, aber die vorgegebenen tun dies alle.) Die zweite Form ist die gleiche wie die erste, da `ALL` die Standardvorgabe ist. Die dritte Form führt die Aggregatfunktion für alle unterschiedlichen Nicht-NULL-Werte, die sich aus dem Ausdruck für die Eingabezeilen ergeben, aus. Die letzte Form führt die Aggregatfunktion einmal für jede Eingabezeile aus, unabhängig davon, ob der Wert NULL ist oder nicht. Da kein bestimmter Eingabewert angegeben wird, ist diese Form nur für die Aggregatfunktion `count()` nützlich.

Ein Beispiel: `count(*)` ergibt die Gesamtzahl der Eingabezeilen; `count(f1)` ergibt die Gesamtzahl der Eingabezeilen, wo `f1` nicht NULL ist; `count(distinct f1)` ergibt die Gesamtzahl der unterschiedlichen Werte von `f1`, die nicht NULL sind.

Die vordefinierten Aggregatfunktionen sind in Abschnitt 9.14 beschrieben. Andere Aggregatfunktionen können von Benutzern hinzugefügt werden.

4.2.6 Typumwandlungen

Eine Typumwandlung (englisch *type cast*) beschreibt die Umwandlung eines Datentyps in einen anderen. PostgreSQL akzeptiert zwei gleichbedeutende Syntaxen für Typumwandlungen:

```
CAST ( ausdruck AS typ )
expression::type
```

Die Syntax mit `CAST` stimmt mit dem SQL-Standard überein; die Syntax mit `::` ist die historische PostgreSQL-Variante.

Wenn eine Typumwandlung auf einen Wertausdruck mit bekanntem Typ angewendet wird, dann ist das eine Laufzeit-Umwandlung. Die Umwandlung ist nur erfolgreich, wenn eine passende Umwandlungsfunktion verfügbar ist. Das ist auf subtile Weise anders als die Verwendung von Typumwandlungen mit Konstanten, wie in Abschnitt *Konstanten anderer Typen* beschrieben. Eine Typumwandlung, die auf eine einfache Zeichenkettenkonstante angewendet wird, ist die anfängliche Zuweisung eines Typs für die Konstante und ist mit jedem Typ erfolgreich (wenn der Inhalt der Zeichenkette gültige Syntax für den Datentyp ist).

Eine ausdrückliche Typumwandlung kann normalerweise ausgelassen werden, wenn keine Zweideutigkeit über den Typ, den der Wertausdruck produzieren muss, besteht (wenn er zum Beispiel einer Tabellen-

spalte zugewiesen wird); in solchen Fällen wendet das System eine Umwandlung automatisch an. Diese automatische Umwandlung wird allerdings nur für Umwandlungen angewendet, die im Systemkatalog zur »automatischen Umwandlung freigegeben« sind. Andere Umwandlungen müssen mit der ausdrücklichen Umwandlungssyntax durchgeführt werden. Diese Einschränkung besteht, damit überraschende Umwandlung nicht ohne Benachrichtigung ausgeführt werden.

Es ist auch möglich, eine Typumwandlung in einer funktionsähnlichen Syntax zu schreiben:

```
typname ( ausdruck )
```

Das funktioniert allerdings nur für Typnamen, die auch als Funktionsnamen zulässig sind. `double precision` kann zum Beispiel auf diese Art nicht verwendet werden, im Gegensatz zum gleichbedeutenden `float8`. Außerdem können die Namen `interval`, `time` und `timestamp` aufgrund von Syntaxkonflikten so nicht verwendet werden, es sei denn, sie werden von Anführungszeichen eingeschlossen. Daher führt die Verwendung der funktionsähnlichen Syntax zu Ungereimtheiten und sollte wahrscheinlich in neuen Anwendungen vermieden werden. (Die funktionsähnliche Syntax ist tatsächlich ein normaler Funktionsaufruf. Wenn einer der beiden normalen Umwandlungsyntaxen verwendet wird, um eine Laufzeitumwandlung durchzuführen, dann wird intern eine eingetragene Funktion aufgerufen, um die Umwandlung durchzuführen. Es ist Brauch, dass diese Umwandlungsfunktionen den gleichen Namen haben wie ihr Ergebnistyp, aber portierbare Anwendungen sollten sich nicht darauf verlassen.)

4.2.7 Skalare Unteranfragen

Eine sklare Unteranfrage ist eine normale `SELECT`-Anfrage in Klammern, die genau eine Zeile mit einer Spalte ergibt. (Siehe in Kapitel 7 für Informationen, wie man Anfragen schreibt.) Der `SELECT`-Befehl wird ausgeführt und der einzelne Ergebniswert wird in dem übergeordneten Wertausdruck verwendet. Es ist ein Fehler, wenn eine Anfrage als skalare Unteranfrage verwendet wird, aber mehr als eine Zeile oder eine Spalte ergibt. (Aber wenn in einem bestimmten Aufruf die Unteranfrage gar keine Zeilen ergibt, dann ist das kein Fehler; das skalare Ergebnis ist dann der NULL-Wert.) Die Unteranfrage kann sich auf Variablen in der äußeren Anfrage beziehen, welche sich dann bei jeder einzelnen Auswertung der Unteranfrage als Konstanten verhalten. Siehe auch unter Abschnitt 9.15.

Das folgende Beispiel findet die Stadt mit der größten Einwohnerzahl in jedem Land:

```
SELECT name, (SELECT max(einwohner) FROM städte WHERE städte.land = länder.name)
    FROM länder;
```

4.2.8 Auswertung von Ausdrücken

Die Reihenfolge, in der Ausdrücke ausgewertet werden, ist nicht definiert. Insbesondere werden die Argumente einer Funktion oder eines Operators nicht unbedingt von links nach rechts oder in einer anderen festgelegten Reihenfolge ausgewertet.

Ferner gilt: Wenn das Ergebnis eines Ausdrucks auch bestimmt werden kann, wenn nur Teile davon ausgewertet werden, dann könnten andere Teilausdrücke womöglich gar nicht ausgewertet werden. Wenn man zum Beispiel schreiben würde

```
SELECT true OR einefunktion();
```

dann würde `einefunktion()` (wahrscheinlich) gar nicht aufgerufen werden. Das Gleiche wäre der Fall wenn man schreiben würde

```
SELECT einefunktion() OR true;
```

Beachten Sie, dass das nicht das Gleiche ist wie die »*Short-Circuiting*«-Technik von Boole'schen Operationen von links nach rechts, die in manchen Programmiersprachen angewendet wird.

Folglich ist es unklug, Funktionen mit Nebenwirkungen in komplexen Ausdrücken zu verwenden. Es ist besonders gefährlich, sich auf die Nebenwirkungen oder die Auswertungsreihenfolge in `WHERE`- und `HAVING`-Klauseln zu verlassen, da diese Klauseln umfassend umgearbeitet werden, während der Ausführungsplan einer Anfrage entwickelt wird. Boole'sche Operationen (Kombinationen aus `AND`, `OR`, `NOT`) in diesen Klauseln können nach allen Möglichkeiten, die von den Gesetzen der Boole'schen Algebra erlaubt werden, umgeordnet werden.

Wenn es erforderlich ist, die Reihenfolge der Auswertung von Ausdrücken festzulegen, dann kann die `CASE`-Konstruktion (siehe Abschnitt 9.12) verwendet werden. Hier ist zum Beispiel eine unverlässliche Methode, eine Division durch null in einer `WHERE`-Klausel zu vermeiden:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

Aber dies hier ist sicher:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Eine `CASE`-Konstruktion, die auf diese Art verwendet wird, verhindert Optimierungen und sollte daher nur wenn unbedingt nötig angewendet werden.

5 Datendefinition

Dieses Kapitel behandelt, wie man die Datenbankstrukturen erzeugt, die dann die zu verarbeitenden Daten enthalten. In einer relationalen Datenbank werden die Daten in Tabellen abgespeichert, und folglich erklärt der Großteil dieses Kapitels, wie man Tabellen erzeugt und verändert und welche Fähigkeiten zur Verfügung stehen, um zu kontrollieren, welche Daten in den Tabellen gespeichert werden können. Anschließend besprechen wir, wie Tabellen organisiert werden können, indem man sie auf verschiedene Schemas verteilt, und wie Zugriffsrechte für Tabellen zugewiesen werden können. Am Schluss werden wir uns kurz weiteren Fähigkeiten, die sich auf die Datenbankstrukturen auswirken, wie zum Beispiel Sichten, Funktionen und Trigger, zuwenden. Weitere Informationen über diese Themen können Sie in Teil V finden.

5.1 Tabellengrundlagen

Eine Tabelle in einer relationalen Datenbank ist einer Tabelle auf Papier ziemlich ähnlich: Sie besteht aus Zeilen und Spalten. Die Anzahl und die Reihenfolge der Spalten ist festgelegt und jede Spalte hat einen Namen. Die Anzahl der Zeilen ist veränderlich: Sie hängt davon ab, wie viele Daten gerade gespeichert werden. SQL gibt aber keine Garantie über die Reihenfolge der Zeilen in einer Tabelle. Wenn eine Tabelle ausgelesen wird, erscheinen die Zeilen in zufälliger Reihenfolge, es sei denn eine Sortierung wird ausdrücklich angefordert. Dies ist in Kapitel 7 besprochen. Außerdem weist SQL den Zeilen keine eindeutige Identifizierung zu, sodass es möglich ist, mehrere vollständig identische Zeilen in einer Tabelle zu haben. Das ist eine Folge des mathematischen Modells, das SQL zugrunde liegt, aber es ist in der Regel nicht wünschenswert. Später in diesem Kapitel werden wir sehen, wie wir mit diesem Problem fertig werden können.

Jede Spalte hat einen Datentyp. Dieser Datentyp beschränkt die Menge der möglichen Werte, die der Spalte zugewiesen werden können, und weist den Daten, die in der Spalte gespeichert werden, einen semantischen Wert zu, sodass sie in Berechnungen verwendet werden können. Ein Spalte, die zum Beispiel als Zahlentyp deklariert wurde, akzeptiert keine beliebigen Zeichenketten, aber die Daten in einer solchen Spalte können für mathematische Berechnungen verwendet werden. Im Gegensatz dazu kann eine Spalte, die als Zeichenkettentyp deklariert wurde, so ziemlich alle Arten von Daten aufnehmen, aber nicht für mathematische Berechnungen verwendet werden. Dagegen stehen andere Operationen, wie Zeichenkettenverknüpfung, zur Verfügung.

PostgreSQL enthält eine Menge eingebauter Datentypen, die für viele Anwendungen passend sind. Benutzer können auch ihre eigenen Datentypen definieren. Die meisten eingebauten Datentypen haben offensichtliche Namen und Bedeutungen, sodass wir mit einer detaillierten Erklärung bis Kapitel 8 warten. Einige häufig verwendete Datentypen sind `integer` für Ganzzahlen, `numeric` für Zahlen mit möglichem

Bruchteil, `text` für Zeichenketten, date für Datumsangaben, `time` für Tageszeitangaben und `timestamp` für Werte mit Datum und Zeit.

Um eine Tabelle zu erzeugen, verwenden Sie den Befehl `CREATE TABLE`. In diesem Befehl geben Sie mindestens den Namen der neuen Tabelle, die Namen der Spalten und die Datentypen jeder Spalte an. Zum Beispiel:

```
CREATE TABLE meine_erste_tabelle (
    erste_spalte text,
    zweite_spalte integer
);
```

Das erzeugt eine Tabelle namens `meine_erste_tabelle` mit zwei Spalten. Die erste Spalte heißt `erste_spalte` und hat den Datentyp text; die zweite Spalte heißt `zweite_spalte` und hat den Typ integer. Die Namen der Tabelle und der Spalten müssen den in Abschnitt 4.1.1 beschriebenen Regel für Namen folgen. Die Typnamen folgen diesen Regeln normalerweise auch, aber es gibt Ausnahmen. Beachten Sie, dass die Spalteneinträge durch Kommas getrennt sind und die Spaltenliste in Klammern steht.

Das obige Beispiel ist natürlich gekünstelt. Normalerweise werden Sie Ihren Tabellen und Spalten Namen geben, die vermitteln, welche Daten sie speichern. Schauen wir uns also ein etwas realistischeres Beispiel an:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric
);
```

(Der Typ numeric kann Nachkommastellen speichern, wie sie in Preisangaben auftreten.)

Tipp

Wenn Sie viele Tabellen erstellen, die untereinander in Verbindung stehen, dann ist es sinnvoll, ein einheitliches Muster für die Namen von Tabellen und Spalten zu verwenden. Es gibt zum Beispiel die Wahl zwischen Einzahl und Mehrzahl für die Tabellennamen, wobei beides von dem einen oder anderen Theoretiker bevorzugt wird.

Es gibt eine Begrenzung, wie viele Spalten eine Tabelle enthalten kann. In Abhängigkeit von den Spaltentypen ist diese Grenze zwischen 250 und 1600. Ein Tabelle mit so vielen Spalten zu erzeugen, ist aber sehr ungewöhnlich und oft ein ziemlich fragwürdiger Entwurf.

Wenn Sie eine Tabelle nicht mehr benötigen, dann können Sie sie mit dem Befehl `DROP TABLE` entfernen. Zum Beispiel:

```
DROP TABLE meine_erste_tabelle;
DROP TABLE produkte;
```

Der Versuch, eine nicht vorhandene Tabelle zu entfernen, ist ein Fehler. Trotzdem ist es gebräuchlich, in Stapeldateien mit SQL-Befehlen, Tabellen vor der Erzeugung bedingungslos zu entfernen und die Fehlermeldungen zu ignorieren.

Wenn Sie eine bestehende Tabelle verändern wollen, dann schauen Sie in Abschnitt 5.5 weiter unten in diesem Kapitel.

Mit den bis jetzt besprochenen Werkzeugen können Sie voll funktionsfähige Tabellen erzeugen. Der Rest dieses Kapitels beschäftigt sich mit weiteren Merkmalen, die Sie Ihren Tabellen hinzufügen können, um

die Integrität oder Sicherheit der Daten sicherzustellen oder die Datenverwaltung bequemer zu gestalten. Wenn Sie ungeduldig sind und Ihre Tabellen jetzt mit Daten füllen wollen, dann springen Sie zu Kapitel 6 und lesen Sie den Rest dieses Kapitels später.

5.2 Systemspalten

Jede Tabelle hat mehrere **Systemspalten**, die von System automatisch erzeugt werden. Diese Namen können daher nicht als Namen von benutzerdefinierten Spalten verwendet werden. (Beachten Sie, dass diese Einschränkung nichts damit zu tun hat, ob der Name ein Schlüsselwort ist; Sie können diese Einschränkung nicht umgehen, indem Sie den Namen in Anführungszeichen setzen.) Sie müssen sich um diese Spalten nicht weiter kümmern, außer zu wissen, dass sie existieren.

`oid`

Der Objekt-Identifikator (englisch *Object Identifier*) einer Zeile. Das ist eine Seriennummer, die von PostgreSQL jeder Zeile automatisch hinzugefügt wird (es sei denn, die Tabelle wurde mit `WITHOUT OIDS` erzeugt, dann ist diese Spalte nicht vorhanden). Diese Spalte hat den Typ oid (gleicher Name wie die Spalte); siehe unter Abschnitt 8.10 für weitere Informationen über den Typ.

`tableoid`

Die OID der Tabelle, die diese Zeile enthält. Diese Spalte ist besonders nützlich in Anfragen, die aus Vererbungshierarchien auswählen, denn ohne sie ist es schwer zu ermitteln, aus welcher einzelnen Tabelle die Zeile kam. Die `tableoid` kann mit der `oid`-Spalte der Tabelle `pg_class` verbunden werden, um den Tabellennamen zu ermitteln.

`xmin`

Die Identität (Transaktionsnummer) der Transaktion, die diese Zeilenversion eingefügt hat. (Jede Aktualisierung erzeugt eine neue Version derselben logischen Zeile.)

`cmin`

Die Befehlsnummer (englisch *Command Identifier*) (von null an zählend) innerhalb der einfügenden Transaktion.

`xmax`

Die Identität (Transaktionsnummer) der löschenden Transaktion oder null, wenn die Zeilenversion nicht gelöscht ist. Diese Spalte kann auch für eine sichtbare Zeilenversion von null verschieden sein: Das bedeutet in der Regel, dass die löschende Transaktion noch nicht abgeschlossen oder dass das versuchte Löschen zurückgerollt wurde.

`cmax`

Die Befehlsnummer innerhalb der löschenden Transaktion oder null.

`ctid`

Die physikalische Speicherungsstelle der Zeilenversion innerhalb der Tabelle. Beachten Sie, dass `ctid` zwar verwendet werden kann, um die Zeile schnell zu finden, aber die `ctid` einer Zeile ändert sich jedes Mal, wenn sie aktualisiert oder durch `VACUUM FULL` bewegt wird. Daher ist die `ctid` als Zeilenidentifikation über längere Zeiträume hinweg nicht zu gebrauchen. Die OID oder besser noch eine benutzerdefinierte Seriennummer sollte zur Identifizierung von logischen Zeilen verwendet werden.

OIDs sind 32-Bit-Größen und werden von einem einzigen Zähler in jedem Cluster erzeugt. In einer großen oder langlebigen Datenbank ist es deshalb möglich, dass der Zähler überläuft und wieder von vorne anfängt. Daher ist es eine schlechte Strategie, anzunehmen, dass OIDs eindeutig sind, außer wenn man Schritte unternimmt, die Eindeutigkeit abzusichern. Wenn man OIDs zur Zeilenidentifikation verwendet, ist es zu empfehlen, einen Unique Constraint für die OID-Spalte jeder Tabelle zu erzeugen, in der die

OID verwendet werden soll. Gehen Sie niemals davon aus, dass OIDs über mehrere Tabellen hinweg eindeutig sind; verwenden Sie die Kombination aus `tableoid` und der Zeilen-OID, wenn Sie einen Zähler brauchen, der in der ganzen Datenbank eindeutig ist. (Zukünftige Versionen von PostgreSQL werden wahrscheinlich einen getrennten OID-Zähler für jede Tabelle verwenden, sodass `tableoid` dann verwendet werden *muss*, um einen überall eindeutigen Zähler zu konstruieren.)

Transaktionsnummern sind auch 32 Bit groß. In einer langlebigen Datenbank ist es daher möglich, dass die Transaktionsnummern überlaufen und von vorne zu zählen anfangen. Das ist kein schwerwiegendes Problem, wenn man die Wartungsprozeduren regelmäßig durchführt; Einzelheiten sind in Kapitel 21 zu finden. Es ist allerdings nicht empfehlenswert, sich auf die Eindeutigkeit von Transaktionsnummern über lange Zeiträume (mehr als eine Milliarde Transaktionen) zu verlassen.

Befehlsnummern sind ebenso 32 Bit groß. Daraus ergibt sich eine Grenze von 2^{32} (4 Milliarden) SQL-Befehlen innerhalb einer einzelnen Transaktion. Das ist in der Praxis kein Problem. Beachten Sie, dass diese Grenze sich auf die Anzahl der SQL-Befehle bezieht und nicht die Anzahl der verarbeiteten Zeilen.

5.3 Vorgabewerte

Einer Spalte kann ein Vorgabewert (englisch *default value*) zugewiesen werden. Wenn eine neue Zeile erzeugt wird und für einige Spalten keine Werte angegeben sind, werden die Spalten mit ihren entsprechenden Vorgabewerten gefüllt. Ein Datenmanipulationsbefehl kann auch ausdrücklich verlangen, dass eine Spalte auf ihren Vorgabewert gesetzt wird, ohne zu wissen, was dieser Wert ist. (Einzelheiten zu Datenmanipulationsbefehlen finden Sie in Kapitel 6.)

Wenn kein Vorgabewert ausdrücklich definiert wurde, wird der NULL-Wert als Vorgabe angenommen. Das ist normalerweise sinnvoll, weil ein NULL-Wert eingesetzt wird, wo die Daten unbekannt sind.

In einer Tabellendefinition werden die Vorgabewerte nach dem Spaltendatentyp aufgelistet. Zum Beispiel:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric DEFAULT 9.99
);
```

Der Vorgabewert kann ein skalarer Ausdruck sein, welcher immer ausgewertet wird, wenn ein Vorgabewert in die Tabelle eingefügt wird (also *nicht* schon, wenn die Tabelle erzeugt wird).

5.4 Constraints

Mit der Auswahl des Datentyps kann man die Art der Daten, die in einer Spalte gespeichert werden können, beschränken. Für viele Anwendungen ist diese Art der Beschränkung aber zu grob. Ein Spalte mit einem Produktpreis sollte zum Beispiel wahrscheinlich nur positive Werte aufnehmen dürfen. Aber es gibt keinen Datentyp, der nur positive Zahlen akzeptiert. Eine andere Problematik ist, dass Sie die Spaltendaten vielleicht auch in Bezug auf die Daten in anderen Spalten oder Zeilen beschränken wollen. In einer Tabelle mit Produktinformationen sollte zum Beispiel nur eine Zeile für jede Produktnummer auftauchen.

Aus diesem Grund erlaubt SQL es Ihnen, Constraints (englisch für Beschränkungen) für Spalten und Tabellen zu definieren. Constraints geben Ihnen genau soviel Kontrolle über Ihre Daten, wie sie wollen. Wenn ein Benutzer versucht, Daten in eine Spalte zu schreiben, die den Constraint verletzen würde, wird ein Fehler ausgegeben. Das gilt auch dann, wenn der Wert aus der Definition des Vorgabewerts kam.

5.4.1 Check-Constraints

Ein Check-Constraint ist der allgemeinste Constraint-Typ. Er erlaubt es Ihnen, einen beliebigen Ausdruck anzugeben, den jeder Wert einer Spalte erfüllen muss. Um zum Beispiel positive Produktpreise zu erzwingen, könnten Sie Folgendes tun:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric CHECK (preis > 0)
);
```

Wie Sie sehen, steht die Constraint-Definition nach dem Datentyp, genauso wie die Definition des Vorgabewerts. Vorgabewerte und Constraints können in beliebiger Reihenfolge gelistet werden. Ein Check-Constraint besteht aus dem Schlüsselwort `CHECK`, gefolgt von einem Ausdruck in Klammern. Der Ausdruck sollte die auf diese Art beschränkte Spalte beinhalten, ansonsten wäre der Constraint nicht sehr sinnvoll.

Sie können einem Constraint auch einen getrennten Namen geben. Das kann Fehlermeldungen klarer gestalten und erlaubt es Ihnen, auf den Constraint zu verweisen, wenn Sie ihn ändern müssen. Die Syntax ist:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric CONSTRAINT positiver_preis CHECK (preis > 0)
);
```

Um einen Namen für einen Constraint zu definieren, schreiben Sie das Schlüsselwort `CONSTRAINT` gefolgt von dem Namen und der Definition des Constraints.

Ein Check-Constraint kann sich auch auf mehrere Spalten beziehen. Nehmen wir an, Sie speichern einen normalen Preis und einen Rabattpreis und sie wollen sicherstellen, dass der Rabattpreis immer niedriger als der Normalpreis ist.

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric CHECK (preis > 0),
    rabattpreis numeric CHECK (rabattpreis > 0),
    CHECK (preis > rabattpreis)
);
```

Die ersten zwei Constraints sollten Ihnen bekannt vorkommen. Der dritte verwendet eine neue Syntax. Er steht nicht bei einer bestimmten Spalte, sondern als getrenntes Element in der Spaltenliste. Spaltendefinitionen und diese Art der Constraint-Definition können in gemischter Reihenfolge aufgelistet werden.

Man sagt, dass die ersten beiden Constraints Spalten-Constraints sind, während der dritte ein Tabellen-Constraint ist, weil er getrennt von den Spaltendefinitionen geschrieben wird. Spalten-Constraints können auch als Tabellen-Constraints geschrieben werden, aber umgekehrt ist das nicht immer möglich. Das obige Beispiel kann auch so geschrieben werden:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
```

```
    preis numeric,
    CHECK (preis > 0),
    rabattpreis numeric,
    CHECK (rabattpreis > 0),
    CHECK (preis > rabattpreis)
);
```

oder gar so:

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric CHECK (preis > 0),
    rabattpreis numeric,
    CHECK (rabattpreis > 0 AND preis > rabattpreis)
);
```

Es ist letztendlich Geschmackssache.

Man sollte beachten, dass der Check-Constraint passiert wird, wenn das Ergebnis des Check-Ausdrucks logisch wahr oder der NULL-Wert ist. Da die meisten Ausdrücke den NULL-Wert ergeben, wenn ein Operand der NULL-Wert ist, verhindern sie keine NULL-Werte in den beschränkten Spalten. Um zu versichern, dass eine Spalte keine NULL-Werte enthalten kann, sollte der NOT-NULL-Constraint, der im folgenden Abschnitt beschrieben wird, angewendet werden.

5.4.2 NOT-NULL-Constraints

Ein NOT-NULL-Constraint gibt ganz einfach an, dass eine Spalte den NULL-Wert nicht aufnehmen darf. Ein Syntaxbeispiel:

```
CREATE TABLE produkte (
produkt_nr integer NOT NULL,
name text NOT NULL,
preis numeric
 );
```

Ein NOT-NULL-Constraint wird immer als Spalten-Constraint geschrieben. Ein NOT-NULL-Constraint hat den gleichen Effekt wie ein Check-Constraint der Form `CHECK (spaltenname IS NOT NULL)`, aber in PostgreSQL ist der ausdrückliche NOT-NULL-Constraint effizienter. Der Nachteil ist, dass man dieser Art von NOT-NULL-Constraint keinen gesonderten Namen geben kann.

Natürlich kann eine Spalte mehrere Constraints haben. Schreiben Sie sie einfach hintereinander:

```
CREATE TABLE produkte (
    produkt_nr integer NOT NULL,
    name text NOT NULL,
    preis numeric NOT NULL CHECK (preis > 0)
);
```

Die Reihenfolge ist egal. Sie beeinflusst auch nicht unbedingt die Reihenfolge, in der die Constraints geprüft werden.

Der NOT-NULL-Constraint hat eine Umkehrung: den NULL-Constraint. Der bedeutet aber nicht, dass die Spalte nur den NULL-Wert akzeptiert, was ziemlich sinnlos wäre. Vielmehr drückt er einfach den Normalzustand aus, dass eine Spalte den NULL-Wert aufnehmen darf. Der NULL-Constraint ist im SQL-Standard nicht definiert und sollte in portierbaren Anwendungen nicht verwendet werden. (Er würde nur in PostgreSQL eingebaut, um mit anderen Datenbanksystemen kompatibel zu sein.) Einige Benutzer mögen diesen Constraints allerdings, weil er es erleichtert, in einer Textdatei den Sinn des Constraint umzukehren. Sie könnten zum Beispiel mit dieser Textdatei anfangen:

```
CREATE TABLE produkte (
    produkt_nr integer NULL,
    name text NULL,
    preis numeric NULL
);
```

und dann fügen Sie einfach daas Schlüsselwort NOT dort ein, wo Sie es wünschen.

5.4.3 Unique Constraints

Tipp

In den meisten Datenbankentwürfen sollte die Mehrzahl der Spalten als `NOT NULL` markiert sein.

Unique Constraints stellen sicher, dass die Daten einer Spalte oder einer Gruppe von Spalten von allen anderen Zeilen der Tabelle voneinander verschieden sind. Die Syntax ist

```
CREATE TABLE produkte (
    produkt_nr integer UNIQUE,
    name text,
    preis numeric
);
```

wenn man es als Spalten-Constraint schreibt, und

```
CREATE TABLE produkte (
    produkt_nr integer,
    name text,
    preis numeric,
    UNIQUE (produkt_nr)
);
```

wenn man es als Tabellen-Constraint schreibt.

Wenn sich ein Unique Constraint auf eine Gruppe von Spalten bezieht, dann werden die Spalten durch Kommas getrennt aufgelistet:

```
CREATE TABLE beispiel (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Man kann Unique Constraints auch Namen zuweisen:

```
CREATE TABLE produkte (
    produkt_nr integer CONSTRAINT müssen_verschieden_sein UNIQUE,
    name text,
    preis numeric
);
```

Generell ist ein Unique Constraint verletzt, wenn es (mindestens) zwei Zeilen in der Tabelle gibt, wo die Werte der einander entsprechenden Spalten, die Teil des Constraints sind, gleich sind. NULL-Werte werden hier allerdings nicht als gleich erachtet. Das bedeutet, dass man trotz eines mehrspaltigen Unique Constraints eine unbegrenzte Anzahl Zeilen erzeugen kann, in denen mindestens eine der beschränkten Spalten den NULL-Wert hat. Dieses Verhalten entspricht dem SQL-Standard, aber wir haben vernommen, dass einige SQL-Datenbanken dieser Regel wohl nicht folgen. Also seien Sie vorsichtig, wenn Sie Anwendungen entwickeln, die portierbar sein sollen.

5.4.4 Primärschlüssel

Technisch gesehen ist ein Primärschlüssel-Constraint (englisch *primary key*) einfach eine Kombination aus Unique Constraint und NOT-NULL-Constraint. Die folgenden beiden Tabellendefinitionen können also die gleichen Daten aufnehmen:

```
CREATE TABLE produkte (
    produkt_nr integer UNIQUE NOT NULL,
    name text,
    preis numeric
);

CREATE TABLE produkte (
    produkt_nr integer PRIMARY KEY,
    name text,
    preis numeric
);
```

Primärschlüssel können auch über mehrere Spalten gehen; die Syntax ist ähnlich der des Unique Constraints:

```
CREATE TABLE beispiel (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

Ein Primärschlüssel zeigt an, dass eine Spalte oder eine Gruppe von Spalten als eindeutige Identifikation einer Zeile in der Tabelle verwendet werden kann. (Das ist eine direkte Folge aus der Definition des Primärschlüssels. Beachten Sie, dass ein Unique Constraint keine eindeutige Identifikation erlaubt, da er NULL-Werte nicht ausschließt.) Das ist sowohl zu Zwecken der Dokumentation der Datenbankstruktur als auch für Clientanwendungen nützlich. Wenn zum Beispiel eine grafische Anwendung es Ihnen erlaubt, Zeilen zu verändern, dann muss diese Anwendung höchstwahrscheinlich den Primärschlüssel der Tabelle kennen, um die Zeilen eindeutig identifizieren zu können.

Eine Tabelle kann höchstens einen Primärschlüssel haben (aber sie kann mehrere Unique- und NOT-NULL-Constraints haben). Die Theorie der relationalen Datenbanken schreibt vor, dass jede Tabelle einen Primärschlüssel haben muss. Diese Regel wird von PostgreSQL nicht durchgesetzt, aber es ist meistens am besten, ihr zu folgen.

5.4.5 Fremdschlüssel

Ein Fremdschlüssel-Constraint (englisch *foreign key*) gibt an, dass die Werte in einer Spalte (oder einer Gruppe von Spalten) mit den Werten in irgendeiner Zeile in einer anderen Tabelle übereinstimmen müssen. Man sagt, dass dadurch die **referenzielle Integrität** zwischen zwei zusammenhängenden Tabellen gesichert wird.

Nehmen wir die Produkttabelle, die wir schon mehrere Male verwendet haben:

```
CREATE TABLE produkte (
    produkt_nr integer PRIMARY KEY,
    name text,
    preis numeric
);
```

Nehmen wir ferner an, dass Sie eine Tabelle haben, die Bestellungen dieser Produkte speichert. Wir wollen sichergehen, dass die Bestellungstabelle nur Bestellungen von tatsächlich existierenden Produkten aufnimmt. Also definieren wir einen Fremdschlüssel-Constraint in der Bestellungstabelle, der sich auf die Produkttabelle bezieht:

```
CREATE TABLE bestellungen (
    bestell_nr integer PRIMARY KEY,
    produkt_nr integer REFERENCES produkte (produkt_nr),
    menge integer
);
```

Jetzt ist es unmöglich, Bestellungen mit `produkt_nr`-Einträgen zu erzeugen, die nirgendwo in der Produkttabelle auftauchen.

Bei dieser Gliederung sagt man, dass die Tabelle `bestellungen` auf die Tabelle `produkte` **verweist**. Genauso verweisen auch die Spalten aufeinander.

Der obige Befehl kann auch abgekürzt werden:

```
CREATE TABLE bestellungen (
    bestell_nr integer PRIMARY KEY,
    produkt_nr integer REFERENCES produkte,
    menge integer
);
```

Wenn die Spaltenliste ausgelassen wird, dann wird der Primärschlüssel der Tabelle, auf die man verweist, als Zielspalte verwendet.

Ein Fremdschlüssel kann auch eine Gruppe von Spalten beschränken bzw. auf eine Gruppe von Spalten verweisen. Er muss dann wie gewohnt als Tabellen-Constraint geschrieben werden. Hier ist ein bedeutungsloses Syntaxbeispiel:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
```

```
    b integer,
    c integer,
    FOREIGN KEY (b, c) REFERENCES andere_tabelle (c1, c2)
);
```

Die Zahl und der Typ der beschränkten Spalten muss natürlich mit der Zahl und dem Typ der Spalten übereinstimmen, auf die man verweist.

Eine Tabelle kann mehrere Fremdschlüssel-Constraints enthalten. Diese Fähigkeit verwendet man, um so genannte m:n-Beziehungen (viele-zu-viele) zwischen Tabellen umzusetzen. Nehmen wir an, Sie haben Tabellen mit Produkten und mit Bestellungen, aber jetzt wollen Sie auch zulassen, dass eine Bestellung mehrere Produkte aufnehmen kann (was in der oben verwendeten Struktur nicht möglich war). Dazu könnten Sie folgende Tabellenstruktur verwenden:

```
CREATE TABLE produkte (
    produkt_nr integer PRIMARY KEY,
    name text,
    preis numeric
);

CREATE TABLE bestellungen (
    bestell_nr integer PRIMARY KEY,
    lieferadresse text,
    ...
);

CREATE TABLE bestell_artikel (
    produkt_nr integer REFERENCES produkte,
    bestell_br integer REFERENCES bestellungen,
    menge integer,
    PRIMARY KEY (produkt_nr, bestell_nr)
);
```

Sie werden auch bemerken, dass sich in der letzten Tabelle der Primärschlüssel mit den Fremdschlüsseln überschneidet.

Wir haben jetzt erreicht, dass die Fremdschlüssel das Erzeugen von Bestellungen, die zu keinen Produkten passen, verhindern. Aber was passiert, wenn ein Produkt gelöscht wird, während noch eine Bestellung darauf verweist? In SQL können Sie auch das angeben. Intuitiv haben wir einige Möglichkeiten:

- das Löschen des Produktes verhindern
- die Bestellungen ebenso löschen
- Noch etwas?

Um dies zu erklären, setzen wir folgende Regeln am obigen m:n-Beispiel um: Wenn jemand ein Produkt entfernen möchte, das noch in einer Bestellung verwendet wird (über `bestell_artikel`), wird das verboten. Wenn jemand eine Bestellung entfernt, dann werden die Bestellartikel auch gelöscht.

```
CREATE TABLE produkte (
    produkt_nr integer PRIMARY KEY,
    name text,
    preis numeric
```

```
);

CREATE TABLE bestellungen (
    bestell_nr integer PRIMARY KEY,
    lieferadresse text,
    ...
);

CREATE TABLE bestell_artikel (
    produkt_nr integer REFERENCES produkte ON DELETE RESTRICT,
    bestell_nr integer REFERENCES bestellungen ON DELETE CASCADE,
    menge integer,
    PRIMARY KEY (produkt_nr, bestell_nr)
);
```

`RESTRICT` steht für "einschränken", also faktisch verbieten; `CASCADE` bedeutet, dass die jeweilige Aktion auch in den anderen Tabellen durchgeführt wird. (Für *cascade* lässt sich kaum eine gute deutsche Übersetzung finden. Das Wort steht eigentlich für eine Reihe kleinerer, zusammenhängender Wasserfälle.) `RESTRICT` und `CASCADE` sind die zwei am häufigsten verwendeten Möglichkeiten. `RESTRICT` kann auch als `NO ACTION` geschrieben werden und ist auch die Vorgabe, wenn Sie gar nichts schreiben. Es gibt zwei weitere Möglichkeiten, was mit den Fremdschlüsselspalten passieren soll, wenn ein Primärschlüssel gelöscht wird: `SET NULL` (auf den NULL-Wert setzen) und `SET DEFAULT` (auf den Vorgabewert setzen). Beachten Sie, dass dadurch aber keine anderen Constraints umgangen werden können. Wenn zum Beispiel `SET DEFAULT` angegeben ist, aber der Vorgabewert den Fremdschlüssel-Constraint verletzen würde, dann wird das Löschen des Primärschlüssels fehlschlagen.

Analog zu `ON DELETE` gibt es auch noch `ON UPDATE`, womit bestimmt wird, was passiert, wenn ein Primärschlüssel aktualisiert wird (englisch *update*). Die möglichen Aktionen sind dieselben.

Weitere Informationen über das Aktualisieren und das Löschen von Daten finden Sie in Kapitel 6.

Am Schluss sollten wir noch erwähnen, dass ein Fremdschlüssel immer auf Spalten verweisen muss, die entweder einen Primärschlüssel oder einen Unique Constraint bilden. Wenn ein Fremdschlüssel auf einen Unique Constraint verweist, gibt es noch einige weitere Möglichkeiten, wie NULL-Werte verarbeitet werden. Diese sind in Teil VI im Eintrag für `CREATE TABLE` erläutert.

5.5 Tabellen verändern

Wenn Sie eine Tabelle erzeugen und feststellen, dass sie einen Fehler gemacht haben oder wenn sich die Anforderungen an Ihre Anwendung verändert haben, dann können Sie die Tabelle entfernen und neu erzeugen. Aber das ist keine passable Lösung, wenn die Tabelle bereits mit Daten gefüllt ist oder wenn die Tabelle von anderen Datenbankobjekten benötigt wird (zum Beispiel von einem Fremdschlüssel). Deswegen hat PostgreSQL eine Sammlung von Befehlen, um Änderungen an bestehenden Tabellen vorzunehmen.

Sie können

- Spalten hinzufügen,
- Spalten entfernen,
- Constraints hinzufügen,
- Constraints entfernen,
- Vorgabewerte ändern,

- Spalten umbenennen,
- Tabellen umbenennen.

Die Aktionen werden alle mit dem Befehl `ALTER TABLE` vorgenommen.

5.5.1 Eine Spalte hinzufügen

Um eine Spalte hinzuzufügen, verwenden Sie diesen Befehl:

```
ALTER TABLE produkte ADD COLUMN beschreibung text;
```

Die neue Spalte wird in den bestehenden Zeilen anfänglich mit NULL-Werten gefüllt.

Sie können für die Spalte auch gleich einen Constraint mit definieren, indem Sie die bekannte Syntax verwenden:

```
ALTER TABLE produkte ADD COLUMN beschreibung text CHECK (beschreibung <> '');
```

Eine neue Spalte kann keinen NOT-NULL-Constraint haben, da die Spalte anfänglich NULL-Werte enthalten muss. Aber Sie können einen NOT-NULL-Constraint später hinzufügen. Eine neue Spalte kann auch keinen Vorgabewert haben. Dem SQL-Standard nach müsste dann die neue Spalte in den bestehenden Zeilen auf dem Vorgabewert gesetzt werden, was so noch nicht umgesetzt ist. Aber Sie können den Vorgabewert immer noch nachher ändern.

5.5.2 Eine Spalte entfernen

Um eine Spalte zu entfernen, verwenden Sie diesen Befehl:

```
ALTER TABLE produkte DROP COLUMN beschreibung;
```

5.5.3 Einen Constraint hinzufügen

Um einen Constraint hinzuzufügen, verwenden Sie die Syntax für Tabellen-Constraints. Zum Beispiel:

```
ALTER TABLE produkte ADD CHECK (name <> '');
ALTER TABLE produkte ADD CONSTRAINT irgendein_name UNIQUE (produkt_nr);
ALTER TABLE produkte ADD FOREIGN KEY (produktgruppen_nr) REFERENCES
produktgruppen;
```

Um einen NOT-NULL-Constraint, der ja nicht als Tabellen-Constraint geschrieben werden kann, hinzuzufügen, verwenden Sie diese Syntax:

```
ALTER TABLE produkte ALTER COLUMN produkt_nr SET NOT NULL;
```

Der neue Constraint wird sofort geprüft, also muss die Tabelle den Constraint erfüllen, bevor dieser erstellt werden kann.

5.5.4 Einen Constraint entfernen

Um einen Constraint entfernen zu können, müssen Sie seinen Namen kennen. Wenn Sie ihm einen Namen gegeben hatten, dann ist das ganz einfach. Ansonsten hat das System einen Namen erzeugt, den

Sie herausfinden müssen. Der `psql`-Befehl `\d` *`tabellenname`* kann hier helfen; andere Schnittstellen haben möglicherweise auch Wege, um sich die Tabellendaten anzusehen. Der Befehl ist dann:

ALTER TABLE produkte CONSTRAINT irgendein_name;

```
DROP
```

Das funktioniert bei allen Constraint-Typen außer NOT-NULL-Constraints. Um einen NOT-NULL-Constraint zu entfernen, verwenden Sie

```
ALTER TABLE produkte ALTER COLUMN produkt_nr DROP NOT NULL;
```

(Erinnern Sie sich, dass NOT-NULL-Constraints keine Namen haben.)

5.5.5 Den Vorgabewert ändern

Um einer Spalte einen neuen Vorgabewert zuzuweisen, verwenden Sie einen Befehl wie diesen:

```
ALTER TABLE produkte ALTER COLUMN preis SET DEFAULT 7.77;
```

Um einen Vorgabewert zu entfernen, verwenden Sie

```
ALTER TABLE produkte ALTER COLUMN preis DROP DEFAULT;
```

Das hat die gleiche Bedeutung, als wenn man den Vorgabewert auf den NULL-Wert setzt, zumindest in PostgreSQL. Daraus folgt, dass es kein Fehler ist, einen Vorgabewert zu entfernen, wo keiner definiert war, weil der Vorgabewert automatisch der NULL-Wert ist.

5.5.6 Eine Spalte umbenennen

Der Befehl, um eine Spalte umzubenennen, ist:

```
ALTER TABLE produkte RENAME COLUMN produkt_nr TO produktnummer;
```

5.5.7 Eine Tabelle umbenennen

Der Befehl, um eine Tabelle umzubenennen, ist:

```
ALTER TABLE produkte RENAME TO artikel;
```

5.6 Privilegien

Wenn Sie ein Datenbankobjekt erzeugen, sind Sie sein Eigentümer. In der Ausgangslage kann nur der Eigentümer eines Objekts etwas damit machen. Um anderen Benutzern die Verwendung zu erlauben, müssen Sie **Privilegien** erteilen. (Es gibt auch Benutzer, die das Superuser-Privileg haben. Diese Benutzer können immer auf alle Objekte zugreifen.)

Es gibt mehrere verschiedene Privilegien: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE`, `USAGE` und `ALL PRIVILEGES`. Vollständige Informationen

Anmerkung

Um den Eigentümer einer Tabelle, eines Index, einer Sequenz oder einer Sicht zu ändern, verwenden Sie den Befehl `ALTER TABLE`.

über die verschiedenen Privilegientypen, die von PostgreSQL unterstützt werden, finden auf der `GRANT`-Referenzseite. Die folgenden Abschnitte und Kapitel werden Ihnen auch zeigen, wie die Privilegien verwendet werden.

Das Recht, ein Objekt zu verändern oder zu entfernen, ist immer das ausschließliche Privileg des Eigentümers.

Um Privilegien zu gewähren, wird der Befehl `GRANT` verwendet. Wenn also `fred` ein bestehender Benutzer ist und `konten` eine vorhandene Tabelle, dann kann das Privileg, die Tabelle aktualisieren zu dürfen, folgendermaßen gewährt werden:

```
GRANT UPDATE ON konten TO fred;
```

Der Benutzer, der diesen Befehl ausführt, muss der Eigentümer der Tabelle sein. Um das Privileg einer Gruppe zu gewähren, verwenden Sie

```
GRANT SELECT ON konten TO GROUP innendienst;
```

Der spezielle "Benutzername" `PUBLIC` kann verwendet werden, um ein Privileg allen Benutzern im System zu gewähren. Wenn Sie `ALL` anstelle der einzelnen Privilegien schreiben, dann werden alle Arten von Privilegien gewährt.

Um ein Privileg zu entziehen, verwenden Sie den Befehl `REVOKE`:

```
REVOKE ALL ON konten FROM PUBLIC;
```

Die besonderen Privilegien des Eigentümers (das heißt, das Recht, `DROP`, `GRANT`, `REVOKE` usw. auszuführen) sind dem Eigentümer immer automatisch gewährt und können nicht weitergegeben oder entzogen werden. Ein Tabelleneigentümer kann allerdings seine eigenen normalen Privilegien entziehen, um zum Beispiel zu verhindern, dass er aus Versehen in die Tabelle schreibt.

5.7 Schemas

Ein PostgreSQL-Datenbankcluster enthält eine oder mehrere benannte Datenbanken. Benutzerkonten und Gruppenkonten sind im gesamten Cluster verfügbar, aber weitere Daten haben die Datenbanken nicht gemeinsam. Jede Verbindung eines Clients kann nur auf eine einzige Datenbank zugreifen, nämlich die, die in der Verbindungsanfrage angegeben wurde.

Anmerkung

Die Benutzer eines Clusters haben nicht unbedingt das Privileg, auf jede Datenbank in dem Cluster zuzugreifen. Dass die Benutzerkonten in allen Datenbanken gelten, heißt, dass es keine unterschiedlichen Benutzer, sagen wir mit Namen `fred`, in zwei Datenbanken des Clusters geben kann, aber das System kann so eingestellt werden, dass `fred` nur auf einige Datenbanken Zugriff hat.

Eine Datenbank enthält ein oder mehrere **Schemas**, welche wiederum Tabellen enthalten. Schemas enthalten auch andere Arten von Objekten, einschließlich Typen, Funktionen und Operatoren. Der gleiche Objektname kann in verschiedenen Schemas verwendet werden, ohne einen Konflikt zu erzeugen; zum Beispiel können sowohl `schema1` also auch `meinschema` Tabellen namens `meinetabelle` enthalten. Im

Gegensatz zu Datenbanken sind Schemas nicht strikt voneinander getrennt: Ein Benutzer kann auf Objekte in allen Schemas in der Datenbank, mit der er verbunden ist, zugreifen, vorausgesetzt, er hat die Privilegien dazu.

Es gibt mehrere Gründe, warum Sie Schemas vielleicht verwenden wollen:

- ❏ um es mehreren Benutzern zu gestatten, eine Datenbank zu verwenden, ohne sich gegenseitig zu stören
- ❏ um Datenbankobjekte in logische Gruppen aufzuteilen, um sie besser verwalten zu können.
- ❏ Anwendungen von Fremdanbietern können in getrennten Schemas abgelegt werden, damit sie keine Konflikte mit den Namen anderer Objekte verursachen.

Schemas entsprechen Verzeichnissen auf Betriebssystemebene, außer dass man Schemas nicht verschachteln kann.

5.7.1 Ein Schema erzeugen

Um ein getrenntes Schema zu erzeugen, verwenden Sie den Befehl `CREATE SCHEMA`. Geben Sie dem Schema einen Namen Ihrer Wahl. Zum Beispiel:

```
CREATE SCHEMA meinschema;
```

Um ein Objekt in einem Schema zu erzeugen oder auf eines zuzugreifen, schreiben Sie einen **qualifizierten Namen**, welcher aus dem Schemanamen und dem Tabellennamen, getrennt von einem Punkt, besteht:

```
schema.tabelle
```

Die noch allgemeinere Syntax

datenbank`.`*schema*`.`*tabelle*

kann im Prinzip auch verwendet werden, was aber gegenwärtig nur zur Pro-forma-Übereinstimmung mit dem SQL-Standard vorgesehen ist. Wenn Sie einen Datenbanknamen angeben, muss das der Name der Datenbank sein, mit der Sie gerade verbunden sind.

Um also eine Tabelle in dem neuen Schema zu erzeugen, verwenden Sie

```
CREATE TABLE meinschema.meinetabelle (
 ...
);
```

Das funktioniert überall, wo ein Tabellenname erwartet wird, einschließlich der Tabellenmodifikationsbefehle und der Datenzugriffsbefehle, die in den folgenden Kapiteln besprochen werden.

Um ein Schema zu entfernen, wenn es leer ist (alle darin enthaltenen Objekte sind gelöscht worden), verwenden Sie

```
DROP SCHEMA meinschema;
```

Um ein Schema einschließlich aller darin enthaltenen Objekte zu löschen, verwenden Sie

```
DROP SCHEMA meinschema CASCADE;
```

Abschnitt 5.9 beschreibt den Mechanismus, der dieser Aktion zugrunde liegt.

Oft werden Sie ein Schema, das jemandem anderen gehören soll, erstellen wollen (da dies eine Möglichkeit ist, die Aktivitäten Ihrer Benutzer auf bestimmte Namensräume zu beschränken). Die Syntax dafür ist:

```
CREATE SCHEMA schemaname AUTHORIZATION benutzername;
```

Sie können den Schemanamen auch auslassen, wodurch der Schemaname mit dem Benutzernamen gleichgesetzt wird. In Abschnitt 5.7.6 werden Sie sehen, wie nützlich das sein kann.

Schemanamen, die mit `pg_` beginnen, sind für Systemaufgaben reserviert und können von Benutzern nicht erzeugt werden.

5.7.2 Das Schema public

In den vorangegangenen Abschnitten haben wir Tabellen erzeugt, ohne einen Schemanamen anzugeben. Solche Tabellen (und andere Objekte) werden in der Voreinstellung automatisch in einem Schema namens `public` abgelegt. Jede neue Datenbank enthält so ein Schema. Die folgenden Befehle sind also gleichbedeutend:

```
CREATE TABLE produkte ( ... );
```

und

```
CREATE TABLE public.produkte ( ... );
```

5.7.3 Der Schema-Suchpfad

Qualifizierte Namen zu schreiben kann, lästig werden und oft ist es auch besser, die Schemanamen nicht fest in eine Anwendung einzubauen. Deshalb schreibt man Tabellennamen oft als **unqualifizierte Namen**, welche einfach aus dem Tabellennamen bestehen. Das System ermittelt, welche Tabelle gemeint ist, indem es dem **Suchpfad** folgt. Der Suchpfad ist eine Liste von Schemas, in der das System nachschauen soll. Die erste passende Tabelle, die im Suchpfad gefunden wurde, wird verwendet. Wenn es keine passende Tabelle im Suchpfad gibt, dann wird ein Fehler erzeugt, selbst wenn es passende Tabellennamen in anderen Schemas der Datenbank gibt.

Das erste Schema, das im Suchpfad steht, wird als das aktuelle Schema bezeichnet. Es ist nicht nur das erste Schema, das durchsucht wird, sondern dort werden auch neue mit `CREATE TABLE` erzeugte Tabellen abgelegt, wenn der Befehl keinen Schemanamen angibt.

Um den aktuellen Schemasuchpfad anzuzeigen, verwenden Sie folgenden Befehl:

```
SHOW search_path;
```

In der Voreinstellung ergibt das:

```
 search_path
--------------
 $user,public
```

Das erste Element gibt an, dass ein Schemaname mit dem gleichen Namen wie der aktuelle Benutzer zu durchsuchen ist. Da ein solches Schema noch nicht existiert, wird dieser Eintrag ignoriert. Das zweite Element bezieht sich auf das bereits gesehene Schema `public`.

Das erste Schema im Suchpfad, das tatsächlich existiert, ist das Schema, wo neue Objekte erstellt werden, wenn kein anderes Schema ausdrücklich angegeben ist. Deshalb werden Objekte in der Voreinstellung im Schema `public` erstellt. Wenn ein Objekt in einem anderen Zusammenhang (Tabellenveränderung, Datenmodifikation, Anfragebefehl) ohne Schemaqualifikation verwendet wird, dann wird der Suchpfad abgesucht, bis ein passendes Objekt gefunden wurde. Deshalb können unqualifizierte Namen in den Voreinstellungen auch nur auf das Schema `public` zugreifen.

Um unser neues Schema in den Pfad zu setzen, verwenden wir

```
SET search_path TO meinschema,public;
```

(Wir haben `$user` hier ausgelassen, weil wir es gegenwärtig nicht brauchen.) Danach können wir auf die Tabelle ohne Qualifikation zugreifen:

```
DROP TABLE meinetabelle;
```

Da `meinschema` auch das erste Element im Pfad ist, würden dort neue Objekte erstellt werden, wenn kein anderes Schema angegeben wird.

Wir hätten auch schreiben können

```
SET search_path TO meinschema;
```

Dann hätten wir keinen Zugriff auf das Schema `public` mehr gehabt, ohne es ausdrücklich zu nennen. Das Schema `public` ist nichts Besonderes, außer dass es in der Ausgangskonfiguration schon vorhanden ist. Sie können es auch ganz entfernen.

In Abschnitt 9.13 finden Sie auch noch weitere Möglichkeiten, um auf den Schemasuchpfad zuzugreifen.

Der Suchpfad funktioniert auf die gleiche Art und Weise für Datentypnamen, Funktionsnamen und Operatornamen, wie er für Tabellennamen funktioniert. Die Namen von Datentypen und Funktionen können auf die gleiche Art qualifiziert werden wie Tabellennamen. Wenn Sie in einem Ausdruck einen qualifizierten Operatornamen schreiben müssen, dann gibt es eine Sonderregel: Sie müssen schreiben

`OPERATOR(`*schema*`.`*operator*`)`

Das ist erforderlich, um syntaktische Zweideutigkeiten auszuschließen. Ein Beispiel wäre

SELECT 3 OPERATOR(pg_catalog.+) 4;

In der Praxis verwendet man normalerweise den Suchpfad für Operatoren, damit man nicht so etwas Umständliches schreiben muss.

5.7.4 Schemas und Privilegien

In der Voreinstellung kann ein Benutzer die Objekte in Schemas, die ihm nicht gehören, nicht sehen. Um das zu erlauben, muss der Eigentümer des Schemas das Privileg `USAGE` gewähren. Um den anderen Benutzern auch zu erlauben, die Objekte in dem Schema zu verwenden, müssen eventuell noch weitere Privilegien gewährt werden, je nach Objekttyp.

Man kann einem Benutzer auch gestatten, Objekte im Schema von jemand anderem zu erzeugen. Um das zu erlauben, muss das Privileg `CREATE` für das Schema gewährt werden. Beachten Sie, dass in der Voreinstellung jeder das `CREATE`-Privileg für das Schema `public` hat. Das ermöglicht allen Benutzern, die es schaffen, sich mit einer bestimmten Datenbank zu verbinden, dort Objekte zu erzeugen. Wenn Sie das nicht gestatten wollen, dann können Sie das Privileg entziehen:

```
REVOKE CREATE ON public FROM PUBLIC;
```

(Das erste `public` ist das Schema, das zweite `public` bedeutet “alle Benutzer”. Im ersten Sinne ist es ein Name, im zweiten Sinne ist es ein Schlüsselwort, daher die unterschiedliche Schreibung. Erinnern Sie sich an die Richtlinien aus Abschnitt 4.1.1.)

5.7.5 Das Systemkatalogschema

Neben `public` und benutzerdefinierten Schemas enthält jede Datenbank ein Schema namens `pg_catalog`, welches die Systemtabellen und die eingebauten Datentypen, Funktionen und Operatoren enthält. `pg_catalog` ist immer automatisch Teil des Suchpfads. Wenn es im Pfad nicht ausdrücklich erwähnt wird, dann wird es automatisch *vor* den Schemas im Pfad durchsucht. Damit wird sichergestellt, dass die eingebauten Namen im auffindbar sind. Sie können `pg_catalog` allerdings ausdrücklich ans Ende Ihres Suchpfads setzen, wenn Sie möchten, dass benutzerdefinierte Namen vor den eingebauten Vorrang haben.

In PostgreSQL-Versionen vor 7.3 waren Tabellennamen, die mit `pg_` anfangen, reserviert. Das ist nicht mehr der Fall: Sie können solche Tabellen erzeugen, wenn Sie wünschen, außer im Systemkatalogschema. Es ist allerdings besser, solche Namen weiterhin zu vermeiden, damit auch in der Zukunft keine Konflikte entstehen, wenn einmal eine neue Systemtabelle den gleichen Namen wie eine Ihrer Tabellen haben sollte. (Mit dem voreingestellten Suchpfad würde sich ein unqualifizierter Verweis auf Ihren Tabellennamen in Wahrheit auf die Systemtabelle beziehen.) Die Systemtabellen werden weiterhin Namen haben, die mit `pg_` beginnen, und somit keine Konflikte mit benutzerdefinierten Namen verursachen, wenn der Präfix `pg_` von Benutzern vermieden wird.

5.7.6 Verwendungsmöglichkeiten

Mit Schemas können Sie Ihren Daten auf viele Arten organisieren. Es gibt einige Verwendungsmöglichkeiten, die empfehlenswert und mit den Voreinstellungen einfach umzusetzen sind:

- Wenn Sie keine Schemas erzeugen, greifen alle Benutzer automatisch auf das Schema `public` zu. Dadurch simulieren Sie eine Umgebung, in der gar keine Schemas vorhanden sind. Diese Gliederung ist hauptsächlich zu empfehlen, wenn Sie nur einen einzigen oder einige wenige zusammen arbeitende Benutzer haben. Mit diesen Einstellungen können Sie auch den Übergang von einer Umgebung ohne Schemas in eine Umgebung mit Schemas problemlos bewältigen.
- Sie können für jeden Benutzer ein Schema mit dem gleichen Namen wie der Benutzer erzeugen. Erinnern Sie sich zurück, dass der voreingestellte Suchpfad mit `$user` anfängt, was für den aktuellen Benutzernamen steht. Wenn also jeder Benutzer ein eigenes Schema hat, dann greift automatisch jeder auf sein eigenes zu.
- Wenn Sie diese Gliederung verwenden, sollten Sie vielleicht auch den Zugriff auf das Schema `public` entziehen (oder es gleich ganz löschen), damit die Benutzer wirklich auf ihr eigenes Schema beschränkt werden.
- Um Anwendungen zu installieren, die von allen verwendet werden können (zum Beispiel gemeinsame Tabellen oder zusätzliche Funktionen von Fremdanbietern), legen Sie sie in getrennten Schemas ab. Vergessen Sie nicht, die passenden Privilegien zu gewähren, damit andere Benutzer darauf Zugriff haben. Die Benutzer können dann auf diese Objekte zugreifen, indem Sie qualifizierte Namen schreiben, oder Sie können die zusätzlichen Schemas in ihren Pfad aufnehmen, je nachdem, wie sie es wünschen.

5.7.7 Portierbarkeit

Im SQL-Standard ist die Vorstellung, das Objekte im gleichen Schema verschiedene Eigentümer haben können, nicht vorhanden. Außerdem gestatten es einige Systeme nicht, Schemas zu erzeugen, deren Name nicht mit dem Name des Eigentümers übereinstimmt. Tatsächlich sind die Begriffe Schema und Benutzer in Datenbanksystemen, die nur die einfachste Schemaunterstützung, die im Standard vorgesehen ist, anbieten, nahezu gleichbedeutend. Deshalb denken viele Benutzer, dass qualifizierte Namen in Wirklichkeit so aussehen: *`benutzername.tabellenname`*. So verhält sich PostgreSQL im Endeffekt auch, wenn Sie für jeden Benutzer ein eigenes Schema erzeugen.

Ferner ist darauf hinzuweisen, dass im SQL-Standard kein Schema `public` vorgesehen ist. Um dem Standard so weit wie möglich zu folgen, sollten Sie das Schema `public` nicht verwenden (und vielleicht sogar löschen).

Natürlich bieten womöglich manche SQL-Datenbanksysteme Schemas gar nicht an oder sie erlauben die Aufteilung der Objekte in Namensräume, indem sie (möglicherweise eingeschränkten) Zugriff auf mehrere Datenbanken gleichzeitig erlauben. Wenn Sie mit solchen Systemen zusammenarbeiten müssen, erreichen Sie die beste Portierbarkeit, wenn Sie Schemas gar nicht verwenden.

5.8 Andere Datenbankobjekte

Tabellen sind die zentralen Objekte einer relationalen Datenbankstruktur, weil sie es sind, die Ihre Daten enthalten. Aber sie sind nicht die einzige Art von Objekt, dass in einer Datenbank zu finden ist. Viele andere Arten von Objekten können erzeugt werden, um die Verwendung und Verwaltung der Daten effizienter und bequemer zu gestalten. Diese werden in diesem Kapitel nicht besprochen, aber wir führen Sie hier auf, um Ihnen zu zeigen, was möglich ist.

- Sichten
- Funktionen, Operatoren, Datentypen, Domänen
- Trigger und Umschreiberegeln (*Rules*)

5.9 Verfolgung von Abhängigkeiten

Wenn Sie komplexe Datenbankstrukturen aus vielen Tabellen mit Fremdschlüsseln, Sichten, Triggern, Funktionen usw. erzeugen, dann erzeugen Sie automatisch ein Netz von Abhängigkeiten zwischen den Objekten. Eine Tabelle mit einem Fremdschlüssel ist zum Beispiel von der Tabelle abhängig, auf die der Fremdschlüssel verweist.

Um die Integrität der gesamten Datenbankstruktur abzusichern, verhindert PostgreSQL, dass Sie Objekte entfernen, von denen andere Objekte abhängen. Wenn Sie zum Beispiel die Produkttabelle, die wir in Abschnitt 5.4.5 betrachtet hatten, entfernen wollen und die Bestellungstabelle ist noch von ihr abhängig, dann würde das eine Fehlermeldung wie diese verursachen:

```
DROP TABLE produkte;

NOTICE:  constraint $1 on table bestellungen depends on table produkte
ERROR:  Cannot drop table produkte because other objects depend on it
        Use DROP ... CASCADE to drop the dependent objects too
```

Diese Fehlermeldung enthält einen nützlichen Hinweis in der letzten Zeile: Wenn Sie die anhängigen Objekte nicht alle einzeln entfernen wollen, können Sie

```
DROP TABLE produkte CASCADE;
```

ausführen und die anhängigen Objekte werden mit entfernt. In diesem Fall wird die Bestellungstabelle dadurch nicht gelöscht, sondern nur der Fremdschlüssel-Constraint. (Wenn Sie überprüfen wollen, was `DROP ... CASCADE` für Auswirkungen haben würde, dann führen Sie `DROP` ohne `CASCADE` aus und lesen Sie die `NOTICE`-Mitteilungen.)

Alle `DROP`-Befehle in PostgreSQL unterstützen die `CASCADE`-Option. Die Art der möglichen Abhängigkeiten ist natürlich bei jedem Objekttyp verschieden. Sie können statt `CASCADE` auch `RESTRICT` schreiben, um das normale Verhalten zu erreichen, welches das Entfernen von Objekten mit Abhängigkeiten verhindert.

Anmerkung

Laut SQL-Standard muss entweder `RESTRICT` oder `CASCADE` immer angegeben werden. Es gibt kein Datenbanksystem, das dies wirklich verlangt, aber es ist unterschiedlich, ob das Normalverhalten `RESTRICT` oder `CASCADE` entspricht.

Anmerkung

Abhängigkeiten von Fremdschlüsseln und Spalten vom Typ serial aus PostgreSQL-Versionen vor 7.3 werden im Upgrade-Prozess *nicht* erhalten oder erzeugt. Alle anderen Abhängigkeiten werden beim Upgrade ordnungsgemäß erstellt.

6

Datenmanipulation

Im vorangegangenen Kapitel haben wir besprochen, wie man Tabellen und andere Strukturen erzeugt, um Ihre aufzunehmen. Jetzt ist es an der Zeit, die Tabellen mit Daten zu füllen. Dieses Kapitel behandelt, wie man Tabellendaten einfügt, aktualisiert und löscht. Wir werden auch Wege vorstellen, wie man automatische Datenmanipulationen verursachen kann, wenn bestimmte Ereignisse auftreten: Trigger und Umschreiberegeln. Das Kapitel nach diesem wird dann endlich erklären, wie Sie Ihre lange verschollenen Daten wieder aus der Datenbank herausbekommen.

6.1 Daten einfügen

Wenn eine Tabelle erzeugt wird, enthält sie keine Daten. Das Erste, was man machen muss, bevor eine Datenbank von Nutzen sein kann, ist Daten einzufügen. Daten werden Zeile für Zeile eingefügt. Natürlich können Sie auch mehrere Zeilen einfügen, aber es ist nicht möglich, weniger als eine Zeile einzufügen. Selbst wenn Sie nur einige Spaltenwerte wissen, müssen Sie eine ganze Zeile erzeugen.

Um eine neue Zeile einzufügen (englisch *insert*), verwenden Sie den Befehl `INSERT`. Dieser Befehl benötigt den Namen der Tabelle und Werte für jede der Spalten in der Tabelle. Betrachten Sie zum Beispiel die Produkttabelle ausKapitel 5:

```
CREATE TABLE produkte (
    prodult_nr integer,
    name text,
    preis numeric
);
```

Ein Beispielbefehl, um eine Zeile einzufügen, wäre:

```
INSERT INTO produkte VALUES (1, 'Käse', 9.99);
```

Die Werte werden in der Reihenfolge aufgelistet, in der die Spalten in der Tabelle angeordnet sind, und werden durch Kommas getrennt. Meistens werden die Werte Konstante sein, aber Wertausdrücke sind auch zulässig.

Die obige Syntax hat den Nachteil, dass man die Reihenfolge der Spalten in der Tabelle kennen muss. Um das zu vermeiden, können Sie die Spalten auch ausdrücklich aufzählen. Zum Beispiel haben beide der folgenden Befehle die gleiche Wirkung wie der obige:

```
INSERT INTO produkte (produkt_nr, name, preis) VALUES (1, 'Käse', 9.99);
INSERT INTO produkte (name, preis, produkt_nr) VALUES ('Käse', 9.99, 1);
```

Viele Benutzer halten es für empfehlenswert, die Spalten immer aufzulisten.

Wenn Sie nicht für alle Spalten Werte haben, können Sie einige auslassen. In diesem Fall werden die Spalten mit ihren Vorgabewerten gefüllt. Zum Beispiel:

```
INSERT INTO produkte (produkt_nr, name) VALUES (1, 'Käse');
INSERT INTO produkte VALUES (1, 'Käse');
```

Die zweite Form ist eine Erweiterung von PostgreSQL. Sie füllt die Spalten von links mit so vielen Werten auf, wie angegeben wurden, und weist den Übrigen die Vorgabewerte zu.

Der Klarheit halber können Sie die Vorgabewerte auch ausdrücklich anfordern, entweder für einzelne Spalten oder für die ganze Zeile:

```
INSERT INTO produkte (produkt_nr, name, preis) VALUES (1, 'Käse', DEFAULT);
INSERT INTO produkte DEFAULT VALUES;
```

Tipp

Um viele Daten auf einmal zu laden (so genanntes *Bulk Load*), schauen Sie sich den Befehl `COPY` an (siehe Teil VI). Er ist nicht ganz so vielseitig wie der `INSERT`-Befehl, aber effizienter..

6.2 Daten aktualisieren

Das Modifizieren von Daten, die sich bereits in der Datenbank befinden, nennt man Aktualisieren (englisch *update*). Sie können einzelne Zeilen, alle Zeilen einer Tabelle oder eine Teilmenge aller Zeilen aktualisieren. Jede Spalte kann getrennt aktualisiert werden; auf die anderen Spalten hat das keine Auswirkung.

Um eine Aktualisierung durchzuführen, brauchen Sie drei Informationen:

- die Namen der Tabelle und der Spalten, die Sie aktualisieren möchten,
- den neuen Wert der Spalte,
- welche Zeile(n) Sie aktualisieren wollen.

In Kapitel 5 wurde bereits erwähnt, dass SQL generell keine eindeutige Identifikation der Zeilen zulässt. Daher ist es nicht immer unbedingt möglich, direkt anzugeben, welche Zeile zu aktualisieren ist. Vielmehr geben Sie die Bedingungen an, die eine Zeile erfüllen muss, um aktualisiert zu werden. Wenn die Tabelle einen Primärschlüssel hat (ganz gleich, ob Sie Ihn ausdrücklich angegeben haben), können Sie einzelne Zeilen verlässlich auswählen, indem Sie eine Bedingung verwenden, die mit dem Primärschlüssel übereinstimmt. Anwendungen, die grafischen Zugriff auf die Datenbank gewähren, machen sich diese Tatsache zunutze, um Ihnen zu gestatten, einzelne Zeilen zu aktualisieren.

Dieser Befehl zum Beispiel aktualisiert den Preis aller Produkte, die einen Preis von 5 haben, auf 10:

```
UPDATE produkte SET preis = 10 WHERE preis = 5;
```

Je nach Umstand könnten dadurch null, eine oder mehrere Zeilen aktualisiert werden. Eine Aktualisierung, die am Ende auf keine Zeilen zutrifft, ist kein Fehler.

Schauen wir uns den Befehl genauer an: Am Anfang steht das Schlüsselwort `UPDATE` gefolgt vom Namen der Tabelle. Wie gewohnt kann der Tabellenname mit einem Schemanamen qualifiziert werden, ansonsten wird er im Pfad gesucht. Als Nächstes kommt das Schlüsselwort `SET` gefolgt vom Spaltennamen, einem Gleichheitszeichen und dem neuen Wert der Spalte. Der neue Spaltenwert kann ein beliebiger Wertausdruck sein, nicht nur eine Konstante. Wenn Sie zum Beispiel den Preis aller Produkte um 10% erhöhen wollen, dann könnten Sie das so machen:

```
UPDATE produkte SET preis = preis * 1.10;
```

Wie Sie sehen, kann sich der Ausdruck für den neuen Wert auch auf den alten Wert beziehen. Wir haben hier auch die `WHERE`-Klausel ausgelassen. Wenn sie fehlt, dann werden alle Zeilen in der Tabelle aktualisiert. Wenn sie da ist, dann werden nur jene Zeilen aktualisiert, die die Bedingung nach der `WHERE`-Klausel erfüllen. Beachten Sie, dass das Gleichheitszeichen in der `SET`-Klausel eine Wertzuweisung ist, während das in der `WHERE`-Klausel ein Vergleich ist; aber dadurch kann keine Zweideutigkeit entstehen. Natürlich muss die Bedingung kein Test auf Gleichheit sein. Viele andere Operatoren stehen zur Verfügung (siehe Kapitel 9). Aber der Ausdruck muss ein Boole'sches Ergebnis haben.

Sie können in einem `UPDATE`-Befehl auch mehrere Spalten aktualisieren, indem Sie mehrere Zuweisungen in der `SET`-Klausel auflisten. Zum Beispiel:

```
UPDATE meinetabelle SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3 Daten löschen

Soweit haben wir erläutert, wie man Daten in Tabellen einfügt und wie man Daten ändert. Was noch übrig bleibt, ist zu besprechen, wie man Daten, die man nicht länger braucht, löschen kann (englisch *delete*). Genauso wie es nur möglich ist, ganze Zeilen einzufügen, kann man nur ganze Zeilen löschen. Im vorigen Abschnitt haben wir besprochen, dass es in SQL nicht möglich ist, einzelne Zeilen direkt anzusprechen. Deshalb kann man Zeilen nur löschen, indem man eine Bedingung angibt, die die zu löschenden Zeilen erfüllen müssen. Wenn die Tabelle einen Primärschlüssel hat, dann können Sie eine Zeile genau auswählen. Aber Sie können auch eine Gruppe von Zeilen, die eine Bedingung erfüllen, oder alle Zeilen in einer Tabelle auf einmal löschen.

Man verwendet den Befehl `DELETE`, um Zeilen zu löschen; die Syntax ist dem `UPDATE`-Befehl sehr ähnlich. Um zum Beispiel alle Zeilen der Produkttabelle zu löschen, die eine Preis von 10 haben, verwenden Sie

```
DELETE FROM produkte WHERE preis = 10;
```

Wenn Sie einfach schreiben

```
DELETE FROM produkte;
```

werden alle Zeilen in der Tabelle gelöscht. Sein Sie vorsichtig!

7 Anfragen

Die vorangegangenen Kapitel haben erklärt, wie man Tabellen erzeugt, sie mit Daten füllt und diese Daten manipulieren kann. Jetzt besprechen wir endlich, wie man diese Daten wieder aus der Datenbank auslesen kann.

7.1 Überblick

Den Vorgang oder der Befehl, Daten aus der Datenbank auszulesen, nennt man eine **Anfrage** (englisch *query*). In SQL wird der Befehl `SELECT` dazu verwendet, Anfragen zu formulieren. Die allgemeine Syntax des `SELECT`-Befehls ist

```
SELECT select_liste FROM tabellen_ausdruck [sortierung]
```

Die folgenden Abschnitte beschreiben die Select-Liste, den Tabellenausdruck und die Sortierangabe im Einzelnen.

Die einfachste Anfrage hat die Form

```
SELECT * FROM tabelle1;
```

Vorausgesetzt, es gibt eine Tabelle namens `tabelle1`, ergibt dieser Befehl alle Zeilen und alle Spalten aus `tabelle1`. (Die genaue Art und Weise der Ergebnisausgabe hängt von der Clientanwendung ab. Das `psql`-Programm gibt die Tabelle zum Beispiel mit ASCII-Zeichen aus und Clientbibliotheken bieten Funktionen an, um auf einzelne Zeilen und Spalten zuzugreifen.) Eine Select-Liste, die nur aus `*` besteht, steht für alle Spalten, die der Tabellenausdruck gerade zur Verfügung stellt. Eine Select-Liste kann auch nur einige der verfügbaren Spalten auswählen oder mit den Spalten Berechnungen vornehmen. Wenn zum Beispiel `tabelle1` Spalten namens `a`, `b` und `c` hat (und möglicherweise noch weitere), können Sie die folgende Anfrage durchführen:

```
SELECT a, b + c FROM tabelle1;
```

(vorausgesetzt, dass `b` und `c` einen Zahlentyp haben). Einzelheiten finden Sie in Abschnitt 7.3.

`FROM table1` ist ein besonders einfacher Tabellenausdruck: Er liest nur aus einer Tabelle. Prinzipiell können Tabellenausdrücke komplexe Konstruktionen aus Basistabellen, Verbunde und Unteranfragen sein. Sie können den Tabellenausdruck auch ganz auslassen und den `SELECT`-Befehl als Taschenrechner verwenden:

```
SELECT 3 * 4;
```

Das ist aber sinnvoller, wenn die Ausdrücke in der Select-Liste veränderliche Ergebnisse haben. Sie könnten auf diese Art zum Beispiel eine Funktion aufrufen:

```
SELECT random();
```

7.2 Tabellenausdrücke

Ein **Tabellenausdruck** berechnet eine Tabelle. Der Tabellenausdruck enthält eine FROM-Klausel, wahlweise gefolgt von WHERE-, GROUP BY- und HAVING-Klauseln. Einfache Tabellenausdrücke beziehen sich ganz einfach auf eine Tabelle auf der Festplatte, eine so genannte Basistabelle, aber komplexere Ausdrücke können angewendet werden, um Basistabellen auf verschiedene Arten zu verändern oder zu kombinieren.

Die optionalen WHERE-, GROUP BY- und HAVING-Klauseln im Tabellenausdruck ergeben sozusagen ein Fließband aufeinander folgender Transformationen, die sich auf die in der FROM-Klausel ermittelte Tabelle auswirken. Diese Transformationen erzeugen eine virtuelle Tabelle, die die Zeilen enthält, die in der Select-Liste verwendet werden, um die Ergebniszeilen der Anfrage zu berechnen.

7.2.1 Die FROM-Klausel

Die FROM-Klausel errechnet eine Tabelle aus einer oder mehreren Tabellen, die durch Kommas getrennt in der Liste aufgezählt werden.

```
FROM tabellen_verweis [, tabellen_verweis [, ...]]
```

Ein Tabellenverweis kann ein Tabellenname (möglicherweise mit Schemaname) oder eine abgeleitete Tabelle, wie zum Beispiel eine Unteranfrage, ein Tabellenverbund oder eine komplexe Kombination daraus sein. Wenn mehrere Tabellenverweise in der FROM-Klausel aufgelistet werden, wird der Kreuzverbund (*cross join*) der Tabellen ermittelt (siehe unten). Diese Tabelle ist dann die virtuelle Zwischentabelle, die den Transformationen in den WHERE-, GROUP BY- und HAVING-Klauseln zugeführt wird und schließlich das Ergebnis des Tabellenausdrucks ist.

Wenn ein Tabellenverweis eine Tabelle nennt, die eine Übertabelle einer Vererbungshierarchie ist, dann erzeugt der Tabellenverweis nicht nur die Zeilen der Tabelle selbst, sondern auch die aller Untertabellen, außer wenn das Schlüsselwort ONLY vor dem Tabellennamen steht. Allerdings erzeugt der Tabellenverweis nur die Spalten, die in der genannten Tabelle vorhanden sind. Spalten, die in Untertabellen hinzugefügt wurden, werden nicht berücksichtigt.

Verbundene Tabellen

Eine verbundene Tabelle wird von zwei anderen (echten oder abgeleiteten) Tabellen nach den Regeln des jeweiligen Verbundtyp abgeleitet. Es gibt innere, äußere und Kreuzverbunde.

Verbundtypen

Kreuzverbund (*cross join*)

```
T1 CROSS JOIN T2
```

Für jede Kombination aus Zeilen von *T1* und *T2* wird die abgeleitete Tabelle eine Zeile enthalten, die aus allen Spalten in *T1* gefolgt von allen Spalten in *T2* besteht. Wenn die Tabellen also N beziehungsweise M Zeilen haben, dann hat die verbundene Tabelle N * M Zeilen.

`FROM` *`T1`* `CROSS JOIN` *`T2`* ist gleichbedeutend mit `FROM` *`T1, T2`*. Es ist auch gleichbedeutend mit `FROM` *`T1`* `INNER JOIN` *`T2`* `ON TRUE` (siehe unten).

Bedingte Verbunde

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolescher_ausdruck
T1 { [INNER]  |  { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( spaltenliste )
T1  NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Die Worte `INNER` und `OUTER` sind in allen Formen optional. `INNER` ist die Vorgabe; wenn `LEFT`, `RIGHT` oder `FULL` angegeben sind, dann wird `OUTER` angenommen.

Die **Verbundbedingung** wird in der `ON`- oder der `USING`-Klausel oder implizit mit dem Wort `NATURAL` angegeben. Die Verbundbedingung bestimmt, welche Zeilen der zwei Ausgangstabellen zusammenpassen, wie im Einzelnen weiter unten besprochen wird.

Die `ON`-Klausel ist die allgemeinste Art der Verbundbedingung: Sie enthält einen Wertausdruck mit einem Ergebnis vom Typ boolean, genauso, wie er in einer `WHERE`-Klausel verwendet wird. Ein Zeilenpaar aus *`T1`* und *`T2`* passt zusammen, wenn der Ausdruck nach `ON` logisch wahr ist.

`USING` ist eine abgekürzte Schreibweise: Sie enthält eine Liste aus Spaltennamen, getrennt durch Kommas, welche beide der zu verbindenden Tabellen haben müssen; die Verbundbedingung ist, dass all diese Spalten gleich sein müssen. Das Ergebnis von `JOIN USING` hat eine Spalte für jede der gleichgesetzten Eingabespalten, gefolgt von den übrigen Spalten der Tabellen. Also ist `USING (a, b, c)` gleichbedeutend mit `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)`, außer man verwendet ON, dann hat das Ergebnis je zwei Spalten `a`, `b` und `c`, anstatt nur je eine mit `USING`.

Schließlich ist `NATURAL` eine Abkürzung von `USING`: Es steht für eine `USING`-Liste mit genau den Spaltennamen, die in beiden Tabellen vorhanden sind. Genauso wie bei `USING` erscheinen diese Spalten nur einmal im Ergebnis.

Es gibt die folgenden Arten von bedingten Verbunden:

Innerer Verbund (`INNER JOIN`)

Für jede Zeile Z1 der Tabelle T1 hat die verbundene Tabelle eine Zeile für jede Zeile in T2, die die Verbundbedingung mit Z1 erfüllt.

Linker äußerer Verbund (`LEFT OUTER JOIN`)

Als Erstes wird ein innerer Verbund ausgeführt. Danach wird für jede Zeile in T1, die die Verbundbedingung mit keiner Zeile in T2 erfüllt, eine Ergebniszeile erzeugt, die NULL-Werte in den Spalten von T2 hat. Folglich hat die verbundene Tabelle auf jeden Fall eine Zeile für jede Zeile in T1.

Rechter äußerer Verbund (`RIGHT OUTER JOIN`)

Als Erstes wird ein innerer Verbund ausgeführt. Danach wird für jede Zeile in T2, die die Verbundbedingung mit keiner Zeile in T1 erfüllt, eine Ergebniszeile erzeugt, die NULL-Werte in den Spalten von T1 hat. Das ist das Gegenteil eines linken Verbunds: Die Ergebnistabelle hat auf jeden Fall eine Zeile für jede Zeile in T2.

Voller äußerer Verbund (`FULL OUTER JOIN`)

Als Erstes wird ein innerer Verbund durchgeführt. Danach wird für jede Zeile in T1, die die Verbundbedingung mit keiner Zeile in T2 erfüllt, eine Ergebniszeile erzeugt, die NULL-Werte in den Spalten von T2 hat. Ebenso wird für jede Zeile in T2, die die Verbundbedingung mit keiner Zeile in T1 erfüllt, eine Ergebniszeile mit NULL-Werten in den Spalten von T1 erzeugt.

Verbunde aller Typen können verkettet oder verschachtelt werden, das heißt, sowohl *`T1`* als auch *`T2`* können selbst verbundene Tabellen sein. Um die Verbundreihenfolge zu bestimmen, kann man Klammern um die `JOIN`-Klauseln setzen. Ansonsten werden die `JOIN`-Klauseln von links nach rechts ausgewertet.

Um all das zusammenzufassen, nehmen wir an, wir haben Tabellen `t1`

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

Dann erhalten wir folgende Ergebnisse für die verschiedenen Verbunde:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | wert
-----+------+------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | wert
-----+------+------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
```

```
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | wert
-----+------+------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

Die Verbundbedingung, die nach `ON` angegeben wird, kann auch Bedingungen enthalten, die nicht direkt mit dem Verbund zu tun haben. Das kann in manchen Anfragen nützlich sein, muss aber genau durchdacht werden. Zum Beispiel:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.wert = 'xxx';
 num | name | num | wert
-----+------+-----+------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Tabellen- und Spaltenaliasnamen

Man kann Tabellen und komplexen Tabellenverweisen einen vorübergehenden Namen geben, der in der weiteren Verarbeitung verwendet werden kann, um auf die abgeleitete Tabelle zu verweisen. Das wird **Tabellenaliasname** genannt.

Um ein Tabellenaliasnamen zu erzeugen, verwenden Sie

```
FROM tabellen_verweis AS alias
```

oder

```
FROM tabellen_verweis alias
```

Das Schlüsselwort AS kann also weggelassen werden. *alias* kann ein beliebiger Name sein.

Eine typische Verwendung von Tabellenaliasnamen ist es, langen Tabellennamen kürzere Bezeichnungen zu geben, um die Verbundbedingungen lesbar zu halten. Zum Beispiel:

```
SELECT * FROM ein_sehr_langer_tabellenname e JOIN noch_ein_ziemlich_langer_name n
ON e.id = n.num;
```

Wenn ein Aliasname verwendet wird, dann wird es der neue Name des Tabellenverweises für die gegenwärtige Anfrage. Es ist dann nicht mehr möglich, auf die Tabelle mit ihrem ursprünglichen Namen zu verweisen. Folgendes wäre also keine gültige SQL-Syntax:

```
SELECT * FROM meine_tabelle AS m WHERE meine_tabelle.a > 5;
```

Was wirklich passiert (das ist eine Erweiterung von PostgreSQL gegenüber dem Standard), ist dass ein impliziter Tabellenverweis zur FROM-Klausel hinzugefügt wird, sodass die Anfrage verarbeitet wird, also wäre sie so geschrieben worden:

```
SELECT * FROM meine_tabelle AS m, meine_tabelle AS meine_tabelle WHERE
meine_tabelle.a > 5;
```

Das ergibt einen Kreuzverbund, was Sie in der Regel sicher nicht wollen.

Tabellenaliasnamen sind hauptsächlich dafür da, um Anfragen übersichtlicher und kürzer schreiben zu können. Aber man muss sie verwenden, wenn eine Tabelle mit sich selbst verbunden werden soll, zum Beispiel:

```
SELECT * FROM meine_tabelle AS a CROSS JOIN meine_tabelle AS b ...
```

Außerdem sind Aliasnamen notwendig, wenn der Tabellenverweis eine Unteranfrage ist (siehe Abschnitt *Unteranfragen*).

Klammern können verwendet werden, um Zweideutigkeiten auszuschließen. Der folgende Befehl weist den Aliasnamen b dem Verbundergebnis zu, im Gegensatz zum vorigen Beispiel:

```
SELECT * FROM (meine_tabelle AS a CROSS JOIN meine_tabelle) AS b ...
```

Zusammen mit Tabellenaliasnamen kann man auch den Spalten einer Tabelle vorübergehende Namen zuweisen:

```
FROM tabellen_verweis [AS] alias ( spalte1 [, spalte2 [, ...]] )
```

Wenn weniger Spaltenaliasnamen angegeben sind, als die eigentliche Tabelle Spalten hat, dann werden die restlichen Spalten nicht umbenannt. Diese Syntax ist besonders für Selbstverbunde und Unteranfragen nützlich.

Wenn dem Ergebnis eines Verbunds mit JOIN ein Aliasname zugewiesen wird, dann werden dadurch die ursprünglichen Namen innerhalb der Verbundklausel versteckt. Folgender Befehl ist zum Beispiel in Ordnung:

```
SELECT a.* FROM meine_tabelle AS a JOIN deine_tabelle AS b ON ...
```

Aber folgender Befehl ist ungültig:

```
SELECT a.* FROM (meine_tabelle AS a JOIN deine_tabelle AS b ON ...) AS c
```

Der Tabellenaliasname a ist außerhalb des Aliasnamen c nicht sichtbar.

Unteranfragen

Unteranfragen, die als abgeleitete Tabelle aufgeführt werden, müssen in Klammern stehen und ihnen *muss* eine Tabellenaliasname zugewiesen werden. (Siehe Abschnitt *Tabellen- und Spaltenaliasnamen*.) Zum Beispiel:

```
FROM (SELECT * FROM tabelle1) AS alias_name
```

Dieses Beispiel hat das gleiche Ergebnis wie `FROM tabelle1 AS alias_name`. Interessantere Fälle, die nicht auf eine einfache Verbundoperation reduziert werden können, treten auf, wenn die Unteranfrage eine Gruppierung oder eine Aggregatberechnung enthält.

Tabellenfunktionen

Tabellenfunktionen sind Funktionen, die eine Ergebnismenge aus mehreren Zeilen zurückgeben, entweder aus Basisdatentypen (skalaren Typen) oder aus zusammengesetzten Datentypen (Tabellenzeilen). Sie werden wie eine Tabelle, Sicht oder Unteranfrage in der `FROM`-Klausel einer Anfrage verwendet. Die von einer Tabellenfunktion zurückgegebenen Spalten können genauso in `SELECT`-, `JOIN`- oder `WHERE`-Klauseln verwendet werden wie Spalten aus Tabellen, Sichten oder Unteranfragen.

Wenn die Tabellenfunktion einen Basisdatentyp zurückgibt, erhält die einzelne Ergebnisspalte denselben Namen wie die Funktion. Wenn die Funktion einen zusammengesetzten Typ zurückgibt, erhalten die Ergebnisspalten dieselben Namen wie die einzelnen Attribute des Typs.

Eine Tabellenfunktion kann in der `FROM`-Klausel Aliasnamen erhalten, kann aber auch ohne Aliasnamen belassen werden. Wenn eine Funktion in der `FROM`-Klausel ohne Aliasname belassen wird, dann wird der Name der Funktion als Name der Ergebnistabelle verwendet.

Einige Beispiele:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (select foosubid from getfoo(foo.fooid) z
                       where z.fooid = foo.fooid);

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
SELECT * FROM vw_getfoo;
```

In manchen Fällen ist es nützlich, eine Tabellenfunktion zu definieren, die, je nachdem, wie sie aufgerufen wird, einen anderen Satz Spalten zurückgibt. Dazu kann eine Tabellenfunktion mit dem Pseudotyp record als Rückgabetyp definiert werden. Wenn eine solche Funktion in einer Anfrage verwendet wird, muss die erwartete Zeilenstruktur in der Anfrage selbst angegeben werden, damit das System weiß, wie die Anfrage verstanden und geplant werden soll. Betrachten Sie dieses Beispiel:

```
SELECT *
FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
```

```
    AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

Die Funktion `dblink` führt eine Anfrage in einer anderen Datenbank aus (siehe `contrib/dblink`). Sie hat den Rückgabetyp record, weil Sie für jede Art von Anfrage verwendet werden kann. Die tatsächlich erwarteten Spalten müssen in der aufrufenden Anfrage angegeben werden, damit der Parser zum Beispiel weiß, wie * interpretiert werden soll.

7.2.2 Die WHERE-Klausel

Die Syntax der `WHERE`-Klausel ist

```
WHERE suchbedingung
```

wobei die *suchbedingung* ein beliebiger Wertausdruck, wie in Abschnitt 4.2 definiert, ist, der ein Ergebnis vom Typ boolean hat.

Nachdem die Verarbeitung der `FROM`-Klausel abgeschlossen ist, wird jede Zeile der virtuellen Tabelle mit der Suchbedingung geprüft. Wenn das Ergebnis der Bedingung logisch wahr ist, wird die Zeile in der Ergebnistabelle behalten, ansonsten (das heißt das Ergebnis ist logisch falsch oder der NULL-Wert), dann wird sie verworfen. Die Suchbedingung bezieht sich normalerweise auf mindestens ein paar der Spalten der in der `FROM`-Klausel erzeugten Tabelle; das ist zwar nicht zwingend, aber ansonsten wäre die `WHERE`-Klausel ziemlich nutzlos.

Anmerkung

Vor der Verwirklichung der `JOIN`-Syntax war es notwendig, die Verbundbedingung in die `WHERE`-Klausel zu schreiben. Die folgenden Tabellenausdrücke sind zum Beispiel alle gleichbedeutend:

und

```
FROM a, b WHERE a.id = b.id AND b.wert > 5
```

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.wert > 5
```

oder vielleicht sogar

```
FROM a NATURAL JOIN b WHERE b.wert > 5
```

Welche dieser Formen Sie verwenden, ist hauptsächlich eine Stilfrage. Die `JOIN`-Syntax in der `FROM`-Klausel ist wahrscheinlich nicht ganz so portierbar auf andere SQL-Datenbanken. Für äußere Verbunde gibt es keine Wahl: Sie müssen in der `FROM`-Klausel geschrieben werden. Die `ON`/`USING`-Klausel eines äußeren Verbunds kann nicht als `WHERE`-Bedingung geschrieben werden, weil damit nicht nur Zeilen herausgefiltert, sondern auch zum Endergebnis hinzufügt werden (für Zeilen die keine passende Zeile in der anderen Tabelle haben).

Hier sind einige Beispiele von `WHERE`-Klauseln:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` ist hier irgendeine in der `FROM`-Klausel ermittelte Tabelle. Zeilen, die die Bedingung in der `WHERE`-Klausel nicht erfüllen, werden aus `fdt` entfernt. Beachten Sie die Verwendung von skalaren Unteranfragen als Wertausdrücke. Unteranfragen können genauso wie andere Anfragen komplexe Tabellenausdrücke verwenden. Beachten Sie auch, wie auf `fdt` in den Unteranfragen Bezug genommen wird. Die Qualifikation von `c1` als `fdt.c1` ist nur notwendig, wenn `c1` auch eine Spalte in der abgeleiteten Eingabetabelle der Unteranfrage ist. Aber die Qualifikation des Spaltennamens fördert die Übersichtlichkeit, selbst wenn sie nicht zwingend erforderlich ist. Dieses Beispiel zeigt, wie der Namensraum der Spalten der äußeren Anfrage sich auf die inneren Anfragen ausdehnt.

7.2.3 Die GROUP BY- und HAVING-Klauseln

Nachdem die abgeleitete Tabelle den `WHERE`-Filter durchlaufen hat, kann die Tabelle mit der `GROUP BY`-Klausel gruppiert und Gruppen mit der `HAVING`-Klausel eliminiert werden.

```
SELECT select_liste
    FROM ...
    [WHERE ...]
    GROUP BY spalten_gruppierungs_verweis [, spalten_gruppierungs_verweis]...
```

Die `GROUP BY`-Klausel wird dazu verwendet, jene Zeilen einer Tabelle zusammen zu gruppieren, die die gleichen Werte in allen aufgezählten Spalten haben. Die Reihenfolge, in der die Spalten aufgelistet werden, ist unerheblich. Der Zweck des Ganzen ist es, jede Gruppe von Zeilen, die die gleichen Werte haben, auf eine Gruppenzeile zu reduzieren, die für alle Zeilen in der Gruppe stehen kann. Das kann man verwenden, um doppelte Ergebniszeilen zu entfernen und/oder um Aggregatberechnungen an den Gruppen vorzunehmen. Zum Beispiel:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Die zweite Anfrage hätte man nicht als `SELECT * FROM test1 GROUP BY x` schreiben können, weil es keinen einzelnen Wert der Spalte y gibt, der der Gruppe hätte zugeordnet werden können. Die Spalten, nach denen gruppiert wurde, können in der Select-Liste verwendet werden, da sie einen bekannten Wert pro Gruppe haben.

Generell gilt, wenn eine Tabelle gruppiert wird, dann können die Spalten, nach denen nicht gruppiert wird, nur in Aggregatausdrücken verwendet werden. Ein Beispiel mit einem Aggregatausdruck ist:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Hier ist `sum` eine Aggregatfunktion, die einen einzelnen Wert aus den Werten der gesamten Gruppe errechnet. Weitere Informationen über die verfügbaren Aggregatfunktionen können Sie in Abschnitt 9.14 finden.

Tipp

Gruppierung ohne Aggregatausdruck berechnet im Endeffekt die Menge der voreinander verschiedenen Werte in einer Spalte. Das kann man auch mit der `DISTINCT`-Klausel erreichen (siehe Abschnitt 7.3.3).

Hier ist noch ein Beispiel: Es berechnet den Gesamtumsatz für jedes Produkt (und nicht etwa für alle Produkte zusammen).

```
SELECT produkt_nr, p.name, (sum(v.menge) * p.preis) AS umsatz
FROM produkte p LEFT JOIN verkäufe v USING (produkt_nr)
GROUP BY produkt_nr, p.name, p.preis;
```

In diesem Beispiel müssen die Spalten `produkt_nr`, `p.name` und `p.preis` in der `GROUP BY`-Klausel stehen, weil sie auch in der Select-Liste der Anfrage verwendet werden. (Je nachdem, wie die Produkttabelle genau aufgebaut ist, könnten Name und Preis völlig von der Produktnummer abhängen, sodass die zusätzlichen Gruppierungen unnötig wären, aber diese Erkennung ist noch nicht verwirklicht.) Die Spalte `v.menge` muss nicht in der `GROUP BY`-Klausel stehen, da sie nur in einem Aggregatausdruck (`sum(...)`) verwendet wird, welcher für den Gesamtumsatz mit einem Produkt steht. Für jedes Produkt ergibt die Anfrage eine zusammenfassende Zeile für den Umsatz dieses Produkts.

Im strikten SQL kann `GROUP BY` nur auf Spalten in der Ausgangstabelle angewendet werden, aber in PostgreSQL kann man `GROUP BY` auch mit Spalten aus der Select-Liste verwenden. Das Gruppieren nach Wertausdrücken anstelle von einfachen Spaltenverweisen ist auch möglich.

Wenn eine Tabelle mit `GROUP BY` gruppiert wurde, aber nur bestimmte Gruppen von Interesse sind, dann kann die `HAVING`-Klausel ähnlich wie die `WHERE`-Klausel verwendet werden, um Gruppen aus der gruppierten Tabelle zu entfernen. Die Syntax ist:

```
SELECT select_liste FROM ... [WHERE ...] GROUP BY ... HAVING boolescher_ausdruck
```

Ausdrücke in der `HAVING`-Klausel können sowohl die gruppierten als auch ungruppierte Ausdrücke (welche dann notwendigerweise eine Aggregatfunktion beinhalten) verwenden.

Ein Beispiel:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
```

up ...

... up ... update

Nutzen Sie den UPDATE-SERVICE des mitp-Teams bei vmi-Buch.

Registrieren Sie sich JETZT!

Unsere Bücher sind mit großer Sorgfalt erstellt. Wir sind stets darauf bedacht, Sie mit den aktuellsten Inhalten zu versorgen, weil wir wissen, dass Sie gerade darauf großen Wert legen. Unsere Bücher geben den top-aktuellen Wissens- und Praxisstand wieder.

Um Sie auch über das vorliegende Buch hinaus regelmäßig über die relevanten Entwicklungen am IT-Markt zu informieren, haben wir einen besonderen Leser-Service eingeführt.

Lassen Sie sich professionell, zuverlässig und fundiert auf den neuesten Stand bringen.
Registrieren Sie sich jetzt auf www.mitp.de oder **www.vmi-buch.de** und Sie erhalten zukünftig einen E-Mail-Newsletter mit Hinweisen auf Aktivitäten des Verlages wie zum Beispiel unsere aktuellen, kostenlosen Downloads.

Ihr Team von mitp

```
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

Nochmal ein etwas realistischeres Beispiel:

```
SELECT produkt_nr, p.name, (sum(v.menge) * (p.preis - p.kosten)) AS profit
    FROM produkte p LEFT JOIN verkäufe v USING (produkt_nr)
    WHERE v.datum > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY produkt_nr, p.name, p.preis, p.kosten
    HAVING sum(p.preis * v.menge) > 5000;
```

Im obigen Beispiel wählt die `WHERE`-Klausel Zeilen aus, indem sie eine ungruppierte Spalte verwendet (der Ausdruck ist nur für Verkäufe in den letzten vier Wochen wahr), während die `HAVING`-Klausel das Ergebnis auf Gruppen, deren Umsatz größer als 5000 war, beschränkt. Beachten Sie auch, dass die Aggregatausdrücke nicht unbedingt in allen Teilen der Anfrage gleich sein müssen.

7.3 Select-Listen

Wie im vorigen Abschnitt gezeigt wurde, ermittelt der Tabellenausdruck im `SELECT`-Befehl eine virtuelle Zwischentabelle, indem möglicherweise Tabellen und Sichten kombiniert wurden, Zeilen entfernt oder gruppiert wurden usw. Diese Tabelle wird letztendlich der **Select-Liste** zur Verarbeitung übergeben. Die Select-Liste bestimmt, welche *Spalten* der Zwischentabelle tatsächlich ausgegeben werden.

7.3.1 Elemente der Select-Liste

Die einfachste Select-Liste ist `*`, welche alle Spalten ausgibt, die der Tabellenausdruck erzeugt. Ansonsten ist die Select-Liste eine Liste von Wertausdrücken (wie in Abschnitt 4.2 definiert), durch Kommas voneinander getrennt. Sie könnte zum Beispiel eine Liste von Spaltennamen sein:

```
SELECT a, b, c FROM ...
```

Die Spaltennamen `a`, `b` und `c` sind entweder die wirklichen Namen von Spalten der in der `FROM`-Klausel verwendeten Tabellen oder die Aliasnamen, die ihnen wie in Abschnitt *Tabellen- und Spaltenaliasnamen* zugewiesen wurden. Der Namensraum, der in der Select-Liste zur Verfügung steht, ist der gleiche wie in der `WHERE`-Klausel, außer wenn eine Gruppierung vorgenommen wird, dann ist es der gleiche wie in der `HAVING`-Klausel.

Wenn mehrere Tabellen eine Spalte mit dem gleichen Namen haben, dann muss der Tabellenname auch angegeben werden, wie in ws

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

(Siehe auch Abschnitt 7.2.2.)

Wenn ein beliebiger Wertausdruck in der Select-Liste verwendet wird, kann man sich das so vorstellen, dass eine neue virtuelle Spalte an die Ergebnistabelle angefügt wird. Der Ausdruck wird für jede Ergebniszeile ausgewertet, wobei die Werte der Zeile für etwaige Spaltenverweise eingesetzt werden. Aber die Ausdrücke in der Select-Liste müssen nicht unbedingt eine Spalte aus dem Tabellenausdruck aus der `FROM`-Klausel verwenden; sie können zum Beispiel auch konstante mathematische Ausdrücke sein.

7.3.2 Ergebnisspaltennamen

Den Einträgen in der Select-Liste kann man Namen für die weitere Verarbeitung geben. Die »weitere Verarbeitung« ist in diesem Fall eine mögliche Sortierklausel und Clientanwendungen (zum Beispiel als Spaltenköpfe, wenn die Ergebnistabelle ausgegeben wird). Zum Beispiel:

```
SELECT a AS wert, b + c AS summe FROM ...
```

Wenn kein Spaltenname mit `AS` angegeben wird, dann wird vom System ein Name zugewiesen. Für einfache Spaltenverweise ist das der Namen der Spalte. Für Funktionsaufrufe ist das der Name der Funktion. Für komplexe Ausdrücke wird vom System ein Name erzeugt.

Anmerkung

Die hier beschriebene Vergabe von Namen an die Spalten ist nicht das Gleiche wie die Vergabe von Aliasnamen in der `FROM`-Klausel (siehe Abschnitt *Tabellen- und Spaltenaliasnamen*). Durch diesen Ablauf können Sie eine Spalte praktisch zweimal umbenennen, aber der Name, der in der Select-Liste bestimmt wird, wird am Ende weitergegeben.

7.3.3 DISTINCT

Nachdem die Select-Liste verarbeitet wurde, können aus der Ergebnistabelle wahlweise etwaige Duplikate entfernt werden. Dazu wird das Schlüsselwort `DISTINCT` gleich hinter das `SELECT` geschrieben:

```
SELECT DISTINCT select_liste ...
```

(Anstelle von `DISTINCT` kann das Wort `ALL` verwendet werden, um das normale Verhalten, dass alle Zeilen beibehalten werden, auszuwählen.)

Selbstverständlich gelten zwei Zeilen als dann verschieden voneinander, wenn sie sich in mindestens einem Spaltenwert voneinander unterscheiden. NULL-Werte gelten in dieser Betrachtung als gleich.

Als Alternative kann auch ein beliebiger Ausdruck bestimmen, welche Zeilen als gleich betrachtet werden sollen:

```
SELECT DISTINCT ON (ausdruck [, ausdruck ...]) select_liste ...
```

Hierbei ist *ausdruck* ein beliebiger Wertausdruck, der für alle Zeilen ausgewertet wird. Die Zeilen, für die alle Ausdrücke gleich sind, werden als gleich betrachtet, und nur die erste der Zeilen wird in der Ergebnismenge behalten. Beachten Sie, dass die »erste Zeile« einer Menge nicht vorhergesagt werden kann, außer wenn die Anfrage nach ausreichend vielen Spalten sortiert wird, damit die Zeilen in einer eindeutigen Reihenfolge am `DISTINCT`-Filter ankommen. (Die Verarbeitung von `DISTINCT ON` geschieht nach der Sortierung durch `ORDER BY`.)

Die `DISTINCT ON`-Klausel ist nicht Teil des SQL-Standards und manche halten sie wegen der möglicherweise unvorhersehbaren Ergebnisse für schlechten Stil. Wenn man `GROUP BY` und Unteranfragen in der `FROM`-Klausel geschickt anwendet, kann man diese Klausel vermeiden, aber sie ist oft eine bequeme Alternative.

7.4 Anfragen kombinieren

Die Ergebnisse zweier Anfragen können mit Mengenoperationen vereinigt werden. Die möglichen Operationen sind Vereinigungsmenge (englisch *union*), Schnittmenge (*intersection*) und Differenzmenge. Die Syntax ist

```
anfrage1 UNION [ALL]
anfrage2 anfrage1 INTERSECT [ALL]
anfrage2 anfrage1 EXCEPT [ALL] anfrage2
```

anfrage1 und *anfrage2* sind Anfragen, die von jeder der bisher beschriebenen Fähigkeiten Gebrauch machen können. Mengenoperationen können auch verschachtelt und verkettet werden, wie zum Beispiel:

```
anfrage1 UNION anfrage2 UNION anfrage3
```

Das bedeutet das Gleiche wie:

```
(anfrage1 UNION anfrage2) UNION anfrage3
```

`UNION`, die Vereinigungsmengenoperation, hängt das Ergebnis von *anfrage2* im Prinzip an das Ergebnis von *anfrage1* an (obwohl es keine Gewähr dafür gibt, dass die Zeilen auch in dieser Reihenfolge zurückgegeben werden). Darüber hinaus werden alle doppelten Zeilen, im Sinne von `DISTINCT`, entfernt, außer wenn `UNION ALL` verwendet wird.

`INTERSECT`, die Schnittmengenoperation, ermittelt alle Zeilen, die sowohl im Ergebnis von *anfrage1* als auch im Ergebnis von *anfrage2* sind. Doppelte Zeilen werden entfernt, außer wenn `INTERSECT ALL` verwendet wird.

`EXCEPT`, die Differenzmengenoperation, ermittelt alle Zeilen, die im Ergebnis von *anfrage1,* aber nicht im Ergebnis von *anfrage2* sind. Doppelte Zeilen werden wiederum entfernt, außer wenn `EXCEPT ALL` verwendet wird.

Um die Vereinigungsmenge, Schnittmenge oder Differenzmenge zweier Anfrage berechnen zu können, müssen die Anfragen vereinigungskompatibel (englisch *union compatible*) sein, was heißt, dass beide die gleiche Anzahl von Spalten ergeben und dass die entsprechenden Spalten kompatible Datentypen haben, wie in Abschnitt 10.5 erklärt.

7.5 Zeilen sortieren

Nachdem eine Anfrage eine Ergebnistabelle erzeugt hat (nachdem die Select-Liste verarbeitet wurde), kann das Ergebnis wahlweise sortiert werden. Wenn das Sortieren nicht gewählt wird, dann werden die Zeilen in zufälliger Reihenfolge zurückgegeben. Die tatsächliche Reihenfolge hängt in diesem Fall von den Scan- und Verbund-Plantypen und der Reihenfolge auf der Festplatte ab, aber darauf darf man sich auf keinen Fall verlassen. Eine bestimmte Reihenfolge kann nur gewährleistet werden, wenn der Sortierschritt ausdrücklich angefordert wird.

Die `ORDER BY`-Klausel gibt die Sortierreihenfolge an:

```
SELECT select_liste
    FROM tabellen_ausdruck
    ORDER BY spalte1 [ASC | DESC] [, spalte2 [ASC | DESC] ...]
```

spalte1 usw. verweisen auf Spalten in der Select-Liste. Sie können entweder der Ausgabename einer Spalte (siehe Abschnitt 7.3.2) oder die Nummer einer Spalte sein. Einige Beispiele:

```
SELECT a, b FROM tabelle1 ORDER BY a;
SELECT a + b AS summe, c FROM tabelle1 ORDER BY summe;
SELECT a, sum(b) FROM tabelle1 GROUP BY a ORDER BY 1;
```

Als Erweiterung gegenüber dem SQL-Standard erlaubt PostgreSQL die Sortierung nach beliebigen Ausdrücken:

```
SELECT a, b FROM tabelle1 ORDER BY a + b;
```

Verweise auf Spaltennamen aus der FROM-Klausel, die in der Select-Liste umbenannt wurden, sind auch erlaubt:

```
SELECT a AS b FROM tabelle1 ORDER BY a;
```

Aber diese Erweiterungen funktionieren nicht in Anfragen mit UNION, INTERSECT oder EXCEPT und sind nicht auf andere SQL-Datenbanken portierbar.

Nach jeder Spaltenangabe kann wahlweise ASC oder DESC stehen, um die Sortierrichtung auf aufsteigend (englisch *ascending*) bzw. abfallend (*descending*) zu setzen. Die aufsteigende Reihenfolge stellt kleinere Werte an den Anfang, wobei »kleiner« durch den Operator < festgelegt wird. Entsprechend wird die abfallende Reihenfolge durch den Operator > festgelegt.

Wenn mehrere Spalten zum Sortieren angegeben werden, dann werden die hinteren Einträge verwendet, um Zeilen zu sortieren, die der von den vorderen Einträgen festgelegten Reihenfolge nach gleich sind.

7.6 LIMIT und OFFSET

LIMIT und OFFSET ermöglichen Ihnen, nur einen Teil der Zeilen, die von der übrigen Anfrage erzeugt werden, zurückzugeben:

```
SELECT select_liste
    FROM tabellen_ausdruck
    [LIMIT { zahl | ALL }] [OFFSET zahl]
```

Wenn eine Limit-Zahl angegeben wird, dann werden höchstens so viele Zeilen zurückgegeben (aber möglicherweise weniger, falls die Anfrage selbst weniger Zeilen liefert). LIMIT ALL hat die gleiche Auswirkung, als wenn man die LIMIT-Klausel auslässt.

OFFSET bedeutet, dass so viele Zeilen ausgelassen werden, bevor Zeilen zurückgegeben werden. OFFSET 0 hat die gleiche Auswirkung, als wenn man die OFFSET-Klausel auslässt. Wenn sowohl OFFSET als auch LIMIT verwendet werden, dann werden so viele Zeilen ausgelassen, wie in OFFSET angegeben, bevor die Zählung für LIMIT beginnt.

Wenn LIMIT verwendet wird, dann ist es empfehlenswert, eine ORDER BY-Klausel zu verwenden, die die Ergebniszeilen in eine eindeutige Ordnung bringt. Ansonsten erhalten Sie eine nicht vorhersagbare Teilmenge der Ergebniszeilen der Anfrage. Sie könnten zum Beispiel die Zeilen 10 bis 20 anfordern, aber 10 bis 20 nach welcher Reihenfolge? Die Reihenfolge ist unbekannt, wenn ORDER BY nicht verwendet wird.

Der Anfrageoptimierer zieht die LIMIT-Klausel mit in Betracht, wenn er einen Ausführungsplan für eine Anfrage erstellt, was heißt, dass man sehr wahrscheinlich unterschiedliche Pläne (die unterschiedliche Zeilenreihenfolgen ergeben) erhält, wenn man die LIMIT- und OFFSET-Werte verändert. Wenn man folglich

die `LIMIT`- und `OFFSET`-Werte variiert, um verschiedene Teilmengen eines Anfrageergebnisses auszuwählen, *wird man widersprüchliche Ergebnisse erhalten*, wenn man keine voraussagbare Reihenfolge mit `ORDER BY` erzwingt. Das ist kein Fehler; das Verhalten folgt aus der Tatsache, dass SQL keine Gewähr dafür gibt, dass das Ergebnis einer Anfrage in einer bestimmten Reihenfolge abgeliefert wird, außer wenn `ORDER BY` verwendet wird, um die Reihenfolge zu beeinflussen.

8

Datentypen

PostgreSQL stellt seinen Benutzern eine reichhaltige Menge eingebauter Datentypen zur Verfügung. Benutzer können weitere Typen mit `CREATE TYPE` zu PostgreSQL hinzufügen.

Tabelle 8.1 zeigt alle eingebauten, zur allgemeinen Verwendung gedachten Datentypen. Die meisten alternativen Namen in der Spalte "Alias" sind interne Namen, die von PostgreSQL aus historischen Gründen verwendet werden. Darüber hinaus gibt es einige für interne Zwecke gebrauchte und einige veraltete Typen, die hier nicht gezeigt sind.

Name	Alias	Beschreibung
`bigint`	`int8`	8-Byte-Ganzzahl mit Vorzeichen
`bigserial`	`serial8`	selbstzählende 8-Byte-Ganzzahl
`bit`		Bitkette mit fester Länge
`bit varying(n)`	`varbit(n)`	Bitkette mit variabler Länge
`boolean`	`bool`	Boole'scher (logischer) Wert (wahr/falsch)
`box`		Rechteck
`bytea`		binäre Daten
`character varying(n)`	`varchar(n)`	Zeichenkette mit variabler Länge
`character(n)`	`char(n)`	Zeichenkette mit fester Länge
`cidr`		IP-Netzwerkadresse
`circle`		Kreis in der Ebene
`date`		Kalenderdatum (Jahr, Monat, Tag)
`double precision`	`float8`	Fließkommazahl mit doppelter Präzision
`inet`		IP-Hostadresse
`integer`	`int, int4`	4-Byte-Ganzzahl mit Vorzeichen
`interval(p)`		Zeitspanne
`line`		Gerade in der Ebene (nicht voll implementiert)
`lseg`		endliche Strecke in der Ebene
`macaddr`		MAC-Adresse

Tabelle 8.1: Datentypen

Name	Alias	Beschreibung
money		Geldbetrag
numeric [(p, s)]	decimal [(p, s)]	exakte Zahl mit wählbarer Präzision
path		offener oder geschlossener Pfad in der Ebene
point		geometrischer Punkt in der Ebene
polygon		Polygon
real	float4	Fließkommazahl mit einfacher Präzision
smallint	int2	2-Byte-Ganzzahl mit Vorzeichen
serial	serial4	selbstzählende 4-Byte-Ganzzahl
text		Zeichenkette mit variabler Länge
time [(p)] [without time zone]		Tageszeit
time [(p)] with time zone	timetz	Tageszeit mit Zeitzone
timestamp [(p)] without time zone	timestamp	Datum und Zeit
timestamp [(p)] [with time zone]	timestamptz	Datum und Zeit mit Zeitzone

Tabelle 8.1: Datentypen (Forts.)

Kompatibilität: Die folgenden Typen (oder deren Schreibweisen) sind von SQL vorgesehen: `bit`, `bit varying`, `boolean`, `char`, `character varying`, `character`, `varchar`, `date`, `double precision`, `integer`, `interval`, `numeric`, `decimal`, `real`, `smallint`, `time`, `timestamp` (beide mit und ohne `with time zone`).

Jeder Datentyp hat eine externe Darstellung, die von seinen Eingabe- und Ausgabefunktionen bestimmt wird. Viele eingebaute Datentypen haben offensichtliche externe Formate. Einige Typen sind allerdings nur in PostgreSQL vorhanden, wie zum Beispiel offene oder geschlossene Pfade, oder haben mehrere mögliche Eingabeformate, wie zum Beispiel die Datums- und Zeittypen. Einige Eingabe- und Ausgabefunktionen sind nicht umkehrbar. Das heißt, dass das Ergebnis einer Ausgabefunktion gegenüber dem ursprünglichen Wert an Genauigkeit verlieren kann.

Einige Operatoren und Funktionen (z.B. Addition und Multiplikation) führen zur Laufzeit keine Fehlerprüfung durch, um die Geschwindigkeit zu verbessernAuf einigen Systemen können die numerischen Operationen bei einigen Datentypen zum Beispiel ohne Benachrichtigung Überlauf oder Unterlauf verursachen.

8.1 Numerische Typen

Numerische Typen bestehen aus 2-, 4- und 8-Byte-Ganzzahlen (englisch *integer*), 4- und 8-Byte-Fließkommazahlen (*floating point*) und Zahlen mit Dezimalbrüchen mit fester Präzision. Tabelle 8.2 listet die verfügbaren Typen.

Name	Speichergröße	Beschreibung	Reichweite
smallint	2 Bytes	ganze Zahl mit kleiner Reichweite	-32768 bis +32767
integer	4 Bytes	normale Wahl für ganze Zahlen	-2147483648 bis +2147483647

Tabelle 8.2: Numerische Typen

Name	Speichergröße	Beschreibung	Reichweite
bigint	8 Bytes	ganze Zahl mit großer Reichweite	-9223372036854775808 bis 9223372036854775807
decimal	variabel	benutzerdefinierte Präzision, exakt	keine Begrenzung
numeric	variabel	benutzerdefinierte Präzision, exakt	keine Begrenzung
real	4 Bytes	variable Präzision, inexakt	6 Dezimalstellen Präzision
double precision	8 Bytes	variable Präzision, inexakt	15 Dezimalstellen Präzision
serial	4 Bytes	selbstzählende ganze Zahl	1 bis 2147483647
bigserial	8 Bytes	große selbstzählende ganze Zahl	1 bis 9223372036854775807

Tabelle 8.2: Numerische Typen (Forts.)

Die Syntax für numerische Typen ist in Abschnitt 4.1.2 beschrieben. Die numerischen Typen haben einen vollen Satz entsprechender Operatoren und Funktionen. Schauen Sie in Kapitel 9 für weitere Informationen. Die folgenden Abschnitte beschreiben die Typen im Einzelnen.

8.1.1 Ganzzahlentypen

Die Typen smallint, integer und bigint speichern ganze Zahlen (englisch *integer*), das heißt Zahlen ohne Nachkommastellen, mit unterschiedlichen Reichweiten. Versuche, einen Wert außerhalb des Gültigkeitsbereichs zu speichern, verursachen einen Fehler.

Der Typ integer ist die häufigste Wahl, da er die beste Balance zwischen Reichweite, Größe im Speicher und Effizienz bietet. Der Typ smallint wird normalerweise nur verwendet, wenn der Speicherplatz knapp ist. Der Typ bigint sollte nur verwendet werden, wenn die Reichweite des Typs integer nicht ausreicht, weil Letzterer auf jeden Fall schneller ist.

Der Type bigint funktioniert möglicherweise nicht auf allen Plattformen, weil er voraussetzt, dass der Compiler 8-Byte-Ganzzahlen unterstützt. Auf Maschinen, die diese Unterstützung nicht haben, verhält sich bigint genau so wie integer (aber benötigt trotzdem 8 Bytes Speicherplatz). Allerdings kennen wir keine vernünftige Plattform, wo dies tatsächlich der Fall ist.

SQL gibt nur die Ganzzahlentypen integer (oder int) und smallint an. Der Typ bigint und die Typnamen int2, int4 und int8 sind Erweiterungen, die PostgreSQL aber mit diversen anderen SQL-Datenbanksystemen gemeinsam hat.

Anmerkung

Wenn Sie eine Spalte vom Typ smallint oder bigint mit einem Index haben, dann werden Sie sicher auf Probleme stoßen, das System dazu zu bewegen, von diesem Index Gebrauch zu machen. Eine Klausel, die zum Beispiel eine Form wie

```
... WHERE smallint_spalte = 42
```

hat, wird den Index nicht verwenden, weil das System der Konstante 42 den Typ integer zuweist, und PostgreSQL kann gegenwärtig keine Indexe verwenden, wenn zwei verschiedene Datentypen im Spiel sind. Eine Möglichkeit, dieses Problem zu umgehen, ist, die Konstante in Apostrophe zu setzen, wie hier:

```
... WHERE smallint_spalte = '42'
```

Dadurch zögert das System die Typauflösung etwas hinaus und wird der Konstante den richtigen Typ zuweisen.

8.1.2 Zahlen mit beliebiger Präzision

Der Type numeric kann Zahlen mit bis zu 1000 Ziffern Präzision speichern und exakte Berechnungen ausführen. Er ist besonders empfehlenswert für die Speicherung von Geldbeträgen und anderer Größen, wo Genauigkeit erforderlich ist. Der Typ numeric ist allerdings verglichen mit den im nächsten Abschnitt beschriebenen Fließkommatypen sehr langsam.

Sowohl die Zahl der Nachkommastellen und die Zahl der insgesamt zu speichernden Zifferstellen können angegeben werden. Um eine Spalte vom Typ numeric zu erstellen, verwenden Sie folgende Syntax:

```
NUMERIC(gesamtstellen, nachkommastellen)
```

Die Zahl der Gesamtstellen muss positiv, die der Nachkommastellen null oder positiv sein. Wenn Sie schreiben

```
NUMERIC(stellen)
```

werden null Nachkommastellen vorgesehen. Und wenn Sie ganz einfach schreiben

```
NUMERIC
```

ohne irgendwelche Stellenzahlen, wird eine Spalte erzeugt, in der Zahlen mit einer beliebigen Anzahl Stellen vor oder nach dem Komma gespeichert werden können. Normalerweise werden die Eingabewerte einer numeric-Spalte auf die vorgesehene Anzahl von Nachkommastellen gebracht, außer wenn die Spalte wie im letzten Fall erzeugt wurde. (Der SQL-Standard sieht hier eigentlich vor, dass null Nachkommastellen die Vorgabe sind, das heißt im Endeffekt eine ganze Zahl. Wir halten das für nicht so sinnvoll. Wenn Sie auf Portierbarkeit Wert legen, dann geben Sie die gewünschte Anzahl von Stellen immer an.)

Wenn die Anzahl der Stellen (insgesamt oder nach dem Komma) in einem Wert die angegebenen Grenzen einer Spalte überschreitet, dann wird das System versuchen, den Wert zu runden. Wenn er nicht so gerundet werden kann, dass er in die Grenzen passt, dann ist das ein Fehler.

Die Typen decimal und numeric sind identisch. Beide Typen sind Teil des SQL-Standards.

8.1.3 Fließkommatypen

Die Datentypen real und double precision sind inexakte numerische Typen mit variabler Genauigkeit. In der Praxis sind diese Typen normalerweise Umsetzungen des IEEE-Standards 754 für binäre Fließkomma-Arithmetik (einfache bzw. doppelte Genauigkeit), insoweit das vom Prozessor, Betriebssystem und Compiler unterstützt wird.

Inexakt bedeutet, dass einige Werte nicht exakt in das interne Format umgewandelt werden können und als Annäherungen gespeichert werden, sodass das Speichern und wieder Auslesen eines Wertes leichte Diskrepanzen zeigen kann. Das Verwalten dieser Abweichungen und der Art, wie sie sich in Berechnungen verhalten, ist das Thema eines ganzen Zweigs der Mathematik und der Informatik und soll hier nicht weiter besprochen werden. Wir wollen lediglich folgende Hinweise geben:

- Wenn Sie exakte Speicherung und Berechnungen benötigen (zum Beispiel für Geldbeträge), dann sollten Sie den Typ numeric verwenden.
- Wenn Sie komplizierte Berechnungen mit diesen Typen anstellen und auf ein bestimmtes Verhalten in Grenzfällen (Unendlichkeit, Unterlauf) angewiesen sind, dann sollten Sie die Implementierung dieser Typen ganz genau auswerten.
- Das Vergleichen zweier Fließkommawerte (mit =) muss nicht unbedingt die Ergebnisse liefern, die man erwarten könnte.

Der Typ real hat normalerweise einen Gültigkeitsbereich von mindestens -1E+37 bis +1E+37 und eine Genauigkeit von mindestens sechs Dezimalstellen (in der gesamten Zahl). Der Typ double precision hat

normalerweise etwa einen Bereich von -1E+308 bis +1E+308 mit mindestens 15 Dezimalstellen Genauigkeit. Werte, die zu groß oder zu klein sind, verursachen einen Fehler. Wenn die Genauigkeit eines Eingabewerts zu hoch ist, dann wird versucht, den Wert zu runden. Zahlen, die zu nahe an null sind und nicht als verschieden von null abgespeichert werden können, verursachen einen Unterlauffehler.

8.1.4 Selbstzählende Datentypen

Die Datentypen serial und bigserial sind keine richtigen Datentypen, sondern abgekürzte Schreibweisen, um Spalten einzurichten, die automatisch eine eindeutige Nummerierung erzeugen. In der gegenwärtigen Umsetzung ist

```
CREATE TABLE tabellenname (
 spaltenname serial );
```

gleichbedeutend mit

```
CREATE SEQUENCE tabellenname_spaltenname_seq;
CREATE TABLE tabellenname (
    spaltenname integer DEFAULT nextval('tabellenname_spaltenname_seq') NOT NULL
);
```

Das heißt also, eine integer-Spalte und ihr Vorgabewert werden aus einer Sequenz erzeugt. Außerdem wird ein `NOT NULL`-Constraint erzeugt, um zu verhindern, dass NULL-Werte manuell eingefügt werden können. In der Regel sollten Sie auch einen `UNIQUE`- oder `PRIMARY KEY`-Constraint erzeugen, um zu verhindern, dass aus Versehen doppelte Werte eingefügt werden; das passiert nicht automatisch.

Anmerkung

Vor PostgreSQL 7.3 hatte serial automatisch einen Unique Constraint. Das ist nicht mehr automatisch. Wenn Sie einen Unique Constraint oder einen Primärschlüssel für die Spalte haben wollen, dann müssen Sie es genauso wie bei jedem anderen Datentyp ausdrücklich angeben.

Die Typnamen serial und serial4 sind gleichbedeutend: Beide erzeugen Spalten vom Typ integer. Die Typnamen bigserial und serial8 funktionieren genauso, außer dass sie eine Spalte vom Typ bigint erzeugen. bigserial sollte verwendet werden, wenn Sie erwarten, dass mehr als 2^{31} Nummern innerhalb der Lebensdauer einer Tabelle benötigt werden.

Die Sequenz, die durch einen serial-Typ erzeugt wird, wird automatisch wieder gelöscht, wenn die zugehörige Spalte entfernt wird, und kann auf direktem Wege nicht gelöscht werden. (Das war vor PostgreSQL 7.3 nicht der Fall. Beachten Sie, dass diese automatische Kopplung des Löschens nicht erfolgt, wenn die Sequenz aus einer Dumpdatei einer Datenbank vor 7.3 erzeugt wurde, weil die Dumpdatei die Informationen, die benötigt werden, um die Abhängigkeit zwischen der Sequenz und der Spalte herzustellen, nicht enthält.)

8.2 Typen für Geldbeträge

Anmerkung

Der Typ money ist veraltet. Verwenden Sie anstelle dessen numeric oder decimal in Verbindung mit der Funktion `to_char`.

Der Typ money speichert einen Geldbetrag mit einer festen Nachkommagenauigkeit; siehe Tabelle 8.3. Die Eingabe kann in verschiedenen Formaten getätigt werden, einschließlich Ganzzahlen- und Fließkommazahlen-Konstanten sowie "typischen" Schreibweisen für Geldbeträge, wie zum Beispiel `'$1,000.00'`. Die Ausgabe ist im Allgemeinen im letzteren Format, hängt allerdings von den Locale-Einstellungen ab.

Name	Speichergröße	Beschreibung	Reichweite
money	4 Bytes	Geldbetrag	-21474836.48 bis +21474836.47

Tabelle 8.3: Typen für Geldbeträge

8.3 Zeichenkettentypen

Name	Beschreibung
character varying(*n*), varchar(*n*)	variable Länge mit Höchstgrenze
character(*n*), char(*n*)	feste Länge, mit Leerzeichen aufgefüllt
text	variable Länge ohne Höchstgrenze

Tabelle 8.4: Zeichenkettentypen

Tabelle 8.4 zeigt die zur allgemeinen Verwendung gedachten Zeichenkettentypen in PostgreSQL

SQL definiert zwei Zeichenkettentypen: `character varying`(*n*) und `character`(*n*), wobei *n* eine positive ganze Zahl ist. Beide Typen können Zeichenketten bis zu *n* Zeichen Länge speichern. Der Versuch, eine längere Zeichenkette zu speichern, ergibt einen Fehler, es sei denn, die überschüssigen Zeichen sind alle Leerzeichen, dann werden sie bis zur zugelassenen Länge der Zeichenkette abgeschnitten. (Diese etwas bizarre Ausnahme wird vom SQL-Standard vorgeschrieben.) Wenn eine Zeichenkette zugewiesen wird, die kürzer als die angegebene Länge ist, dann werden Werte vom Typ character mit Leerzeichen aufgefüllt, während Werte vom Typ character varying einfach den kürzeren Wert speichern.

Wenn ein Wert ausdrücklich in den Typ character varying(*n*) oder character(*n*) umgewandelt wird (Cast), dann werden zu lange Werte ohne Fehlermeldung auf *n* Zeichen abgeschnitten. (Das ist auch vom SQL-Standard vorgeschrieben.)

Anmerkung

Vor PostgreSQL 7.2 wurden zu lange Zeichenketten immer ohne Fehlermeldung abgeschnitten, egal in welchem Zusammenhang die Umwandlung oder Zuweisung erfolgte.

Die Schreibweisen `varchar(n)` und `char`(n) sind Aliasnamen für `character varying`(n) bzw. `character`(n). character ohne Längenangabe entspricht `character`(1); wenn character varying ohne Längenangabe verwendet wird, dann akzeptiert der Typ Zeichenketten beliebiger Länge. Letzteres ist eine Erweiterung von PostgreSQL.

Darüber hinaus stellt PostgreSQL den Typ text zur Verfügung, welcher Zeichenketten beliebiger Länge aufnehmen kann. Obwohl der Typ text nicht im SQL-Standard steht, haben mehrere andere SQL-Datenbankprodukte ihn auch.

Die Speicheranforderungen für Daten dieser Typen ist 4 Bytes zuzüglich der eigentlichen Zeichen, und im Falle von character zuzüglich der auffüllenden Leerzeichen. Lange Zeichenketten werden vom System automatisch komprimiert, sodass die Speicherplatzanforderungen auf der Festplatte womöglich geringer sind. Außerdem werden lange Werte in separaten Tabellen gespeichert, damit sie den schnellen Zugriff auf kürzere Spaltenwerte nicht erschweren. Auf jeden Fall ist die längste mögliche Zeichenkette, die gespeichert werden kann, rund 1 GB lang. (Der höchste Wert, der für *n* in der Deklaration des Datentyps ver-

wendet werden kann, ist aber kleiner. Es wäre nicht sonderlich nützlich, das zu ändern, denn mit mehrbytigen Zeichenkodierungen kann die Zahl der Zeichen und die Zahl der Bytes sowieso ziemlich unterschiedlich sein. Wenn Sie längere Zeichenketten ohne bestimmte Obergrenze speichern wollen, dann verwenden Sie text oder character varying ohne Längenangabe, anstatt willkürlich eine Obergrenze zu erfinden.)

Tipp

Es gibt keine Leistungsunterschiede zwischen diesen Typen, außer den höheren Speicheranforderungen des Typs character wegen der Auffüllung mit Leerzeichen.

In Kapitel 4 Abschnitt *Zeichenkettenkonstanten* finden Sie Informationen über die Syntax von Zeichenkettenkonstanten und in Kapitel 9 Informationen über die verfügbaren Operatoren und Funktionen.

Beispiel 8.1: Verwendung der Zeichenkettentypen

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; --
   a | char_length
------+-------------
  ok | 4

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('gut ');
INSERT INTO test2 VALUES ('zu lang');
ERROR: value too long for type character varying(5)
INSERT INTO test2 VALUES ('zu lang'::varchar(5)); -- ausdrückliche Umwandlung
SELECT b, char_length(b) FROM test2;
b      | char_length
-------+-------------
ok     |           2
gut    |           5
zu la  |           5
```

Listing 8.1: Verwendung der Zeichenkettentypen

Die Funktion `char_length` ist in Abschnitt 9.4 besprochen.

Es gibt zwei weitere Typen für Zeichenketten mit fester Länge in PostgreSQL, welche in Tabelle 8.5 gezeigt werden. Der Typ name existiert *ausschließlich* für die Speicherung von Namen in den internen Systemkatalogen und ist nicht für Endanwender vorgesehen. Seine Länge ist 64 Bytes (63 verwendbare Bytes und eine Abschlussmarkierung) definiert und ist in der Konstante *NAMEDATALEN* festgelegt. Die Länge wird bei der Compilierung festgelegt (und kann daher für besondere Zwecke verändert werden); der Standardwert könnte in der Zukunft einmal ein anderer sein. Der Typ "char" (achten Sie auf die Anführungszeichen) unterscheidet sich von char(1) dadurch, dass er nur ein Byte Speicherplatz braucht. In den internen Systemkatalogen wird er quasi als Aufzählungstyp für Arme verwendet.

Name	Speichergröße	Beschreibung
`"char"`	1 Byte	interner Typ für ein Zeichen
`name`	64 Bytes	interner Typ für Objektnamen

Tabelle 8.5: Spezielle Zeichenkettentypen

Die Funktion char_length ist in Abschnitt 9.4 besprochen.

8.4 Typen für binäre Daten

Der Typ bytea ermöglicht die Speicherung von binären Daten; siehe Tabelle 8.6.

Name	Speichergröße	Beschreibung
`bytea`	4 Bytes plus die eigentlichen Daten	binäre Daten mit variabler Länge

Tabelle 8.6: Typen für binäre Daten

Binäre Daten sind eine Reihe von Bytes. Sie unterscheiden sich von Zeichenketten durch zwei Merkmale: Erstens können binäre Daten Bytes mit dem Wert null und andere "nicht druckbare" Bytes enthalten. Zweitens beziehen sich Operationen an binären Daten auf die tatsächlichen Bytes, wogegen die Kodierung und Verarbeitung von Zeichenketten von Locale-Einstellungen abhängt.

Wenn ein bytea-Wert als Zeichenkettenkonstante in einem SQL-Befehl eingegeben wird, *müssen* bestimmte Bytewerte durch Fluchtfolgen (*escape sequences*) umschrieben werden (aber alle Werte *dürfen* so umschrieben werden). Um ein Byte zu umschreiben, wandeln Sie es in eine dreistellige Oktalzahl um und schreiben davor zwei Backslashes. Einige Bytewerte haben alternative Fluchtfolgen, wie in Tabelle 8.7 gezeigt.

Dezimaler Bytewert	Beschreibung	Eingabedarstellung als Fluchtfolge	Beispiel	Ausgabedarstellung
0	Null-Byte	`'\\000'`	SELECT '\\000'::bytea;	\000
39	Apostroph	`'\''` or `'\\047'`	SELECT '\''::bytea;	'
92	Backslash	`'\\\\'` or `'\\134'`	SELECT '\\\\'::bytea;	\\

Tabelle 8.7: Fluchtfolgen für bytea-Konstanten

Beachten Sie, dass das Ergebnis jedes Beispiels in Tabelle 8.7 genau eine Byte Länge hat, obwohl die Ausgabedarstellung des Null-Bytes und des Backslashes mehrere Zeichen lang ist.

Der Grund, dass Sie so viele Backslashes, wie in Tabelle 8.7 gezeigt, schreiben müssen, ist, dass eine Eingabe, die als Zeichenkettenkonstante geschrieben wird, zwei Parseschritte im PostgreSQL-Server durchläuft. Der erste Backslash jedes Paars wird bei der Analyse der Zeichenkettenkonstante als Fluchtzeichen erkannt und wird entfernt, wodurch der zweite Backslash übrig bleibt. Der verbliebene Backslash wird dann von der bytea-Eingabefunktion erkannt, wo er entweder eine Folge von drei Oktalziffern anführt oder als Fluchtzeichen für noch einen Backslash steht. Wenn man zum Beispiel eine Zeichenkettenkonstante `'\\001'` an den Server schickt, bleibt davon nach der Verarbeitung der Zeichenkettenkonstante `\001` übrig. Der Wert `\001` wird dann an die Eingabefunktion des Typs bytea übergeben, wo er in ein einzelnes Byte mit dem Wert 1 verwandelt wird. Beachten Sie, dass das Apostrophzeichen von bytea nicht gesondert behandelt wird, sondern den normalen Regeln für Zeichenkettenkonstanten folgt. (Siehe auch Kapitel 4 Abschnitt *Zeichenkettenkonstanten*.)

Bytea-Werte werden auch in der Ausgabe durch Fluchtfolgen dargestellt. Prinzipiell wird jedes "nicht druckbare" Byte in seinen entsprechenden dreistelligen Oktalwert umgewandelt und ein Backslash vorangestellt. Die meisten "druckbaren" Bytes werden ihre normale Darstellung im Zeichensatz des Clients behalten. Das Byte mit dem Dezimalwert 92 (Backslash) hat eine besondere Darstellung in der Ausgabe. Einzelheiten stehen in Tabelle 8.8.

Dezimaler Bytewert	Beschreibung	Ausgabe durch Fluchtfolge	Beispiel	Ausgabeergebnis
92	Backslash	\\	`SELECT '\\134'::bytea;`	\\
0 bis 31 und 127 bis 255	"nicht druckbare" Bytes	*xxx* (oktaler Wert)	`SELECT '\\001'::bytea;`	\001
32 bis 126	"druckbare" Bytes	ASCII-Zeichen	`SELECT '\\176'::bytea;`	~

Tabelle 8.8: Fluchtfolgen in bytea-Ausgaben

Je nachdem, welche Clientschnittstelle für PostgreSQL Sie verwenden, haben Sie vielleicht noch etwas mehr Arbeit, um die bytea-Werte ordentlich verarbeiten zu können. Wenn zum Beispiel die Zeilenumbrüche automatisch angepasst werden, müssen Sie diese womöglich auch durch Fluchtfolgen darstellen, um das zu verhindern.

Der SQL-Standard sieht einen anderen Typ für binäre Daten vor, nämlich BLOB oder BINARY LARGE OBJECT. Das Eingabeformat unterscheidet sich von bytea, aber die verfügbaren Funktionen und Operatoren sind im Großen und Ganzen gleich.

8.5 Datums- und Zeittypen

PostgreSQL unterstützt die komplette Sammlung an Datums- und Zeittypen aus dem SQL-Standard, welche in Tabelle 8.9 gezeigt werden.

Name	Speichergröße	Beschreibung	Niedrigster Wert	Höchster Wert	Genauigkeit
`timestamp [ (`*p*`) ] [ without time zone ]`	8 Bytes	Datum und Zeit	4713 v.u.Z.	1465001 u.Z.	1 Mikrosekunde / 14 Stellen
`timestamp [ (`*p*`) ] with time zone`	8 Bytes	Datum und Zeit	4713 v.u.Z.	1465001 u.Z.	1 Mikrosekunde / 14 Stellen
`interval [ (`*p*`) ]`	12 Bytes	Zeitspannen	-178000000 Jahre	178000000 Jahre	1 Mikrosekunde
`date`	4 Bytes	nur Datum	4713 v.u.Z.	32767 u.Z.	1 Tag
`time [ (`*p*`) ] [ without time zone ]`	8 Bytes	nur Tageszeit	00:00:00.00	23:59:59.99	1 Mikrosekunde
`time [ (`*p*`) ] with time zone`	12 Bytes	nur Tageszeit	00:00:00.00+12	23:59:59.99-12	1 Mikrosekunde

Tabelle 8.9: Datums- und Zeittypen

Bei den Typen time, timestamp und interval kann wahlweise die Genauigkeit *p* angegeben werden, welche aussagt, wie viele Nachkommastellen im Sekundenfeld abgespeichert werden. Ohne diese Angabe gibt es keine ausdrückliche Beschränkung. Der erlaubte Bereich für *p* ist 0 bis 6 für die Typen timestamp und interval, und 0 bis 13 für die time-Typen.

Anmerkung

Wenn timestamp-Werte als Fließkommazahlen gespeichert werden (gegenwärtig die Vorgabe), kann die tatsächliche Genauigkeit weniger als 6 Stellen sein. Die Werte werden als Sekunden seit dem 1.1.2000 gespeichert und die Mikrosekundengenauigkeit wird für Werte innerhalb einiger Jahre um den 1.1.2000 herum erreicht, aber die Genauigkeit lässt für weiter entfernte Zeitpunkte nach. Wenn timestamp-Werte als 8-Byte-Ganzzahlen gespeichert werden (eine Option bei der Compilierung), dann ist die Mikrosekundengenauigkeit im gesamten Wertebereich verfügbar.

Anmerkung

Vor PostgreSQL 7.3 war timestamp gleichbedeutend mit timestamp with time zone. Das wurde geändert, um dem SQL-Standard zu entsprechen.

Der Typ time with time zone ist vom SQL-Standard definiert, aber die Definition hat einige Eigenschaften, die die Nützlichkeit dieses Typs infrage stellen. In der Regel reichen die Typen date, time, timestamp without time zone und timestamp with time zone für die Anforderungen aller Anwendungen aus.

Die Typen abstime und reltime sind Typen mit niedrigerer Genauigkeit für interne Aufgaben. Sie sollten diese Typen in neuen Anwendungen nicht verwenden und etwaige alte Anwendungen auf die neuen Typen umändern. Diese internen Typen könnten in einer zukünftigen Version verschwinden.

8.5.1 Eingabeformate für Datum und Zeit

Datum und Zeit können in vielen verschiedenen Formaten eingegeben werden, einschließlich ISO 8601, SQL-kompatibel, dem traditionellen POSTGRES-Format und anderer. In einigen Formaten kann die Reihenfolge von Monat und Tag in Datumsangaben zweideutig sein und daher gibt es eine Möglichkeit, die erwartete Reihenfolge dieser Felder einzustellen. Der Befehl `SET datestyle TO 'European'` wählt die Variante “Tag vor Monat”, der Befehl `SET datestyle TO 'US'` oder `SET datestyle TO 'NonEuropean'` wählt die Variante “Monat vor Tag”.

PostgreSQL ist bei der Verarbeitung von Datums- und Zeiteingaben viel flexibler, als der SQL-Standard erfordert. Die genauen Parseregeln finden Sie in Anhang A, wo auch die erlaubten Werte für die Textfelder der Monat, Wochentag und Zeitzone gelistet sind.

Beachten Sie, dass Datums- und Zeitkonstanten wie Zeichenketten zwischen Apostrophen stehen müssen. Genauere Informationen finden Sie in Kapitel 4 Abschnitt *Konstanten anderer Typen*. SQL verlangt die folgende Syntax:

```
typ [ (p) ] 'wert'
```

wobei *p* eine ganze Zahl ist, die wahlweise die Anzahl der Nachkommastellen im Sekundenfeld bestimmt. Die Genauigkeit kann für time, timestamp und interval angegeben werden. Die zulässigen Werte finden Sie oben. Wenn keine Genauigkeit angegeben wurde, dann wird die Genauigkeit des angegebenen Werts verwendet.

date-Werte

Tabelle 8.10 zeigt mögliche Eingabewert für den Typ date.

Beispiel	Beschreibung
January 8, 1999	englisch ausgeschrieben
1999-01-08	ISO-8601-Format, bevorzugt

Tabelle 8.10: Datumseingabe

Beispiel	Beschreibung
1/8/1999	zweideutig (8. Januar im US-Modus, 1. August im europäischen Modus)
1/18/1999	US-Schreibweise; 18. Januar in jedem Modus
19990108	ISO-8601; Jahr, Monat, Tag
990108	ISO-8601; Jahr, Monat, Tag
1999.008	Jahr und Tag innerhalb des Jahres
99008	Jahr und Tag innerhalb des Jahres
J2451187	Julianischer Tag
January 8, 99 BC	Jahr 99 vor unserer Zeitrechnung

Tabelle 8.10: Datumseingabe (Forts.)

time-Werte

Tabelle 8.11 zeigt gü *(Forts.)*ltige Eingabewerte für den Typ time.

Beispiel	Beschreibung
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	identisch mit 04:05; AM hat keinen Einfluss
`04:05 PM`	identisch mit 16:05; Stunde muss <= 12 sein
`allballs`	identisch mit 00:00:00

Tabelle 8.11: Zeiteingabe

Der Typ time with time zone akzeptiert alle Eingabewerte, die für time gültig sind, gefolgt von einer gültigen Zeitzonenangabe, wie in Tabelle 8.12 gezeigt.

Beispiel	Beschreibung
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601

Tabelle 8.12: Zeiteingabe mit Zeitzone

Mehr Beispiele für Zeitzonen finden Sie in Tabelle 8.13.

timestamp-Werte

Gültige Eingabewerte für die Typen timestamp und timestamp with time zone bestehen aus einem Datum gefolgt von einer Zeitangabe und wahlweise einem `AD` (englisch für unsere Zeitrechnung) oder `BC` (vor unserer Zeitrechnung), alles wahlweise gefolgt von einer Zeitzonenangabe. (Siehe Tabelle 8.13.) Also sind

```
1999-01-08 04:05:06
```

und

```
1999-01-08 04:05:06 -8:00
```

gültige Werte, welche dem ISO-8601-Standard folgen. Darüber hinaus wird das weit verbreitete Format

```
January 8 04:05:06 1999 PST
```

unterstützt.

Bei `timestamp [without time zone]` werden ausdrückliche Zeitzonenangaben still ignoriert. Das heißt, der Ergebniswert wird nur aus den Datums- und Zeitfeldern im Eingabewert ermittelt und nicht der Zeitzone wegen angepasst.

Bei timestamp with time zone wird der Wert intern immer in UTC (*Universal Coordinated Time*, auch bekannt als *Greenwich Mean Time*/GMT) abgespeichert. Ein Eingabewert, der eine Zeitzonenangabe enthält, wird entsprechend der Zeitzone in UTC umgewandelt. Wenn keine Zeitzone angegeben ist, wird angenommen, dass der Wert in der aktuellen Systemzeitzone ist (welche durch den Parameter `timezone` festgelegt ist) und wird dementsprechend nach UTC umgewandelt.

Wenn ein Wert des Typs timestamp with time zone ausgegeben werden soll, dann wird der Wert wiederum von UTC in die aktuelle Systemzeitzone umgewandelt und als lokale Zeit in dieser Zone ausgegeben. Um die Zeit in einer anderen Zeitzone zu sehen, ändern Sie entweder den Parameter `timezone` oder verwenden die Klausel `AT TIME ZONE` (siehe Abschnitt 9.8.3).

Umwandlungen zwischen timestamp without time zone und timestamp with time zone gehen normalerweise davon aus, dass der Wert für timestamp without time zone in der aktuellen Systemzeitzone stehen soll. Eine andere Zeitzone kann für die Umwandlungen mit der Klausel `AT TIME ZONE` angegeben werden.

Beispiel	Beschreibung
MEZ	Mitteleuropäische Zeit
+1:00	Zeitunterschied für MEZ nach ISO 8601
+100	Zeitunterschied für MEZ nach ISO 8601
+1	Zeitunterschied für MEZ nach ISO 8601

Tabelle 8.13: Zeitzoneneingabe

interval-Werte

Werte vom Typ interval können mit der folgenden Syntax geschrieben werden:

```
[@] zahl einheit [zahl einheit...] [richtung]
```

Wobei: *zahl* eine Zahl (möglicherweise mit Vorzeichen) ist, *einheit* eines der Folgenden ist: `second` (Sekunde), `minute` (Minute), `hour` (Stunde), `day` (Tag), `week` (Woche), `month` (Monat), `year` (Jahr), `decade` (Jahrzehnt), `century` (Jahrhundert), `millennium` (Jahrtausend), oder eine Abkürzung oder die Mehrzahl von einer dieser Einheiten; *richtung* kann `ago` oder leer sein, wobei `ago` für eine negative Zeitspanne steht. Das At-Zeichen (`@`) ist optional und hat keine Bedeutung. Die Werte in den unterschiedlichen Einheiten werden automatisch unter Beachtung der Vorzeichen zusammengezählt.

Werte für Tage, Stunden, Minuten und Sekunden können ohne ausdrückliche Einheiten angegeben werden. Zum Beispiel: `'1 12:59:10'` entspricht `'1 day 12 hours 59 min 10 sec'`.

Besondere Werte

Die folgenden SQL-kompatiblen Funktionen können als Datums- und Zeitangaben der entsprechenden Datentypen verwendet werden: `CURRENT_DATE` (aktuelles Datum), `CURRENT_TIME` (aktuelle Zeit),

`CURRENT_TIMESTAMP` (aktuelle Zeit mit Datum). Die letzteren beiden akzeptieren auch eine optionale Genauigkeitsangabe. (Siehe auch Abschnitt 9.8.4.)

PostgreSQL unterstützt auch mehrere besondere Datums- und Zeiteingabewerte, die der Bequemlichkeit dienen können. Sie werden in Tabelle 8.14 gezeigt. Die Werte `infinity` und `-infinity` werden im System gesondert gespeichert und auch genauso wieder ausgegeben, wogegen die anderen einfach Abkürzungen sind, die bei der Eingabe in normale Daten und Zeiten umgewandelt werden. Alle diese Werte werden wie normale Konstanten behandelt und müssen in halbe Anführungszeichen gesetzt werden.

Eingabe	Gültige Typen	Beschreibung
`epoch`	`date, timestamp`	1970-01-01 00:00:00+00 (Unix-Systemzeit null)
`infinity`	`timestamp`	unendlich; später als alle anderen Zeiten
`-infinity`	`timestamp`	negativ unendlich; früher als alle anderen Zeiten
`now`	`date, time, timestamp`	Zeit der aktuellen Transaktion
`today`	`date, timestamp`	heute um Mitternacht
`tomorrow`	`date, timestamp`	morgen um Mitternacht
`yesterday`	`date, timestamp`	gestern um Mitternacht
`zulu, allballs, z`	`time`	00:00:00.00 UTC

Tabelle 8.14: Besondere Datums- und Zeiteingabewerte

8.5.2 Ausgabeformate für Datum und Zeit

Das Ausgabeformat der Datums- und Zeittypen kann auf vier Stile gesetzt werden: ISO 8601, SQL (Ingres), traditioneller POSTGRES-Stil und Deutsch (`German`). Dazu wird der Befehl `SET datestyle` verwendet. Die Vorgabe ist das ISO-Format. (Der SQL-Standard verlangt die Verwendung des ISO-8601-Formats. Der Name des Formats “SQL” ist ein Fehler der Geschichte.) Tabelle 8.15 zeigt Beispiele für jeden Stil. Die Ausgabe der Typen date und time ist natürlich immer nur der jeweilige Datums- oder Zeitteil der Beispiele.

Stilangabe	Beschreibung	Beispiel
`ISO`	ISO-8601-/SQL-Standard	1997-12-17 07:37:16-08
`SQL`	traditioneller Stil	17/12/1997 07:37:16.00 PST
`POSTGRES`	ursprünglicher Stil	Wed Dec 17 07:37:16 1997 PST
`German`	deutscher Stil	17.12.1997 07:37:16.00 PST

Tabelle 8.15: Ausgabestile für Datum und Zei

Der Stil SQL hat eine europäische und eine nicht-europäische (US) Variante, welche bestimmt, ob der Monat vor oder nach dem Tag steht. (Lesen Sie auch in Abschnitt 8.5.1, wie diese Einstellung die Interpretation der Eingabewerte beeinflusst.) Tabelle 8.16 zeigt ein Beispiel.

Stilangabe	Beschreibung	Beispiel
European	*tag/monat/jahr*	17/12/1997 15:37:16.00 CET
US	*monat/tag/jahr*	12/17/1997 07:37:16.00 PST

Tabelle 8.16: Datumsformate

Das Ausgabeformat des Typs interval sieht aus wie das Eingabeformat, außer dass Einheiten wie `week` (Woche) oder `century` (Jahrhundert) in Jahre und Tage umgewandelt werden. Im ISO-Modus sieht die Ausgabe so aus:

```
[ zahl einheit [ ... ] ] [ tage ] [ stunden:minuten:sekunden ]
```

Die Ausgabestile können vom Benutzer mit dem Befehl `SET datestyle`, dem Parameter `datestyle` in der Konfigurationsdatei `postgresql.conf` und der Umgebungsvariable `PGDATESTYLE` auf dem Server oder Client gesetzt werden. Außerdem ist die Formatierungsfunktion `to_char` (siehe Abschnitt 9.7) ein flexiblerer Weg, Datums- und Zeitangaben zu formatieren.

8.5.3 Zeitzonen

Zeitzonen und ihr Gebrauch werden von politischen Entscheidungen beeinflusst, und nicht nur durch die Geometrie der Erde. Die Zeitzonen wurden rund um die Welt während des 20. Jahrhunderts einigermaßen standardisiert, sind aber immer noch für willkürliche Änderungen anfällig. PostgreSQL verwendet die Funktionen des Betriebssystems, um Zeitzonenunterstützung anzubieten, und diese Systeme haben in der Regel nur Informationen für die Zeit von 1902 bis 2038 (was der unterstützten Spanne der herkömmlichen Unix-Systemzeit entspricht). timestamp with time zone und time with time zone verwenden Zeitzoneninformationen nur in diesem Bereich und nehmen an, dass die Zeiten außerhalb dieses Bereichs in UTC sind. Da PostgreSQL die Zeitzonenunterstützung aus dem Betriebssystem verwendet, können dadurch aber Sommerzeit und andere Besonderheiten unterstützt werden.

PostgreSQLstrebt an, dem SQL-Standard in Bereichen, die für normale Anwendungen von Belang sind, zu entsprechen. Der SQL-Standard hat allerdings eine eigenartige Mischung aus Datums- und Zeittypen und -fähigkeiten. Zwei offensichtliche Probleme sind:

- Obwohl der Typ date keine Zeitzone hat, kann der Typ time eine haben. In der Realität ist eine Zeitzone aber wertlos, wenn sie nicht sowohl zu einem Datum als auch zu einer Zeit gehört, weil die Zeitzonenunterschiede sich während des Jahres ändern können (Sommerzeit).
- Die Standardzeitzone wir als Zahlenkonstante als Unterschied zu UTC angegeben. Dadurch wird die Anpassung an die Sommerzeit erschwert, wenn sich arithmetische Berechnungen über die Umstellungsdaten erstrecken.

Um diesen Problemen aus dem Weg zu gehen, empfehlen wir, dass, wenn Sie Zeitzonen verarbeiten müssen, Sie dann Typen mit Datum und Zeit verwenden. Wir empfehlen, dass Sie den Typ time with time zone *nicht* verwenden (obwohl er in PostgreSQL zur Unterstützung bestehender Anwendungen und zur Kompatibilität mit anderen SQL-Implementierungen vorhanden ist). PostgreSQL geht davon aus, dass Typen, die nur ein Datum oder nur die Zeit enthalten, in der lokalen Zeitzone sind.

Alle Daten und Zeiten werden intern in UTC abgespeichert. Die Zeiten werden vom Datenbankserver in die lokale Zeitzone umgewandelt, bevor Sie an den Client geschickt werden, und sind folglich, wenn keine Clientzeitzone definiert ist, in der Zeitzone des Servers.

Es gibt mehrere Wege, die vom Server verwendete Zeitzone festzulegen:

- Wenn nichts anderes angegeben ist, wird die Umgebungsvariable `TZ` auf der Servermaschine als Vorgabe verwendet.
- Der Parameter `timezone` kann in der Konfigurationsdatei `postgresql.conf` gesetzt werden.

- Wenn die Umgebungsvariable `PGTZ` beim Client gesetzt ist, wird sie von `libpq`-Anwendungen dazu verwendet, nach der Verbindung einen `SET TIME ZONE`-Befehl an den Server zu schicken.
- Der SQL-Befehl `SET TIME ZONE` setzt die Zeitzone für die aktuelle Sitzung.

Anmerkung

Wenn eine ungültige Zeitzone angegeben wurde, wird UTC angenommen (jedenfalls auf den meisten Systemen).

In Anhang A finden Sie eine vollständige Liste aller verfügbaren Zeitzonen.

8.5.4 Interna

PostgreSQL verwendet das Julianische Datum für alle Datums- und Zeitberechnungen. Das hat die praktische Eigenschaft, dass alle Daten von 4713 v.u.Z. bis in die ferne Zukunft korrekt vorhergesagt und berechnet werden können, unter der Annahme, dass ein Jahr 365,2425 Tage lang ist.

Die Kalender, die vor dem 19. Jahrhundert in Gebrauch waren, geben eine interessante Lektüre ab, sind aber nicht beständig genug, um Sie in eine moderne Datenbank einzubauen.

8.6 Der Typ boolean

PostgreSQL verfügt über den standardisierten SQL-Typ boolean. boolean hat immer einen von nur zwei Zuständen: “wahr” oder “falsch”. Ein dritter Zustand, “unbekannt”, wird in SQL durch den NULL-Wert ausgedrückt.

Gültige Konstanten für den Zustand “wahr” (englisch *true*) sind:

`TRUE`

't'

'true'

'y'

'yes'

'1'

Für den Zustand “falsch” (englisch *false*) können die folgenden Werte verwendet werden:

`FALSE`
`'f'`
`'false'`
`'n'`
`'no'`
`'0'`

Die Verwendung der Schlüsselwörter `TRUE` und `FALSE` wird bevorzugt (und entspricht dem SQL-Standard).

Beispiel 8.2: Verwendung des Typs boolean

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
```

```
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
a | b
---+---------
t | sic est
f | non est

SELECT * FROM test1 WHERE a;
a | b
---+---------
t | sic est
```

Beispiel 8.2: zeigt, dass Werte vom Typ boolean mit den Buchstaben `t` und `f` ausgegeben werden.

Tipp

Werte vom Typ boolean können nicht unmittelbar in andere Typen umgewandelt werden (zum Beispiel `CAST (boolwert AS integer)` funktioniert nicht). Man kann dies aber durch den `CASE`-Ausdruck erreichen: `CASE WHEN boolwert THEN 'Wert wenn wahr' ELSE 'Wert wenn falsch' END`. Siehe auch Abschnitt 9.12.

8.7 Geometrische Typen

```
boolean benötigt 1 Byte Speicherplatz.
```

Geometrische Typen stellen zweidimensionale räumliche Objekte dar. Tabelle 8.17 zeigt die in PostgreSQL verfügbaren geometrischen Typen. Der elementarste Typ, der Punkt (point), bildet die Grundlage für alle anderen Typen.

Name	Speichergröße	Beschreibung	Darstellung
point	16 Bytes	Punkt auf der Ebene	(x,y)
line	32 Bytes	Gerade (nicht voll implementiert)	((x1,y1),(x2,y2))
lseg	32 Bytes	endliche Strecke	((x1,y1),(x2,y2))
box	32 Bytes	Rechteck	((x1,y1),(x2,y2))
path	16+16n Bytes	geschlossener Pfad (ähnlich Polygon)	((x1,y1),...)
path	16+16n Bytes	offener Pfad	[(x1,y1),...]
polygon	40+16n Bytes	Polygon (ähnlich geschlossener Pfad)	((x1,y1),...)
circle	24 Bytes	Kreis	<(x,y),r> (Mittelpunkt und Radius)

Tabelle 8.17: Geometrische Typen

Eine große Zahl Funktionen und Operatoren sind verfügbar, um diverse geometrische Operationen, wie Dehnung, Verschiebung, Drehung und die Ermittlung von Schnittpunkten, durchzuführen. Sie werden in Abschnitt 9.9 beschrieben.

8.7.1 Punkte

Der Typ point speichert einen Punkt. Punkte sind auch die grundlegenden Bausteine aller anderen geometrischen Typen. Werte vom Typ point werden mit der folgenden Syntax angegeben:

```
( x , y ) x , y
```

wo *x* und *y* die entsprechenden Koordinaten als Fließkommazahlen sind.

8.7.2 Strecken

Der Typ lseg speichert eine Strecke (englisch *line segment*). Strecken werden durch ein Paar Punkte dargestellt. Die Syntax ist folgende:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   , x2 , y2
```

wobei `(x1,y1)` und `(x2,y2)` die Endpunkte der Strecke sind.

8.7.3 Rechtecke

Der Typ box speichert ein Rechteck. Rechtecke werden durch ein Paar gegenüberliegende Ecken dargestellt. Die Syntax ist folgende:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   , x2 , y2
```

wobei `(x1,y1)` und `(x2,y2)` die gegenüberliegenden Ecken des Rechtecks sind.

Das Ausgabeformat entspricht der ersten Syntax. Die Ecken werden sortiert, damit zuerst die obere rechte Ecke und dann die untere linke Ecke steht. Andere Ecken können auch eingegeben werden, aber die obere rechte und die untere linke Ecke werden aus den eingegebenen und gespeicherten Ecken berechnet.

8.7.4 Pfade

Der Typ path speichert einen Pfad. Ein Pfad kann **offen** sein, wenn der erste und der letzte Punkt nicht verbunden sind, oder **geschlossen**, wenn der erste und der letzte Punkt verbunden sind. Die Funktionen `popen(`*p*`)` und `pclose(`*p*`)` stehen zur Verfügung, um festzulegen, ob ein Pfad offen oder geschlossen sein soll, und die Funktionen `isopen(`*p*`)` und `isclosed(`*p*`)` stehen zur Verfügung um die Art in einem Ausdruck zu prüfen.

Werte vom Typ path werden mit der folgenden Syntax angegeben:

```
( ( x1 , y1 ) , ... , ( xn , yn ) ) [
  ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

wobei die Punkte die Endpunkte der Strecken sind, aus denen sich der Pfad zusammensetzt. Eckige Klammern (`[]`) ergeben einen offenen und runde Klammern (`()`) einen geschlossenen Pfad.

Das Ausgabeformat entspricht der ersten Syntax.

8.7.5 Polygone

Der Typ polygon speichert ein Polygon (Vieleck). Polygone entsprechen in gewisser Weise geschlossenen Pfaden, werden aber anders gespeichert und haben einen getrennten Satz Funktionen verfügbar.

Werte vom Typ polygon werden mit der folgenden Syntax angegeben:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1 , ... ,     xn , yn
```

wobei die Punkte die Endpunkte der Strecken sind, die das Polygon begrenzen.

Das Ausgabeformat entspricht der ersten Syntax.

8.7.6 Kreise

Der Typ circle speichert einen Kreis. Kreise werden durch den Mittelpunkt und den Radius dargestellt. Die Syntax ist folgende:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

wobei `(x,y)` der Mittelpunkt und *r* der Radius des Kreises sind.

Das Ausgabeformat entspricht der ersten Syntax.

8.8 Typen für Netzwerkadressen

PostgreSQL bietet Datentypen, um IP- und MAC-Adressen zu speichern, welche in Tabelle 8.18 gezeigt werden. Es ist empfehlenswert, diese Typen statt normaler Texttypen zu verwenden, da diese Typen die Eingabewerte prüfen und es spezialisierte Operatoren und Funktionen für sie gibt.

Name	Speichergröße	Beschreibung
cidr	12 Bytes	IP-Netzwerke
inet	12 Bytes	IP-Hosts und -Netzwerke
macaddr	6 Bytes	MAC-Adressen

Tabelle 8.18: Typen für Netzwerkadressen

IPv6 wird noch nicht unterstützt.

8.8.1 inet

Der Typ inet speichert eine IP-Hostadresse und wahlweise die Identität des Subnetzes, in dem der Host ist, beides in einem Wert. Die Subnetzidentität wird ausgedrückt, indem angegeben wird, wie viele Bits der Hostadresse die Netzwerkadresse darstellen (die “Netzmaske”). Wenn die Netzmaske 32 ist, gibt der Wert kein Subnetz an, sondern nur einen einzelnen Host. Wenn Sie nur Netzwerke speichern wollen, sollten Sie den Typ cidr statt inet verwenden.

Das Eingabeformat für diesen Typ ist *x.x.x.x/y*, wobei *x.x.x.x* eine IP-Adresse ist und *y* die Anzahl der Bits in der Netzmaske. Wenn */y* ausgelassen wird, dann ist die Netzmaske 32 und der Wert steht für einen einzelnen Host. Bei der Ausgabe wird der */y*-Teil weggelassen, wenn die Netzmaske 32 ist.

8.8.2 cidr

Der Typ cidr speichert die Identität eines IP-Netzwerks. Eingabe- und Ausgabeformate folgen den Vorgaben des Classless Internet Domain Routing Systems. Das Format, um ein Netzwerk anzugeben, ist *x.x.x.x/y*, wobei *x.x.x.x* ein Netzwerk und *y* die Anzahl von Bits in der Netzmaske ist. Wenn *y* ausgelassen wird, dann wird es aus den Annahmen des alten klassen-orientierten Netzwerknummerierungssystems errechnet, mit der Ausnahme, dass die Bitmaske mindestens groß genug sein muss, um alle Oktetts im Eingabewert einzuschließen. Es ist ein Fehler, eine Netzwerkadresse anzugeben, die Bits rechts von der angegebenen Netzmaske gesetzt hat.

Tabelle 8.19 zeigt einige Beispiele.

cidr-Eingabe	cidr-Ausgabe	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8

Tabelle 8.19: Eingabebeispiele für den Typ cidr

8.8.3 inet vs. cidr

Der Hauptunterschied zwischen den Datentypen inet und cidr ist, dass inet gesetzte Bits rechts von der Netzmaske zulässt und cidr nicht.

Tipp

Wenn Sie die Ausgabeformate von inet oder cidr nicht mögen, dann versuchen Sie die Funktionen `host`, `text` und `abbrev`.

8.8.4 macaddr

Der Typ macaddr speichert MAC-Adressen, das heißt die Hardwareadressen von Ethernetkarten (obwohl MAC-Adressen auch für andere Zwecke verwendet werden). Als Eingabe werden verschiedene gebräuchliche Formate akzeptiert, einschließlich

```
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'
```

welche alle die gleiche Adresse angeben würden. Für die Ziffern `a` bis `f` werden Groß- und Kleinbuchstaben akzeptiert. Das Ausgabeformat entspricht immer der letzten gezeigten Form.

Das Verzeichnis `contrib/mac` in der PostgreSQL-Quelltext-Distribution enthält einige Werkzeuge, die verwendet werden können, um aus MAC-Adressen die Namen der Hardwarehersteller zu ermitteln.

8.9 Bitkettentypen

Bitketten sind Ketten von Einsen und Nullen. Sie können verwendet werden, um Bitmasken zu speichern oder darzustellen. Es gibt zwei SQL-Typen für Bitketten: bit(*n*) und bit varying(*n*), wo *n* eine positive ganze Zahl ist.

Daten vom Typ bit müssen genau der Länge *n* entsprechen; es ist ein Fehler zu versuchen, einen kürzeren oder längeren Wert zu speichern. Daten vom Typ bit varying sind variabel in der Länge bis zur Höchstlänge *n*; längere Werte werden nicht akzeptiert. Eine Typangabe bit ist gleichbedeutend mit `bit(1)`, wogegen bit varying ohne Längenangabe eine unbegrenzte Länge zult Nullen erweitert, egal in welchem

Anmerkung

Wenn man einen Bitkettenwert ausdrücklich in bit(*n*) umwandelt (Cast), wird er rechts gekürzt oder mit Nullen erweitert, damit er genau *n* Bits lang wird. Ebenso gilt, wenn ein Bitkettenwert ausdrücklich in bit varying(*n*) umgewandelt wird, dann wird er rechts abgeschnitten, wenn er länger als *n* Bits ist.

Anmerkung

Vor PostgreSQL 7.2 wurden Bitwerte immer still abgeschnitten oder mit Nullen erweitert, egal in welchem Zusammenhang die Zuweisung oder Umwandlung erfolgte. Das wurde geändert, um dem SQL-Standard zu entsprechen.

Schauen Sie in Kapitel 4 Abschnitt *Bitkettenkonstanten* für Informationen über die Syntax von Bitkettenkonstanten. Bitlogische Operatoren sowie Funktionen zur Bitkettenmanipulation sind auch verfügbar; siehe Kapitel 9.

Beispiel 8.3: Verwendung der Bitkettentypens

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR: Bit string length 2 does not match type BIT(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
   a | b
-----+-----
 101 | 00
 100 | 101
```

8.10 Objekt-Identifikator-Typen

Objekt-Identifikatoren (OIDs) werden von PostgreSQL intern als Primärschlüssel in diversen Systemtabellen verwendet. Eine OID-Systemspalte wird auch jeder benutzerdefinierten Tabelle hinzugefügt (außer wenn `WITHOUT OIDS` angegeben wird, wenn die Tabelle erzeugt wird). Der Typ oid speichert einen Objekt-Identifikator. Es gibt auch mehrere Aliastypen für oid: regproc, regprocedure, regoper, regoperator, regclass und regtype. Tabelle 8.20 zeigt einen Überblick.

Der Typ oid ist gegenwärtig ein 4-Byte-Ganzzahlentyp ohne Vorzeichen. Daher ist er nicht groß genug, um in der ganzen Datenbank oder sogar in einzelnen großen Tabellen eindeutig sein zu können. Daher ist es nicht empfehlenswert, die OID-Spalte einer benutzerdefinierten Tabelle als Primärschlüssel zu verwenden. Am besten sollten OIDs nur für Verweise auf die Systemtabellen verwendet werden.

Der Typ oid hat selbst kaum Operationen außer Vergleichsoperationen. Man kann ihn aber in integer umwandeln und ihn mit den normalen numerischen Operationen bearbeiten. (Wenn Sie das tun, achten Sie aber auf mögliche Verwirrungen über die Vorzeichen.)

Die OID-Aliastypen haben keine eigenen Operationen außer den spezialisierten Eingabe- und Ausgaberoutinen. Die Aliastypen gestatten eine vereinfachte Suche nach der OID von Systemobjekten: Zum Beispiel kann man schreiben `'meinetabelle'::regclass,` um die OID der Tabelle `meinetabelle` zu erfahren, anstatt `SELECT oid FROM pg_class WHERE relname = 'meinetabelle'` schreiben zu müssen. (In Wirklichkeit müsste man eine viel kompliziertere Anfrage verwenden, um die richtige OID zu finden, wenn es mehrere Tabellen namens `meinetabelle` in verschiedenen Schemas gibt.)

Name	Verweis	Beschreibung	Wertbeispiel
oid	beliebig	numerischer Objekt-Identifikator	564182
regproc	pg_proc	Funktionsname	sum
regprocedure	pg_proc	Funktion mit Argumenttypen	sum(int4)
regoper	pg_operator	Operatorname	+
regoperator	pg_operator	Operator mit Argumenttypen	`*(integer,integer)` oder `-(NONE,integer)`
regclass	pg_class	Relationsname	pg_type
regtype	pg_type	Datentypname	integer

Tabelle 8.20: Objekt-Identifikator-Typen

Alle OID-Aliastypen akzeptieren schemaqualifizierte Namen und geben diese aus, wenn das Objekt im aktuellen Suchpfad nicht ohne ausdrückliche Schemaangabe gefunden würde. Die Typen regproc und regoper akzeptieren nur Namen als Eingabe, die eindeutig (nicht überladen) sind, sind also von begrenztem Nutzen; meistens sind regprocedure bzw. regoperator sinnvoller. Bei regoperator kann man unäre Operatoren angeben, indem man `NONE` für den nicht benutzten Operand schreibt.

Ein weiterer Identifikator-Typ, der vom System verwendet wird, ist xid, der Transaktions-Identifikator. Das ist der Datentyp der Systemspalten `xmin` und `xmax`. Transaktionsnummern sind 32 Bit groß.

Ein dritter Identifikator-Typ, der vom System verwendet wird, ist cid (*command identifier*). Das ist der Datentyp der Systemspalten `cmin` und `cmax`. cid-Werte sind auch 32 Bit groß.

Ein letzter Identifikator-Typ, der vom System verwendet wird, ist tid (*tuple identifier*). Das ist der Datentyp der Systemspalte `ctid`. Eine ctid ist ein Paar (Blocknummer, Index im Block), das die physikalische Stelle der Zeilenversion innerhalb seiner Tabelle angibt.

(Die Systemspalten werden im Einzelnen in Abschnitt 5.2 erläutert.)

8.11 Arrays

PostgreSQL erlaubt die Definition von Spalten als mehrdimensionalen Arraytyp mit variabler Länge. Arrays können aus eingebauten oder benutzerdefinierten Typen erzeugt werden.

8.11.1 Deklaration von Arraytypen

Um die Verwendung von Arrays zu erläutern, erzeugen wir diese Tabelle:

```
CREATE TABLE angestellte (
    name            text,
    quartal_gehalt integer[],
    plan text[][]
);
```

Wie gezeigt, werden Arraytypen bezeichnet, indem eckige Klammern ([]) an den Typnamen der Arrayelemente angehängt werden. Der obige Befehl erzeugt eine Tabelle namens `angestellte` mit einer Spalte vom Typ text (`name`), einem eindimensionalen Array vom Typ integer (`quartal_gehalt`), welches das Gehalt pro Quartal des Angestellten speichert, und einem zweidimensionalen Array vom Typ text (`plan`), welches den Wochenplan des Angestellten darstellt.

8.11.2 Eingabe von Arraywerten

Jetzt können wir einige Daten einfügen. Arrayelemente werden von geschweiften Klammern eingeschlossen und durch Kommas voneinander getrennt. Wenn Sie C kennen, dann werden Sie diese Syntax vielleicht wiedererkennen. (Weiter Einzelheiten folgen unten.)

```
INSERT INTO angestellte
    VALUES ('Wilhelm',
    '{10000, 10000, 10000, 10000}',
  '{{"Meeting", "Mittag"}, {}}'); INSERT INTO angestellte VALUES ('Carola',
'{20000, 25000, 25000, 25000}',
   '{{"Präsentation", "Beratung"}, {"Meeting"}}');
```

Anmerkung

Gegenwärtig können einzelne Arrayelemente nicht den SQL-NULL-Wert enthalten. Das ganze Array kann auf den NULL-Wert gesetzt werden, aber Sie können kein Array haben, wo einzelne Werte NULL sind und andere nicht. Es ist vorgesehen, dies irgendwann zu verbessern.

8.11.3 Verweise auf Arraywerte

Jetzt können wir einige Anfragen durchführen. Zuerst zeigen wir, wie man auf ein einzelnes Element im Array zugreifen kann. Die folgende Anfrage ergibt zum Beispiel die Namen der Angestellten, deren Gehalt sich im zweiten Quartal verändert hat:

```
SELECT name FROM angestellte WHERE quartal_gehalt[1] <> quartal_gehalt[2];
name
--------
Carola (1 row)
```

Die Arrayindexzahlen werden in eckige Klammern gesetzt. PostgreSQL zählt die Arrayelemente normalerweise von eins an, das heißt, in einem Array von Größe *n* fangen die Elemente bei `array[1]` an und enden bei `array[n]`.

Die folgende Anfrage ermittelt das Gehalt von allen Angestellten im dritten Quartal:

```
SELECT quartal_gehalt[3] FROM angestellte;

quartal_gehalt
----------------
         10000
         25000

(2 rows)
```

Man kann auch auf beliebige Teilarrays eines Arrays zugreifen. Ein Teilarray wird mit der folgenden Schreibweise bestimmt: *`untergrenze:obergrenze`*, jeweils für eine oder mehrere Arraydimensionen. Diese Anfrage ermittelt zum Beispiel den ersten Eintrag auf Wilhelms Wochenplan für die ersten zwei Tage der Woche:

```
SELECT plan[1:2][1:1] FROM angestellte WHERE name = 'Wilhelm';

        plan
--------------------
 {{Meeting},{""}}
(1 row)
```

Man hätte auch schreiben können

```
SELECT plan[1:2][1] FROM angestellte WHERE name = 'Wilhelm';
```

und hätte das gleiche Ergebnis erhalten. Eine Arrayindizierung ergibt immer ein Teilarray, wenn mindestens ein Indexwert die Form *`untergrenze:obergrenze`* hat. Die Untergrenze 1 wird angenommen, wenn ein Index nur einen Wert angibt.

Ein Arraywert kann komplett ersetzt werden:

```
UPDATE angestellte SET quartal_gehalt = '{25000,25000,27000,27000}'
    WHERE name = 'Carola';
```

Oder einzelne Elemente können aktualisiert werden:

```
UPDATE angestellte SET quartal_gehalt[4] = 15000
    WHERE name = 'Wilhelm';
```

Oder Teilarrays können aktualisiert werden:

```
UPDATE angestellte SET quartal_gehalt[1:2] = '{27000,27000}'
    WHERE name = 'Carola';
```

Ein Array kann vergrößert werden, indem man einen Wert einem Element zuweist, das an ein bereits vorhandenes angrenzt, oder indem man in ein Teilarray, das an vorhandene Daten angrenzt oder sich mit ihnen überschneidet, schreibt. Wenn zum Beispiel ein Array vorher vier Elemente hat, wird es fünf Ele-

mente haben, nachdem eine Aktualisierung in `array[5]` schreibt. Gegenwärtig können auf diese Art nur eindimensionale Arrays vergrößert werden, nicht aber mehrdimensionale.

Indem man Teilarrays Daten zuweist, kann man Arrays erzeugen, wo die Elemente nicht von eins an zählen. So kann man zum Beispiel Daten in `array[-2:7]` schreiben, um ein Array zu erzeugen, das Indexwerte von -2 bis 7 hat.

Die Syntax von `CREATE TABLE` erlaubt auch die Erzeugung von Arrays mit fester Größe:

```
CREATE TABLE tictactoe (
    kästchen integer[3][3]
);
```

Gegenwärtig werden diese Arraygrößengrenzen aber nicht durchgesetzt. Das Verhalten ist genauso wie bei Arrays ohne Größenangabe.

Die angegebene Zahl der Dimensionen wird im Übrigen auch nicht durchgesetzt. Arrays mit einem bestimmten Elementtyp werden alle als derselbe Typ betrachtet, ungeachtet der Zahl oder der Größe der Dimensionen. Die Angabe der Dimensionen oder Größen in `CREATE TABLE` dient also einfach nur der Dokumentation und beeinflusst das Laufzeitverhalten nicht.

Die aktuellen Dimensionen eines Arraywerts können mit der Funktion `array_dims` ermittelt werden:

```
SELECT array_dims(plan) FROM angestellte WHERE name = 'Carola';

 array_dims
------------ [
 1:2][1:1]
(1 row)
```

`array_dims` erzeugt ein Ergebnis vom Typ text, was für Leute gut zu lesen, aber vielleicht nicht so praktisch für Programme ist.

8.11.4 In Arrays suchen

Um einen Wert in einem Array zu finden, müssen Sie jeden Wert des Arrays prüfen. Das kann von Hand gemacht werden (wenn Sie die Größe des Arrays kennen). Zum Beispiel:

```
SELECT * FROM angestellte WHERE quartal_gehalt[1] = 10000 OR
                                quartal_gehalt[2] = 10000 OR
                                quartal_gehalt[3] = 10000 OR
                                quartal_gehalt[4] = 10000;
```

Das kann aber sehr umständlich sein, wenn die Arrays sehr groß sind, und wenn die Größe des Arrays nicht bekannt ist, funktioniert das gar nicht. Es gibt eine Erweiterung, die neue Funktionen und Operatoren definiert, die dafür verwendet werden kann, die aber nicht in PostgreSQL eingebaut ist. Damit könnte man die obige Anfrage so schreiben:

```
SELECT * FROM angestellte WHERE quartal_gehalt[1:4] *= 10000;
```

Um das ganze Array zu durchsuchen (und nicht nur Teilarrays), könnten Sie Folgendes machen:

```
SELECT * FROM angestellte WHERE quartal_gehalt *= 10000;
```

Außerdem könnten Sie alle Zeilen finden, wo die Arrayelemente alle gleich 10000 sind:

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

Um dieses optionale Modul zu installieren, schauen Sie in das Verzeichnis `conArtrib/array` in der PostgreSQL-Quelltext-Distribution.

8.11.5 Array-Eingabe- und Ausgabesyntax

Tipp

Arrays sind keine Mengen. Die Verwendung von Arrays in der beschriebenen Art und Weise ist oft ein Zeichen, dass die Datenbank falsch entworfen wurde. Das Arrayfeld sollte normalerweise als eigene belle abgetrennt werden. Tabellen können Sie logischerweise ganz einfach durchsuchen.

Die externe Darstellung eines Arrays besteht aus den Elementen, die durch die Eingabe- und Ausgaberoutinen des entsprechenden Datentyps interpretiert werden, und der Dekoration, die die Struktur des Arrays angibt. Diese Dekoration besteht aus geschweiften Klammern ({ und }) um den Arraywert sowie den Trennzeichen zwischen den Elementen. Das Trennzeichen ist normalerweise das Komma (,), kann aber auch ein anderes sein: Es wird durch den `typdelim`-Wert des Arrayelementtyps festgelegt. (Unter den Standarddatentypen in der PostgreSQL-Distribution verwendet der Typ `box` das Semikolon (;), aber alle anderen Typen verwenden das Komma.) In mehrdimensionalen Arrays muss um jede Dimension ein Paar geschweifte Klammern stehen, und Trennzeichen müssen zwischen den eingeklammerten Einheiten stehen. Leerzeichen können vor den linken oder rechten Klammern oder vor den einzelnen Elementen stehen. Leerzeichen nach Elementen werden allerdings nicht ignoriert: Nachdem die vorangehenden Leerzeichen übersprungen wurden, zählt alles bis zum nächsten Trennzeichen oder der rechten Klammer als zum Wert gehörend.

8.11.6 Sonderzeichen in Arrayelementen

Wie oben gezeigt, können Sie in einen Arraywert Anführungszeichen um die einzelnen Elemente schreiben. Das ist erforderlich, wenn der Elementwert den Arrayparser verwirren könnte. So müssen zum Beispiel Elemente mit geschweiften Klammern, Kommas (oder was das Trennzeichen sein mag), Anführungszeichen, Backslashes oder Whitespace am Anfang in Anführungszeichen gesetzt werden. Um ein Anführungszeichen oder einen Backslash in ein Arrayelement zu schreiben, setzen Sie einen Backslash davor. Als Alternative können Sie auch einen Backslash vor alle anderen Zeichen setzen, die sonst Teil der Arraysyntaxdekoration oder ignorierte Leerzeichen sind.

Die Arrayausgabefunktion setzt Anführungszeichen um Elementwerte, wenn diese leere Zeichenketten sind oder geschweifte Klammern, Trennzeichen, Anführungszeichen, Backslashes oder Whitespace enthalten. Vor Anführungszeichen und Backslashes in Elementwerten wird ein Backslash gestellt. Bei Zahlentypen kann man mit Sicherheit davon ausgehen, dass Anführungszeichen niemals verwendet werden, aber bei Textdatentypen sollte man darauf vorbereitet sein, dass sie auftreten können oder auch nicht. (Das ist eine Veränderung im Verhalten gegenüber PostgreSQL-Versionen vor 7.2.)

```
INSERT ... VALUES ('{"\\\\","\\""}');
```

Anmerkung

Beachten Sie, dass das, was Sie in einem SQL-Befehl schreiben, zuerst als Zeichenkettenkonstante verarbeitet wird und dann als Array. Dadurch verdoppelt sich die Anzahl der Backslashes, die Sie verwenden müssen. Um zum Beispiel einen Arraywert einzufügen, der aus einem Backslash und einem Anführungszeichen besteht, müssten Sie schreiben:

Bei der Verarbeitung der Zeichenkettenkonstante wird ein Satz Backslashes entfernt, sodass der Wert, der bei der Arrayeingabefunktion ankommt, wie {"\\","\""} aussieht. Die Werte, die wiederum bei der Eingabefunktion vom Typ text ankommen, sind \ bzw. ". (Wenn man Datentypen verwendet, deren Eingaberoutine Backslashes besonders behandelt, wie zum Beispiel bytea, dann muss man mithin bis zu acht Backslashes verwenden, um am Ende einen im Arrayelementwert zu speichern.)

8.12 Pseudotypen

Das Typensystem in PostgreSQL enthält einige Typen mit besonderen Aufgaben, die zusammenfassend **Pseudotypen** genannt werden. Ein Pseudotyp kann nicht als Spaltentyp verwendet werden, aber er kann als Argument- oder Ergebnistyp einer Funktion angegeben werden. Jeder der vorhandenen Pseudotypen ist in einer Situation von Nutzen, wo das Verhalten der Funktion nicht einfach durch einen normalen SQL-Datentyp als Eingabe oder Ausgabe ausgedrückt werden kann. Tabelle 8.21 listet die bestehenden Pseudotypen.

Name	Beschreibung
record	Identifiziert eine Funktion, die einen nicht näher definierten Zeilentyp zurückgibt.
any	Gibt an, dass die Funktion jeden Datentyp akzeptiert.
anyarray	Gibt an, dass die Funktion jeden beliebigen Arraydatentyp akzeptiert.
void	Gibt an, dass die Funktion keinen Wert zurückgibt.
trigger	Eine Triggerfunktion wird so deklariert, dass sie trigger zurückgibt.
language_handler	Ein Handler für eine prozedurale Sprache wird so deklariert, dass er language_handler zurückgibt.
cstring	Gibt an, dass die Funktion eine C-Zeichenkette (mit Null-Byte abgeschlossen) akzeptiert oder zurückgibt.
internal	Gibt an, dass die Funktion einen internen Datentyp akzeptiert oder zurückgibt.
opaque	Ein obsoleter Typname, der früher alle oben genannten Funktionen ausfüllte.

Tabelle 8.21: Pseudotypen

Funktionen, die in C geschrieben wurden (egal ob eingebaut oder dynamisch geladen), können so deklariert werden, dass sie einen dieser Pseudodatentypen als Argument oder Rückgabewert verwenden. Der Autor der Funktion muss selbst sicherstellen, dass die Funktion sich ordnungsgemäß verhält, wenn einer dieser Pseudotypen als Argument verwendet wird.

Funktionen, die in prozeduralen Sprachen geschrieben wurden, können Pseudotypen nur verwenden, wenn es die Sprachen gestatten. Gegenwärtig verbieten die prozeduralen Sprachen alle die Verwendung von Pseudotypen als Argument und gestatten nur void als Ergebnistyp (sowie trigger, wenn die Funktion als Trigger verwendet werden soll).

Der Pseudotyp internal wird zur Deklaration von Funktionen verwendet, die nur intern vom Datenbanksystem aufgerufen werden sollen und nicht von einem SQL-Befehl aus. Wenn eine Funktion mindestens ein Argument vom Typ internal hat, dann kann es von SQL aus nicht aufgerufen werden. Um in die Typensicherheit dieser Beschränkung zu erhalten, ist es wichtig, folgende Regel zu beachten: Deklarieren Sie keine Funktion mit einen Rückgabetyp internal, wenn die Funktion nicht mindestens ein Argument vom Typ internal hat.

9 Funktionen und Operatoren

PostgreSQL bietet eine große Zahl Funktionen und Operatoren für die eingebauten Datentypen. Benutzer können auch eigene Funktionen und Operatoren definieren, wie in Teil V beschrieben wird. Die Befehle `\df` und `\do` in `psql` können verwendet werden, um alle tatsächlich vorhandenen Funktionen bzw. Operatoren anzuzeigen.

Wenn Sie auf Portierbarkeit Wert legen, sollten Sie beachten, dass die meisten hier beschriebenen Funktionen und Operatoren, mit Ausnahme der einfachsten arithmetischen und Vergleichsoperatoren und einiger ausdrücklich markierter Funktionen, nicht im SQL-Standard definiert sind. Einige der Erweiterungen sind auch in anderen SQL-Datenbanken vorhanden und in vielen Fällen ist die Funktionalität zwischen den Produkten einheitlich und kompatibel.

9.1 Logische Operatoren

Die gebräuchlichen logischen Operatoren sind vorhanden:

`AND` (Konjunktion)

`OR` (Disjunktion)

`NOT` (Negation)

SQL verwendet eine Boole'sche Logik mit drei Wertigkeiten, wobei der NULL-Wert den Zustand “unbekannt” darstellt. Beachten Sie die folgenden Wahrheitstabellen:

a	**b**	*a* **AND** *b*	*a* **OR** *b*
wahr	wahr	wahr	wahr
wahr	falsch	falsch	wahr
wahr	unbekannt	unbekannt	wahr
falsch	falsch	falsch	falsch
falsch	unbekannt	falsch	unbekannt
unbekannt	unbekannt	unbekannt	unbekannt

a	**NOT** *a*
wahr	falsch
falsch	wahr
unbekannt	unbekannt

9.2 Vergleichsoperatoren

Die gewöhnlichen Vergleichsoperatoren sind vorhanden und werden in Tabelle 9.1 gelistet.

Operator	**Beschreibung**
<	kleiner als
>	größer als
<=	kleiner als oder gleich
>=	größer als oder gleich
=	gleich
<> oder !=	ungleich

Tabelle 9.1: Vergleichsoperatoren

Anmerkung

Der Operator != wird vom Parser in <> umgewandelt. Es ist nicht möglich, Operatoren mit den Namen != und <> zu erzeugen, die sich verschieden verhalten.

Vergleichsoperatoren gibt es für alle Datentypen, wo dies sinnvoll ist. Alle Vergleichsoperatoren sind binäre Operatoren (mit zwei Operanden) und haben ein Ergebnis vom Typ boolean; Ausdrükke wie `1 < 2 < 3` sind ungültig (weil es keinen Operator < zum Vergleichen eines Werts vom Typ boolean mit 3 gibt).

Neben den Vergleichsoperatoren gibt es die spezielle Klausel BETWEEN. So ist

```
a BETWEEN x AND y
```

gleichbedeutend mit

```
a >= x AND a <= y
```

Ebenso ist

```
a NOT BETWEEN x AND y
```

gleichbedeutend mit

```
a < x OR a > y
```

Es gibt keinen Unterschied zwischen den beiden Formen, außer der CPU-Leistung, die aufgewendet werden muss, um die erste Form intern in die zweite Form umzuschreiben.

Um zu prüfen, ob ein Wert NULL ist, verwenden Sie die Klauseln

```
ausdruck IS NULL
ausdruck IS NOT NULL
```

oder die gleichbedeutenden, aber nicht standardisierten Formen

```
ausdruck ISNULL
ausdruck NOTNULL
```

Verwenden Sie *nicht* `ausdruck = NULL`, weil `NULL` nicht "gleich" `NULL` ist. (Der NULL-Wert stellt einen unbekannten Wert dar und man kann nicht bestimmen, ob zwei unbekannte Werte gleich sind.)

Einige Anwendungen gehen (fälschlicherweise) davon aus, dass *`ausdruck`* `= NULL` wahr ist, wenn *ausdruck* den NULL-Wert ergibt. Um diese Anwendungen zu unterstützen, kann man den Konfigurationsparameter `transform_null_equals` anstellen (z.B. `SET transform_null_equals TO ON;`). In dem Fall wird

PostgreSQL Ausdrücke der Form `x = NULL` in `x IS NULL` umwandeln. Das war das voreingestellte Verhalten in PostgreSQL 6.5 bis 7.1.

Werte vom Typ boolean können mit folgenden Ausdrücken getestet werden:

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Diese sind ähnlich `IS NULL` in dem Sinne, dass sie immer wahr oder falsch ergeben und niemals den NULL-Wert, selbst wenn ein Operand NULL ist. Ein NULL-Wert wird als der logische Wert "unbekannt" (*unknown*) behandelt.

9.3 Mathematische Funktionen und Operatoren

Mathematische Operatoren stehen für viele PostgreSQLPostgreSQL-Typen zur Verfügung. Für Typen ohne weithin bekannte mathematische Definitionen für alle Varianten (z.B. Datum und Zeit) beschreiben wir das Verhalten in den folgenden Abschnitten.

Tabelle 9.2 zeigt die vorhandenen mathematischen Operatoren.

Operator	Beschreibung	Beispiel	Ergebnis
`+`	Addition	2 + 3	5
`-`	Subtraktion	2 - 3	-1
`*`	Multiplikation	2 * 3	6
`/`	Division (Ganzzahldivision schneidet Ergebnis ab)	4 / 2	2
`%`	Rest nach Division	5 % 4	1
`^`	Potenzierung	2.0 ^ 3.0	8
`|/`	Quadratwurzel	|/ 25.0	5

Tabelle 9.2: Mathematische Operatoren

Operator	Beschreibung	Beispiel	Ergebnis
`||/`	Kubikwurzel	||/ 27.0	3
`!`	Fakultät	5 !	120
`!!`	Fakultät (Präfixoperator)	!! 5	120
`@`	Betrag	@ -5.0	5
`&`	bitweise Konjunktion (“und”)	91 & 15	11
`|`	bitweise Disjunktion (“oder”)	32 | 3	35
`#`	bitweise Alternative (“exklusiv-oder”)	17 # 5	20
`~`	bitweise Negation (“nicht”)	~1	-2
`<<`	bitweises Linksschieben	1 << 4	16
`>>`	bitweises Rechtsschieben	8 >> 2	2

Tabelle 9.2: Mathematische Operatoren (Forts.)

Die bitweisen Operatoren gibt es auch für die Bitkettentypen bit und bit varying, wie in Tabelle 9.3 gezeigt. Die Bitkettenoperanden von `&`, `|` und # müssen jeweils die gleiche Länge haben. Beim Bitschieben wird die ursprüngliche Länge der Bitkette erhalten, wie in der Tabelle gezeigt.

Beispiel	Ergebnis
`B'10001' & B'01101'`	`00001`
`B'10001' | B'01101'`	`11101`
`B'10001' # B'01101'`	`11110`
`~ B'10001'`	`01110`
`B'10001' << 3`	`01000`
`B'10001' >> 2`	`00100`

Tabelle 9.3: Bitweise Operatoren für Bitketten

Tabelle 9.4 zeigt die verfügbaren mathematischen Funktionen. In der Tabelle steht `dp` für double precision. Viele dieser Funktionen gibt es in mehreren Formen mit unterschiedlichen Argumentdatentypen. Wenn nichts anderes angegeben wurde, ist der Rückgabewert von jeder Variante dieser Funktionen derselbe wie der Argumentdatentyp. Die Funktionen, die mit dem Datentyp double precision umgehen, sind meistens mit den Funktionen der C-Bibliothek des Betriebssystems implementiert; die Genauigkeit und das Verhalten in Grenzfällen kann daher vom System abhängen.

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
`abs(`*x*`)`	(gleicher wie *x*)	Betrag	`abs(-17.4)`	`17.4`
`cbrt(dp)`	`dp`	Kubikwurzel	`cbrt(27.0)`	`3`
`ceil(dp oder numeric)`	`numeric`	kleinste ganze Zahl, die nicht kleiner als das Argument ist	`ceil(-42.8)`	`-42`
`degrees(dp)`	`dp`	Radiant in Grad umwandeln	`degrees(0.5)`	`28.6478897565412`
`exp(dp oder numeric)`	`dp`	Potenzierung	`exp(1.0)`	`2.71828182845905`

Tabelle 9.4: Mathematische Funktionen

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
floor(dp oder numeric)	numeric	größte ganze Zahl, die nicht größer als das Argument ist	floor(-42.8)	-43
ln(dp oder numeric)	dp	natürlicher Logarithmus	ln(2.0)	0.693147180559945
log(dp oder numeric)	dp	Logarithmus zur Basis 10	log(100.0)	2
log(*b* numeric, *x* numeric)	numeric	Logarithmus zur Basis *b*	log(2.0, 64.0)	6.0000000000
mod(**y**, *x*)	(gleicher wie Argumenttypen)	Rest von *y*/*x*	mod(9,4)	1
pi()	dp	Konstante ®π”	pi()	3.14159265358979
pow(*a* dp, *b* dp)	dp	*a* hoch *b*	pow(9.0, 3.0)	729
pow(*a* numeric, *b* numeric)	numeric	*a* hoch *b*	pow(9.0, 3.0)	729
radians(dp)	dp	Grad in Radiant umwandeln	radians(45.0)	0.785398163397448
random()	dp	zufälliger Wert zwischen 0.0 und 1.0	random()	
round(dp or numeric)	(wie Argument)	zur nächsten ganzen Zahl runden	round(42.4)	42
round(*v* numeric, *s* integer)	numeric	auf *s* Dezimalstellen runden	round(42.4382, 2)	42.44
sign(dp oder numeric)	(wie Argument)	Vorzeichen des Arguments (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp)	dp	Quadratwurzel	sqrt(2.0)	1.4142135623731
trunc(dp)	dp	Richtung null runden (abschneiden)	trunc(42.8)	42
trunc(numeric, *r* integer)	numeric	auf *r* Dezimalstellen abschneiden	trunc(42.4382, 2)	42.43

Tabelle 9.4: Mathematische Funktionen (Forts.)

Schließlich zeigt Tabelle 9.5 die verfügbaren trigonometrischen Funktionen. Alle trigonometrischen Funktionen haben Argument- und Ergebnistypen double precision.

Funktion	Beschreibung
acos(*x*)	Arkuskosinus
asin(*x*)	Arkussinus
atan(*x*)	Arkustangens
atan2(*x*, *y*)	Arkustangens von *x*/*y*
cos(*x*)	Kosinus
cot(*x*)	Kotangens
sin(*x*)	Sinus
tan(*x*)	Tangens

Tabelle 9.5: Trigonometrische Funktionen

9.4 Zeichenkettenfunktionen und -operatoren

Dieser Abschnitt beschreibt Funktionen und Operatoren zur Auswertung und Manipulation von Zeichenketten. Damit sind in diesem Zusammenhang die Typen character, character varying und text gemeint. Wenn nicht anders angegeben, sind alle gelisteten Funktionen für alle Typen verfügbar, aber achten Sie auf die möglichen Auswirkungen der automatischen Auffüllung mit Leerzeichen beim Typ character. Im Allgemeinen kann man die hier beschriebenen Funktionen auch mit anderen Datentypen verwenden, wobei die Werte dann automatisch in eine Zeichenkettendarstellung umgewandelt werden. Einige Funktionen gibt es auch direkt für die Bitkettentypen.

SQL definiert einige Funktionen mit spezieller Syntax, wo bestimmte Schlüsselwörter anstatt Kommas verwendet werden, um die Argumente zu trennen. Einzelheiten sehen Sie in Tabelle 9.6. Diese Funktionen gibt es auch mit der normalen Syntax für Funktionsaufrufe. (Siehe Tabelle 9.7.)

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
string || *string*	text	Zeichenketten aneinander hängen	'Post' || 'greSQL'	PostgreSQL
bit_length(*string*)	integer	Anzahl Bits in Zeichenkette	bit_length('jose')	32
char_length(*string*) or character_length(*string*)	integer	Anzahl Zeichen in Zeichenkette	char_length('jose')	4
convert(*string* using *konversionsname*)	text	Ändert die Kodierung mit der angegebenen Konversion. Konversionen können mit CREATE CONVERSION erzeugt werden. Es gibt auch vordefinierte Konversionen; siehe Tabelle 9.8.	convert('PostgreSQL' using iso_8859_1_to_utf_8)	'PostgreSQL' in Unicode-Kodierung (UTF-8)
lower(*string*)	text	in Kleinbuchstaben umwandeln	lower('TOM')	tom
octet_length(*string*)	integer	Anzahl Bytes in Zeichenkette	octet_length('jose')	4
overlay(*string* placing *string* from integer [for integer])	text	Teilzeichenkette ersetzen	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(*substring* in *string*)	integer	Position der angegebenen Teilzeichenkette	position('om' in 'Thomas')	3
substring(*string* [from integer] [for integer])	text	Teilzeichenkette ermitteln	substring('Thomas' from 2 for 3)	hom
substring(*string* from *pattern*)	text	Teilzeichenkette nach POSIX regulärem Ausdruck ermitteln	substring('Thomas' from '...$')	mas
substring(*string* from *pattern* for *escape*)	text	Teilzeichenkette nach SQL regulärem Ausdruck ermitteln	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*zeichen*] from *string*)	text	Entfernt die längste Zeichenkette, die nur aus zeichen (als Vorgabe ein Leerzeichen) besteht, von Anfang/Ende/beiden Enden von string	trim(both 'x' from 'xTomxx')	Tom
upper(*string*)	text	in Großbuchstaben umwandeln	upper('tom')	TOM

Tabelle 9.6: SQL-Zeichenkettenfunktionen und -operatoren

Weitere Funktionen zur Zeichenkettenmanipulation sind in Tabelle 9.7 aufgeführt. Einige davon werden intern verwendet, um die SQL-Standard-Funktionen aus Tabelle 9.6 zu implementieren.

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
ascii(text)	integer	ASCII-Code des ersten Arguments	ascii('x')	120
btrim(*string* text, *zeichen* text)	text	Entfernt die längste Zeichenkette, die nur aus Zeichen in zeichen steht, vom Anfang und Ende von string	btrim('xyxtrimyyx','xy')	trim
chr(integer)	text	Zeichen mit dem angegebenen ASCII-Code	chr(65)	A
convert(*string* text, [*quellkodierung* name,] *zielkodierung* name)	text	Wandelt Zeichenkette von quellkodierung nach zielkodierung um. Wenn quellkodierung ausgelassen wird, dann wird die Datenbankkodierung angenommen.	convert('text_in_unicode', 'UNICODE', 'LATIN1')	text_in_unicode in der Kodierung ISO 8859-1
decode(*string* text, *typ* text)	bytea	Dekodiert binäre Daten von string, die zuvor mit encode kodiert wurden. Parametertyp ist genauso wie bei encode.	decode('MTIzAAE=', 'base64')	123\000\001
encode(*data* bytea, *typ* text)	text	Kodiert binäre Daten in eine Darstellung nur aus ASCII-Zeichen. Unterstützte Typen sind: base64, hex, escape.	encode('123\\000\\001', 'base64')	MTIzAAE=
initcap(text)	text	Wandelt ersten Buchstaben jedes Worts (durch Whitespace getrennt) in Großbuchstaben um.	initcap('hi thomas')	Hi Thomas
length(*string*)	integer	Anzahl Zeichen in Zeichenkette	length('jose')	4
lpad(*string* text, *länge* integer [, *füllung* text])	text	Füllt string bis zur Länge länge, indem Zeichen füllung (ein Leerzeichen als Vorgabe) vorangestellt werden. Wenn string schon länger als länge ist, dann wird es rechts abgeschnitten.	lpad('hi', 5, 'xy')	xyxhi
ltrim(*string* text, *zeichen* text)	text	Entfernt die längste Zeichenkette, die nur aus Zeichen in zeichen besteht, vom Anfang der Zeichenkette.	ltrim('zzzytrim','xyz')	trim
pg_client_encoding()	name	aktuelle Clientkodierung	pg_client_encoding()	SQL_ASCII
quote_ident(*string* text)	text	Gibt die Zeichenkette so zurück, dass sie als Name in einem SQL-Befehl verwendet werden kann. Anführungszeichen werden hinzugefügt, wenn nötig (d.h. die Zeichenkette enthält Sonderzeichen oder Großbuchstaben). Vorhandene Anführungszeichen werden korrekt verdoppelt.	quote_ident('Foo')	"Foo"
quote_literal(*string* text)	text	Gibt die Zeichenkette so zurück, dass sie als Zeichenkettenkonstante in einem SQL-Befehl verwendet werden kann. Vorhandene Apostrophe und Backslashes werden korrekt verdoppelt.	quote_literal('O\'Reilly')	'O''Reilly'
repeat(text, integer)	text	Wiederholt den Text so oft wie angegeben.	repeat('Pg', 4)	PgPgPgPg

Tabelle 9.7: Andere Zeichenkettenfunktionen

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
replace(*string* text, *vorher* text, *nachher* text)	text	Ersetzt alle Vorkommen von Teilzeichenkette vorher in string mit nachher .	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
rpad(*string* text, *länge* integer [, *füllung* text])	text	Füllt string bis zur Länge länge, indem Zeichen füllung (ein Leerzeichen als Vorgabe) angehängt werden. Wenn string schon länger als länge ist, dann wird es rechts abgeschnitten.	rpad('hi', 5, 'xy')	hixyx
rtrim(*string* text, zeichen text)	text	Entfernt die längste Zeichenkette, die nur aus Zeichen in zeichen besteht, vom Ende der Zeichenkette.	rtrim('trimxxxx','x')	trim
split_part(string text, *trennzeichen* text, *feld* integer)	text	Teilt string an den trennzeichen auf und ergibt das angegebene Feld (von eins an zählend).	split_part('abc~@~def~@~ghi','~@~',2)	def
strpos(*string*, *substring*)	text	Position der angegebenen Teilzeichenkette (das Gleiche wie position(substring in string), aber andere Reihenfolge der Argumente).	strpos('high', 'ig')	2
substr(*string*, *from* [, *count*])	text	Teilzeichenkette ermitteln (das Gleiche wie substring(string from from for count))	substr('alphabet', 3, 2)	ph
to_ascii(text [, *kodierung*])	text	Text von anderer Kodierung in ASCII umwandeln [a] .	to_ascii('Karel')	Karel
to_hex(*zahl* integer or bigint)	text	Wandelt zahl in hexadezimale Darstellung um.	to_hex(9223372036854775807)	7fffffffffffffff
translate(*string* text, *von* text, *nach* text)	text	Jedes Zeichen in string, dass gleich einem Zeichen in von ist, wird durch das entsprechende Zeichen in nach ersetzt.	translate('12345', '14', 'ax')	a23x5

Tabelle 9.7: Andere Zeichenkettenfunktionen (Forts.)

a. Die Funktion to_ascii unterstützt nur die Konversion von LATIN1, LATIN2 und WIN1250.

Konversionsname [a]	Quellkodierung	Zielkodierung
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf_8	SQL_ASCII	UNICODE
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf_8	BIG5	UNICODE
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf_8	EUC_CN	UNICODE
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf_8	EUC_JP	UNICODE

Tabelle 9.8: Eingebaute Konversionen

Konversionsname [a]	Quellkodierung	Zielkodierung
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf_8	EUC_KR	UNICODE
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf_8	EUC_TW	UNICODE
gb18030_to_utf_8	GB18030	UNICODE
gbk_to_utf_8	GBK	UNICODE
iso_8859_10_to_utf_8	LATIN6	UNICODE
iso_8859_13_to_utf_8	LATIN7	UNICODE
iso_8859_14_to_utf_8	LATIN8	UNICODE
iso_8859_15_to_utf_8	LATIN9	UNICODE
iso_8859_16_to_utf_8	LATIN10	UNICODE
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf_8	LATIN1	UNICODE
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf_8	LATIN2	UNICODE
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf_8	LATIN3	UNICODE
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf_8	LATIN4	UNICODE
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf_8	ISO_8859_5	UNICODE
iso_8859_5_to_windows_1251	ISO_8859_5	WIN
iso_8859_5_to_windows_866	ISO_8859_5	ALT
iso_8859_6_to_utf_8	ISO_8859_6	UNICODE
iso_8859_7_to_utf_8	ISO_8859_7	UNICODE
iso_8859_8_to_utf_8	ISO_8859_8	UNICODE
iso_8859_9_to_utf_8	LATIN5	UNICODE
johab_to_utf_8	JOHAB	UNICODE
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf_8	KOI8	UNICODE
koi8_r_to_windows_1251	KOI8	WIN
koi8_r_to_windows_866	KOI8	ALT
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5

Tabelle 9.8: Eingebaute Konversionen (Forts.)

Konversionsname [a]	Quellkodierung	Zielkodierung
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN
mic_to_windows_866	MULE_INTERNAL	ALT
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf_8	SJIS	UNICODE
tcvn_to_utf_8	TCVN	UNICODE
uhc_to_utf_8	UHC	UNICODE
utf_8_to_ascii	UNICODE	SQL_ASCII
utf_8_to_big5	UNICODE	BIG5
utf_8_to_euc_cn	UNICODE	EUC_CN
utf_8_to_euc_jp	UNICODE	EUC_JP
utf_8_to_euc_kr	UNICODE	EUC_KR
utf_8_to_euc_tw	UNICODE	EUC_TW
utf_8_to_gb18030	UNICODE	GB18030
utf_8_to_gbk	UNICODE	GBK
utf_8_to_iso_8859_1	UNICODE	LATIN1
utf_8_to_iso_8859_10	UNICODE	LATIN6
utf_8_to_iso_8859_13	UNICODE	LATIN7
utf_8_to_iso_8859_14	UNICODE	LATIN8
utf_8_to_iso_8859_15	UNICODE	LATIN9
utf_8_to_iso_8859_16	UNICODE	LATIN10
utf_8_to_iso_8859_2	UNICODE	LATIN2
utf_8_to_iso_8859_3	UNICODE	LATIN3
utf_8_to_iso_8859_4	UNICODE	LATIN4
utf_8_to_iso_8859_5	UNICODE	ISO_8859_5
utf_8_to_iso_8859_6	UNICODE	ISO_8859_6
utf_8_to_iso_8859_7	UNICODE	ISO_8859_7

Tabelle 9.8: Eingebaute Konversionen (Forts.)

Konversionsname [a]	Quellkodierung	Zielkodierung
utf_8_to_iso_8859_8	UNICODE	ISO_8859_8
utf_8_to_iso_8859_9	UNICODE	LATIN5
utf_8_to_johab	UNICODE	JOHAB
utf_8_to_koi8_r	UNICODE	KOI8
utf_8_to_sjis	UNICODE	SJIS
utf_8_to_tcvn	UNICODE	TCVN
utf_8_to_uhc	UNICODE	UHC
utf_8_to_windows_1250	UNICODE	WIN1250
utf_8_to_windows_1251	UNICODE	WIN
utf_8_to_windows_1256	UNICODE	WIN1256
utf_8_to_windows_866	UNICODE	ALT
utf_8_to_windows_874	UNICODE	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf_8	WIN1250	UNICODE
windows_1251_to_iso_8859_5	WIN	ISO_8859_5
windows_1251_to_koi8_r	WIN	KOI8
windows_1251_to_mic	WIN	MULE_INTERNAL
windows_1251_to_utf_8	WIN	UNICODE
windows_1251_to_windows_866	WIN	ALT
windows_1256_to_utf_8	WIN1256	UNICODE
windows_866_to_iso_8859_5	ALT	ISO_8859_5
windows_866_to_koi8_r	ALT	KOI8
windows_866_to_mic	ALT	MULE_INTERNAL
windows_866_to_utf_8	ALT	UNICODE
windows_866_to_windows_1251	ALT	WIN
windows_874_to_utf_8	WIN874	UNICODE

Tabelle 9.8: Eingebaute Konversionen (Forts.)

a. Die Konversionsnamen folgen einem einheitlichen Schema: Der offizielle Name der Quellkodierung mit allen nicht alphanumerischen Zeichen durch Unterstriche ersetzt, gefolgt von `_to_`, gefolgt vom auf gleiche Weise umgewandelten Zielkodierungsnamen. Daher könnten die Namen von den gebräuchlichen Kodierungsnamen abweichen.

9.5 Funktionen und Operatoren für binäre Daten

Dieser Abschnitt beschreibt Funktionen und Operatoren zur Bearbeitung von Werten des Typs bytea.

SQL definiert einige Funktionen mit spezieller Syntax, wo bestimmte Schlüsselwörter anstatt Kommas verwendet werden, um die Argumente zu trennen. Einzelheiten sehen Sie in Tabelle 9.9. Diese Funktionen gibt es auch mit der normalen Syntax für Funktionsaufrufe. (Siehe Tabelle 9.10.)

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
daten || daten	bytea	Datenketten aneinander hängen	'\\\\Post'::bytea || '\\047greSQL\\000'::bytea	\\Post'greSQL\000
octet_length(daten)	integer	Anzahl Bytes in Daten	octet_length('jo\\000se'::bytea)	5
position(subdaten in daten)	integer	Position der angegebenen Teildatenkette	position('\\000om'::bytea in 'Th\\000omas'::bytea)	3
substring(daten [from integer] [for integer])	bytea	Teildatenkette ermitteln	substring('Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from daten)	bytea	Entfernt die längste Datenkette, die nur aus Bytes in bytes besteht, vom Anfang und Ende von daten.	trim('\\000'::bytea from '\\000Tom\\000'::bytea)	Tom

Tabelle 9.9: SQL-Funktionen und -Operatoren für binäre Daten

Weitere Funktionen zur Zeichenkettenmanipulation sind in Tabelle 9.10 aufgeführt. Einige davon werden intern verwendet, um die SQL-Standard-Funktionen aus Tabelle 9.9 zu implementieren.

Funktion	Ergebnistyp	Beschränkung	Beispiel	Ergebnis
btrim(daten bytea bytes bytea)	bytea	Entfernt die längste Datenkette, die nur aus Bytes in bytes steht, vom Anfang und Ende von daten .	btrim('\\000trim\\000'::bytea,'\\000'::bytea)	trim
length(daten)	integer	Anzahl Bytes in Daten	length('jo\\000se'::bytea)	5
decode(daten text, typ text)	bytea	Dekodiert binäre Daten in daten, die zuvor mit encode kodiert wurden. Parametertyp ist genauso wie bei encode.	decode('123\\000456', 'escape')	123\000456
encode(daten bytea, typ text)	text	Kodiert binäre Daten in eine Darstellung nur aus ASCII-Zeichen. Unterstützte Typen sind: base64, hex, escape.	encode('123\\000456'::bytea, 'escape')	123\000456

Tabelle 9.10: Andere Funktionen für binäre Daten

9.6 Mustervergleiche

Es gibt drei verschiedene Möglichkeiten, in PostgreSQL Mustervergleiche durchzuführen: der herkömmliche SQL-Operator LIKE, der neuere SQL99-Operator SIMILAR TO und reguläre Ausdrücke nach POSIX. Außerdem gibt es die Funktion substring, die Mustervergleiche mit regulären Ausdrücken entweder nach SQL99 oder POSIX durchführen kann.

Tipp

Wenn sie Anforderungen für Mustervergleiche haben, die diese Fähigkeiten überschreiten, sollten Sie das Schreiben einer benutzerdefinierten Funktion in Perl oder Tcl in Betracht ziehen.

9.6.1 LIKE

```
zeichenkette LIKE muster [ESCAPE fluchtzeichen]
zeichenkette NOT LIKE muster [ESCAPE fluchtzeichen]
```

Jedes *muster* definiert eine Sammlung von Zeichenketten. Der LIKE-Ausdruck ergibt logisch wahr, wenn die *zeichenkette* in der von *muster* bestimmten Sammlung ist. (Wie zu erwarten ist, ergibt der Ausdruck NOT LIKE falsch, wenn LIKE wahr ergibt, und umgekehrt. Ein gleichbedeutender Ausdruck ist NOT (*zeichenkette* LIKE *muster*).)

Wenn *muster* kein Prozentzeichen oder Unterstrich enthält, dann steht das Muster nur für die Zeichenkette selbst; in dem Fall verhält sich LIKE wie die Vergleichsoperation. Ein Unterstrich (_) in *muster* steht für ein einzelnes beliebiges Zeichen; ein Prozentzeichen (%) steht für null oder mehr Zeichen.

Einige Beispiele:

```
'abc' LIKE 'abc' wahr
'abc' LIKE 'a%'  wahr
'abc' LIKE '_b_' wahr
'abc' LIKE 'c'   falsch
```

Mustervergleiche mit LIKE gehen immer über die gesamte Zeichenkette. Um ein Muster irgendwo in einer Zeichenkette zu finden, muss das Muster also am Anfang und Ende ein Prozentzeichen haben.

Um einen Unterstrich oder ein Prozentzeichen, aber keine anderen Zeichen, zu finden, muss vor die entsprechenden Zeichen in *muster* das Fluchtzeichen geschrieben werden. Das normale Fluchtzeichen ist der Backslash, aber ein anderes Zeichen kann mit der ESCAPE-Klausel gewählt werden. Um das Fluchtzeichen selbst zu finden, schreiben Sie es zweimal hintereinander in das Muster.

Beachten Sie, dass der Backslash in Zeichenkettenkonstanten eine besondere Bedeutung hat. Um also eine Musterkonstante mit einem Backslash zu schreiben, müssen Sie in einem SQL-Befehl zwei Backslashes schreiben. Wenn Sie folglich ein Muster schreiben wollen, dass einen Backslash findet, müssen Sie in dem Befehl vier eingeben. Das können Sie vermeiden, indem Sie mit ESCAPE ein anderes Fluchtzeichen bestimmen; dann hat der Backslash keine besondere Bedeutung für LIKE mehr. (Aber er hat immer noch eine besondere Bedeutung in Zeichenkettenkonstanten, also brauchen Sie immer noch zwei.)

Es ist möglich, gar kein Fluchtzeichen auszuwählen, indem man ESCAPE '' schreibt. Dadurch wird der Fluchtzeichenmechanismus im Prinzip ausgeschaltet, wodurch es unmöglich wird, die besondere Bedeutung des Unterstrichs und des Prozentzeichens im Muster zu umgehen.

Das Schlüsselwort ILIKE kann anstelle von LIKE verwendet werden, um die Mustersuche unabhängig von Groß- und Kleinschreibung zu machen (entsprechend der aktuellen Locale-Einstellung). Das steht nicht im SQL-Standard, sondern ist eine Erweiterung von PostgreSQL.

Der Operator ~~ ist gleichbedeutend mit LIKE und ~~* entspricht ILIKE. Es gibt auch die Operatoren !~~ und !~~* die für NOT LIKE bzw. NOT ILIKE stehen. Alle diese Operatoren gibt es so nur in PostgreSQL.

9.6.2 SIMILAR TO und reguläre Ausdrücke nach SQL99

```
zeichenkette SIMILAR TO muster [ESCAPE fluchtzeichen]
zeichenkette NOT SIMILAR TO muster [ESCAPE fluchtzeichen]
```

Der Operator SIMILAR TO ergibt wahr oder falsch abhängig davon, ob das Muster mit der angegebenen Zeichenkette übereinstimmt. Es ist ähnlich LIKE, außer dass es das Muster als regulären Ausdruck nach SQL99 interpretiert. Reguläre Ausdrücke nach SQL99 sind eine merkwürdige Kombination aus der von LIKE bekannten Schreibweise und der allgemein gebräuchlichen Schreibweise von regulären Ausdrücken.

Wie `LIKE`, ist `SIMILAR TO` nur erfolgreich, wenn das Muster auf die gesamte Zeichenkette passt. Das steht im Gegensatz zur gebräuchlichen Praxis bei regulären Ausdrücken, wo das Muster auch auf nur einen Teil der Zeichenkette passen kann. Ebenso wie `LIKE` verwendet `SIMILAR TO` _ und `%` als Wildcard-Zeichen für ein beliebiges einzelnes Zeichen bzw. jede beliebige Zeichenkette. (Diese sind vergleichbar mit `.` und `.*` in regulären Ausdrücken in der POSIX-Schreibweise.)

Neben diesen von `LIKE` übernommenen Möglichkeiten unterstützt `SIMILAR TO` auch folgende von POSIX übernommene Sonderzeichen zur Mustersuche:

- `|` steht für eine Alternative (entweder die eine oder die andere).
- `*` steht für die Wiederholung des vorangegangenen Ausdrucks null oder mehrere Male.
- `+` steht für die Wiederholung des vorangegangenen Ausdrucks ein oder mehrere Male.
- Klammern `()` können verwendet werden, um Ausdrücke zu einem einzigen logischen Ausdruck zusammenzufassen.
- Ein Ausdruck in eckigen Klammern `[...]` ist eine Zeichenklasse, wie bei POSIX.

Beachten Sie, dass begrenzte Wiederholung (`?` und `{...}`) nicht zur Verfügung steht, obwohl sie in POSIX vorhanden sind. Außerdem ist der Punkt (`.`) kein Sonderzeichen.

Genauso wie bei `LIKE` kann der Backslash die besondere Bedeutung dieser Sonderzeichen abschalten; oder ein anderes Fluchtzeichen kann mit `ESCAPE` bestimmt werden.

Einige Beispiele:

```
'abc' SIMILAR TO 'abc'     wahr
'abc' SIMILAR TO 'a'       falsch
'abc' SIMILAR TO '%(b|d)%' wahr
'abc' SIMILAR TO '(b|c)%'  falsch
```

Die Funktion `substring` mit drei Parametern, `substring(zeichenkette from muster for fluchtzeichen)`, erlaubt das Auswählen einer Teilzeichenkette, die mit dem regulären Ausdruck nach SQL99 übereinstimmt. Wie bei `SIMILAR TO` muss das Muster auf die gesamte Zeichenkette passen, ansonsten schlägt die Funktion fehl und gibt den NULL-Wert zurück. Um den Teil des Musters zu markieren, der bei erfolgreicher Übereinstimmung zurückgegeben werden soll, muss das Muster zweimal das Fluchtzeichen jeweils gefolgt von einem Anführungszeichen (") enthalten. Der Text, der mit dem Teil des Musters zwischen diesen Markierungen übereinstimmt, wird zurückgegeben.

Einige Beispiele:

```
substring('foobar' from '%#"o_b#"%' for '#') oob
substring('foobar' from '#"o_b#"%' for '#')  NULL
```

9.6.3 Reguläre Ausdrücke nach POSIX

Tabelle 9.11 listet die verfügbaren Operatoren für Mustervergleiche mit regulären Ausdrücken nach POSIX.

Operator	Beschreibung	Beispiel
~	Vergleich mit regulärem Ausdruck, achtet auf Groß-/Kleinschreibung	`'thomas' ~ '.*thomas.*'`
~*	Vergleich mit regulärem Ausdruck, ignoriert Groß-/Kleinschreibung	`'thomas' ~* '.*Thomas.*'`

Tabelle 9.11: Operatoren für reguläre Ausdrücke

Operator	Beschreibung	Beispiel
`!~`	negierter Vergleich mit regulärem Ausdruck, achtet auf Groß-/Kleinschreibung	`'thomas' !~ '.*Thomas.*'`
`!~*`	negierter Vergleich mit regulärem Ausdruck, ignoriert Groß-/Kleinschreibung	`'thomas' !~* '.*vadim.*'`

Tabelle 9.11: Operatoren für reguläre Ausdrücke (Forts.)

Die regulären Ausdrücke nach dem POSIX-Standard bieten eine mächtigere Schnittstelle für Mustervergleiche als die Operatoren `LIKE` und `SIMILAR TO`. Viele Unix-Werkzeuge, wie `egrep`, `sed` oder `awk`, verwenden eine Schreibweise für reguläre Ausdrücke, die der hier beschriebenen ähnlich ist.

Ein regulärer Ausdruck ist eine Zeichenfolge, die in abgekürzter Schreibweise eine Menge von Zeichenketten definiert (eine **reguläre Menge**). Eine Zeichenkette stimmt mit einem regulären Ausdruck überein, wenn sie ein Element der von dem regulären Ausdruck beschriebenen regulären Menge ist. Wie mit `LIKE` passen Zeichen im Muster genau auf Zeichen in der angegebenen Zeichenkette, außer wenn sie Sonderzeichen von regulären Ausdrücken sind. Aber reguläre Ausdrücke verwenden andere Sonderzeichen als `LIKE`. Im Gegensatz zu `LIKE`-Mustern können reguläre Ausdrücke auch auf Teilzeichenketten mitten in der zu durchsuchenden passen, außer wenn es im regulären Ausdruck explizit angegeben wurde, dass die Übereinstimmung vom Anfang oder bis zum Ende bestehen muss.

Einige Beispiele:

```
'abc' ~ 'abc'    wahr
'abc' ~ '^a'     wahr
'abc' ~ '(b|d)'  wahr
'abc' ~ '^(b|c)' falsch
```

Die Funktion `substring` mit zwei Parametern, `substring(zeichenkette from` *`muster`*`)`, erlaubt das Auswählen einer Teilzeichenkette, die mit dem regulären Ausdruck nach POSIX übereinstimmt. Wenn es keine Übereinstimmung gibt, dann ergibt sie den NULL-Wert, ansonsten ergibt sie den Teil des Textes der übereingestimmt hat. Wenn das Muster Klammern enthält, wird der Teil des Texts zurückgegeben, der mit dem ersten eingeklammerten Teilausdruck (der, von dem die linke Klammer zuerst kam) übereingestimmt hat. Wenn Sie Klammern verwenden wollen, ohne diesen Mechanismus auszulösen, können Sie um den ganzen Ausdruck ein Klammernpaar setzen.

Einige Beispiele:

```
substring('foobar' from 'o.b')   oob
substring('foobar' from 'o(.)b') o
```

Reguläre Ausdrücke (RAs), nach der Definition in POSIX 1003.2, gibt es in zwei Formen: moderne RAs (etwa die von `egrep` bekannten; in 1003.2 werden diese “erweiterte” RAs genannt) und obsolete RAs (etwa die aus `ed`; in 1003.2 “einfache” RAs). PostgreSQL bietet die moderne Form an.

Ein (moderner) RA besteht aus einem oder mehreren nicht leeren **Zweigen**, getrennt durch `|`. Er passt auf alles, was auf einen der Zweige passt.

Ein Zweig besteht aus einem oder mehreren aneinander gehängten **Teilen**. Er passt, wenn der Text auf alle Teile hintereinander passt.

Ein Teil besteht aus einem **Atom**, wahlweise gefolgt von einem einzelnen Zeichen `*`, `+`, `?` oder einer **Begrenzung**. Ein Atom, das von einem `*` gefolgt wird, passt auf eine Folge von 0 oder mehr Übereinstimmungen mit dem Atom. Ein Atom, das von einem `+` gefolgt wird, passt auf eine Folge von 1 oder mehr Übereinstimmungen mit dem Atom. Ein Atom gefolgt von `?` passt auf eine Folge von 0 oder 1 Übereinstimmungen mit dem Atom.

Eine **Begrenzung** besteht aus `{` gefolgt von einer ganzen Zahl ohne Vorzeichen (in Dezimalschreibweise), wahlweise gefolgt von `,` und wahlweise noch einer ganze Zahl ohne Vorzeichen, und am Ende immer `}`. Die Zahlen müssen zwischen einschließlich 0 und *RE_DUP_MAX* (255) liegen, und wenn zwei Zahlen angegeben wurden, darf die erste nicht größer als die zweite sein. Ein Atom gefolgt von einer Begrenzung mit einer Zahl *i* und keinem Komma passt auf eine Folge von genau *i* Übereinstimmungen mit dem Atom. Ein Atom gefolgt von einer Begrenzung mit einer Zahl *i* und einem Komma passt auf eine Folge von *i* oder mehr Übereinstimmungen mit dem Atom. Ein Atom gefolgt von einer Begrenzung mit zwei Zahlen *i* und *j* passt auf eine Folge von einschließlich *i* bis *j* Übereinstimmungen mit dem Atom.

Anmerkung

Ein Wiederholungsoperator (`?`, `*`, `+` oder eine Begrenzung) kann nicht auf einen anderen Wiederholungsoperator folgen. Ein Wiederholungsoperator kann nicht am Anfang eines Ausdrucks oder eines Teilausdrucks oder nach `^` oder `|` stehen.

Ein **Atom** ist ein regulärer Ausdruck in `()` (passt auf alles, was mit dem eingeschlossenen Ausdruck übereinstimmt), ein leeres Paar `()` (passt auf eine leere Zeichenkette), ein Ausdruck in ekkigen Klammern (siehe unten), ein `.` (passt auf ein einzelnes Zeichen), `^` (passt auf den Anfang der angegebenen Zeichenkette), `$` (passt auf das Ende der eingegebenen Zeichenkette), ein `\` gefolgt von eines der Zeichen `^.[$()|*+?{\` (passt auf das Zeichen ungeachtet seiner Stellung als Sonderzeichen), ein `\` gefolgt von einem anderen Zeichen (passt auf dieses Zeichen, als ob der `\` nicht da gewesen wäre) oder ein anderes Zeichen ohne besondere Bedeutung (passt auf dieses Zeichen). Ein `{` gefolgt von einem Zeichen, das keine Ziffer ist, zählt als normales Zeichen und nicht als Anfang einer Begrenzungsangabe. Es ist nicht erlaubt, am Ende eines RA ein `\` stehen zu haben.

Beachten Sie, dass ein Backslash (\) schon eine besondere Bedeutung in Zeichenkettenkonstanten hat. Um also eine Musterkonstante zu schreiben, die einen Backslash enthält, müssen Sie in dem Befehl zwei Backslashes schreiben.

Ein Ausdruck in eckigen Klammern ist eine Liste von Zeichen, die von `[]` eingeschlossen wird. Normalerweise passt er auf ein beliebiges Zeichen aus der Liste (siehe aber unten). Wenn die Liste mit `^` beginnt, passt er auf ein beliebiges Zeichen (siehe aber unten), das nicht in der Liste ist. Wenn zwei Zeichen in der Liste durch – getrennt sind, dann ist das eine Kurzschreibweise für den gesamten Bereich von Zeichen zwischen diesen beiden (einschließend) in der Sortierreihenfolge, z.B. passt `[0-9]` in ASCII auf eine beliebige dezimale Ziffer. Es ist nicht erlaubt, dass zwei Bereiche einen gemeinsamen Endpunkt haben, z.B. `a-c-e`. Die Bereiche hängen stark von Locale-Einstellungen ab und sollten von portierbaren Programmen vermieden werden.

Um ein `]` in die Liste zu setzen, schreiben Sie es als erstes Zeichen (nach einem etwaigen `^`). Um ein – in die Liste zu setzen, schreiben Sie es als erstes oder letztes Zeichen oder als zweiter Endpunkt eines Bereichs. Um ein – als Anfangspunkt eines Bereichs zu verwenden, schreiben Sie es zwischen `[.` und `.]`, um es in ein Sortierelement umzuwandeln (siehe unten). Mit Ausnahme dieser Konstruktion und einigen Kombinationen mit `[` (siehe nächster Absatz) verlieren alle anderen Sonderzeichen, einschließlich `\`, ihre besondere Bedeutung innerhalb von eckigen Klammern.

Wenn innerhalb eines Ausdrucks in eckigen Klammern ein Sortierelement (ein Zeichen, eine Folge aus mehreren Zeichen, die wie ein einzelnes Zeichen sortiert wird, oder ein besonderer Name für eins der beiden) zwischen `[.` und `.]` steht, dann steht es für die Zeichenfolge dieses Sortierelements. Diese Folge ist ein einzelnes Element in der eingeklammerten Liste. Ein Ausdruck in eckigen Klammern kann also auf mehr als ein Zeichen passen. Wenn zum Beispiel die Sortierreihenfolge ein `ch` als Sortierelement enthält, passt der RA `[[.ch.]]*c` auf die ersten fünf Zeichen von `chchcc`.

Wenn innerhalb eines Ausdrucks in eckigen Klammern ein Sortierelement zwischen `[=` und `=]` steht, ist es eine Äquivalenzklasse, die für die Zeichenfolgen aller Sortierelemente, die mit demjenigen äquivalent sind, einschließlich demjenigen selbst, steht. (Wenn es keine äquivalenten Sortierelemente gibt, verhält sich der Ausdruck, als wenn er stattdessen zwischen `[.` und `.]` stehen würde.) Wenn zum Beispiel `o` und `^` Elemente einer Äquivalenzklassen wären, dann wären `[[=o=]]`, `[[=^=]]` und `[o^]` alle gleichbedeutend. Eine Äquivalenzklasse darf nicht der Endpunkt eines Bereichs sein.

Wenn innerhalb eines Ausdrucks in eckigen Klammern der Name einer Zeichenklasse zwischen `[:` und `:]` steht, dann steht das für die Liste aller Zeichen, die zu dieser Klasse gehören. Standardisierte Namen von Zeichenklassen sind: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. Diese stehen für die Zeichenklassen, die in ctype (3) definiert sind. In bestimmten Locale-Einstellungen können weitere vorhanden sein. Eine Zeichenklasse darf nicht als Endpunkt eines Bereichs verwendet werden.

Es gibt zwei besondere Fälle von Ausdrücken in eckigen Klammern: `[[:<:]]` und `[[:>:]]` erzeugen eine Übereinstimmung am Anfang bzw. Ende eines Worts. Ein Wort ist definiert als eine Folge von Wortzeichen, wo weder davor noch danach ein weiteres Wortzeichen steht. Ein Wortzeichen ist ein “alnum”-Zeichen (definiert in ctype (3)) oder ein Unterstrich. Das ist eine Erweiterung, die zwar kompatibel mit POSIX 1003.2 ist, dort aber nicht angegeben wird; daher sollte sie mit Vorsicht verwendet werden, wenn die Software auf andere Systeme portierbar sein soll.

Wenn es sich ergibt, dass ein RA mit mehr als einer Teilzeichenkette übereinstimmt, ergibt der RA eine Übereinstimmung mit der, die zuerst in der Zeichenkette anfängt. Wenn der RA mit mehr als einer Teilzeichenkette an dieser Stelle übereinstimmen kann, nimmt er die längste. Teilausdrücke suchen auch eine Übereinstimmung mit den längstmöglichen Teilzeichenketten, mit der Einschränkung, dass die gesamte Übereinstimmung so lang wie möglich sein soll und dass Teilausdrücke, die eher in dem RA anfangen, Vorrang vor den später Anfangenden haben. Daraus ergibt sich, dass äußere Teilausdrücke Vorrang vor den tiefer in ihnen geschachtelten haben.

Die Länge einer Übereinstimmung wird in Zeichen, nicht Sortierelementen gezählt. Eine leere Zeichenkette wird als länger als gar keine Übereinstimmung erachtet. Zum Beispiel: `bb*` passt auf die drei mittleren Zeichen von `abbbc`; `(wee|week)(knights|nights)` passt auf alle zehn Zeichen von `weeknights`; wenn `(.*).*` mit `abc` verglichen wird, dann passt der Teilausdruck in Klammern auf alle drei Zeichen; und wenn `(a*)*` mit `bc` verglichen wird, dann ergeben der ganze RA wie auch der Teilausdruck in Klammern eine Übereinstimmung mit der leeren Zeichenkette.

Wenn ein Mustervergleich gewünscht wird, der Groß- und Kleinschreibung ignoriert, dann ist die Auswirkung, als ob alle Unterschiede zwischen Groß- und Kleinbuchstaben aus dem Alphabet verschwunden wären. Wenn ein alphabetisches Zeichen, das es als Groß- und als Kleinbuchstabe gibt, als normales Zeichen außerhalb von eckigen Klammern auftritt, dann wird es im Endeffekt in einen Ausdruck in eckigen Klammern umgewandelt, der beide Buchstabenformen enthält; so wird aus `x` zum Beispiel `[xX]`. Wenn es innerhalb einer Liste in eckigen Klammern auftritt, werden alle übrigen Formen des Buchstabens zu der Liste hinzugefügt; so wird zum Beispiel `[x]` in `[xX]` und `[^x]` in `[^xX]` verwandelt.

Es gibt keine bestimmte Höchstgrenze für die Länge eines RA, außer was den Hauptspeicher betrifft. Der Speicherverbrauch ist in etwa direkt proportional zur Länge des RA und im Großen und Ganzen unabhängig von dessen Komplexität, außer bei begrenzten Wiederholungen. Begrenzte Wiederholungen sind als Makroerweiterung implementiert, was viel Zeit und Platz kostet, wenn die Zahlen groß oder die begrenzten Wiederholungen geschachtelt sind. Ein RA wie zum Beispiel `((((a{1,100}){1,100}){1,100}){1,100}){1,100}` wird (letztendlich) den Speicher fast jeder bestehenden Maschine aufbrauchen. [1]

9.7 Datentyp-Formatierungsfunktionen

Die PostgreSQL-Formatierungsfunktionen bieten vielfältige Möglichkeiten, um verschiedene Datentypen (Datum/Zeit, diverse Zahlentypen) in formatierte Zeichenketten oder formatierte Zeichenketten in bestimmte Datentypen umzuwandeln. In Tabelle 9.12 sind sie gelistet. Diese Funktionen werden alle nach

1. Bedenken Sie, dass das 1994 geschrieben wurde. Die Zahlen haben sich wahrscheinlich verändert, aber das Problem bleibt bestehen.

dem gleichen Schema aufgerufen: Das erste Argument ist der zu formatierende Wert und das zweite Argument ist eine Mustervorlage, die das Ausgabe- oder Eingabeformat bestimmt.

Funktion	Ergebnistyp	Beschreibung	Beispiel
`to_char(timestamp, text)`	`text`	`timestamp` in Zeichenkette umwandeln	`to_char(current_timestamp, 'HH12:MI:SS')`
`to_char(interval, text)`	`text`	`interval` in Zeichenkette umwandeln	`to_char(interval '15h 2m 12s', 'HH24:MI:SS')`
`to_char(int, text)`	`text`	`integer` in Zeichenkette umwandeln	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	`real/double precision` in Zeichenkette umwandeln	`to_char(125.8::real, '999D9')`
`to_char(numeric, text)`	`text`	`numeric` in Zeichenkette umwandeln	`to_char(-125.8, '999D99S')`
`to_date(text, text)`	`date`	Zeichenkette in `date` umwandeln	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(text, text)`	`timestamp`	Zeichenkette in `timestamp` umwandeln	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	Zeichenkette in `numeric` umwandeln	`to_number('12,454.8-', '99G999D9S')`

Tabelle 9.12: Formatierungsfunktionen

In einer Ausgabemustervorlage (für `to_char`) werden bestimmte Muster erkannt und durch entsprechend formatierte Daten vom zu formatierenden Wert ersetzt. Jeglicher Text, der kein solches Muster ist, wird unverändert übernommen. Gleichermaßen bestimmen die Muster in einer Eingabemustervorlage (für alles außer `to_char`), welche Teile der angegebenen Zeichenkette zu betrachten sind und welche Werte dort zu finden sind.

Tabelle 9.13 zeigt die Muster, die zur Formatierung von Datums- und Zeitangaben zur Verfügung stehen.

Muster	Beschreibung
`HH`	Stunde (01-12)
`HH12`	Stunde (01-12)
`HH24`	Stunde (00-23)
`MI`	Minute (00-59)
`SS`	Sekunde (00-59)
`MS`	Millisekunde (000-999)
`US`	Mikrosekunde (000000-999999)
`SSSS`	Sekunden nach Mitternacht (0-86399)
`AM` oder `A.M.` oder `PM` oder `P.M.`	Vormittags-/Nachmittagsangabe (Großbuchstaben)
`am` oder `a.m.` oder `pm` oder `p.m.`	Vormittags-/Nachmittagsangabe (Kleinbuchstaben)
`Y,YYY`	Jahr (mind. 4 Ziffern) mit Komma zur Zifferngruppierung
`YYYY`	Jahr (mind. 4 Ziffern)
`YYY`	die letzten 3 Ziffern des Jahres
`YY`	die letzten 2 Ziffern des Jahres

Tabelle 9.13: Mustervorlagen für die Formatierung von Datum/Zeit

Muster	Beschreibung
Y	die letzte Ziffer des Jahres
BC oder B.C. oder AD oder A.D.	englische Angabe der Zeitrechnung (Großbuchstaben)
bc oder b.c. oder ad oder a.d.	englische Angabe der Zeitrechnung (Kleinbuchstaben)
MONTH	voller englischer Monatsname in Großbuchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
Month	voller englischer Monatsname in gemischten Buchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
month	voller englischer Monatsname in Kleinbuchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
MON	abgekürzter englischer Monatsname in Großbuchstaben (3 Zeichen)
Mon	abgekürzter englischer Monatsname in gemischten Buchstaben (3 Zeichen)
mon	abgekürzter englischer Monatsname in Kleinbuchstaben (3 Zeichen)
MM	Nummer des Monats (01-12)
DAY	voller englischer Wochentag in Großbuchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
Day	voller englischer Wochentag in gemischten Buchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
day	voller englischer Wochentag in Kleinbuchstaben (mit Leerzeichen auf 9 Zeichen aufgefüllt)
DY	abgekürzter englischer Wochentag in Großbuchstaben (3 Zeichen)
Dy	abgekürzter englischer Wochentag in gemischten Buchstaben(3 Zeichen)
dy	abgekürzter englischer Wochentag in Kleinbuchstaben (3 Zeichen)
DDD	Tag im Jahr (001-366)
DD	Tag im Monat (01-31)
D	Wochentag (1-7; Sonntag ist 1)
W	Woche im Monat (1-5) (Die erste Woche fängt am ersten Tag des Monats an.)
WW	Woche im Jahr (1-53) (Die erste Woche fängt am ersten Tag des Jahres an.)
IW	ISO-Wochennummer (Der erste Donnerstag des neuen Jahres ist in Woche Nummer 1.)
CC	Jahrhundert (2 Ziffern)
J	Julianischer Tag (Tage seit 1. Januar 4712 v.u.Z.)
Q	Quartal
RM	Monat in römischen Zahlen (I-XII; I=Januar) (Großbuchstaben)
rm	Monat in römischen Zahlen (i-xii; i=Januar) (Kleinbuchstaben)
TZ	Zeitzonenname (Großbuchstaben)
tz	Zeitzonenname (Kleinbuchstaben)

Tabelle 9.13: Mustervorlagen für die Formatierung von Datum/Zeit (Forts.)

Auf alle Muster können bestimmte Abänderungen angewendet werden, um ihr Verhalten zu verändern. Zum Beispiel wäre FMMonth das Muster Month mit der Abänderung FM. Tabelle 9.14 zeigt die Abänderungsmuster für die Formatierung von Datum und Zeit.

Abänderung	Beschreibung	Beispiel
FM Präfix	Füllmodus (unterbindet Auffüllen mit Leerzeichen und Nullen)	FMMonth
TH Suffix	englische Ordnungszahlendung (Großbuchstaben)	DDTH
th Suffix	englische Ordnungszahlendung (Kleinbuchstaben)	DDth
FX Präfix	globale Option Fixformat (siehe Verwendungshinweise)	FX Month DD Day
SP Suffix	(noch nicht implementiert)	DDSP

Tabelle 9.14: Abänderungsmuster für die Formatierung von Datum/Zeit

Verwendungshinweise zur Formatierung von Datum und Zeit:

- FM unterbindet das Füllen mit Nullen links und mit Leerzeichen rechts, was ansonsten verwendet wird, um dem Ergebnis eines Musters eine feste Länge zu geben.
- to_timestamp und to_date ignorieren Folgen von mehreren Leerzeichen im Eingabetext, wenn die Option FX nicht verwendet wird. FX muss das erste Element in der Mustervorlage sein. Zum Beispiel: `to_timestamp('2000 JUN', 'YYYY MON')` ist korrekt, aber `to_timestamp('2000 JUN', 'FXYYYY MON')` ergibt einen Fehler, weil to_timestamp nur ein Leerzeichen erwartet.
- Normaler Text kann in Mustervorlagen für to_char stehen und wird unverändert übernommen. Sie können eine Teilzeichenkette in Anführungszeichen setzen, um zu verhindern, dass Zeichen als eins der Muster verwendet werden. So wird zum Beispiel in `'"Hello Year "YYYY'` das YYYY durch das Jahr ersetzt, nicht aber das einzelne Y in Year.
- Wenn Sie ein Anführungszeichen im Ergebnis haben wollen, müssen Sie einen Backslash davor stellen, zum Beispiel `'\\"YYYY Month\\"'`. (Sie müssen zwei Backslashes eingeben, weil der Backslash schon eine besondere Bedeutung in Zeichenkettenkonstanten hat.)
- Die Umwandlungen mit YYYY von Text in timestamp oder date hat eine Einschränkung, wenn Sie Jahreszahlen mit mehr als vier Ziffern verwenden. Nach dem YYYY muss ein nichtnumerisches Zeichen oder Muster stehen, ansonsten werden immer vier Ziffern als das Jahr genommen. Zum Beispiel (mit dem Jahr 20000): `to_date('200001131', 'YYYYMMDD')` wird als 4-ziffriges Jahr interpretiert; setzen Sie stattdessen ein nichtnumerisches Trennzeichen nach das Jahr, wie `to_date('20000-1131', 'YYYY-MMDD')` oder `to_date('20000Nov31', 'YYYYMonDD')`.
- Die Formatwerte für Millisekunden (MS) und Mikrosekunden (US) in einer Umwandlung von Text in timestamp werden als Teil der Sekunden nach dem Komma verwendet. Zum Beispiel ergibt `to_timestamp('12:3', 'SS:MS')` nicht 3 Millisekunden, sondern 300, weil die Umwandlung 12 + 0,3 Sekunden zählt. Das bedeutet, dass bei der Mustervorlage SS:MS die Eingabewerte 12:3, 12:30 und 12:300 alle die gleiche Anzahl Millisekunden ergeben. Um drei Millisekunden zu erhalten, muss man 12:003 schreiben, was als 12 + 0,003 = 12,003 Sekunden gezählt wird.

  Hier ist ein komplexeres Beispiel: `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` ergibt 15 Stunden, 12 Minuten und 2 Sekunden + 20 Millisekunden + 1230 Mikrosekunden = 2,021230 Sekunden.

Tabelle 9.15 zeigt die Muster, die zur Formatierung von numerischen Werten zur Verfügung stehen.

Muster	Beschreibung
9	Wert mit der angegebenen Anzahl von Ziffern
0	Wert mit führenden Nullen
. (Punkt)	Punkt zur Trennung von Nachkommastellen
, (Komma)	Komma als Tausendergruppierung

Tabelle 9.15: Mustervorlagen für die Formatierung von numerischen Werten

Muster	Beschreibung
PR	negativer Wert in spitzen Klammern
S	Vorzeichen direkt neben der Zahl (verwendet Locale)
L	Währungssymbol (verwendet Locale)
D	Trennzeichen für Nachkommastellen nach Locale (also Komma auf Deutsch)
G	Zeichen zur Tausendergruppierung nach Locale (Punkt auf Deutsch)
MI	Minuszeichen auf angegebener Position (wenn Zahl < 0)
PL	Pluszeichen auf angegebener Position (wenn Zahl > 0)
SG	Plus-/Minuszeichen auf angegebener Position
RN	römische Zahl (Eingabe zwischen 1 und 3999)
TH oder th	englische Ordnungszahlendung
V	um angegebene Anzahl Ziffern verschieben (siehe Hinweise)
EEEE	wissenschaftliche Schreibweise (noch nicht implementiert)

Tabelle 9.15: Mustervorlagen für die Formatierung von numerischen Werten (Forts.)

- Verwendungshinweise zur Formatierung von numerischen Werten:
- Ein mit SG, PL oder MI formatiertes Vorzeichen steht nicht direkt neben der Zahl; zum Beispiel, to_char(-12, 'S9999') ergibt ' -12', aber to_char(-12, 'MI9999') ergibt '- 12'. Die Oracle-Implementierung erlaubt nicht die Verwendung von MI vor 9, sondern erfordert, dass 9 vor MI steht.
- 9 ergibt einen Wert mit der gleichen Zahl von Ziffern, wie 9en vorhanden sind. Wenn keine Ziffer verfügbar ist, dann wird ein Leerzeichen ausgegeben.
- TH funktioniert nicht mit Werten unter null oder mit Zahlen mit Bruchteilen.
- PL, SG und TH sind PostgreSQL-Erweiterungen.
- V multipliziert den Eingabewert mit 10^n, wobei *n* die Anzahl der Ziffern nach V ist. to_char unterstützt nicht die Verwendung von V mit einen Dezimalpunkt. (Zum Beispiel ist 99.9V99 nicht erlaubt.)

Tabelle 9.16 zeigt einige Beispiele, wie die Funktion to_char verwendet werden kann.

Ausdruck	Ergebnis
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'

Tabelle 9.16: Beispiele für to_char

Ausdruck	Ergebnis
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

Tabelle 9.16: Beispiele für `to_char` (Forts.)

9.8 Funktionen und Operatoren für Datum und Zeit

Tabelle 9.18 listet die verfügbaren Operatoren zur Verarbeitung von Datums- und Zeitangaben; Einzelheiten finden Sie in den folgenden Unterabschnitten. Tabelle 9.17 zeigt das Verhalten der grundlegenden arithmetischen Operatoren (+, * usw.). Formatierungsfunktionen finden Sie in Abschnitt 9.7. Sie sollten mit den Hintergrundinformationen über die Datums- und Zeittypen aus Abschnitt 8.5 vertraut sein.

Alle Funktionen, bei denen unten steht, dass sie Werte vom Typ time oder timestamp verwenden, gibt es in Wirklichkeit in zwei Varianten: Eine verwendet time with time zone bzw. timestamp with time zone und eine verwendet time without time zone bzw. timestamp without time zone. Des Platzes halber sind diese Varianten nicht getrennt aufgeführt.

Operator	Beispiel	Ergebnis
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	time '01:00' + interval '3 hours'	time '04:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	interval '2 hours' - time '05:00'	time '03:00:00'

Tabelle 9.17: Operatoren für Datum/Zeit

Operator	Beispiel	Ergebnis
*	`interval '1 hour' * int '3'`	`interval '03:00'`
/	`interval '1 hour' / int '3'`	`interval '00:20'`

Tabelle 9.17: Operatoren für Datum/Zeit (Forts.)

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
`age(timestamp)`	`interval`	Subtraktion vom heutigen Datum	`age(timestamp '1957-06-13')`	`43 years 8 mons 3 days`
`age(timestamp, timestamp)`	`interval`	Subtraktion	`age('2001-04-10', timestamp '1957-06-13')`	`43 years 9 mons 27 days`
`current_date`	`date`	heutiges Datum; siehe Abschnitt 9.8.4		
`current_time`	`time with time zone`	aktuelle Zeit; siehe Abschnitt 9.8.4		
`current_timestamp`	`timestamp with time zone`	aktuelle Zeit mit Datum; siehe Abschnitt 9.8.4		
`date_part(text, timestamp)`	`double precision`	Teilfeld ermitteln (gleichbedeutend mit `extract`); siehe Abschnitt 9.8.1	`date_part('hour', timestamp '2001-02-16 20:38:40')`	`20`
`date_part(text, interval)`	`double precision`	Teilfeld ermitteln (gleichbedeutend mit `extract`); siehe Abschnitt 9.8.1	`date_part('month', interval '2 years 3 months')`	`3`
`date_trunc(text, timestamp)`	`timestamp`	auf angegebene Genauigkeit abschneiden; siehe auch Abschnitt 9.8.2	`date_trunc('hour', timestamp '2001-02-16 20:38:40')`	`2001-02-16 20:00:00+00`
`extract(`*`feld`*` from timestamp)`	`double precision`	Teilfeld ermitteln; siehe auch Abschnitt 9.8.1	`extract(hour from timestamp '2001-02-16 20:38:40')`	`20`
`extract(`*`feld`*` from interval)`	`double precision`	Teilfeld ermitteln; siehe auch Abschnitt 9.8.1	`extract(month from interval '2 years 3 months')`	`3`
`isfinite(timestamp)`	`boolean`	auf endlichen Wert prüfen (ungleich `infinity`)	`isfinite(timestamp '2001-02-16 21:28:30')`	`true`
`isfinite(interval)`	`boolean`	auf endlichen Wert prüfen (ungleich `infinity`)	`isfinite(interval '4 hours')`	`true`
`localtime`	`time`	aktuelle Zeit; siehe Abschnitt 9.8.4		
`localtimestamp`	`timestamp`	aktuelle Zeit mit Datum; siehe Abschnitt 9.8.4		
`now()`	`timestamp with time zone`	aktuelle Zeit mit Datum (gleichbedeutend mit `current_timestamp`); siehe Abschnitt 9.8.4		
`timeofday()`	`text`	aktuelle Zeit mit Datum; siehe Abschnitt 9.8.4	`timeofday()`	`Wed Feb 21 17:01:13.000126 2001 EST`

Tabelle 9.18: Funktionen für Datum/Zeit

9.8.1 EXTRACT, date_part

```
EXTRACT (feld FROM quelle)
```

Die Funktion `extract` ermittelt aus Datums- und Zeitangaben Teilfelder, wie zum Beispiel das Jahr oder die Stunde. *quelle* ist ein Wertausdruck vom Typ timestamp oder interval. (Ausdrücke mit Ergebnistyp date oder time werden automatisch in timestamp umgewandelt und können daher auch verwendet werden.) *feld* ist ein Name oder eine Zeichenkette, die angibt, welches Feld aus dem Eingabewert ermittelt werden soll. Der Ergebnistyp der Funktion `extract` ist double precision. Gültige Feldnamen sind folgende:

`century`

Das Jahr geteilt durch 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 20
```

Beachten Sie, dass das Ergebnis einfach das Jahr geteilt durch 100 ist und nicht die gebräuchliche Definition eines Jahrhunderts, nach der mindestens die Jahre 1900 bis 1999 ins 20. Jahrhundert gehören.

`day`

Die Nummer des Tages im Monat (1-31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 16
```

`decade`

Das Jahr geteilt durch 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 200
```

`dow`

Der Wochentag (0-6; Sonntag ist 0) (nur für timestamp-Werte)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 5
```

`doy`

Die Nummer des Tages im Jahr (1-365/366) (nur für timestamp-Werte)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 47
```

`epoch`

Bei date- und timestamp-Werten die Anzahl der Sekunden seit 1970-01-01 00:00:00-00 (möglicherweise negativ); bei interval-Werten die Gesamtzahl der Sekunden in der Zeitspanne

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 982352320
```

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Ergebnis: 442800
```

`hour`

Das Stundenfeld (0-23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 20
```

`microseconds`

Das Sekundenfeld, einschließlich Bruchteile, multipliziert mit 1 000 000. Beinhaltet also auch ganze Sekunden.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Ergebnis: 28500000
```

`millennium`

Das Jahr geteilt durch 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 2
```

Beachten Sie, dass das Ergebnis einfach das Jahr geteilt durch 1000 ist und nicht die gebräuchliche Definition eines Jahrtausends, nach der zum Beispiel die Jahre 1900 bis 1999 ins zweite Jahrtausend gehören.

`milliseconds`

Das Sekundenfeld, einschließlich Bruchteile, multipliziert mit 1000. Beinhaltet also auch ganze Sekunden.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Ergebnis: 28500
```

`minute`

Das Minutenfeld (0-59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 38
```

`month`

Bei timestamp-Werten die Nummer des Monats im Jahr (1-12); bei interval-Werten der Rest, wenn man die Anzahl der Monate durch 12 teilt (0-11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Ergebnis: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Ergebnis: 1
```

`quarter`

Das Quartal (1-4) in dem sich der Tag innerhalb des Jahres befindet (nur für timestamp-Werte)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 1
```

`second`

Das Sekundenfeld, einschließlich Bruchteile (0-59 [2])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Ergebnis: 28.5
```

`timezone_hour`

Der Stundenteil des Zeitzonenunterschieds

`timezone_minute`

Der Minutenteil des Zeitzonenunterschieds

`week`

Die Nummer der Woche, in der sich der Tag befindet, innerhalb des Jahres. Laut Definition (ISO 8601) enthält die erste Woche eines Jahres den 4. Januar dieses Jahres. (Nach ISO 8601 fängt eine Woche am Montag an.) Anders ausgedrückt, ist der erste Donnerstag eines Jahres in der Woche Nummer 1 dieses Jahres (nur für timestamp-Werte).

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 7
```

`year`

Das Jahr

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 2001
```

Die Funktion `extract` ist vornehmlich für Berechnungen gedacht. Um Datums- und Zeitangaben für die Ausgabe zu formatieren, schauen Sie in Abschnitt 9.7.

Die Funktion `date_part` ist der alten Funktion aus Ingres nachempfunden, entspricht aber der Funktion `extract` aus dem SQL-Standard:

```
date_part('feld', quelle)
```

Beachten Sie, dass der Parameter *feld* ein Zeichenkettenwert sein muss, kein Name. Die gültigen Feldnamen für `date_part` sind dieselben wie bei `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 16
```

2. 60 wenn das Betriebssystem Schaltsekunden unterstützt

```
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Ergebnis: 4
```

9.8.2 date_trunc

Die Funktion `date_trunc` entspricht in etwa der Funktion `trunc` für Zahlen.

```
date_trunc('feld', quelle)
```

quelle ist ein Wertausdruck vom Typ timestamp. (Werte vom Typ date und time werden automatisch umgewandelt.) *feld* bestimmt, auf welche Genauigkeit der Wert abgeschnitten werden soll. Der Rückgabewert ist vom Typ timestamp, wobei alle Felder, die kleiner als das angegebene sind, auf null gesetzt sind (oder eins, bei Tag und Monat).

Gültige Werte für *feld* sind:

```
microseconds
milliseconds
second
minute
hour
day
month
year
decade
century
millennium
```

Beispiele:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Ergebnis: 2001-01-01 00:00:00+00
```

9.8.3 AT TIME ZONE

Die Klausel `AT TIME ZONE` ermöglicht die Umrechnung von Zeitangaben in andere Zeitzonen. Tabelle 9.19 zeigt ihre Varianten.

Ausdruck	Ergebnistyp	Beschreibung
`timestamp without time zone AT TIME ZONE zone`	`timestamp with time zone`	Ortszeit in angegebener Zeitzone in UTC umwandeln
`timestamp with time zone AT TIME ZONE zone`	`timestamp without time zone`	UTC in Ortszeit in angegebener Zeitzone umwandeln
`time with time zone AT TIME ZONE zone`	`time with time zone`	Ortszeit in andere Zeitzone übertragen

Tabelle 9.19: Varianten von `AT TIME ZONE`

In diesen Ausdrücken kann die gewünschte Zeitzone *zone* als Zeichenkettenkonstante (z.B. `'PST'`) oder als interval-Wert (z.B. `INTERVAL '-08:00'`) angegeben werden.

Beispiele (mit aktueller Zeitzone `PST8PDT`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Ergebnis: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Ergebnis: 2001-02-16 18:38:40
```

Das erste Beispiel nimmt eine Zeit ohne Zeitzone, nimmt die Zeitzone MST (UTC-7) an, berechnet die Zeit in UTC und rechnet sie dann in PST (UTC-8) zur Ausgabe um. Das zweite Beispiel nimmt eine Zeit in der Zone EST (UTC-5) und wandelt sie in eine Ortszeit in der Zone MST (UTC-7) um.

Die Funktion `timezone(`*`zone, timestamp`*`)` ist gleichbedeutend mit der Konstruktion *`timestamp`* `AT TIME ZONE` *`zone`*. Letztere entspricht dem SQL-Standard.

9.8.4 Aktuelle Zeit

Die folgenden Funktionen stehen zur Verfügung, um die aktuelle Zeit und/oder das aktuelle Datum abzufragen:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME ( präzision )
CURRENT_TIMESTAMP ( präzision )
LOCALTIME
LOCALTIMESTAMP
LOCALTIME ( präzision )
LOCALTIMESTAMP ( präzision )
```

Die Ergebnisse von `CURRENT_TIME` und `CURRENT_TIMESTAMP` sind mit Zeitzone; die von `LOCALTIME` und `LOCALTIMESTAMP` sind ohne Zeitzone.

Nach `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` und `LOCALTIMESTAMP` kann wahlweise ein Parameter stehen, der die Präzision angibt, wodurch das Ergebnis auf die angegebene Zahl von Nachkommastellen im Sekundenfeld gerundet wird. Ohne den Präzisionsparameter enthält das Ergebnis die volle mögliche Präzision.

Anmerkung

Vor PostgreSQL 7.2 gab es die Präzisionsparameter nicht und das Ergebnis hatte immer nur ganzzahlige Sekunden.

Einige Beispiele:

```
SELECT CURRENT_TIME;
Ergebnis: 14:39:53.662522-05

SELECT CURRENT_DATE;
```

```
Ergebnis: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Ergebnis: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Ergebnis: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Ergebnis: 2001-12-23 14:39:53.662522
```

Die Funktion `now()` ist gleichbedeutend mit `CURRENT_TIMESTAMP`. Sie ist eine traditionelle, PostgreSQL-spezifische Variante.

Ferner gibt es die Funktion `timeofday()`, welche aus historischen Gründen einen Wert vom Typ text anstatt timestamp zurückgibt:

```
SELECT timeofday();
Ergebnis: Sat Feb 17 19:07:32.000126 2001 EST
```

Es ist wichtig zu wissen, dass `CURRENT_TIMESTAMP` und die verwandten Funktionen die Startzeit der aktuellen Transaktion ergeben; ihre Ergebnisse ändern sich während einer Transaktion nicht. Das Ergebnis von `timeofday()` ist allerdings die tatsächliche Zeit, die sich auch während einer Transaktion ändert.

Anmerkung

In anderen Datenbanksystemen könnte es sein, dass diese Werte sich auch während einer Transaktion ändern.

Alle Datums- und Zeittypen akzeptieren außerdem die spezielle Konstante `now` als Eingabe, um die aktuelle Zeit anzugeben. Folglich haben alle drei dieser Befehle das gleiche Ergebnis:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

Anmerkung

Die dritte Form sollten Sie höchstwahrscheinlich nicht in einer `DEFAULT`-Klausel verwenden, wenn Sie eine Tabelle erstellen. Das System wandelt `now` in timestamp um, sobald die Konstante gelesen wird, sodass, wenn der Vorgabewert benötigt wird, die Zeit der Erzeugung der Tabelle herauskommen würde! Die ersten beiden Formen werden erst ausgewertet, wenn der Vorgabewert verwendet wird, weil sie als Funktionsaufrufe zählen. Daher bieten sie das gewünschte Verhalten, dass der Vorgabewert die Zeit des Einfügens der Zeile ergibt.

9.9 Geometrische Funktionen und Operatoren

Die geometrischen Typen point, box, lseg, line, path, polygon und circle haben eine große Zahl zugehöriger Funktionen und Operatoren, welche in Tabelle 9.20, Tabelle 9.21 und Tabelle 9.22 gezeigt werden.

Operator	Beschreibung	Beispiel
+	Verschiebung	box '((0,0),(1,1))' + point '(2.0,0)'
-	Verschiebung	box '((0,0),(1,1))' - point '(2.0,0)'
*	Dehnung/Drehung	box '((0,0),(1,1))' * point '(2.0,0)'
/	Dehnung/Drehung	box '((0,0),(2,2))' / point '(2.0,0)'
#	Schnittpunkt oder -rechteck	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Anzahl der Punkte in Pfad oder Polygon	# '((1,0),(0,1),(-1,0))'
##	zu erstem Operanden nächstliegender Punkt auf zweitem Operanden	point '(0,0)' ## lseg '((2,0),(0,2))'
&&	liegt Überschneidung vor?	box '((0,0),(1,1))' && box '((0,0),(2,2))'
&<	liegt Überschneidung links vor?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	liegt Überschneidung rechts vor?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<->	Abstand	circle '((0,0),1)' <-> circle '((5,0),1)'
<<	liegt links von?	circle '((0,0),1)' << circle '((5,0),1)'
<^	liegt unterhalb von?	circle '((0,0),1)' <^ circle '((0,5),1)'
>>	liegt rechts von?	circle '((5,0),1)' >> circle '((0,0),1)'
>^	liegt oberhalb von?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	schneidet?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	ist horizontal?	point '(1,0)' ?- point '(0,0)'
?-|	steht senkrecht auf?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
@-@	Länge oder Umfang	@-@ path '((0,0),(1,0))'
?|	ist vertikal?	point '(0,1)' ?| point '(0,0)'
?||	sind parallel?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@	liegt innerhalb oder darauf?	point '(1,1)' @ circle '((0,0),2)'
@@	Mittelpunkt	@@ circle '((0,0),10)'
~=	identisch?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Tabelle 9.20: Geometrische Operatoren

Funktion	Ergebnistyp	Beschreibung	Beispiele
area(*objekt*)	double precision	Flächeninhalt	area(box '((0,0),(1,1))')
box(box, box)	box	Schnittrechteck	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(*objekt*)	point	Mittelpunkt	center(box '((0,0),(1,2))')
diameter(circle)	double precision	Durchmesser des Kreises	diameter(circle '((0,0),2.0)')
height(box)	double precision	Höhe des Rechtecks	height(box '((0,0),(1,1))')
isclosed(path)	boolean	geschlossener Pfad?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	offener Pfad?	isopen(path '[(0,0),(1,1),(2,0)]')
length(*objekt*)	double precision	Länge	length(path '((-1,0),(1,0))')
npoints(path)	integer	Anzahl der Punkte	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	Anzahl der Punkte	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	Pfad in geschlossenen umwandeln	popen(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	Pfad in offenen umwandeln	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	Radius des Kreises	radius(circle '((0,0),2.0)')
width(box)	double precision	Breite des Rechtecks	width(box '((0,0),(1,1))')

Tabelle 9.21: Geometrische Funktionen

Funktion	Ergebnistyp	Beschreibung	Beispiel
box(circle)	box	Kreis in Rechteck	box(circle '((0,0),2.0)')
box(point, point)	box	Punkte in Rechteck	box(point '(0,0)', point '(1,1)')
box(polygon)	box	Polygon in Rechteck	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	Rechteck in Kreis	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	Punkt und Radius in Kreis	circle(point '(0,0)', 2.0)
lseg(box)	lseg	Rechteckdiagonale in Strecke	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	Punkte in Strecke	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	Polygon in Pfad	path(polygon '((0,0),(1,1),(2,0))')
point(circle)	point	Mittelpunkt des Kreises	point(circle '((0,0),2.0)')
point(lseg, lseg)	point	Schnittpunkt	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	Mittelpunkt des Polygons	point(polygon '((0,0),(1,1),(2,0))')

Tabelle 9.22: Geometrische Typumwandlungsfunktionen

Funktion	Ergebnistyp	Beschreibung	Beispiel
`polygon(box)`	`polygon`	Rechteck in 4-Punkte-Polygon	`polygon(box '((0,0),(1,1))')`
`polygon(circle)`	`polygon`	Kreis in 12-Punkte Polygon	`polygon(circle '((0,0),2.0)')`
`polygon(`*`npkte`*`, circle)`	`polygon`	Kreis in *npkte*-Punkte-Polygon	`polygon(12, circle '((0,0),2.0)')`
`polygon(path)`	`polygon`	Pfad in Polygon	`polygon(path '((0,0),(1,1),(2,0))')`

Tabelle 9.22: Geometrische Typumwandlungsfunktionen (Forts.)

Es ist möglich, auf die beiden Zahlen in einem Wert des Typs point zuzugreifen, als ob point ein Array mit Indizes 0 und 1 wäre. Wenn zum Beispiel `t.p` eine Spalte vom Typ point ist, dann ergibt `SELECT p[0] FROM t` die X-Koordinate und `UPDATE t SET p[1] = ...` ändert die Y-Koordinate. Ebenso können Werte vom Typ box und lseg als Array zweier Werte vom Typ point behandelt werden.

9.10 Funktionen für Netzwerkadresstypen

Tabelle 9.23 zeigt die Operatoren, die für die Typen cidr und inet zur Verfügung stehen. Die Operatoren <<, <<=, >> und >>= überprüfen, ob ein Subnetz in einem anderen Netz enthalten ist. Sie betrachten dafür nur den Netzwerkteil der beiden Adressen und ignorieren einen möglichen Host-Teil.

Operator	Beschreibung	Beispiel
`<`	ist kleiner als	`inet '192.168.1.5' < inet '192.168.1.6'`
`<=`	ist kleiner als oder gleich	`inet '192.168.1.5' <= inet '192.168.1.5'`
`=`	ist gleich	`inet '192.168.1.5' = inet '192.168.1.5'`
`>=`	ist größer als oder gleich	`inet '192.168.1.5' >= inet '192.168.1.5'`
`>`	ist größer als	`inet '192.168.1.5' > inet '192.168.1.4'`
`<>`	ist ungleich	`inet '192.168.1.5' <> inet '192.168.1.4'`
`<<`	ist enthalten in	`inet '192.168.1.5' << inet '192.168.1/24'`
`<<=`	ist enthalten in oder ist gleich	`inet '192.168.1/24' <<= inet '192.168.1/24'`
`>>`	enthält	`inet'192.168.1/24' >> inet '192.168.1.5'`
`>>=`	enthält oder ist gleich	`inet '192.168.1/24' >>= inet '192.168.1/24'`

Tabelle 9.23: Operatoren für cidr und inet

Tabelle 9.24 zeigt die Funktionen, die für die Typen cidr und inet zur Verfügung stehen. Die Funktionen `host`, `text` und `abbrev` bieten in der Hauptsache alternative Ausgabeformate für Werte dieser Typen. Sie können einen Textwert mit der normalen Umwandlungssyntax in inet umwandeln: `inet(`*`ausdruck`*`)` oder *`spaltennamen`*`::inet`.

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
`broadcast(inet)`	`inet`	Broadcast-Adresse des Netzwerks	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`host(inet)`	`text`	IP-Adresse als Text auslesen	`host('192.168.1.5/24')`	`192.168.1.5`

Tabelle 9.24: Funktionen für cidr und inet

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
`masklen(inet)`	`integer`	Netzmaskenlänge auslesen	`masklen('192.168.1.5/24')`	`24`
`set_masklen(inet,integer)`	`inet`	Netzmaskenlänge für `inet`-Wert setzen	`set_masklen('192.168.1.5/24',16)`	`192.168.1.5/16`
`netmask(inet)`	`inet`	Netzmaske des Netzwerks ermitteln	`netmask('192.168.1.5/24')`	`255.255.255.0`
`network(inet)`	`cidr`	Netzwerkteil der Adresse auslesen	`network('192.168.1.5/24')`	`192.168.1.0/24`
`text(inet)`	`text`	IP-Adresse und Netzmaskenlänge als Text auslesen	`text(inet '192.168.1.5')`	`192.168.1.5/32`
`abbrev(inet)`	`text`	abgekürztes Ausgabeformat als Text	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`

Tabelle 9.24: Funktionen für cidr und inet (Forts.)

Tabelle 9.25 zeigt die Funktionen, die für den Typ macaddr zur Verfügung stehen. Die Funktion `trunc(macaddr)` ergibt die MAC-Adresse mit den letzten 3 Bytes auf null gesetzt. Damit kann der übrig gebliebene Präfix mit einem Hersteller in Verbindung gebracht werden. Das Verzeichnis `contrib/mac` in der Quelltext-Distribution enthält einige Hilfsmittel, um eine solche Assoziierungstabelle zu erstellen und zu pflegen.

Funktion	Ergebnistyp	Beschreibung	Beispiel	Ergebnis
`trunc(macaddr)`	`macaddr`	letzte 3 Bytes auf null setzen	`trunc(macaddr '12:34:56:78:90:ab')`	`12:34:56:00:00:00`

Tabelle 9.25: Funktionen für macaddr

Der Typ macaddr hat auch die normalen Vergleichsoperatoren (>, <= usw.) zur lexikographischen Sortierung.

9.11 Funktionen zur Bearbeitung von Sequenzen

Dieser Abschnitt beschreibt die Funktion von PostgreSQL zur Verwendung mit **Sequenzobjekten**. Sequenzobjekte (auch Sequenzgeneratoren oder einfach Sequenzen genannt) sind besondere Tabellen mit einer einzigen Zeile, die mit dem Befehl `CREATE SEQUENCE` erzeugt werden. Ein Sequenzobjekt wird normalerweise verwendet, um für die Zeilen einer Tabelle laufende Nummern zur eindeutigen Identifikation zu erzeugen. Die Sequenzfunktionen, welche in Tabelle 9.26 gelistet sind, bieten einfache und für den Mehrbenutzerbetrieb geeignete Methoden, um laufende Werte aus Sequenzobjekten zu erhalten.

Funktion	Ergebnistyp	Beschreibung
`nextval(text)`	`bigint`	Sequenz erhöhen und neuen Wert zurückgeben
`currval(text)`	`bigint`	Wert, der zuletzt durch `nextval` erzeugt wurde
`setval(text, bigint)`	`bigint`	aktuellen Wert der Sequenz setzen
`setval(text, bigint, boolean)`	`bigint`	aktuellen Wert der Sequenz und den Parameter `is_called` setzen

Tabelle 9.26: Sequenzfunktionen

Aus im Großen und Ganzen historischen Gründen wird die Sequenz, die durch eine Sequenzfunktion bearbeitet wird, durch ein Textargument angegeben. Um eine gewisse Kompatibilität mit der Behandlung von normalen SQL-Namen zu erreichen, wandeln die Sequenzfunktionen ihre Argumente in Kleinbuchstaben um, außer wenn die Zeichenkette in Anführungszeichen steht. Also gilt:

```
nextval('foo')   betrifft die Sequenz foo
nextval('FOO')   betrifft die Sequenz foo
nextval('"Foo"') betrifft die Sequenz Foo
```

Der Sequenzname kann, wenn notwendig, um einen Schemanamen ergänzt werden:

```
nextval('meinschema.foo')   betrifft meinschema.foo
nextval('"meinschema".foo') ebenso
nextval('foo')              sucht im Pfad nach foo
```

Natürlich kann das Textargument ein Ausdruck sein und nicht nur eine einfach Konstante. Das kann gelegentlich nützlich sein.

Die verfügbaren Sequenzfunktionen sind:

`nextval`

Erhöhe die Sequenz zum nächsten Wert und gib diesen Wert zurück. Das geschieht atomar: Selbst wenn mehrere Sitzungen `nextval` gleichzeitig aufrufen, erhält jede Sitzung einen unterschiedlichen Sequenzwert.

`currval`

Gib den Wert zurück, der zuletzt in der aktuellen Sitzung von `nextval` für diese Sequenz zurückgegeben wurde. (Ein Fehler wird ausgegeben, wenn `nextval` für diese Sequenz in dieser Sitzung noch nie aufgerufen wurde.) Beachten Sie, dass das Ergebnis speziell für diese Sitzung gilt und auch noch gültig bleibt, wenn zur selben Zeit in anderen Sitzungen `nextval` ausgerufen wird.

`setval`

Stellt den Zähler für diese Sequenz zurück. Die Form mit zwei Argumenten setzt das Feld `last_value` (letzter Wert) der Sequenz auf den angegebenen Wert und setzt das Feld `is_called` auf logisch wahr, was heißt, dass der nächste Aufruf von `nextval` den Sequenzwert erst erhöht, bevor ein Wert zurückgegeben wird. In der Form mit drei Argumenten kann `is_called` auf wahr oder falsch gesetzt werden. Wenn es auf falsch gesetzt wird, dann ergibt der nächste Aufruf von `nextval` genau den angegebenen Wert und die Erhöhung der Sequenz beginnt erst mit den folgenden Aufruf von `nextval`. Zum Beispiel:

```
SELECT setval('foo', 42);           Nächster nextval ergibt 43
SELECT setval('foo', 42, true);     ebenso
SELECT setval('foo', 42, false);    Nächster nextval ergibt 42
```

Das Ergebnis von `setval` ist einfach der Wert des zweiten Arguments.

Wichtig

Um zu verhindern, dass Transaktionen, die gleichzeitig Zahlen von einer Sequenz erhalten wollen, sich gegenseitig blockieren, werden `nextval`-Operationen nicht zurückgerollt. Das heißt, wenn ein Wert einmal abgegeben wurde, dann wird er als verwendet betrachtet, selbst wenn die Transaktion, die `nextval` aufgerufen hatte, später abgebrochen wird. Das bedeutet, dass abgebrochene Transaktionen unbenutzte “Löcher” in der Reihe der erzeugten Werte verursachen können. `setval`-Operationen werden auch nicht zurückgerollt.

Wenn ein Sequenzobjekt mit den Vorgabewerten erstellt wurde, ergeben Aufrufe von `nextval` aufeinander folgende Werte bei 1 beginnend. Andere Verhalten können mit besonderen Parametern in dem Befehl `CREATE SEQUENCE` eingestellt werden; siehe die Referenzseite des Befehls für weitere Informationen.

9.12 Konditionale Ausdrücke

Dieser Abschnitt beschreibt die in PostgreSQL verfügbaren konditionalen Ausdrücke, welche auch dem SQL-Standard entsprechen.

Tipp

Wenn Ihre Anforderungen die von diesen konditionalen Ausdrücken gebotenen Fähigkeiten überschreiten, sollte Sie in Erwägung ziehen, eine serverseitige Prozedur in einer ausdrucksfähigeren Sprache zu schreiben.

9.12.1 CASE

Der CASE-Ausdruck ist ein allgemeiner konditionaler Ausdruck, ähnlich `if`/`else`-Anweisungen in anderen Sprachen:

```
CASE WHEN bedingung THEN ergebnis
     [WHEN ...]
     [ELSE ergebnis]
END
```

CASE-Klauseln können überall verwendet werden, wo ein Ausdruck gültig ist. *bedingung* ist ein Ausdruck, der ein Ergebnis vom Typ boolean liefert. Wenn das Ergebnis logisch wahr ist, dann ist das Ergebnis des CASE-Ausdrucks das nach der Bedingung stehende *ergebnis*. Wenn das Ergebnis aber falsch ist, werden etwaige folgende WHERE-Klauseln auf dieselbe Art durchsucht. Wenn keine der Bedingungen in WHERE wahr ist, dann ist das Ergebnis des CASE-Ausdrucks das *ergebnis* in der ELSE-Klausel. Wenn die ELSE-Klausel weggelassen wurde und keine Bedingung stimmt, ist das Ergebnis der NULL-Wert.

Ein Beispiel:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
```

```
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Es muss möglich sein, die Datentypen aller *ergebnis*-Ausdrücke in einen einzigen Ergebnistyp umzuwandeln. Siehe Abschnitt 10.5 bezüglich Einzelheiten.

Der folgende "einfache" CASE-Ausdruck ist eine spezialisierte Variante der oben beschriebenen Form:

```
CASE ausdruck
    WHEN wert THEN ergebnis
    [WHEN ...]
    [ELSE ergebnis]
END
```

ausdruck wird berechnet und mit allen *wert*-Angaben in den WHERE-Klauseln verglichen, bis eine gefunden wird, die gleich ist. Wenn keine Übereinstimmung gefunden wird, dann wird das *ergebnis* in der ELSE-Klausel (bzw. der NULL-Wert) zurückgegeben. Diese Konstruktion ist also mit der `switch`-Anweisung in C vergleichbar.

Das obige Beispiel kann mit der einfachen CASE-Syntax geschrieben werden:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

9.12.2 COALESCE

```
COALESCE(wert [, ...])
```

Die Funktion `COALESCE` gibt das erste ihrer Argumente zurück, welches nicht NULL ist. Das kann oft nützlich sein, um bei der Ausgabe für NULL-Werte einen Vorgabewert einzusetzen, zum Beispiel:

```
SELECT COALESCE(beschreibung, kurzbeschreibung, '(keine)') ...
```

9.12.3 NULLIF

```
NULLIF(wert1, wert2)
```

Die Funktion `NULLIF` ergibt den NULL-Wert genau dann, wenn *wert1* und *wert2* gleich sind. Ansonsten wird *wert1* zurückgegeben. Diese Funktion kann verwendet werden, um die Umkehrung des `COALESCE`-Beispiels von oben durchzuführen:

```
SELECT NULLIF(wert, '(keine)') ...
```

Tipp

`COALESCE` und `NULLIF` sind eigentlich nur Kurzformen von CASE-Ausdrücken. Sie werden beim Parsen früh in CASE-Ausdrücke umgewandelt und in der folgenden Verarbeitung wird dann gedacht, man hat es mit wirklichen CASE-Ausdrücken zu tun. Eine fehlerhafte Verwendung von `COALESCE` oder `NULLIF` kann daher zu Fehlermeldungen führen, die sich auf CASE beziehen.

9.13 Diverse Funktionen

Tabelle 9.27 zeigt einige Funktionen, die Sitzungs- und Systeminformationen ermitteln.

Name	Ergebnistyp	Beschreibung
`current_database()`	`name`	Name der aktuellen Datenbank
`current_schema()`	`name`	Name des aktuellen Schemas
`current_schemas(boolean)`	`name[]`	Namen der Schemas in Suchpfad, wahlweise einschließlich der impliziten Schemas
`current_user`	`name`	Benutzername der aktuellen Ausführungsumgebung
`session_user`	`name`	Benutzername der Sitzung
`user`	`name`	äquivalent mit `current_user`
`version()`	`text`	PostgreSQL-Versionsinformationen

Tabelle 9.27: Sitzungsinformationsfunktionen

Der Benutzer, der von der Funktion `session_user` ermittelt wird, ist der Benutzer, der die Datenbankverbindung eingeleitet hat; er ändert sich während der gesamten Verbindung nicht. Der Benutzer, der von der Funktion `current_user` zurückgegeben wird, ist der Benutzer, der für die Prüfung von Zugriffsrechten verwendet wird. Normalerweise sind beide Benutzer gleich, aber `current_user` ändert sich während der Ausführung von Funktionen mit dem Attribut `SECURITY DEFINER`. Im Unix-Jargon wäre `session_user` der "reale Benutzer" und `current_user` der "effektive Benutzer".

Anmerkung

`current_user`, `session_user` und `user` haben einen besonderen syntaktischen Status in SQL: Sie müssen ohne Klammern aufgerufen werden.

`current_schema` gibt das Schema zurück, das am Anfang des Suchpfads steht (oder den NULL-Wert, wenn der Suchpfad leer ist). Das ist das Schema, in dem Tabellen oder andere Objekte angelegt werden, wenn bei deren Erzeugung kein Schema angegeben wird. `current_schemas(boolean)` ergibt ein

Array mit den Namen aller Schemas, die gegenwärtig im Suchpfad sind. Die boolean-Option bestimmt, ob implizite Systemschemas, wie zum Beispiel `pg_catalog`, im Ergebnis mit auftauchen.

Anmerkung

Der Suchpfad kann zur Laufzeit geändert werden. Der Befehl lautet:

```
SET search_path TO schema [,schema , ...]
```

Die Funktion `version()` ergibt eine Zeichenkette, die die Version des PostgreSQL-Servers beschreibt.

Tabelle 9.28 zeigt die Funktionen zur Abfrage und Veränderung von Konfigurationsparametern.

Name	Ergebnistyp	Beschreibung
`current_setting(parametername)`	`text`	aktueller Wert des Parameters
`set_config(parametername, neuer_wert, lokal)`	`text`	setzt Parameter und gibt neuen Wert zurück

Tabelle 9.28: Funktionen zur Kontrolle von Konfigurationsparametern

Die Funktion `current_setting` ergibt den aktuellen Wert des Konfigurationsparameters `parametername` als Teil eines Anfrageergebnisses. Sie entspricht dem SQL-Befehl `SHOW`. Ein Beispiel:

```
SELECT current_setting('datestyle');

            current_setting
-----------------------------------
 ISO with US (NonEuropean) conventions
(1 row)
```

`set_config` setzt den Konfigurationsparameter `parametername` auf `neuer_wert`. Wenn `lokal` logisch wahr ist, dann gilt der neue Wert nur für die aktuelle Transaktion. Wenn der neue Wert für die aktuelle Sitzung gelten soll, dann geben Sie logisch falsch an. Die Funktion entspricht dem SQL-Befehl `SET`. Ein Beispiel:

```
SELECT set_config('show_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

Tabelle 9.29 listet Funktionen, die es dem Benutzer ermöglichen, die Zugriffsprivilegien für Objekte programmatisch zu untersuchen. Weitere Informationen über Privilegien finden Sie in Abschnitt 5.6.

Name	Ergebnistyp	Beschreibung
`has_table_privilege(benutzer, tabelle, privileg)`	`boolean`	Benutzer hat Privileg für Tabelle
`has_table_privilege(tabelle, privileg)`	`boolean`	aktueller Benutzer hat Privileg für Tabelle

Tabelle 9.29: Funktionen zur Abfrage von Zugriffsprivilegien

Name	Ergebnistyp	Beschreibung
has_database_privilege(*benutzer*, *datenbank*, *privileg*)	boolean	Benutzer hat Privileg für Datenbank
has_database_privilege(*datenbank*, *privileg*)	boolean	aktueller Benutzer hat Privileg für Datenbank
has_function_privilege(*benutzer*, *funktion*, *privileg*)	boolean	Benutzer hat Privileg für Funktion
has_function_privilege(*funktion*, *privileg*)	boolean	aktueller Benutzer hat Privileg für Funktion
has_language_privilege(*benutzer*, *sprache*, *privileg*)	boolean	Benutzer hat Privileg für Sprache
has_language_privilege(*sprache*, *privileg*)	boolean	aktueller Benutzer hat Privileg für Sprache
has_schema_privilege(*benutzer*, *schema*, *privileg*)	boolean	Benutzer hat Privileg für Schema
has_schema_privilege(*schema*, *privileg*)	boolean	aktueller Benutzer hat Privileg für Schema

Tabelle 9.29: Funktionen zur Abfrage von Zugriffsprivilegien (Forts.)

has_table_privilege prüft, ob ein Benutzer auf eine Tabelle auf eine bestimmte Art zugreifen darf. Der Benutzer kann als Name oder als ID (pg_user.usesysid) angegeben werden; wenn das Argument ausgelassen wird, dann wird current_user angenommen. Die Tabelle kann als Name oder als OID angegeben werden. (Es gibt also eigentlich sechs Varianten von has_table_privilege, welche sich durch die Zahl und die Typen der Argumente unterscheiden.) Wenn ein Name angegeben wird, kann er, wenn nötig, durch ein Schema ergänzt werden. Das gewünschte Zugriffsprivileg wird durch eine Zeichenkette angegeben, welche gleich einem der folgenden Werte sein muss: SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES oder TRIGGER. (Groß- oder Kleinschreibung ist unerheblich.) Ein Beispiel ist:

```
SELECT has_table_privilege('meinschema.meinetabelle', 'select');
```

has_database_privilege prüft, ob ein Benutzer auf eine Datenbank auf eine bestimmte Art zugreifen darf. Die möglichen Argumente sind analog zu has_table_privilege. Das gewünschte Zugriffsprivileg muss gleich CREATE, TEMPORARY oder TEMP (was äquivalent mit TEMPORARY ist) sein.

has_function_privilege prüft, ob ein Benutzer auf eine Funktion auf eine bestimmte Art zugreifen darf. Die möglichen Argumente sind analog zu has_table_privilege. Wenn die Funktion als Zeichenkette statt als OID angegeben wird, sind die erlaubten Eingabewerte die gleichen wie beim Datentyp regprocedure. Das gewünschte Zugriffsprivileg muss gegenwärtig immer gleich EXECUTE sein.

has_language_privilege prüft, ob ein Benutzer auf eine prozedurale Sprache auf eine bestimmte Art zugreifen darf. Die möglichen Argumente sind analog zu has_table_privilege. Das gewünschte Zugriffsprivileg muss gegenwärtig immer gleich USAGE sein.

has_schema_privilege prüft, ob ein Benutzer auf ein Schema auf eine bestimmte Art zugreifen darf. Die möglichen Argumente sind analog zu has_table_privilege. Das gewünschte Zugriffsprivileg muss gleich CREATE oder USAGE sein.

Tabelle 9.30 zeigt Funktionen, die feststellen, ob ein bestimmtes Objekt im aktuellen Schemasuchpfad **sichtbar** ist. Eine Tabelle wird als sichtbar bezeichnet, wenn das Schema, das sie enthält, im Suchpfad steht und keine Tabelle mit dem gleichen Namen vorher im Suchpfads steht. Gleichbedeutend heißt, dass auf die Tabelle verwiesen werden kann, ohne einen Schemanamen anzugeben. Folgende Anfrage kann zum Beispiel verwendet werden, um die Namen aller sichtbaren Tabellen anzuzeigen:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Name	Ergebnistyp	Beschreibung
pg_table_is_visible(*tabellen_oid*)	boolean	ist Tabelle im Suchpfad sichtbar
pg_type_is_visible(*typ_oid*)	boolean	ist Typ im Suchpfad sichtbar
pg_function_is_visible(*funktions_oid*)	boolean	ist Funktion im Suchpfad sichtbar
pg_operator_is_visible(*operator_oid*)	boolean	ist Operator im Suchpfad sichtbar
pg_opclass_is_visible(*opklassen_oid*)	boolean	ist Operatorklasse im Suchpfad sichtbar

Tabelle 9.30: Funktionen zur Überprüfung der Schemasichtbarkeit

pg_table_is_visible führt die Überprüfung für Tabellen (oder Sichten oder andere Objekte mit Eintrag in pg_class) aus. pg_type_is_visible, pg_function_is_visible, pg_operator_is_visible und pg_opclass_is_visible führen die gleiche Art von Sichtbarkeitsprüfung für Datentypen, Funktionen, Operatoren bzw. Operatorklassen aus. Bei Funktionen und Operatoren gilt, ein Objekt im Suchpfad ist sichtbar, wenn es kein anderes Objekt mit dem gleichen Namen *und den gleichen Argumentdatentypen* vorher im Pfad gibt. Bei Operatorklassen wird sowohl der Name als auch die zugehörige Indexmethode betrachtet.

Alle diese Funktionen verlangen die OID, um das zu überprüfende Objekt zu identifizieren. Wenn Sie ein Objekt dem Namen nach überprüfen wollen, können Sie die OID-Aliastypen (regclass, regtype, regprocedure oder regoperator) verwenden, zum Beispiel:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Beachten Sie, dass es keinen Sinn macht, einen unqualifizierten Namen auf diese Art zu überprüfen, denn wenn der Name überhaupt erkannt würde, muss er auch sichtbar sein.

Tabelle 9.31 listet Funktionen, die Informationen aus den Systemkatalogen herausholen. pg_get_viewdef, pg_get_ruledef, pg_get_indexdef bzw. pg_get_constraintdef rekonstruieren den Befehl, der zur Erzeugung einer Sicht, einer Regel, eines Index bzw. eines Constraints erforderlich ist. (Beachten Sie, dass dies eine Wiederzusammensetzung ist und nicht der Originaltext des Befehls.) Gegenwärtig funktioniert pg_get_constraintdef nur mit Fremdschlüssel-Constraints. pg_get_userbyid ermittelt den Benutzernamen, der zur angegebenen Benutzernummer gehört.

Name	Ergebnistyp	Beschreibung
pg_get_viewdef(*sicht_name*)	text	ermittle CREATE VIEW-Befehl für Sicht (*veraltet*)
pg_get_viewdef(*sicht_oid*)	text	ermittle CREATE VIEW-Befehl für Sicht
pg_get_ruledef(*regel_oid*)	text	ermittle CREATE RULE-Befehl für Sicht
pg_get_indexdef(*index_oid*)	text	ermittle CREATE INDEX-Befehl für Index
pg_get_constraintdef(*constraint_oid*)	text	ermittle Definition des Constraints
pg_get_userbyid(*benutzer_id*)	name	ermittle Benutzername für Benutzernummer

Tabelle 9.31: Systemkataloginformationsfunktionen

Die in Tabelle 9.32 gezeigten Funktionen ermitteln Kommentare, welche vorher mit dem Befehl COMMENT erzeugt wurden. Ein NULL-Wert wird zurückgegeben, wenn kein passender Kommentar für die angegebenen Parameter gefunden werden konnte.

Name	Ergebnistyp	Beschreibung
obj_description(*objekt_oid*, *katalog_name*)	text	ermittle Kommentar für Datenbankobjekt
obj_description(*objekt_oid*)	text	ermittle Kommentar für Datenbankobjekt (*veraltet*)
col_description(*tabellen_oid*, *spaltennummer*)	text	ermittle Kommentar für Tabellenspalte

Tabelle 9.32: Funktionen zur Ermittlung von Kommentaren

Die Form von obj_description mit zwei Parametern gibt den Kommentar des Datenbankobjekts zurück, der die angegebene OID hat und der im angegebenen Systemkatalog gespeichert ist. Zum Beispiel würde obj_description(123456,'pg_class') den Kommentar für die Tabelle mit der OID 123456 ermitteln. Die Form von obj_description mit einem Parameter braucht nur die OID des Objekts. Diese Form ist jetzt nicht mehr empfohlen, da es keine Garantie dafür gibt, dass OIDs über alle Systemkataloge eindeutig sind; daher könnte der falsche Kommentar zurückgegeben werden.

col_description gibt den Kommentar für eine Tabellenspalte zurück, welche durch die OID ihrer Tabelle und der Spaltennummer angegeben wird. obj_description kann für Tabellenspalten nicht verwendet werden, da Spalten keine eigenen OIDs haben.

9.14 Aggregatfunktionen

Aggregatfunktionen berechnen ein einzelnes Ergebnis aus einer Sammlung von Eingabewerten. Tabelle 9.33 zeigt die eingebauten Aggregatfunktionen. Die besonderen Syntaxanforderungen für Aggregatfunktionen werden in Abschnitt 4.2.5 erklärt. In Teil I finden Sie weiteres einführendes Material dazu.

Funktion	Argumenttyp	Ergebnistyp	Beschreibung
avg(*ausdruck*)	smallint, integer, bigint, real, double precision, numeric oder interval	numeric für ganzzahlige Argumente, double precision für Fließkommaargumente, ansonsten gleich dem Argumentdatentyp	Durchschnitt (arithmetisches Mittel) aller Eingabewerte
count(*)		bigint	Anzahl der Eingabewerte
count(*ausdruck*)	egal	bigint	Anzahl der Eingabewerte, für die *ausdruck* nicht gleich dem NULL-Wert ist
max(*ausdruck*)	numerisch, Zeichenkette oder Datum/Zeit	gleich dem Argumenttyp	Maximalwert von *ausdruck* aus allen Eingabewerten
min(*ausdruck*)	numerisch, Zeichenkette oder Datum/Zeit	gleich dem Argumenttyp	Minimalwert von *ausdruck* aus allen Eingabewerten
stddev(*ausdruck*)	smallint, integer, bigint, real, double precision oder numeric	double precision für Fließkommaargumente, ansonsten numeric	Standardabweichung der Eingabewerte

Tabelle 9.33: Aggregatfunktionen

Funktion	Argumenttyp	Ergebnistyp	Beschreibung
sum(*ausdruck*)	smallint, integer, bigint, real, double precision, numeric oder interval	bigint für smallint- oder integer-Argumente, numeric für bigint-Argumente, double precision für Fließkommaargumente, ansonsten gleich dem Argumenttyp	Summe von *ausdruck* für alle Eingabewerte
variance(*ausdruck*)	smallint, integer, bigint, real, double precision oder numeric	double precision für Fließkommaargumente, ansonsten numeric	Varianz der Eingabewerte (Quadrat der Standardabweichung)

Tabelle 9.33: Aggregatfunktionen (Forts.)

Es ist anzumerken, dass außer count all diese Funktionen den NULL-Wert zurückgeben, wenn keine Zeilen ausgewählt sind. Insbesondere ergibt sum ohne Zeilen den NULL-Wert und nicht numerisch null, was man vielleicht erwartet hätte. Die Funktion coalesce kann verwendet werden, um NULL-Werte, wenn gewünscht, durch andere Werte zu ersetzen.

9.15 Ausdrücke mit Unteranfragen

Dieser Abschnitt beschreibt die in PostgreSQL vorhandenen, SQL-konformen Ausdrücke mit Unteranfragen. Alle hier beschriebenen Ausdrücke haben einen Rückgabewert vom Typ boolean (wahr/falsch).

9.15.1 EXISTS

```
EXISTS ( unteranfrage )
```

Das Argument von EXISTS ist ein beliebiger SELECT-Befehl oder eine **Unteranfrage**. Die Unteranfrage wird ausgewertet, um festzustellen, ob sie irgendwelche Zeilen ergibt. Wenn Sie mindestens eine Zeile liefert, ist das Ergebnis von EXISTS “wahr”; wenn die Unteranfrage keine Zeilen liefert, ist das Ergebnis von EXISTS “falsch”.

Die Unteranfrage kann auf Variablen aus der übergeordneten Anfrage verweisen, welche sich dann bei jeder einzelnen Auswertung der Unteranfrage als Konstanten verhalten.

Die Unteranfrage wird im Allgemeinen nur so weit ausgeführt, bis festgestellt werden kann, ob mindestens eine Zeile zurückgegeben wird, nicht bis zum Ende. Es ist nicht zu empfehlen, eine Unteranfrage zu schreiben, die Nebeneffekte hat (z.B. das Aufrufen von Sequenzfunktionen); ob die Nebeneffekte auftreten, könnte schwer vorherzusagen sein.

Da das Ergebnis nur davon abhängt, ob irgendwelche Zeilen zurückgegeben werden, und nicht, was der Inhalt der Zeilen ist, ist die Ergebnismenge der Unteranfrage normalerweise uninteressant. Es ist daher gebräuchlich, alle EXISTS-Prüfungen in der Form EXISTS(SELECT 1 WHERE ...) zu schreiben. Es gibt von dieser Regel jedoch Ausnahmen, wie zum Beispiel bei Unteranfragen, die INTERSECT verwenden.

Dieses einfache Beispiel ist wie ein innerer Verbund über col2, aber es erzeugt höchstens eine Ergebniszeile für jede Zeile in tab1, selbst wenn es mehrere passende Zeilen in tab2 gibt:

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.15.2 IN (skalare Form)

```
ausdruck IN (wert[, ...])
```

Die rechte Seite dieser Form von IN ist eine Liste skalarer Ausdrücke in Klammern. Das Ergebnis ist "wahr", wenn das Ergebnis des Ausdrucks auf der linken Seite gleich irgendeinem Ausdruck auf der rechten Seite ist. Dies ist eine Abkürzung für

```
ausdruck = wert1
OR
ausdruck = wert2
OR
...
```

Beachten Sie, dass, wenn der linke Ausdruck den NULL-Wert ergibt oder wenn es keinen gleichen Ausdruck auf der rechten Seite gibt und mindestens ein rechter Ausdruck den NULL-Wert ergibt, dass dann das Ergebnis von IN nicht "falsch", sondern der NULL-Wert ist. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Anmerkung

Diese Form von IN ist eigentlich kein Ausdruck mit Unteranfrage, aber es scheint besser, ihn in der Nähe des IN-Ausdrucks für Unteranfragen zu dokumentieren.

9.15.3 IN (Unteranfrageform)

```
ausdruck IN (unteranfrage)
```

Die rechte Seite dieser Form von IN ist eine Unteranfrage in Klammern, die genau eine Spalte ergeben muss. Der Ausdruck auf der linken Seite wird ausgewertet und mit jeder Zeile des Anfrageergebnisses verglichen. Das Ergebnis von IN ist "wahr", wenn irgendeine übereinstimmende Zeile gefunden wird. Das Ergebnis ist "falsch", wenn keine übereinstimmende Zeile gefunden wird (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert).

Beachten Sie, dass, wenn der linke Ausdruck den NULL-Wert ergibt oder wenn es keinen gleichen Ausdruck auf der rechten Seite gibt und mindestens eine Zeile rechts den NULL-Wert ergibt, dass dann das Ergebnis von IN nicht "falsch", sondern der NULL-Wert ist. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Wie bei EXISTS sollte man nicht davon ausgehen, dass die Unteranfrage komplett ausgewertet werden wird.

```
(ausdruck [, ausdruck ...]) IN (unteranfrage)
```

Die rechte Seite dieser Form von IN ist eine Unteranfrage in Klammern, welche genau so viele Zeilen ergeben muss, wie es Ausdrücke in der Liste auf der linken Seite gibt. Die Ausdrücke links werden ausgewertet und zeilenweise mit jeder Zeile des Anfrageergebnisses verglichen. Das Ergebnis von IN ist "wahr", wenn irgendeine übereinstimmende Zeile gefunden wird. Das Ergebnis ist "falsch", wenn keine übereinstimmende Zeile gefunden wird (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert).

Wie gewohnt, werden NULL-Werte in den Ausdrücken und Anfragezeilen gemäß den normalen SQL-Regeln für logische Ausdrücke kombiniert. Zwei Zeilen gelten als gleich, wenn alle ihre einander entsprechenden Elemente nicht NULL und gleich sind; die Zeilen sind ungleich, wenn irgendwelche einander entsprechenden Elemente nicht NULL und ungleich sind; ansonsten ist das Ergebnis des Zeilenvergleichs

unbekannt (NULL). Wenn die Ergebnisse bei allen Zeilen ungleich oder unbekannt sind, mit mindestens einem unbekannt, dann ist das Ergebnis von IN der NULL-Wert.

9.15.4 NOT IN (skalare Form)

```
ausdruck NOT IN (wert[, ...])
```

Die rechte Seite dieser Form von NOT IN ist eine Liste von skalaren Ausdrücken in Klammern. Das Ergebnis ist "wahr", wenn das Ergebnis des Ausdrucks auf der linken Seite ungleich aller Ausdrücke auf der rechten Seite ist. Dies ist eine Abkürzung für

```
ausdruck <> wert1
AND
ausdruck <> wert2
AND
...
```

Beachten Sie, dass, wenn der linke Ausdruck den NULL-Wert ergibt oder wenn es keinen gleichen Ausdruck auf der rechten Seite gibt und mindestens ein rechter Ausdruck den NULL-Wert ergibt, dass dann das Ergebnis der NOT IN-Konstruktion der NULL-Wert ist, und nicht "wahr", wie man naiverweise meinen könnte. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Tipp

`x NOT IN y` ist immer äquivalent mit `NOT (x IN y)`. NULL-Werte werden den Anfänger aber bei NOT IN viel wahrscheinlicher verwirren als bei IN. Drücken Sie Bedingungen daher am besten, wenn möglich, positiv aus.

9.15.5 NOT IN (Unteranfrageform)

```
ausdruck NOT IN (unteranfrage)
```

Die rechte Seite dieser Form von NOT IN ist eine Unteranfrage in Klammern, die genau eine Spalte ergeben muss. Der Ausdruck auf der linken Seite wird ausgewertet und mit jeder Zeile des Anfrageergebnisses verglichen. Das Ergebnis von NOT IN ist "wahr", wenn nur nicht übereinstimmende Zeilen gefunden werden (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert). Das Ergebnis ist "falsch", wenn irgendeine übereinstimmende Zeile gefunden wird.

Beachten Sie, dass, wenn der linke Ausdruck den NULL-Wert ergibt oder wenn es keinen gleichen Ausdruck auf der rechten Seite gibt und mindestens eine Zeile rechts den NULL-Wert ergibt, dass dann das Ergebnis von NOT IN nicht "wahr", sondern der NULL-Wert ist. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Wie bei EXISTS, sollte man nicht davon ausgehen, dass die Unteranfrage komplett ausgewertet werden wird.

```
(ausdruck [, ausdruck ...]) NOT IN (unteranfrage)
```

Die rechte Seite dieser Form von NOT IN ist eine Unteranfrage in Klammern, welche genau so viele Zeilen ergeben muss, wie es Ausdrücke in der Liste auf der linken Seite gibt. Die Ausdrücke links werden ausgewertet und zeilenweise mit jeder Zeile des Anfrageergebnisses verglichen. Das Ergebnis von NOT IN ist "wahr", wenn nur nicht übereinstimmende Zeilen gefunden werden (einschließlich des Sonderfalls, dass

die Unteranfrage keine Zeilen liefert). Das Ergebnis ist “falsch”, wenn irgendeine übereinstimmende Zeile gefunden wird.

Wie gewohnt, werden NULL-Werte in den Ausdrücken und Anfragezeilen gemäß den normalen SQL-Regeln für logische Ausdrücke kombiniert. Zwei Zeilen gelten als gleich, wenn alle ihre einander entsprechenden Elemente nicht NULL und gleich sind; die Zeilen sind ungleich, wenn irgendwelche einander entsprechenden Elemente nicht NULL und ungleich sind; ansonsten ist das Ergebnis des Zeilenvergleichs unbekannt (NULL). Wenn die Ergebnisse bei allen Zeilen ungleich oder unbekannt sind, mit mindestens einem unbekannt, dann ist das Ergebnis von NOT IN der NULL-Wert.

9.15.6 ANY/SOME

```
ausdruck operator ANY (unteranfrage)
ausdruck operator SOME (unteranfrage)
```

Die rechte Seite dieser Form von ANY ist eine Unteranfrage in Klammern, die genau eine Spalte ergeben muss. Der Ausdruck auf der linken Seite wird ausgewertet und mit jeder Zeile des Anfrageergebnisses unter Verwendung von *operator*, welcher ein Ergebnis des Typs boolean haben muss, verglichen. Das Ergebnis von ANY ist “wahr”, wenn irgendein wahres Ergebnis zurückgegeben wird. Das Ergebnis ist “falsch”, wenn kein wahres Ergebnis gefunden wird (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert).

SOME ist ein Synonym für ANY. IN ist äquivalent mit `= ANY`.

Beachten Sie, dass, wenn keine wahren Ergebnisse gefunden werden und mindestens eine Zeile auf der rechten Seite für den Operator den NULL-Wert ergibt, dass dann das Ergebnis der ANY-Konstruktion nicht falsch, sondern der NULL-Wert ist. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Wie bei EXISTS, sollte man nicht davon ausgehen, dass die Unteranfrage komplett ausgewertet werden wird.

```
(ausdruck [, ausdruck ...]) operator ANY (unteranfrage)
(ausdruck [, ausdruck ...]) operator SOME (unteranfrage)
```

Die rechte Seite dieser Form von ANY ist eine Unteranfrage in Klammern, welche genau so viele Zeilen ergeben muss wie es Ausdrücke in der Liste auf der linken Seite gibt. Die Ausdrücke links werden ausgewertet, und zeilenweise mit jeder Zeile des Anfrageergebnisses unter Verwendung von *operator* verglichen. Gegenwärtig sind nur die Operatoren = und <> in zeilenweisen ANY-Konstruktionen erlaubt. Das Ergebnis von ANY ist “wahr”, wenn irgendeine gleiche bzw. ungleiche Zeile gefunden wird. Das Ergebnis ist “falsch”, wenn keine solche Zeile gefunden wird (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert).

Wie gewohnt, werden NULL-Werte in den Ausdrücken und Anfragezeilen gemäß den normalen SQL-Regeln für logische Ausdrücke kombiniert. Zwei Zeilen gelten als gleich, wenn alle ihre einander entsprechenden Elemente nicht NULL und gleich sind; die Zeilen sind ungleich, wenn irgendwelche einander entsprechenden Elemente nicht NULL und ungleich sind; ansonsten ist das Ergebnis des Zeilenvergleichs unbekannt (NULL). Wenn das Ergebnis bei mindestens einer Zeile unbekannt ist, dann kann das Ergebnis von ANY nicht falsch sein; es wird wahr oder der NULL-Wert sein.

9.15.7 ALL

```
ausdruck operator ALL (unteranfrage)
```

Die rechte Seite dieser Form von ALL ist eine Unteranfrage in Klammern, die genau eine Spalte ergeben muss. Der Ausdruck auf der linken Seite wird ausgewertet und mit jeder Zeile des Anfrageergebnisses unter Verwendung von *operator*, welcher ein Ergebnis des Typs boolean haben muss, verglichen. Das Ergebnis von ALL ist "wahr", wenn für alle Zeilen ein wahres Ergebnis zurückgegeben wird (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert). Das Ergebnis ist "falsch", wenn irgendein falsches Ergebnis gefunden wird.

NOT IN ist äquivalent mit `<> ALL`.

Beachten Sie, dass, wenn es keine falschen Ergebnisse gibt, aber mindestens eine Zeile auf der rechten Seite für den Operator den NULL-Wert ergibt, dass dann das Ergebnis der ALL-Konstruktion nicht wahr, sondern der NULL-Wert ist. Das entspricht den normalen Regeln zur Kombinierung logischer Werte in SQL.

Wie bei EXISTS sollte man nicht davon ausgehen, dass die Unteranfrage komplett ausgewertet werden wird.

```
(ausdruck [, ausdruck ...]) operator ALL (unteranfrage)
```

Die rechte Seite dieser Form von ALL ist eine Unteranfrage in Klammern, welche genau so viele Zeilen ergeben muss, wie es Ausdrücke in der Liste auf der linken Seite gibt. Die Ausdrücke links werden ausgewertet und zeilenweise mit jeder Zeile des Anfrageergebnisses unter Verwendung von *operator* verglichen. Gegenwärtig sind nur die Operatoren = und <> in zeilenweisen ALL-Konstruktionen erlaubt. Das Ergebnis von ALL ist "wahr", wenn alle Zeilen gleich bzw. ungleich sind (einschließlich des Sonderfalls, dass die Unteranfrage keine Zeilen liefert). Das Ergebnis ist "falsch", wenn mindestens eine Zeile, die ungleich bzw. gleich ist, gefunden wird.

Wie gewohnt, werden NULL-Werte in den Ausdrücken und Anfragezeilen gemäß den normalen SQL-Regeln für logische Ausdrücke kombiniert. Zwei Zeilen gelten als gleich, wenn alle ihre einander entsprechenden Elemente nicht NULL und gleich sind; die Zeilen sind ungleich, wenn irgendwelche einander entsprechenden Elemente nicht NULL und ungleich sind; ansonsten ist das Ergebnis des Zeilenvergleichs unbekannt (NULL). Wenn das Ergebnis bei mindestens einer Zeile unbekannt ist, dann kann das Ergebnis von ALL nicht wahr sein; es wird dann falsch oder der NULL-Wert sein.

9.15.8 Zeilenweiser Vergleich

```
(ausdruck [, ausdruck ...]) operator (unteranfrage)
(ausdruck [, ausdruck ...]) operator (ausdruck [, ausdruck ...])
```

Die linke Seite ist eine Liste von skalaren Ausdrücken. Die rechte Seite kann entweder eine gleich lange Liste von skalaren Ausdrücken sein oder eine Unteranfrage in Klammern, welche genau so viele Zeilen ergeben muss, wie es Ausdrücke in der Liste auf der linken Seite gibt. Außerdem darf die Anfrage nicht mehr als eine Zeile ergeben. (Wenn sie keine Zeilen ergibt, dann wird das Ergebnis als NULL-Wert interpretiert.) Die Ausdrücke links werden ausgewertet und zeilenweise mit der einzigen Ergebniszeile der Unteranfrage bzw. den Ausdrücken auf der rechten Seite verglichen. Gegenwärtig sind nur die Operatoren = und <> in zeilenweisen Vergleichen erlaubt. Das Ergebnis ist "wahr", wenn die beiden Zeilen gleich bzw. ungleich sind.

Wie gewohnt, werden NULL-Werte in den Ausdrücken und Anfragezeilen gemäß den normalen SQL-Regeln für logische Ausdrücke kombiniert. Zwei Zeilen gelten als gleich, wenn alle ihre einander entsprechenden Elemente nicht NULL und gleich sind; die Zeilen sind ungleich, wenn irgendwelche einander entsprechenden Elemente nicht NULL und ungleich sind; ansonsten ist das Ergebnis des Zeilenvergleichs unbekannt (NULL).

10 Typumwandlung

In SQL-Befehlen kann es, absichtlich oder nicht, vorkommen, dass unterschiedliche Datentypen in einem Ausdruck auftreten. PostgreSQL hat diverse Mechanismen, um Ausdrücke mit gemischten Typen auszuwerten.

In vielen Fällen müssen Endanwender die Einzelheiten der Typumwandlungsmechanismen nicht kennen. Die impliziten Umwandlungen, die von PostgreSQL vorgenommen werden, können allerdings die Ergebnisse einer Anfrage beeinflussen. Wenn es notwendig ist, kann der Benutzer oder Programmierer diese Ergebnisse mit *expliziten* Umwandlungen kontrollieren.

Dieses Kapitel gibt eine Einführung in die Typumwandlungsmechanismen und -konventionen in PostgreSQL. Mehr Informationen über bestimmte Datentypen und die verfügbaren Funktionen und Operatoren finden Sie in Kapitel 8 bzw. Kapitel 9.

10.1 Überblick

SQL ist eine Sprache mit einem starken Typensystem. Das heißt, jedes Stück Daten hat einen zugehörigen Datentyp, der das Verhalten und die erlaubte Verwendung bestimmt. PostgreSQL hat ein erweiterbares Typensystem, dass allgemeiner und flexibler als in anderen SQL-Implementierungen ist. Daher sollten die meisten Typumwandlungen in PostgreSQL von Regeln bestimmt werden, die allgemeingültig sind und nicht aus einer bestimmten Situation heraus aufgestellt wurden, damit Ausdrücke mit gemischten Datentypen auch mit benutzerdefinierten Typen funktionieren können.

Der Parser in PostgreSQL kategorisiert lexikalische Elemente in eine von nur fünf allgemeinen Gruppen: ganze Zahlen, Fließkommazahlen, Zeichenketten, Namen und Schlüsselwörter. Die meisten Erweiterungstypen werden am Anfang als Zeichenkette kategorisiert. Die Definition der SQL-Sprache erlaubt, dass man bei der Zeichenkette den Typ angeben kann, und damit kann man in PostgreSQL erreichen, dass der Parser gleich den korrekten Typ erfährt. Zum Beispiel hat die Anfrage

```
SELECT text 'Ursprung' AS "bezeichnung", point '(0,0)' AS "wert";

 bezeichnung | wert
-------------+-------
 Ursprung    | (0,0)
(1 row)
```

zwei Konstanten, mit den Typen text und point. Wenn kein Typ für eine Zeichenkettenkonstante angegeben wurde, dann wird erstmal der Platzhaltertyp unknown zugewiesen, der in späteren Phasen, wie unten beschrieben, aufgelöst wird.

Es gibt grundsätzlich vier SQL-Konstruktionen, wo im PostgreSQL-Parser unterschiedliche Regeln zur Umwandlung von Typen vonnöten sind:

Operatoren

> PostgreSQL erlaubt Ausdrücke mit Präfix- und Postfix-Operatoren (unär/ein Operand), sowie binären (zwei Operanden) Operatoren.

Funktionsaufrufe

> Ein Großteil des PostgreSQL-Typensystems ist um eine große Zahl Funktionen aufgebaut. Funktionsaufrufe können ein oder mehrere Argumente haben. Da PostgreSQLPostgreSQL das Überladen von Funktionen erlaubt, kann die aufzurufende Funktion nicht allein durch den Funktionsnamen identifiziert werden; das System wählt die richtige Funktion auf Grundlage der Datentypen der angegebenen Funktionsargumente.

Wertspeicherung

> Die SQL-Befehle `INSERT` und `UPDATE` speichern die Ergebnisse von Ausdrücken in einer Tabelle. Die Ausdrücke in dem Befehl müssen an die Datentypen der Zielspalten angepasst oder womöglich in diese umgewandelt werden.

`UNION`- und `CASE`-Konstruktionen

> Da alle Anfrageergebnisse eines `SELECT`-Befehls mit Mengenoperationen in den selben Ergebnisspalten erscheinen müssen, müssen die Ergebnistypen eines jeden `SELECT` angepasst und in einen jeweils einheitlichen Endergebnistyp umgewandelt werden. Auf ähnliche Art müssen die Teilausdrücke eines `CASE`-Ausdrucks in einen einheitlichen Typ umgewandelt werden, damit der gesamte `CASE`-Ausdruck einen vorhersehbaren Ergebnistyp haben kann.

Die Systemkatalogtabellen speichern Informationen darüber, welche Umwandlungen (englisch *cast*) zwischen Datentypen gültig sind und wie diese Umwandlungen durchzuführen sind. Zusätzliche Umwandlungen können vom Benutzer mit dem Befehl `CREATE CAST` bestimmt werden. (Das wird normalerweise in Verbindung mit der Erzeugung eines neuen Datentyps gemacht. Die Umwandlungsmöglichkeiten zwischen den eingebauten Typen wurden sehr sorgfältig ausgewählt und sollten nicht verändert werden.)

Außerdem gibt es einige eingebaute Regeln im Parser, um das erwartete Verhalten von Typen aus dem SQL-Standard zu ermöglichen. Dafür wurden mehrere allgemeine **Typenkategorien** festgelegt: boolean, numeric, string, bitstring, datetime, timespan, geometric, network und benutzerdefiniert. Jede Kategorie, mit Ausnahme der benutzerdefinierten Typen, hat einen **bevorzugten Typ**, welcher bevorzugt ausgewählt wird, wenn es Unklarheiten gibt. In der Kategorie der benutzerdefinierten Typen ist jeder Typ sein eigener bevorzugter Typ. Zweideutige Ausdrücke (mit mehreren möglichen Parse-Ergebnissen) können daher bei mehreren möglichen eingebauten Typen oft aufgelöst werden, aber bei mehreren möglichen benutzerdefinierten Typen gibt es eine Fehlermeldung.

Alle Regeln zur Typumwandlung wurden aufgrund mehrerer Prinzipien entworfen:

- Implizite Umwandlungen sollten niemals überraschende oder unvorhersehbare Ergebnisse haben.
- Benutzerdefinierte Typen, von denen der Parser von vornherein keine Kenntnis hat, sollten “höher” in der Typenhierarchie stehen. In Ausdrücken mit gemischten Typen sollten eingebaute Typen immer in benutzerdefinierte Typen umgewandelt werden (natürlich nur wenn eine Umwandlung nötig ist).
- Benutzerdefinierte Typen stehen in keinem Verhältnis zueinander. Gegenwärtig verfügt PostgreSQL über keine Informationen über Verhältnisse zwischen Typen, außer den Regeln für eingebaute Typen und den Beziehungen, die sich aus vorhandenen Funktionen ergeben.
- Parser oder Executor sollten keine zusätzliche Arbeit haben, wenn eine Anfrage keine implizite Typumwandlung benötigt. Das heißt, wenn die Anfrage gut formuliert wurde und die Typen alle passen, dann sollte es möglich sein, die Anfrage zu verarbeiten, ohne zusätzliche Zeit im Parser aufzubringen oder unnötige implizite Umwandlungsfunktionen anzuwenden.

Zusätzlich sollte gelten, wenn eine Anfrage normalerweise für eine Funktion eine implizite Umwandlung nötig hat und der Benutzer dann eine neue Funktion mit den passenden Argumenttypen erzeugt, dann sollte der Parser diese neue Funktion verwenden und nicht mehr die alte Funktion mit der nötigen impliziten Umwandlung.

10.2 Operatoren

Die Operandentypen eines Operatoraufrufs werden nach der unten beschriebenen Prozedur aufgelöst. Beachten Sie, dass diese Prozedur indirekt vom Vorrang der betroffenen Operatoren beeinflusst wird. Siehe auch Abschnitt 4.1.6.

Typauflösung von Operanden

1. Wähle die zu betrachtenden Operatoren aus der Systemtabelle `pg_operator` aus. Wenn ein unqualifizierter Operatorname verwendet wurde (der normale Fall), dann werden die Operatoren ausgewählt, die den richtigen Namen und die richtige Anzahl Operanden haben und im Suchpfad sichtbar sind (siehe Abschnitt 5.7.3). Wenn ein qualifizierter Operatorname verwendet wurde, dann werden nur Operatoren im angegebenen Schema in Betracht gezogen.

   a. Wenn der Suchpfad mehrere Operatoren mitidentischen Operandentypen findet, dann wird nur der behalten, der als erster im Pfad gefunden wurde. Aber Operatoren mit verschiedenen Operandentypen werden unabhängig von ihrer Suchpfadposition in Betracht gezogen.

2. Prüfe, ob es einen Operator mit den genauen Typen der Operanden gibt. Wenn es einen gibt (in der Menge der Operatoren, die in Betracht gezogen werden, kann es nur einen passenden geben), dann verwende ihn.

   a. Wenn ein Operandentyp eines binären Operatoraufrufs vom Typ unknown ist, dann nimm für diese Prüfung an, dass er den gleichen Typ wie der andere Operand hat. Andere Fälle mit dem Typ unknown finden in diesem Schritt nie eine Lösung.

3. Suche nach der am besten passenden Lösung.

   a. Verwerfe Kandidaten, wo die Operandentypen nicht übereinstimmen und nicht entsprechend umgewandelt werden können (mit einer impliziten Umwandlung). Bei Werten vom Typ unknown wird hier angenommen, dass sie in alles umgewandelt werden können. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   b. Prüfe alle Kandidaten und behalte die mit den meisten exakten Übereinstimmungen bei den Operandentypen. Wenn es keine exakten Übereinstimmungen gibt, behalte alle Kandidaten. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   c. Prüfe alle Kandidaten und behalte die mit den meisten exakten oder binärkompatiblen Übereinstimmungen bei den Operandentypen. Wenn es keine Kandidaten mit exakten oder binärkompatiblen Übereinstimmungen gibt, behalte alle. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   d. Prüfe alle Kandidaten und behalte die, die bevorzugte Typen auf den meisten Positionen, auf den Typumwandlungen erforderlich sein werden, akzeptieren. Wenn kein Kandidat bevorzugte Typen akzeptiert, behalte alle. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   e. Wenn irgendeiner der angegebenen Operanden vom Typ unknown ist, prüfe die Typenkategorien, die von den übrig gebliebenen Kandidaten auf diesen Positionen akzeptiert werden. Wähle auf jeder Position die Kategorie `string`, wenn ein Kandidat diese Kategorie akzeptiert. (Die Bevorzugung dieser Kategorie ist sinnvoll, weil eine Konstante vom Typ unknown wie ein String, d.h. eine Zeichenkette, aussieht.) Ansonsten, wenn alle übrigen Kandidaten die gleiche Kategorie akzeptieren, dann wähle diese Kategorie. Wenn nicht, dann erzeuge einen Fehler, weil die richtige Wahl

nicht ohne weitere Hinweise getroffen werden kann. Verwerfe jetzt die Kandidaten, die nicht die gewählte Typenkategorie akzeptieren. Wenn außerdem irgendein Kandidat einen bevorzugten Typ auf einer gegebenen Operandenposition akzeptiert, verwerfe Kandidaten, die nicht den bevorzugten Typ für diesen Operanden akzeptieren.

f. Wenn nur ein Kandidat übrig bleibt, verwende ihn. Wenn kein oder mehr als ein Kandidat übrig bleibt, dann erzeuge einen Fehler.

Einige Beispiele folgen.

Beispiel 10.1: Typauflösung beim Potenzierungsoperator

Es gibt nur einen Potenzierungsoperator im Systemkatalog, und er hat Operanden vom Typ double precision. Der Parser weist anfänglich beiden Operanden in dieser Anfrage den Typ integer zu:

```
SELECT 2 ^ 3 AS "exp";

 exp
-----
   8
(1 row)
```

Also führt das System eine Typumwandlung an beiden Operanden durch und die Anfrage wird gleichbedeutend mit

```
SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "exp";
```

Beispiel 10.2: Typauflösung beim Zeichenkettenverknüpfungsoperator

Eine zeichenkettenähnliche Syntax wird sowohl für Zeichenkettentypen als auch für komplexe Erweiterungstypen verwendet. Zeichenketten ohne angegebenen Typ müssen mit möglichen Operatorkandidaten in Übereinstimmung gebracht werden.

Ein Beispiel mit einem Operanden ohne ausdrücklichen Typ:

```
SELECT text 'abc' || 'def' AS "text und unknown";

 text und unknown
------------------
 abcdef
(1 row)
```

In diesem Fall prüft der Parser, ob es einen Operator gibt, der den Typ text für beide Operanden akzeptiert. Da das der Fall ist, nimmt er an, dass der zweite Operand als Typ text interpretiert werden soll.

Hier ist eine Verknüpfung ohne Typangaben:

```
SELECT 'abc' || 'def' AS "unbekannt";

 unbekannt
-----------
 abcdef
(1 row)
```

In diesem Fall gibt es keine Anfangshinweise darüber, welcher Typ zu verwenden ist, da in der Anfrage keine Typen angegeben sind. Also sucht der Parser nach allen Kandidaten und stellt fest, dass es Kandidaten gibt, die Operanden der Kategorie `string` und der Kategorie `bitstring` akzeptieren. Da die Kategorie `string` bevorzugt wird, wird sie ausgewählt, und der bevorzugte Typ dieser Kategorie, text, wird den Konstanten mit unbekanntem Typ zugewiesen.

Beispiel 10.3: Typauflösung bei den Betrags- und Fakultätsoperatoren

Der PostgreSQL-Operatorkatalog hat mehrere Einträge für den Präfixoperator `@`, die alle den Betrag von diversen numerischen Datentypen berechnen. Einer dieser Einträge ist für den Typ float8, welcher der bevorzugte Typ in der Kategorie `numeric` ist. Daher wird PostgreSQL diesen Eintrag verwenden, wenn ein nicht-numerischer Wert vorliegt:

```
SELECT @ '-4.5' AS "abs";

 abs
-----
 4.5
(1 row)
```

Hier hat das System eine implizite Umwandlung von text in float8 vorgenommen, bevor der Operator angewendet wird. Wir können auch überprüfen, dass wirklich float8 und nicht ein anderer Typ verwendet wurde:

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  Input '-4.5e500' is out of range for float8
```

Andererseits gibt es den Postfixoperator `!` (Fakultät) nur für Ganzzahltypen, nicht für float8. Wenn wir also einen ähnlichen Fall mit `!` versuchen, erhalten wir:

```
SELECT '20' ! AS "fakultät";

ERROR:  Unable to identify a postfix operator '!' for type '"unknown"'
        You may need to add parentheses or an explicit cast
```

Das passiert, weil das System nicht entscheiden kann, welchen der möglichen Operatoren es bevorzugen soll. Wir können mit einer ausdrücklichen Typumwandlung aushelfen:

```
SELECT CAST('20' AS int8) ! AS "fakultät";

      fakultät
--------------------
 2432902008176640000
(1 row)
```

10.3 Funktionen

Die Argumenttypen eines Funktionsaufrufs werden nach der unten beschriebenen Prozedur aufgelöst.

Typauflösung von Funktionsargumenten

1. Wähle die zu betrachtenden Funktionen aus der Systemtabelle `pg_proc` aus. Wenn ein unqualifizierter Funktionsname verwendet wurde, werden die Funktionen ausgewählt, die den richtigen Namen und die richtige Anzahl Argumente haben und im Suchpfad sichtbar sind (siehe Abschnitt 5.7.3). Wenn ein qualifizierter Funktionsname verwendet wurde, werden nur Funktionen im angegebenen Schema in Betracht gezogen.

   a. Wenn der Suchpfad mehrere Funktionen mit identischen Argumenttypen findet, wird nur die behalten, die als erster im Pfad gefunden wurde. Aber Funktionen mit verschiedenen Argumenttypen werden unabhängig von ihrer Suchpfadposition in Betracht gezogen.

2. Prüfe, ob es eine Funktion mit den genauen Typen der Argumente gibt. Wenn es eine gibt (in der Menge der Funktionen, die in Betracht gezogen werden, kann es nur eine passende geben), dann verwende sie. (Fälle mit dem Typ unknown finden in diesem Schritt nie eine Lösung.)
3. Wenn keine exakte Übereinstimmung gefunden wurde, dann prüfe, ob der Funktionsaufruf eine triviale Typumwandlung darstellt. Das ist der Fall, wenn der Funktionsaufruf ein Argument hat und der Funktionsname der gleiche ist wie der (interne) Name eines Datentyps. Außerdem muss das Funktionsargument entweder eine Konstante vom Typ unknown oder ein Typ, der mit dem ersten binärkompatibel ist, sein. Wenn diese Voraussetzungen erfüllt sind, dann wird das Funktionsargument ohne tatsächlichen Funktionsaufruf in den Zieltyp umgewandelt.
4. Suche nach der am besten passenden Lösung.

   a. Verwerfe Kandidaten, wo die Argumenttypen nicht übereinstimmen und nicht entsprechend umgewandelt werden können (mit einer impliziten Umwandlung). Bei Werten vom Typ unknown wird hier angenommen, dass sie in alles umgewandelt werden können. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   b. Prüfe alle Kandidaten und behalte die mit den meisten exakten Übereinstimmungen bei den Argumenttypen. Wenn es keine exakten Übereinstimmungen gibt, behalte alle Kandidaten. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   c. Prüfe alle Kandidaten und behalte die mit den meisten exakten oder binärkompatiblen Übereinstimmungen bei den Argumenttypen. Wenn es keine Kandidaten mit exakten oder binärkompatiblen Übereinstimmungen gibt, behalte alle. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   d. Prüfe alle Kandidaten und behalte die, die bevorzugte Typen auf den meisten Positionen, auf denen Typumwandlungen erforderlich sein werden, akzeptieren. Wenn kein Kandidat bevorzugte Typen akzeptiert, behalte alle. Wenn nur ein Kandidat übrig bleibt, verwende ihn, ansonsten gehe zum nächsten Schritt.

   e. Wenn irgendeiner der angegebenen Argumente vom Typ unknown ist, prüfe die Typenkategorien, die von den übrig gebliebenen Kandidaten auf diesen Positionen akzeptiert werden. Wähle auf jeder Position die Kategorie `string`, wenn ein Kandidat diese Kategorie akzeptiert. (Die Bevorzugung dieser Kategorie ist sinnvoll, weil eine Konstante vom Typ unknown wie ein String, d.h. eine Zeichenkette, aussieht.) Ansonsten, wenn alle übrigen Kandidaten die gleiche Kategorie akzeptieren, wähle diese Kategorie. Wenn nicht, erzeuge einen Fehler, weil die richtige Wahl nicht ohne weitere Hinweise getroffen werden kann. Verwerfe jetzt die Kandidaten, die nicht die gewählte Typenkategorie akzeptieren. Wenn außerdem irgendein Kandidat einen bevorzugten Typ auf einer gegebenen Argumentposition akzeptiert, verwerfe Kandidaten, die nicht den bevorzugten Typ für dieses Argument akzeptieren.

   f. Wenn nur ein Kandidat übrig bleibt, verwende ihn. Wenn kein oder mehr als ein Kandidat übrig bleibt, erzeuge einen Fehler.

Einige Beispiele folgen.

Beispiel 10.4: Typauflösung bei der Rundungsfunktion

Es gibt nur eine Funktion `round` mit zwei Argumenten. (Das erste ist numeric, das zweite integer.) In der folgenden Anfrage wird also das erste Argument vom Typ integer automatisch in den Typ numeric umgewandelt:

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

Die Anfrage wird vom Parser quasi in folgende Anfrage umgewandelt:

```
SELECT round(CAST (4 AS numeric), 4);
```

Da numerische Konstanten mit Nachkommastellen den Typ numeric zugewiesen bekommen, benötigt die folgende Anfrage keine Typumwandlung und könnte daher etwas effizienter sein:

```
SELECT round(4.0, 4);
```

Beispiel 10.5: Typauflösung bei der Funktion substr

Es gibt mehrere Funktionen namens `substr`, wovon eine Argumente der Typen text und integer hat. Wenn `substr` mit einer Zeichenkettenkonstante unbekannten Typs aufgerufen wird, wählt das System die Kandidatenfunktion, die ein Argument der bevorzugten Kategorie `string` (nämlich Typ text) akzeptiert.

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

Wenn die Zeichenkette als Typ varchar ausgewiesen ist, was der Fall sein könnte, wenn sie aus einer Tabelle stammt, wird der Parser versuchen, sie in den Typ text umzuwandeln:

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Das wird vom Parser quasi in folgende Anfrage umgewandelt:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

Anmerkung

Dem Parser ist bekannt, dass text und varchar binärkompatibel sind, das heißt, dass der eine Typ einer Funktion übergeben werden kann, die eigentlich den anderen Typ verlangt, ohne dass eine tatsächliche Umwandlung vorgenommen werden muss. Daher wird in diesem Fall keine wirkliche Typumwandlung eingefügt.

Wenn die Funktion mit einem Argument vom Typ integer aufgerufen wird, dann wird der Parser versuchen, dies in den Typen text umzuwandeln:

```
SELECT substr(1234, 3);

 substr
--------
     34
(1 row)
```

Das wird in Wirklichkeit so ausgeführt:

```
SELECT substr(CAST (1234 AS text), 3);
```

Diese Transformation kann automatisch geschehen, weil es einen impliziten Umwandlungsweg (Cast) von integer nach text gibt.

10.4 Wertspeicherung

Wenn ein Wert in eine Tabelle eingefügt wird, dann wird er in den Typ der Zielspalte nach folgender Prozedur umgewandelt.

Typumwandlung bei der Wertspeicherung

1. Prüfe, ob eine exakte Übereinstimmung zwischen Wert und Zielspalte vorliegt.
2. Ansonsten versuchen den Ausdruck in den Zieltyp umzuwandeln. Das ist erfolgreich, wenn es einen registrierten Umwandlungsweg (Cast) zwischen den Typen gibt. Wenn der Ausdruck den Typ unknown hat, dann wird der Inhalt der Zeichenkette der Eingaberoutine des Zieldatentyps übergeben.
3. Wenn der Zieltyp eine feste Länge hat (zum Beispiel char oder varchar mit Längenangabe), versuche, für den Zieltyp eine Funktion zur Größeneinstellung zu finden. Ein solche Funktion hat den gleichen Namen wie der Datentyp, hat zwei Argumente, wovon der erste der Zieldatentyp und der zweite vom Typ integer ist, und hat den gleichen Ergebnistyp. Wenn eine gefunden wurde, dann wird sie angewendet, wobei die deklarierte Länge der Spalte als zweiter Parameter übergeben wird.

Beispiel 10.6: Typumwandlung bei der Speicherung beim Typ character

Bei einer Zielspalte, die als character(20) erzeugt wurde, wird bei den folgenden Befehlen automatisch darauf geachtet, dass der Wert an die richtige Größe angepasst wird:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, length(v) FROM vv;

          v           | length
```

```
----------------------+--------
 abcdef               |     20
(1 row)
```

Was hier wirklich passierte, war, dass die beiden Zeichenkettenkonstanten als Typ text aufgelöst wurden, wodurch der Operator `||` als Verknüpfung von text-Werten aufgelöst werden konnte. Danach wird das Ergebnis des Operators in den Typ bpchar umgewandelt (der interne Name des Typs character, für *blank-padded char*), um mit der Zielspalte übereinzustimmen. (Da die Typen text und bpchar binärkompatibel sind, wird zu dieser Umwandlung kein Funktionsaufruf benötigt.) Schließlich wird die Funktion zur Größenanpassung `bpchar(bpchar, integer)` im Systemkatalog gefunden und mit dem Ergebnis des Operators und der gespeicherten Spaltenlänge aufgerufen. Diese typenspezifische Funktion führt dann entsprechende Längenprüfungen durch und füllt den Wert mit Leerzeichen auf.

10.5 UNION- und CASE-Konstruktionen

In `UNION`-Konstruktionen müssen möglicherweise verschiedene Typen aneinander angepasst werden, um eine einheitliche Ergebnismenge zu erzeugen. Der Auflösungsalgorithmus wird für jede Spalte einer solchen Anfrage getrennt angewendet. In `INTERSECT`- und `EXCEPT`-Klauseln werden unterschiedliche Typen genauso wie bei `UNION` aufgelöst. Ein `CASE`-Ausdruck verwendet auch denselben Algorithmus, um aus den Teilausdrücken einen Ergebnistyp zu ermitteln.

Typauflösung bei UNION- und CASE-Konstruktionen

1. Wenn alle Eingabewerte den Typ unknown haben, wähle den Typ text (der bevorzugte Typ der Kategorie `string`). Ansonsten ignoriere die Werte vom Typ unknown bei der Ermittlung des Ergebnistyps.
2. Wenn die Eingabewerte, die nicht vom Typ unknown sind, nicht alle in derselben Typenkategorie sind, dann erzeuge einen Fehler.
3. Wähle den ersten der Eingabetypen, außer unknown, der ein bevorzugter Typ in seiner Kategorie ist oder in den alle Eingabetypen, außer unknown, implizit umgewandelt werden können.
4. Wandle alle Eingabewerte in den gewählten Typ um.

Einige Beispiele folgen.

Beispiel 10.7: Typauflösung in einer UNION-Konstruktion mit fehlenden Typen

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Hier wird die Konstante `'b'` mit unbekanntem Typ als Typ text aufgelöst.

Beispiel 10.8: Typauflösung in einer einfachen UNION-Konstruktion

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
```

```
---------
       1
     1.2
(2 rows)
```

Die Konstante `1.2` hat den Typ numeric und die Konstante `1` hat den Typ integer. integer kann implizit in numeric umgewandelt werden, also wird numeric verwendet.

Beispiel 10.9: Typauflösung in einer umgekehrten UNION-Konstruktion

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Hier kann der Typ real nicht implizit in integer umgewandelt werden, aber integer kann implizit in real gewandelt werden. Daher hat das Ergebnis den Typ real.

11 Indexe

Die Verwendung von Indexen ist eine häufig benutzte Möglichkeit, die Leistung der Datenbank zu verbessern. Ein Index gestattet dem Datenbankserver, bestimmte Zeilen viel schneller zu finden und auszugeben, als es ohne Index möglich wäre. Aber Indexe erhöhen auch den Verwaltungsaufwand im Datenbanksystem und sollten daher nicht ohne Überlegung verwendet werden.

11.1 Einführung

Das klassische Beispiel, um die Notwendigkeit eines Index darzustellen, ist mit einer Tabelle wie dieser:

```
CREATE TABLE test1 (
    id integer,
    inhalt varchar
);
```

und einer Anwendung, die viele Anfragen der Form

```
SELECT inhalt FROM test1 WHERE id = konstante;
```

erfordert. Normalerweise müsste das System die gesamte Tabelle `test1` Zeile für Zeile absuchen, um alle passenden Einträge zu finden. Wenn die Anfrage nur wenige Zeilen als Ergebnis liefert (möglicherweise nur eine oder gar keine), ist das ganz klar eine ineffiziente Methode. Wenn dem System mitgeteilt worden wäre, dass es einen Index für die Spalte `id` unterhalten soll, könnte es eine viel effizientere Methode verwenden, um die passenden Zeilen zu finden. Zum Beispiel muss es vielleicht nur ein paar Ebenen in einem Suchbaum überprüfen.

In den meisten Sachbüchern gibt es einen ähnlichen Ansatz: Begriffe, welche die Leser oft nachschlagen wollen, werden in einem alphabetischen Stichwortverzeichnis (auf Englisch: *index*) am Ende des Buchs zusammengestellt. Ein Leser kann das Stichwortverzeichnis relativ schnell überfliegen und dann schnell die richtige Seite aufschlagen; er muss nicht das gesamte Buch durchlesen, um eine relevante Stelle zu finden. Genauso wie es die Aufgabe des Buchautors ist, die Begriffe, die ein Leser am wahrscheinlichsten nachschlagen will, vorherzusehen, ist es die Aufgabe des Datenbankprogrammierers, vorherzubestimmen, welche Indexe von Vorteil sein würden.

Der folgende Befehl wird verwendet, um den besprochenen Index für die Spalte `id` zu erzeugen:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Der Name `test1_id_index` kann frei bestimmt werden, aber Sie sollten einen Namen wählen, an dem Sie später immer noch erkennen können, wofür der Index war.

Um einen Index zu entfernen, verwenden Sie den Befehl `DROP INDEX`. Indexe können jederzeit zu einer Tabelle hinzugefügt oder entfernt werden.

Wenn der Index einmal erzeugt worden ist, ist kein weiteres Eingreifen erforderlich: Das System wird den Index verwenden, wenn es denkt, dass er effizienter als eine sequenzielle Suche durch die Tabelle ist. Aber Sie müssen möglicherweise den Befehl `ANALYZE` regelmäßig ausführen, um die Statistiken zu aktualisieren, die es dem Planer erlauben realitätsnahe Schätzungen abzugeben. Lesen Sie auch in Kapitel 13 ,wie man herausfinden kann, ob ein Index verwendet wird und wann und warum der Planer auch manchmal zu dem Entschluss kommt, von einem Index *nicht* Gebrauch zu machen.

Indexe können auch in `UPDATE`- und `DELETE`-Befehlen mit Suchbedingungen von Nutzen sein. Außerdem können Indexe auch in Verbundanfragen verwendet werden. Ein Index für eine Spalte, die Teil einer Verbundbedingung ist, kann also Anfragen mit Verbundoperationen erheblich beschleunigen.

Wenn ein Index erzeugt wurde, muss er vom System mit der Tabelle in Übereinstimmung gehalten werden. Dadurch wird der Aufwand bei Datenmanipulationsoperationen erhöht. Indexe, die nicht wichtig sind oder überhaupt nicht verwendet werden, sollten daher entfernt werden. Beachten Sie, dass eine Anfrage oder ein Datenmanipulationsbefehl nur höchstens einen Index pro Tabelle verwenden kann.

11.2 Indextypen

In PostgreSQLPostgreSQL gibt es mehrere Indextypen: B-Tree, R-Tree, GiST und Hash. Jeder Indextyp ist wegen seines Algorithmus für bestimmte Arten von Anfragen am besten geeignet. Wenn nichts anderes angegeben wird, erstellt der Befehl `CREATE INDEX` einen B-Tree-Index, welcher für die am häufigsten vorkommenden Situationen geeignet ist. Im Einzelnen heißt das, dass der Anfrageplaner in PostgreSQL-PostgreSQL einen B-Tree-Index in Betracht ziehen wird, wenn eine indizierte Spalte in einem Vergleich unter Verwendung folgender Operatoren vorkommt: < , <= , = , >= , >

R-Tree-Indexe sind besonders für räumliche Daten geeignet. Um einen R-Tree-Index zu erzeugen, verwenden Sie einen Befehl der Form

```
CREATE INDEX name ON tabelle USING RTREE (spalte);
```

Der Anfrageplaner in PostgreSQL wird einen R-Tree-Index in Betracht ziehen, wenn eine indizierte Spalte in einem Ausdruck mit folgenden Operatoren vorkommt: << , &< , &> , >> , @ , ~= , && (Die Bedeutung dieser Operatoren finden Sie in Abschnitt 9.9.)

Der Anfrageplaner wird einen Hash-Index in Betracht ziehen, wenn eine indizierte Spalte in einem Vergleich unter Verwendung des Operators = vorkommt. Folgender Befehl erzeugt einen Hash-Index:

```
CREATE INDEX name ON tabelle USING HASH (spalte);
```

Anmerkung

In Tests hat es sich gezeigt, dass Hash-Indexe in PostgreSQL ähnlich schnell oder langsamer als B-Tree-Indexe sind, und die Indexgröße und die Zeit zum Aufbauen des Index viel schlechter sind. Hash-Indexe zeigen auch einen schlechte Leistung bei vielen gleichzeitigen Zugriffen. Aus diesen Gründen wird die Verwendung eines Hash-Index nicht empfohlen.

Die B-Tree-Indexmethode ist eine Implementierung der hochparallelen B-Trees von Lehman und Yao. Die R-Tree-Indexmethode ist eine Implementierung des normalen R-Trees mit dem quadratischen Split-Algorithmus von Guttman. Die Hash-Indexmethode ist eine Implementierung des linearen Hashs von Litwin. Wir erwähnen die Algorithmen hier nur, um zu zeigen, dass alle Indexmethoden voll dynamisch sind und nicht regelmäßig optimiert werden müssen (wie es zum Beispiel bei statischen Hash-Methoden der Fall ist).

11.3 Mehrspaltige Indexe

Ein Index kann auch für mehrere Spalten definiert werden. Wenn Sie zum Beispiel eine Tabelle dieser Form haben:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(Nehmen wir an, Sie speichern Ihr `/dev` Verzeichnis in einer Datenbank ...) und Sie oft Anfragen der Form

```
SELECT name FROM test2 WHERE major = konstante AND minor = konstante;
```

ausführen, kann es angebracht sein, wenn Sie einen Index für die Spalten `major` und `minor` zusammen definieren. Zum Beispiel:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Gegenwärtig werden mehrspaltige Indexe nur von den B-Tree- und GiST-Implementierungen unterstützt. Bis zu 32 Spalten können angegeben werden. (Diese Grenze kann verändert werden, bevor man PostgreSQL compiliert; siehe die Datei `pg_config.h`.)

Der Anfrageplaner kann einen mehrspaltigen Index verwenden, wenn die Anfrage die in der Indexdefinition ganz links aufgezählte Spalte und beliebig viele weitere Spalten, die rechts von ihr ohne Lücke gelistet wurden, verwendet (mit passenden Operatoren). Zum Beispiel: Ein Index für `(a, b, c)` kann verwendet werden, wenn eine Anfrage `a`, `b` und `c` verwendet oder wenn sie `a` und `b` verwendet oder wenn sie nur `a` verwendet, aber nicht bei anderen Kombinationen. (In einer Anfrage mit `a` und `c` könnte der Anfrageplaner entscheiden, den Index nur für `a` zu verwenden und `c` wie eine normale nichtindizierte Spalte zu behandeln.)

Ein mehrspaltiger Index kann nur verwendet werden, wenn die Ausdrücke, die die indizierten Spalten enthalten, mit `AND` verbunden wurde. Zum Beispiel,in der Anfrage

```
SELECT name FROM test2 WHERE major = konstante OR minor = konstante;
```

kann der oben definierte Index `test2_mm_idx` nicht für die Suche in beiden Spalten verwendet werden. (Er kann aber verwendet werden, um nur in der Spalte `major` zu suchen.)

Mehrspaltige Indexe sollten sparsam verwendet werden. Meistens reicht ein Index für eine einzelne Spalte aus und spart Platz und Zeit. Indexe mit mehr als drei Spalten sind fast immer unangebracht.

11.4 Indexe für Unique Constraints

Indexe können auch verwendet werden, um sicherzustellen, dass ein Spaltenwert oder eine Gruppe von Werten in mehreren Spalten in einer Tabelle nur einmal vorkommt.

```
CREATE UNIQUE INDEX name ON tabelle (spalte [, ...]);
```

Gegenwärtig können nur B-Tree-Indexe als `UNIQUE` deklariert werden.

Wenn ein Index als `UNIQUE` deklariert ist, wird dafür gesorgt, dass in der Tabelle keine Zeilen mit gleichen indizierten Werten auftreten können. NULL-Werte zählen nicht als gleich. PostgreSQL erzeugt automatisch einen solchen `UNIQUE`-Index, wenn eine Tabelle erzeugt wird, die einen Unique Constraint oder einen Primärschlüssel hat, um diesen Constraint umzusetzen. Die Spalten, die den Unique Constraint bzw. den Primärschlüssel ausmachen, bilden die Spalten für den Index (ein mehrspaltiger Index, wenn nötig). Ein solcher Index kann zu einer Tabelle auch später hinzugefügt werden, um einen Unique Constraint zu erzeugen.

Anmerkung

Die bevorzugte Methode, zu einer Tabelle einen Unique Constraint hinzuzufügen, ist `ALTER TABLE ... ADD CONSTRAINT`. Die Tatsache, dass Indexe verwendet werden, um Unique Constraints umzusetzen, könnte man als internes Implementierungsdetail betrachten, auf dass man nicht direkt zugreifen sollte.

11.5 Funktionsindexe

Ein **Funktionsindex** ist ein Index, der über das Ergebnis einer Funktion, die auf eine oder mehrere Spalten einer Tabelle angewendet wird, definiert ist. Funktionsindexe können verwendet werden, um auf Grundlage der Ergebnisse von Funktionsaufrufen schnell auf Daten zugreifen zu können.

Zum Beispiel verwendet man oft, um Vergleiche ohne Beachten der Groß- und Kleinschreibung durchzuführen, die Funktion `lower`:

```
SELECT * FROM test1 WHERE lower(spalte1) = 'wert';
```

Diese Anfrage kann einen Index verwenden, wenn er für das Ergebnis des Ausdrucks `lower(spalte1)` erzeugt wurde:

```
CREATE INDEX test1_lower_spalte1_idx ON test1 (lower(spalte1));
```

Die Funktion in der Indexdefinition kann mehrere Argumente haben, aber diese müssen Spalten der Tabelle sein, nicht aber Konstanten. Funktionsindexe sind immer einspaltig (nämlich das Ergebnis der Funktion), selbst wenn die Funktion mehr als eine Spalte verwendet. Mehrspaltige Indexe mit Funktionsaufrufen sind nicht möglich.

Tipp

Die im letzten Absatz erwähnten Beschränkungen kann man leicht umgehen, indem man selbst eine Funktion erzeugt, die das gewünschte Ergebnis intern berechnet.

11.6 Operatorklassen

Eine Indexdefinition kann für jede Spalte eines Index eine **Operatorklasse** angeben.

```
CREATE INDEX name ON table (spalte opklasse [, ...]);
```

Die Operatorklasse bestimmt, welche Operatoren vom Index für diese Spalte verwendet werden sollen. Ein B-Tree-Index für eine Spalte vom Typ int4 würde die Operatorklasse `int4_ops` verwenden; diese Operatorklasse enthält die Vergleichsfunktionen für Werte vom Typ int4. Normalerweise reicht die vorgegebene Operatorklasse für den Datentyp der Spalte aus. Der Hauptgrund, warum es Operatorklassen gibt, ist, dass es für manche Datentypen mehrere sinnvolle Sortierreihenfolgen geben kann. Zum Beispiel könnte man komplexe Zahlen nach Betrag oder nach dem reellen Teil sortieren. Das könnte man erreichen, indem man zwei Operatorklassen erzeugt und die passende wählt, wenn man den Index definiert.

Es gibt auch eingebaute Operatorklassen neben den Standardklassen:

- Die Operatorklassen `box_ops` und `bigbox_ops` unterstützen beide R-Tree-Indexe für den Datentyp box. Der Unterschied zwischen beiden ist, dass `bigbox_ops` die Koordinaten der Rechtecke herunterskaliert, um Fließkommafehler bei der Multiplikation, Addition und Subtraktion von sehr großen Fließkommazahlen zu vermeiden. Wenn der Bereich, in dem Ihre Rechtecke liegen, größer als etwas 20000 Quadrateinheiten ist, sollten Sie `bigbox_ops` verwenden.

Die folgende Anfrage zeigt alle vorhandenen Operatorklassen:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcamid = am.oid
     ORDER BY index_method, opclass_name;
```

Sie kann auch erweitert werden um alle Operatoren in einer Klasse mit zu zeigen:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opr.oprname AS opclass_operator
    FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
    WHERE opc.opcamid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
     ORDER BY index_method, opclass_name, opclass_operator;
```

11.7 Partielle Indexe

Ein **partieller Index** wird über einen Teil einer Tabelle erstellt; der Teil wird durch einen Bedingungsausdruck (das so genannte **Prädikat** des partiellen Index) bestimmt. Der Index enthält nur Einträge für jene Tabellenzeilen, die das Prädikat erfüllen.

Ein Hauptanliegen der partiellen Indexe ist es, zu verhindern, dass häufig vorkommende Werte indiziert werden. Da eine Anfrage, die nach einem häufig vorkommenden Wert sucht (einer, der mehr als ein paar Prozent aller Tabellenzeilen ausmacht), den Index sowieso nicht verwenden wird, gibt es auch keinen

Grund, diese Zeilen überhaupt im Index zu speichern. Dadurch wir die Größe des Index verringert, wodurch Anfragen, die den Index tatsächlich benutzen, beschleunigt werden. Außerdem können dadurch viele Aktualisierungen der Tabelle beschleunigt werden, weil der Index nicht in jedem Fall mit aktualisiert werden muss. Beispiel 11.1 zeigt eine mögliche Anwendung dieser Idee.

Beispiel 11.1: Einrichtung eines partiellen Index, um häufige Werte auszuschließen

Nehmen wir an, Sie speichern die Zugriffsstatistiken eines Webservers in einer Datenbank. Die meisten Zugriffe sind aus dem IP-Adressbereich Ihrer Organisation, aber einige sind von woandersher (sagen wir Angestellte, die sich von außerhalb einwählen). Wenn Ihren Suchanfragen nach IP-Adresse hauptsächlich die externen Zugriffe betreffen, müssen Sie den IP-Bereich, der dem Subnetz Ihrer Organisation entspricht, wahrscheinlich gar nicht indizieren.

Nehmen wir folgende Tabelle an:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Um einen partiellen Index zu erstellen, der zu unserem Beispiel passt, verwenden Sie folgenden Befehl:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet
'192.168.100.255');
```

Eine typische Anfrage, die diesen Index verwenden kann, ist folgende:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Eine Anfrage, die den Index nicht verwenden kann, ist:

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Sie werden bemerken, dass diese Art eines partiellen Index erfordert, dass die häufigen Werte vorher ermittelt werden. Wenn die Verteilung der Werte sich aus der Anwendung ergibt oder sich nach einer Zeit nicht mehr verändert, dann ist das nicht schwer, aber wenn die häufigen Werte nur auf dem zufälligen Datenaufkommen beruhen, kann das einen hohen Wartungsaufwand mit sich ziehen.

Eine weitere Möglichkeit besteht darin, Werte aus dem Index auszuschließen, die für die meisten Anfragen nicht von Interesse sind; das wird in Beispiel 11.2 gezeigt. Daraus ergeben sich die gleichen Vorteile wie oben, aber man verhindert dadurch, dass man überhaupt mit einem Index auf die “uninteressanten” Werte zugreifen kann, selbst wenn eine Indexsuche in diesem Fall vorteilhaft wäre. Es sollte klar sein, dass ein partieller Index in diesem Szenario eine Menge Sorgfalt und Experimentierfreude voraussetzt.

Beispiel 11.2: Einrichtung eines partiellen Index, um uninteressante Werte auszuschließen

Nehmen wir an, Sie haben eine Tabelle mit Bestellungen, wovon für einige schon die Rechnung gestellt wurde und für andere nicht. Nehmen wir weiter an, die Bestellungen ohne Rechnung sind ein kleiner Teil der Tabelle, aber die Zeilen mit den meisten Zugriffen. Dann können Sie die Leistung verbessern, indem Sie einen Index nur für die Zeilen ohne Rechnung erstellen. Der Befehl um den Index zu erstellen, würde so aussehen:

```
CREATE INDEX bestell_ohne_rechnung_index ON bestellungen (bestell_nr)
    WHERE rechnung is not true;
```

Eine mögliche Anfrage, die diesen Index verwendet, wäre

```
SELECT * FROM bestellungen WHERE rechnung is not true AND bestell_nr < 10000;
```

Der Index kann allerdings auch in Anfragen verwendet werden, die `bestell_nr` gar nicht verwenden, zum Beispiel:

```
SELECT * FROM bestellungen WHERE rechnung is not true AND betrag > 5000.00;
```

Das ist nicht so effizient wie ein partieller Index für die Spalte `betrag` wäre, da das System den gesamten Index durchsuchen muss. Wenn es aber relativ wenige Bestellungen ohne Rechnung gibt, könnte die Verwendung dieses partiellen Index, um diese Bestellungen zu finden trotzdem von Vorteil sein.

Beachten Sie, dass diese Anfrage den Index nicht verwenden kann:

```
SELECT * FROM bestellungen WHERE bestell_nr = 3501;
```

Die Bestellung 3501 könnte unter den Bestellungen mit oder ohne Rechnung sein.

Beispiel 11.2 zeigt auch, dass die indizierte Spalte und die Spalte im Indexprädikat nicht die gleiche sein müssen. PostgreSQL unterstützt partielle Indexe mit beliebigen Prädikaten, soweit nur Spalten der zu indizierenden Tabelle verwendet werden. Beachten Sie aber, dass das Prädikat mit den Suchbedingungen der Anfragen, die aus dem Index Nutzen ziehen sollen, übereinstimmen muss. Genau heißt das, ein partieller Index kann nur verwendet werden, wenn das System erkennen kann, dass das Indexprädikat logisch aus der `WHERE`-Bedingung der Anfrage folgt. PostgreSQL enthält keine sonderlich schlaue Beweismaschine, die mathematisch äquivalente Ausdrücke, die auf unterschiedliche Arten geschrieben wurde, erkennen kann. (Eine solche Beweismaschine ist nicht nur sehr schwer herzustellen, sie wäre wahrscheinlich auch viel zu langsam für die Praxis.) Das System kann einfache Implikationen aus Ungleichungen erkennen, zum Beispiel, dass aus “x < 1” folgt, dass “x < 2”; in anderen Fällen muss das Indexprädikat genau mit der Bedingung in der `WHERE`-Klausel der Anfrage übereinstimmen, ansonsten wird nicht erkannt werden, dass der Index verwendet werden kann.

Eine dritte Möglichkeit zur Verwendung eines partiellen Index verlangt gar nicht danach, dass der Index überhaupt in Anfragen eingesetzt wird. Die Idee ist, einen `UNIQUE`-Index über einen Teil einer Tabelle zu erzeugen, wie in Beispiel 11.3. Dadurch wird die Eindeutigkeit der Zeilen, die das Indexprädikat erfüllen, erreicht, ohne Auswirkungen auf andere Zeilen zu haben.

Beispiel 11.3: Einrichtung eines partiellen UNIQUE-Index

Nehmen wir an, Sie haben eine Tabelle, die Testergebnisse speichert. Wir möchten sicherstellen, dass es nur einen “erfolgreichen” Test für jede Kombination von Thema und Testobjekt gibt, aber es darf beliebig viele “erfolglose” Tests geben. So könnte man das machen:

```
CREATE TABLE tests (
    thema text,
    testobjekt text,
    erfolg boolean,
    ...
);

CREATE UNIQUE INDEX tests_erfolg_constraint ON tests (thema, testobjekt)
    WHERE erfolg;
```

Das ist eine besonders effiziente Art, diese Anforderungen umzusetzen, besonders wenn es wenige erfolgreiche und viele nicht erfolgreiche Tests gibt.

Schließlich kann man einen partiellen Index auch einsetzen, um die Wahl der Anfragepläne im System zu übergehen. Es kann vorkommen, dass Datenmengen mit eigentümlicher Wertverteilung das System dazu bringen, einen Index einzusetzen, wenn es nicht von Vorteil ist. In einem solchen Fall kann der Index so eingerichtet werden, dass er für die betroffene Anfrage nicht verwendet werden kann. PostgreSQL trifft eigentlich vernünftige Einscheidungen darüber, wann ein Index verwendet wird und wann nicht (zum Beispiel wird ein Index nicht verwendet, wenn ein häufiger Wert ausgelesen wird, sodass das Beispiel weiter oben wirklich nur an Indexgröße spart und nicht notwendig ist, um die Verwendung des Index zu vermeiden), und grotesk falsche Planerentscheidungen sind auch ein guter Grund für einen Fehlerbericht an die Entwickler.

Merken Sie sich, dass die Einrichtung eines partiellen Index aussagt, dass Sie mindestens so viel wissen wie der Anfrageplaner, insbesondere dass Sie wissen, wann ein Index von Nutzen sein kann. Der Erwerb dieses Wissens erfordert Erfahrung und ein Verständnis dafür, wie Indexe in PostgreSQL funktionieren. In den meisten Fällen wird der Vorteil gegenüber einem normalen Index nicht viel sein.

Weitere Informationen über partielle Indexe finden Sie in den Abhandlungen *Stonebraker 1989*, *Olson 1993* und *Seshardri & Swami 1995* (siehe Literaturverzeichnis).

11.8 Indexverwendung überprüfen

Trotzdem Indexe in PostgreSQL nicht manuell gewartet oder eingestellt werden müssen, ist es wichtig zu prüfen, welche Indexe von den tatsächlich anfallenden Anfragen wirklich verwendet werden. Die Verwendung der Indexe kann mit dem Befehl `EXPLAIN` betrachtet werden; wie er zu diesem Zweck angewendet wird, ist in Abschnitt 13.1 beschrieben.

Es ist sehr schwer, allgemeingültige Regeln aufzustellen, die bestimmen, wann man einen Index einrichten sollte. Es gibt eine Reihe typischer Fälle, die in den Beispielen der vorangegangenen Abschnitte gezeigt wurden. In den meisten Fällen wird man nicht um das Ausprobieren umhinkommen. Der Rest dieses Abschnitts gibt dazu ein paar Hinweise.

- Führen Sie immer zuerst den Befehl `ANALYZE` aus. Dieser Befehl erstellt Statistiken über die Verteilung der Werte in der Tabelle. Diese Informationen sind notwendig, um vorherzubestimmen, wie viele Zeilen eine Anfrage als Ergebnis haben wird, was vom Planer benötigt wird, um jedem möglichen Anfrageplan einen realistische Kostenfaktor zuzuordnen. Ohne wirkliche Statistiken werden irgendwelche Vorgabewerte verwendet, die ganz bestimmt ungenau sind. Wenn man also die Indexverwendung einer Anwendung untersucht, ohne `ANALYZE` ausgeführt zu haben, ist das ein hoffnungsloses Unterfangen.
- Verwenden Sie echte Daten zum Ausprobieren. Wenn Sie Testdaten verwenden, werden Sie erfahren, welche Indexe Sie für die Testdaten brauchen, aber das war auch schon alles.

  Es ist besonders fatal, proportional verkleinerte Datenmengen zu verwenden. Obwohl beispielsweise die Auswahl von 1000 Zeilen aus 100000 für einen Index geeignet sein könnte, wird das bei 1 aus 100 Zeilen kaum der Fall sein, weil die 100 Zeilen wahrscheinlich in eine einzige Speicherseite auf der Festplatte passen und es keinen Plan gibt, der besser ist als sequenziell 1 Seite von der Festplatte zu laden.

  Seien Sie auch vorsichtig, wenn Sie Testdaten erfinden, was oft nicht zu vermeiden ist, wenn die Anwendung noch nicht eingesetzt wird. Werte, die sehr ähnlich oder komplett zufällig sind oder in vorsortierter Reihenfolge eingefügt werden, können die Statistiken gegenüber praxisnahen Daten verfälschen.
- Wenn Indexe nicht verwendet werden, kann es beim Testen von Nutzen sein, ihre Verwendung zu erzwingen. Es gibt Konfigurationsparameter, mit denen die verschiedenen Plantypen ausgeschaltet werden können (beschrieben in Abschnitt 16.4). Wenn man zum Beispiel sequentielle Suchen (`enable_seqscan`) und Nested-Loop-Verbunde (`enable_nestloop`), welche die einfachsten Plantypen sind, abschaltet, muss das System versuchen, einen anderen Plan zu verwenden. Wenn das System trotzdem eine sequentielle Suche oder einen Nested-Loop-Verbund verwendet, gibt es wahrscheinlich ein grundlegenderes Problem, warum der Index nicht verwendet wird, zum Beispiel weil die Bedin-

gung in der Anfrage nicht auf den Index passt. (Welche Anfragen welche Arten von Index verwenden können, wurde in den vorangegangenen Abschnitten erklärt.)

- Wenn Sie die Verwendung des Index erzwungen haben und dadurch ein Index tatsächlich verwendet wird, gibt es zwei Möglichkeiten: Entweder hat das System Recht und die Verwendung des Index ist tatsächlich nicht angebracht oder die Kostenschätzungen der Anfragepläne entsprechen nicht der Wirklichkeit. Sie sollten also die Zeit Ihrer Anfrage mit und ohne Index messen. Der Befehl `EXPLAIN ANALYZE` kann hier nützlich sein.
- Wenn es sich herausstellt, dass die Kostenschätzungen falsch sind, gibt es wiederum zwei Möglichkeiten. Die Gesamtkosten werden aus den Kosten pro Zeile jedes Planknotens multipliziert mit der Schätzung, wie viele Zeilen der Planknoten ergeben wird (englisch *selectivity estimate*). Die Kosten der Planknoten können mit Konfigurationsparametern eingestellt werden (beschrieben in Abschnitt 16.4). Eine ungenaue Schätzung der Zeilenzahl kommt von unzureichenden Statistiken. Möglicherweise kann man da Abhilfe schaffen, indem man die Parameter zur Statistikeinholung ändert (siehe Referenz von `ALTER TABLE`).

  Wenn es Ihnen nicht gelingt, die Parameter so einzustellen, dass die Kostenschätzungen realistisch werden, dann werden Sie womöglich als letzten Ausweg die Verwendung der Indexe ausdrücklich erzwingen müssen. Vielleicht sollten Sie sich auch mit den PostgreSQL-Entwicklern in Verbindung setzen, um die Angelegenheit zu untersuchen.

12

Konsistenzkontrolle im Mehrbenutzerbetrieb

Dieses Kapitel beschreibt das Verhalten des PostgreSQL-Datenbanksystems, wenn zwei oder mehr Sitzungen versuchen, zur selben Zeit auf die selben Daten zuzugreifen. Das Ziel in dieser Situation ist, jeder Sitzung effizienten Zugriff zu ermöglichen, aber die Integrität der Daten strikt zu gewährleisten. Jeder Entwickler von Datenbankanwendungen sollte mit den Themen dieses Kapitels vertraut sein.

12.1 Einführung

Im Gegensatz zu herkömmlichen Datenbanksystemen, welche Sperren verwenden, um Konsistenz im Mehrbenutzerbetrieb zu gewährleisten, verwendet PostgreSQL ein Multiversionsmodell (*Multiversion Concurrency Control*, MVCC). Das bedeutet, dass während einer Anfrage jede Transaktion einen Schnappschuss der Daten (eine `Datenbankversion`), wie sie vor einiger Zeit waren, sieht, ungeachtet des aktuellen Zustands der zugrunde liegenden Daten. Das schützt eine Transaktion davor, inkonsistente Daten zu lesen, die von anderen Transaktionen durch Änderung an denselben Datenzeilen erzeugt werden könnten. Dadurch erhält jede Datenbanksitzung *Transaktionsisolation.*

Der Hauptunterschied zwischen dem Multiversionsmodell und einem Sperrenmodell ist, dass in MVCC die Sperren zum Lesen von Daten nicht mit Sperren zum Schreiben von Daten in Konflikt stehen, und daher blockiert das Lesen niemals das Schreiben und das Schreiben niemals das Lesen.

Sperren für Tabellen und Zeilen sind in PostgreSQLPostgreSQL auch vorhanden, für Anwendungen, die nicht leicht an das MVCC-Verhalten angepasst werden können. Durch die richtige Verwendung von MVCC kann normalerweise aber eine bessere Leistung erzielt werden.

12.2 Transaktionsisolation

Der SQL definiert Transaktionsisolation in vier Graden durch drei Phänomene, die zwischen gleichzeitigen Transaktionen verhindert werden müssen. Die unerwünschten Phänomene sind:

Dirty Read (Lesen ungültiger Daten)

> Eine Transaktion liest Daten, die von einer gleichzeitigen, aber noch nicht abgeschlossenen Transaktion geschrieben wurden.

Nonrepeatable Read (nichtwiederholbares Lesen)

Eine Transaktion liest Daten, welche sie vorher schon gelesen hat, erneut und stellt fest, dass die Daten von einer anderen Transaktion (die seit dem ersten Lesen abgeschlossen wurde) verändert worden sind.

Phantom Read (Lesen von Phantomdaten)

Eine Transaktion führt eine Anfrage mit Suchbedingung, welche eine Gruppe von Zeilen ergibt, wiederholt aus und stellt fest, dass sich die Gruppe der Zeilen, die die Suchbedingung erfüllen, wegen einer anderen Transaktion, die seitdem abgeschlossen wurde, geändert hat.

Die vier Transaktionsisolationsgrade und die sich daraus ergebenden Verhalten werden in Tabelle 12.1 beschrieben.

Isolationsgrad	*Dirty Read*	*Nonrepeatable Read*	*Phantom Read*
Read Uncommitted	möglich	möglich	möglich
Read Committed	nicht möglich	möglich	möglich
Repeatable Read	nicht möglich	nicht möglich	möglich
Serializable	nicht möglich	nicht möglich	nicht möglich

Tabelle 12.1: SQL-Transaktionsisolationsgrade

PostgreSQL bietet die Isolationsgrade *Read Committed* und *Serializable* an.

12.2.1 Der Isolationsgrad Read Committed

Read Committed ist der Standard-Isolationsgrad in PostgreSQL. Wenn eine Transaktion in diesem Isolationsgrad läuft, dann sieht eine `SELECT`-Anfrage nur Daten aus Transaktionen, welche abgeschlossen wurden (*commit*), bevor die Anfrage begann; sie sieht niemals Daten aus nicht abgeschlossenen Transaktionen oder aus Transaktionen, die während der Ausführung der Anfrage abgeschlossen wurden. (Das `SELECT` sieht allerdings die Auswirkungen von vorherigen Aktualisierungen in der eigenen Transaktion, obwohl die eigene Transaktion ja noch nicht abgeschlossen wurde.) Im Endeffekt sieht die `SELECT`-Anfrage einen Schnappschuss der Datenbank, wie sie in dem Moment, als die Anfrage mit der Ausführung begonnen hat, war. Beachten Sie aber, dass zwei aufeinander folgende `SELECT`-Befehle unterschiedliche Daten sehen könnten, selbst wenn sie in derselben Transaktion ausgeführt werden, wenn andere Transaktionen während der Ausführung des ersten `SELECT` abgeschlossen wurden.

Die Befehle `UPDATE`, `DELETE` und `SELECT FOR UPDATE` verhalten sich genauso wie `SELECT`, wenn es um das Suchen von Zielzeilen geht: Sie finden nur Zeilen, die von Transaktionen stammen, die schon abgeschlossen waren, als die Ausführung des Befehls begann. Es kann aber sein, dass eine solche Zielzeile, ehe sie gefunden wurde, schon von einer gleichzeitig laufenden Transaktion aktualisiert (oder gelöscht oder zur Aktualisierung markiert) worden ist. In diesem Fall wartet die zweite aktualisierende Transaktion, ob die erste Transaktion erfolgreich abgeschlossen (*commit*) oder zurückgerollt (*rollback*) wird (wenn sie überhaupt noch unabgeschlossen ist). Wenn die erste Transaktion zurückgerollt wird, sind ihre Auswirkungen hinfällig und die zweite Transaktion kann fortfahren und die ursprünglich gefundene Zeile aktualisieren. Wenn die erste Transaktion erfolgreich abgeschlossen wird, dann wird die Zeile von der zweiten Transaktion ignoriert, wenn sie von der ersten gelöscht wurde, ansonsten wird die zweite versuchen, ihre Operation an der aktualisierten Zeile auszuführen. Die Suchbedingung der Anfrage (die `WHERE`-Klausel) wird erneut ausgewertet, um zu prüfen, ob sie von der aktualisierten Version der Zeile immer noch erfüllt wird. Wenn ja, dann fährt die zweite Transaktion mit ihrer Operation fort, ausgehend von der aktualisierten Version der Zeile.

Wegen der obigen Regel ist es möglich, dass ein aktualisierender Befehl einen inkonsistenten Schnappschuss sieht: Er sieht die Auswirkungen von gleichzeitig ablaufenden Transaktionen, die die Zeilen betref-

fen, welche der Befehl aktualisieren will, aber er sieht nicht die Auswirkungen der gleichzeitigen Transaktionen auf andere Zeilen in der Datenbank. Dieses Verhalten macht den Modus *Read Committed* für Befehle mit komplexen Suchbedingungen ungeeignet. Für einfachere Fälle ist er aber genau richtig. Betrachten Sie zum Beispiel das Aktualisieren von Kontoständen mit Transaktionen wie

```
BEGIN;
UPDATE konten SET kontostand = kontostand + 100.00 WHERE kontonr = 12345;
UPDATE konten SET kontostand = kontostand - 100.00 WHERE kontonr = 7534;
COMMIT;
```

Wenn zwei solche Transaktionen gleichzeitig versuchen, den Kontostand von Konto 12345 zu ändern, dann wollen wir zweifellos, dass die zweite Transaktion von der aktualisierten Version der Kontozeile ausgeht. Da jeder Befehl nur eine vorherbestimmte Zeile bearbeitet, ergibt sich keine besorgniserregende Inkonsistenz daraus, dass die zweite Transaktion die aktualisierten Daten sieht.

Da im Modus *Read Committed* jeder neue Befehl mit einem neuen Schnappschuss anfängt, welcher die Änderungen aus allen bis zu diesem Zeitpunkt abgeschlossenen Transaktionen enthält, sehen nachfolgende Befehle die Auswirkungen von gleichzeitig ablaufenden, abgeschlossenen Transaktionen sowieso. Der Sinn und Zweck hier ist, dass wir innerhalb eines *einzelnen* Befehls einen absolut konsistenten Blick auf die Datenbank haben.

Die teilweise Transaktionsisolation, die durch den Modus *Read Committed* geboten wird, ist für viele Anwendungen ausreichend und ist auch schnell und leicht zu verwenden. In Anwendungen mit komplexen Anfragen und Aktualisierungen kann es allerdings notwendig sein, eine noch strengere Konsistenz der Datenbankansicht zu garantieren.

12.2.2 Der Isolationsgrad Serializable

Der Isolationsgrad *Serializable* bietet die strikteste Transaktionsisolation. Dieser Grad emuliert die serielle Ausführung von Transaktionen, als ob die Transaktionen nacheinander statt gleichzeitig ausgeführt worden wären. Anwendungen, die diesen Grad verwenden, müssen allerdings darauf vorbereitet sein, Transaktionen neu zu versuchen, wenn die “Serialisierung” fehlschlägt.

Wenn eine Transaktion im Modus *Serializable* ist, dann sieht ein `SELECT`-Befehl nur Daten aus Transaktionen, welche abgeschlossen wurden, bevor die neue Transaktion begann; er sieht niemals Daten aus nicht abgeschlossenen Transaktionen oder Änderungen aus Transaktionen, die während der Ausführung der aktuellen Transaktion abgeschlossen wurden. (Das `SELECT` sieht allerdings die Auswirkungen von vorherigen Aktualisierungen in der eigenen Transaktion, obwohl die eigene Transaktion ja noch nicht abgeschlossen wurde.) Das unterscheidet sich vom Modus *Read Committed* dadurch, dass ein `SELECT` einen Schnappschuss vom Anfang der Transaktion sieht, nicht vom Anfang des aktuellen Befehls innerhalb der Transaktion. Aufeinander folgende `SELECT`-Befehle innerhalb derselben Transaktion sehen also immer die gleichen Daten.

Die Befehle `UPDATE`, `DELETE` und `SELECT FOR UPDATE` verhalten sich genauso wie `SELECT`, wenn es um das Suchen von Zielzeilen geht: Sie finden nur Zeilen, die von Transaktionen stammen, die schon abgeschlossen waren, als die Ausführung der Transaktion begann. Es kann aber sein, dass eine solche Zielzeile, ehe sie gefunden wurde, schon von einer gleichzeitig laufenden Transaktion aktualisiert (oder gelöscht oder zur Aktualisierung markiert) worden ist. In diesem Fall wartet die serialisierbare Transaktion, ob die erste Transaktion erfolgreich abgeschlossen oder zurückgerollt wird (wenn sie überhaupt noch unabgeschlossen ist). Wenn die erste Transaktion zurückgerollt wird, dann sind ihre Auswirkungen hinfällig und die serialisierbare Transaktion kann fortfahren und die ursprünglich gefundene Zeile aktualisieren. Aber wenn die erste Transaktion erfolgreich abgeschlossen wird (und die Zeile tatsächlich aktualisiert oder gelöscht und nicht nur zur Aktualisierung ausgewählt wurde), wird die serialisierbare Transaktion mit der Meldung

```
ERROR:  Can't serialize access due to concurrent update
```

zurückgerollt, weil eine serialisierbare Transaktion keine Zeilen ändern kann, die von anderen Transaktionen nach Beginn der serialisierbaren Transaktion geändert wurden.

Wenn eine Anwendung diese Fehlermeldung sieht, sollte sie die aktuelle Transaktion abbrechen und die ganze Transaktion von vorne versuchen. Im zweiten Versuch sieht die Transaktion die Auswirkungen der anderen Transaktion als Teil ihres anfänglichen Schnappschusses von der Datenbank, sodass es keine logischen Widersprüche gibt, wenn die neue Version der Zeile als Ausgangspunkt für die Aktualisierungen in der neuen Transaktion verwendet wird.

Beachten Sie, dass nur schreibende Transaktionen gelegentlich wiederholt werden müssen; Transaktionen, die nur lesen, haben niemals Serialisierungskonflikte.

Der Modus *Serializable* gibt einem eine rigorose Garantie, dass jede Transaktion einen vollständig konsistenten Blick auf die Datenbank hat. Anwendungen müssen aber bereit sein, Transaktionen zu wiederholen, wenn gleichzeitige Änderungen es unmöglich machen, die Illusion der aufeinander folgenden Ausführung aufrechtzuerhalten. Da es mit enormem Aufwand verbunden sein kann, komplexe Transaktionen zu wiederholen, ist dieser Modus nur empfohlen, wenn aktualisierende Transaktionen Befehle enthalten, die so komplex sind, dass sie im Modus *Read Committed* falsche Ergebnisse liefern könnten. Meistens ist der Modus *Serializable* notwendig, wenn eine Transaktion mehrere aufeinander folgende Befehle ausführt, die einen identischen Blick auf die Datenbank haben müssen.

12.3 Ausdrückliche Sperren

PostgreSQL bietet verschiedene Sperrmodi, um den gleichzeitigen Zugriff auf Tabellendaten zu kontrollieren. Diese Modi können verwendet werden, um Sperren von der Anwendung aus zu kontrollieren, wenn MVCC nicht das gewünschte Verhalten bietet. Die meisten PostgreSQL-Befehle setzen auch automatisch die passenden Sperren, um sicherzustellen, dass die verwendeten Tabellen nicht entfernt oder auf nichtverträgliche Art verändert werden, während der Befehl ausgeführt wird. (Zum Beispiel kann `ALTER TABLE` nicht gleichzeitig mit irgendeiner anderen Operation an der gleichen Tabelle ausgeführt werden.)

12.3.1 Tabellensperren

Die unten folgende Liste zeigt die vorhandenen Sperrmodi und in welchen Zusammenhängen die Sperren von PostgreSQL automatisch gesetzt werden. Denken Sie daran, dass alle diese Sperren auf Tabellenebene wirken, selbst wenn der Name das Wort “row” (Zeile) enthält; die Namen der Sperrmodi sind historisch bedingt. In gewisser Weise spiegeln die Namen die typischen Verwendungssituationen eines Sperrmodus wider, aber die Bedeutung ist bei allen Sperren die gleiche. Der einzige wirkliche Unterschied zwischen den Sperrmodi ist, mit welchen anderen Modi sie in Konflikt stehen. Zwei Transaktionen können nicht zur gleichen Zeit für die gleiche Tabelle Sperren innehaben, die in Konflikt zueinander stehen. (Eine Transaktion kann allerdings nie mit sich selbst in Konflikt stehen. Sie kann zum Beispiel eine `ACCESS EXCLUSIVE`-Sperre setzen und später eine `ACCESS-SHARE` Sperre für dieselbe Tabelle.) Sperren, die nicht in Konflikt miteinander stehen, können von mehreren Transaktionen gleichzeitig gehalten werden. Beachten Sie auch besonders, dass einige Sperrmodi mit sich selbst in Konflikt stehen (zum Beispiel kann eine `ACCESS EXCLUSIVE`-Sperre nicht von mehreren Transaktionen gleichzeitig gehalten werden), aber einige nicht (zum Beispiel kann eine `ACCESS-SHARE` Sperre von mehreren Transaktionen gehalten werden). Wenn eine Sperre gesetzt wurde, bleibt sie bis zum Ende der Transaktion erhalten.

Um zu überprüfen, welche Sperren gerade im Datenbankserver gehalten werden, können Sie die Systemsicht `pg_locks` verwenden. Weitere Informationen, wie Sie das Sperrensystem überwachen können, finden Sie in Kapitel 23.

Tabellensperrmodi

`ACCESS SHARE`

Steht nur in Konflikt mit dem Sperrmodus `ACCESS EXCLUSIVE`.

Der Befehl `SELECT` setzt Sperren dieses Modus für alle verwendeten Tabellen. Generell setzt jede Anfrage, die nur eineTabelle liest und nicht verändert, diesen Sperrmodus.

`ROW SHARE`

Steht in Konflikt mit den Sperrmodi `EXCLUSIVE` und `ACCESS EXCLUSIVE`.

Der Befehl `SELECT FOR UPDATE` setzt Sperren dieses Modus für die Zieltabelle(n) (zusätzlich zu `ACCESS SHARE`-Sperren für alle anderen Tabellen, die verwendet werden, aber nicht in der `FOR UPDATE`-Klausel stehen).

`ROW EXCLUSIVE`

Steht in Konflikt mit den Sperrmodi `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`.

Der Befehl `UPDATE`, `DELETE` und `INSERT` setzen Sperren dieses Modus für die Zieltabelle (zusätzlich zu `ACCESS-SHARE` Sperren für alle anderen verwendeten Tabellen). Generell setzt jeder Befehl, der die Daten in einer Tabelle verändert, diesen Sperrmodus.

`SHARE UPDATE EXCLUSIVE`

Steht in Konflikt mit den Sperrmodi `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`. Dieser Modus schützt eine Tabelle gegen gleichzeitige Schemaveränderungen oder `VACUUM`-Läufe.

Automatisch von `VACUUM` (ohne `FULL`) gesetzt.

`SHARE`

Steht in Konflikt mit den Sperrmodi `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`. Dieser Modus schützt eine Tabelle gegen gleichzeitige Datenänderungen.

Automatisch von `CREATE INDEX` gesetzt.

`SHARE ROW EXCLUSIVE`

Steht in Konflikt mit den Sperrmodi `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`.

Dieser Sperrmodus wird von keinem PostgreSQL-Befehl automatisch verwendet.

`EXCLUSIVE`

Steht in Konflikt mit den Sperrmodi `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`. Dieser Modus erlaubt nur eine gleichzeitige `ACCESS SHARE`-Sperre, das heißt, gleichzeitig mit einer Transaktion, die diesen Sperrmodus hält, können nur Lesevorgänge in der Tabelle stattfinden.

Dieser Sperrmodus wird von keinem PostgreSQL-Befehl automatisch verwendet.

`ACCESS EXCLUSIVE`

Steht in Konflikt mit allen Sperrmodi (`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` und `ACCESS EXCLUSIVE`). Dieser Modus garantiert, dass die Transaktion, die diese Sperre hält, die einzige Transaktion ist, die auf diese Tabelle auf irgendeine Art zugreift.

Wird automatisch von den Befehlen `ALTER TABLE`, `DROP TABLE` und `VACUUM FULL` gesetzt. Dieser Modus wird auch vom Befehl `LOCK TABLE` verwendet, wenn kein anderer Sperrmodus ausdrücklich angegeben wurde.

Tipp

Nur eine `ACCESS EXCLUSIVE`-Sperre blockiert einen `SELECT`-Befehl (ohne `FOR UPDATE`).

12.3.2 Zeilensperren

Neben Tabellensperren gibt es Zeilensperren. Eine Zeilensperre für eine bestimmte Zeile wird automatisch gesetzt, wenn die Zeile aktualisiert (oder gelöscht oder zur Aktualisierung markiert) wird. Die Sperre wird gehalten, bis die Transaktion abgeschlossen oder zurückgerollt wird. Zeilensperren haben keine Auswirkungen auf das Anfragen von Daten; sie blockieren nur *Schreibvorgänge in der selben Zeile*. Um eine Zeilensperre zu setzen, ohne die Zeile wirklich zu verändern, wählen Sie die Zeile mit `SELECT FOR UPDATE` aus. Wenn eine bestimmte Zeilensperre gesetzt wurde, kann eine Transaktion eine Zeile mehrfach aktualisieren, ohne vor Konflikten Angst haben zu müssen. PostgreSQL speichert keine Informationen über veränderte Zeilen im Hauptspeicher und hat daher keine Höchstgrenze, wie viele Zeilen auf einmal gesperrt sein können. Das sperren einer Zeile kann allerdings Schreibvorgänge auf die Festplatte verursachen; ein `SELECT FOR UPDATE` verändert die ausgewählten Zeilen zum Beispiel, und dadurch muss auf die Festplatte geschrieben werden.

Außer Tabellen- und Zeilensperren gibt es noch exklusive und nichtexklusive Sperren auf Speicherseitenebene, welche den Lese- und Schreibzugriff auf die Tabellenspeicherseiten im gemeinsamen Pufferpool kontrollieren. Diese Sperren werden sofort wieder aufgehoben, sobald eine Zeile gelesen oder aktualisiert wurde. Anwendungsentwickler müssen sich in der Regel nicht mit diesen Sperren befassen, aber wir erwähnen sie hier der Vollständigkeit halber.

12.3.3 Verklemmungen

Durch die Verwendung von expliziten Sperren können **Verklemmungen** (englisch *deadlock*) auftreten, wenn zwei (oder mehr) Transaktionen jeweils Sperren halten, die die anderen erlangen wollen. Zum Beispiel, wenn Transaktion 1 eine exklusive Sperre für Tabelle 1 setzt und danach versucht, eine exklusive Sperre für Tabelle B zu erlangen, während Transaktion 2 schon eine exklusive Sperre für Tabelle B hält und jetzt eine exklusive Sperre für Tabelle A möchte, dann kann keine der beiden Transaktionen fortfahren. PostgreSQL entdeckt Verklemmungen automatisch und löst sie, indem eine der verwickelten Transaktionen abgebrochen wird, wodurch die andere(n) fortfahren können. (Welche Transaktion genau abgebrochen wird, ist schwer vorherzusagen und man sollte sich nicht darauf verlassen.)

Die beste Vorsorge gegen Verklemmungen ist im Allgemeinen, dass man sie vermeidet, indem man darauf achtet, dass alle Anwendungen in einer Datenbank Sperren für mehrere Objekte in einer einheitlichen Reihenfolge anfordern. Außerdem sollte man sicherstellen, dass die erste Sperre, die für ein Objekt in einer Transaktion angefordert wird, den höchsten Modus, den man für das Objekt benötigen wird, hat. Wenn es nicht praktikabel ist, dies vorher zu überprüfen, kann man Verklemmungen zur Laufzeit behandeln, indem man Transaktionen, die wegen Verklemmungen abgebrochen wurden, neu startet.

Solange keine Verklemmung entdeckt wird, wird eine Transaktion, die eine Tabellen- oder Zeilensperre angefordert hat, endlos darauf warten, dass in Konflikt stehende Sperren aufgehoben werden. Das bedeutet, dass es keine gute Idee ist, Transaktionen über längere Zeiträume offen zu lassen (z.B. während man auf eine Eingabe des Anwenders wartet).

12.4 Konsistenzkontrolle auf der Anwendungsseite

Da Lesevorgänge in PostgreSQL ungeachtet des Transaktionsisolationsgrads keine Daten sperren, können die Daten, die von einer Transaktion gelesen werden, von einer anderen Transaktion zeitgleich überschrie-

ben werden. Anders ausgedrückt, wenn eine Zeile von einem `SELECT` zurückgegeben wird, heißt das nicht, dass die Zeile im Moment der Rückgabe (d.h. irgendwann nachdem die Anfrage anfing) noch aktuell ist. Die Zeile könnte von einer Transaktion, die nach dieser abgeschlossen wurde, geändert oder gelöscht worden sein. Selbst wenn die Zeile "jetzt" noch gültig ist, könnte sie geändert oder gelöscht werden, bevor die aktuelle Transaktion abgeschlossen oder zurückgerollt wird.

Eine andere Möglichkeit, sich das vorzustellen, ist, dass jede Transaktion einen Schnappschuss des Datenbankinhalts sieht, und gleichzeitig ablaufende Transaktionen können sehr wohl unterschiedliche Schnappschüsse sehen. Der ganze Begriff des "Jetzt" ist also sowieso etwas zweifelhaft. Das ist normalerweise kein großes Problem, wenn die Clientanwendungen voneinander getrennt sind, aber wenn Clients über Kanäle außerhalb der Datenbank miteinander kommunizieren, kann das zu einem ernsthaften Durcheinander führen.

Um den aktuellen Zustand einer Zeile abzusichern und sie gegen gleichzeitig ablaufende Aktualisierungen zu schützen, müssen Sie den Befehl `SELECT FOR UPDATE` oder einen passenden `LOCK TABLE`-Befehl verwenden. (`SELECT FOR UPDATE` sperrt nur die Ergebniszeilen gegen gleichzeitige Aktualisierungen, wogegen `LOCK TABLE` die ganze Tabelle sperrt.) Das sollte man mit in die Überlegungen einfließen lassen, wenn man Anwendungen von anderen Umgebungen nach PostgreSQL übertragt.

Anmerkung

Vor Version 6.5 hat auch PostgreSQL Lesesperren verwendet. Daher gelten die obigen Überlegungen auch, wenn man von PostgreSQL-Versionen vor 6.5 kommt.

Globale Datenprüfungen erfordern unter MVCC etwas mehr Überlegung. Eine Bankanwendung könnte zum Beispiel überprüfen wollen, ob die Summe aller Soll-Beträge in einer Tabelle gleich der Summe aller Haben-Beträge in einer anderen Tabelle ist, wobei beide Tabellen ständig aktualisiert werden. Das Vergleichen zweier aufeinander folgender `SELECT sum(...)`-Anfragen würde im Isolationsgrad *Read Committed* nicht verlässlich sein, da in die zweite Anfrage wahrscheinlich Ergebnisse von Transaktionen mit einfließen, die nicht von der ersten mitgezählt wurden. Wenn man die Summen in einer einzelnen serialisierbaren Transaktion berechnen lässt, dann erhält man ein genaues Bild über die Auswirkungen der Transaktionen, die abgeschlossen wurden, bevor die serialisierbare begann. Aber es stellt sich die legitime Frage, ob die Antwort, wenn sie endlich ausgegeben wird, überhaupt noch relevant ist. Wenn die serialisierbare Transaktion selbst einige Änderungen vornimmt, bevor sie die Prüfung durchführt, dann wird die Bedeutung des Prüfungsergebnisses noch fragwürdiger, weil es jetzt einige, aber nicht alle Änderungen, die nach dem Start der Transaktion getätigt wurden, enthält. In solchen Fällen könnte man, wenn man sehr sorgfältig ist, alle für die Prüfung notwendigen Tabellen sperren, um so ein unstrittiges Bild der gegenwärtigen Tatsachen zu erhalten. Eine Sperre im Modus `SHARE` (oder höher) garantiert, dass es keine unabgeschlossene Transaktion, außer der eigenen, gibt, die Daten in der gesperrten Tabelle verändert hat.

Beachten Sie auch, wenn man ausdrückliche Sperren verwendet, um gleichzeitige Änderungen auszuschließen, dann sollte man den Isolationsgrad *Read Committed* verwenden, oder im Modus *Serializable* darauf achten, dass man die Sperren setzt, bevor man Anfragen ausführt. Eine ausdrückliche Sperre, die von einer serialisierbaren Transaktion gesetzt wurde, garantiert, dass keine anderen Transaktionen, die die Tabelle verändern, noch am laufen sind, aber wenn der von der Transaktion gesehene Schnappschuss vor dem Setzen der Sperre gemacht wurde, dann ist es möglich, dass er auch vor einigen Änderungen gemacht wurde, die aus seit dem abgeschlossenen Transaktionen stammen. Der Schnappschuss einer serialisierbaren Transaktion wird eingefroren, wenn die erste Anfrage oder der erste Datenmodifikationsbefehl (`SELECT`, `INSERT`, `UPDATE` oder `DELETE`) ausgeführt wird, also ist es möglich, Sperren zu setzen, bevor der Schnappschuss eingefroren wird.

12.5 Sperren und Indexe

Obwohl PostgreSQL nicht blockierendes Lesen und Schreiben in Tabellen ermöglicht, wird nicht blockierendes Lesen und Schreiben gegenwärtig nicht für alle in PostgreSQL vorhandene Indextypen geboten. Die verschiedenen Indextypen werden folgendermaßen behandelt:

B-Tree-Indexe

: Es werden kurzzeitige Lese- und Schreibsperren auf Seitenebene verwendet. Die Sperren werden sofort, nachdem eine Indexzeile gelesen oder eingefügt wurde, aufgehoben. B-Tree-Indexe bieten das beste Maß an gleichzeitigem Zugriff ohne Verklemmungsgefahr.

GIST- und R-Tree-Indexe

: Es werden Lese- und Schreibsperren auf Indexebene verwendet. Die Sperren werden aufgehoben, wenn der Befehl beendet ist.

Hash-Indexe

: Es werden Lese- und Schreibsperren auf Seitenebene verwendet. Die Sperren werden aufgehoben, nachdem die Seite verarbeitet worden ist. Seitensperren bieten bessere gleichzeitige Ausführung als Indexsperren, sind aber für Verklemmungen anfällig.

Kurz gesagt sind B-Tree-Indexe empfohlen für Anwendungen, die ein hohes Maß an Parallelität erfordern.

13

Tipps zur Leistungsverbesserung

Die Leistung von Anfragen kann von vielen Dingen beeinflusst werden. Einige davon können vom Benutzer verändert werden, aber andere Faktoren sind fest im Design des Systems verankert. Dieses Kapitel gibt einige Hinweise, wie Sie die Leistungsfähigkeit von PostgreSQL verstehen und verbessern können.

13.1 EXPLAIN verwenden

PostgreSQLPostgreSQL entwirft für jede Anfrage, die ausgeführt werden soll, einen **Anfrageplan** (englisch *query plan*). Die Wahl des richtigen Plans, der der Struktur der Anfrage und den Eigenschaften der Daten entspricht, ist für eine gute Leistung absolut kritisch. Sie können den Befehl `EXPLAIN` verwenden, um zu sehen, welchen Anfrageplan das System für jede beliebige Anfrage erstellt. Das Lesen des Plans ist eine Kunst für sich, für die es eigentlich eine ausführliche Anleitung geben sollte, was dies hier nicht ist, aber wir geben hier einige grundsätzliche Informationen.

Die Zahlen, die gegenwärtig von `EXPLAIN` ausgegeben werden, sind:

- geschätzte Startkosten (Zeit, die benötigt wird, bis die Ausgabe anfangen kann, z.B. Zeit zum Sortieren in einem Sortierknoten)
- geschätzte Gesamtkosten (Wenn alle Zeilen zurückgegeben würden, was nicht sein muss: Eine Anfrage mit einer `LIMIT`-Klausel wird zum Beispiel nicht die gesamten Kosten verursachen.)
- geschätzte Zahl der Zeilen, die dieser Planknoten ausgeben wird (wiederum nur, wenn er ganz ausgeführt wird)
- geschätzte Breite (in Bytes) der Zeilen, die dieser Planknoten ausgibt

Die Einheiten der Kostenwerte ist das Lesen (*fetch*) einer Diskseite (*disk page*). (Die CPU-Bemühungen werden mit relativ willkürlichen Umrechnungsfaktoren in Diskseiten-Einheiten umgewandelt. Wenn Sie mit diesen Faktoren experimentieren wollen, dann schauen Sie sich die Liste der Laufzeiteinstellungen in Abschnitt 16.4 an.)

Es ist wichtig anzumerken, dass die Kosten eines übergeordneten Knotens die Kosten aller untergeordneten Knoten enthalten. Es ist auch wichtig zu bedenken, dass die Kosten nur die Vorgänge einberechnen, um die sich der Planer/Optimierer kümmert. Insbesondere beinhalten die Kosten nicht die Zeit, die benötigt wird, um die Ergebnisse an den Client zu schicken, was in der tatsächlich abgelaufenen Zeit ein ziemlich beherrschender Faktor sein könnte, aber vom Planer ignoriert wird, weil es nicht durch Veränderung des Plans verbessert werden kann. (Jeder korrekte Plan ergibt dieselbe Zeilenmenge, hoffen wir doch.)

Die Zahl der ausgegebenen Zeilen ist etwas knifflig, weil sie *nicht* die Zahl der Zeilen darstellt, die von der Anfrage gelesen oder verarbeitet werden, sie ist normalerweise geringer, weil die geschätzten Effekte einer etwaigen Bedingung in einer WHERE-Klausel in diesem Knoten mit einfließen. Im Idealfall wird die Zeilenschätzung für den obersten Knoten in etwa gleich der Zahl der tatsächlich von dieser Anfrage ausgewählten, aktualisierten oder gelöschten Zeilen sein.

Hier sind einige Beispiele (aus der Regression-Test-Datenbank nach einem VACUUM ANALYZE, mit den Entwicklungsquelltext von 7.3):

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

Das ist ein *Sequential Scan*, also eine Suche durch die Tabelle von Anfang bis Ende, was so ziemlich das einfachste ist, was es gibt. (cost sind die Start- und Gesamtkosten, rows die Zeilenschätzung und width die Zeilenbreite.) Wenn Sie die Anfrage

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

ausführen, werden Sie feststellen, dass tenk1 Diskseiten und 10000 Zeilen hat. Die Kosten werden also berechnet aus 233 zu lesenden Seiten, die per Definition 1,0 pro Stück kosten, plus 10000 mal cpu_tuple_cost (CPU-Kosten pro Zeile), was gegenwärtig 0,01 ist (probieren Sie SHOW cpu_tuple_cost).

Jetzt modifizieren wir die Anfrage und fügen eine WHERE-Klausel hinzu:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..358.00 rows=1033 width=148)
   Filter: (unique1 < 1000)
```

Die Schätzung der Ergebniszeilen hat sich wegen der WHERE-Klausel verringert. Allerdings muss immer noch die gesamte Tabelle mit 10000 Zeilen durchsucht werden, weswegen die Kosten nicht gefallen sind; tatsächlich sind sie sogar ein bisschen gestiegen, um der gestiegenen CPU-Leistung vom Prüfen der WHERE-Bedingung Rechnung zu tragen.

Die tatsächliche Anzahl der Zeilen, die diese Anfrage liefert, ist 1000, aber die Schätzung ist nur annähernd. Wenn Sie versuchen, dieses Experiment selbst zu wiederholen, werden Sie wahrscheinlich geringfügig andere Schätzungen sehen; außerdem wird sich die Schätzung nach jedem ANALYZE-Befehl ändern, da die von ANALYZE erstellten Statistiken aus einer zufälligen Auswahl aus der Tabelle erstellt werden.

Ändern Sie die Anfrage noch einmal, um die Bedingung noch stärker einschränkend zu machen:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.33 rows=49 width=148)
   Index Cond: (unique1 < 50)
```

Sie sehen, dass, wenn Sie die WHERE-Bedingung ausreichend restriktiv machen, wird der Planer irgendwann entscheiden, ein *Index Scan* (eine Indexsuche) billiger ist als eine sequenzielle Suche. Dieser Plan wird wegen des Index nur 50 Zeilen aufsuchen müssen und ist daher besser, obwohl das Aufsuchen jeder einzelnen Zeile teurer ist, als eine Diskseite sequenziell zu lesen.

Fügen Sie der WHERE-Klausel eine weitere Bedingung hinzu:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND stringu1 = 'xxx';

                                QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.45 rows=1 width=148)
   Index Cond: (unique1 < 50)
   Filter: (stringu1 = 'xxx'::name)
```

Die zusätzliche Bedingung `stringu1 = 'xxx'` verringert die Schätzung der Ausgabezeilen, aber nicht die Kosten, weil wir immer noch die gleiche Menge von Zeilen durchsuchen müssen. Beachten Sie, dass die Bedingung mit `stringu1` nicht als Suchbedingung für den Index verwendet werden kann (da der Index nur für die Spalte `unique1` ist). Anstelle dessen wird sie als Filter für die Zeilen, die aus der Indexsuche kommen, verwendet. Daher haben sich die Kosten ein bisschen erhöht, um diese zusätzlichen Berechnungen mit einzuschließen.

Jetzt verbinden wir zwei Tabellen unter Verwendung der besprochenen Spalten:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 =
t2.unique2;

                               QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                                      (cost=0.00..179.33 rows=49 width=148)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                                      (cost=0.00..3.01 rows=1 width=148)
           Index Cond: ("outer".unique2 = t2.unique2)
```

In diesem Nested-Loop-Verbund (geschachtelte Schleifen) verwendet die äußere Schleife die gleiche Indexsuche, die wir im vorletzten Beispiel gesehen hatten, also sind die Kosten und Zeilenzahlen gleich, weil wir die WHERE-Klausel `unique1 < 50` in diesem Knoten anwenden. Die Klausel `t1.unique2 = t2.unique2` ist noch nicht relevant und beeinflusst folglich die Zeilenzahl der äußeren Schleife nicht. In der inneren Schleife wird der Wert von `unique2` in der aktuellen Zeile der äußeren Schleife in die Indexsuche eingesetzt und erzeugt dadurch eine Indexbedingung wie `t2.unique2 = konstante`. Wir sehen folglich den gleichen Plan und die gleichen Kosten, die wir für, sagen wir `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42` sehen würden. Die Kosten des Nested-Loop-Knotens werden dann berechnet aus den Kosten für die äußere Schleife plus einer Wiederholung der inneren Schleife für jede äußere Zeile (hier 49 * 3,01) plus etwas CPU für die Bearbeitung der Verbundoperation.

In diesem Beispiel ist die Ergebniszeilenzahl des Verbunds gleich dem Produkt der Zeilenzahlen aus den beiden Schleifen, aber das muss nicht immer der Fall sein, weil man im Allgemeinen WHERE-Klauseln haben kann, die auf beide Tabellen verweisen und daher nur im Verbundknoten angewendet werden können, nicht in einer der beiden Schleifen. Wenn wir zum Beispiel `WHERE ... AND t1.hundred < t2.hundred` hinzufügen würden, dann würde dadurch die Zahl der Ergebniszeilen sinken, aber die beiden untergeordneten Schleifen bleiben gleich.

Eine Möglichkeit, um sich andere Pläne anzuschauen, ist, den Planer zu zwingen, die Strategie, von der er dachte, dass sie die beste wäre, mithilfe der entsprechenden Laufzeiteinstellungen zu ignorieren. (Das ist eine ziemlich primitive Methode, aber eine sehr nützliche. Siehe auch Abschnitt 13.3.)

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 =
t2.unique2;

                              QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=179.45..563.06 rows=49 width=296)
   Hash Cond: ("outer".unique2 = "inner".unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
   ->  Hash  (cost=179.33..179.33 rows=49 width=148)
         ->  Index Scan using tenk1_unique1 on tenk1 t1
                                  (cost=0.00..179.33 rows=49 width=148)
               Index Cond: (unique1 < 50)
```

Dieser Plan schlägt vor, die 50 interessanten Zeilen aus `tenk1` mit demselben guten alten Indexscan herauszusuchen, diese dann in einer Hash-Tabelle im Hauptspeicher abzulegen, dann eine sequenzielle Suche durch `tenk2` durchzuführen und für jede Reihe von `tenk2` in der Hash-Tabelle nach möglichen Übereinstimmungen für `t1.unique2 = t2.unique2` zu suchen. Die Kosten, um `tenk1` zu lesen und die Hash-Tabelle einzurichten, gehören vollständig zu den Startkosten des Hash-Verbunds, da wir keine Zeilen geliefert bekommen, bis wir mit dem Lesen von `tenk2` anfangen. Die Schätzung der Gesamtzeit für den Verbund enthält auch einen hohen Preis für den CPU-Aufwand, um die Hash-Tabelle 10000 Mal zu prüfen. Beachten Sie aber, dass die Kosten *nicht* 10000 mal 179,33 betragen; die Einrichtung der Hash-Tabelle findet bei diesem Plantyp nur einmal statt.

Man kann die Genauigkeit der vom Planer geschätzten Kosten mit dem Befehl `EXPLAIN ANALYZE` überprüfen. Dieser Befehl führt die Anfrage aus und zeigt dann die in jedem Planknoten gemessene Laufzeit zusammen mit den geschätzten Kosten, die ein einfaches `EXPLAIN` zeigt. Wir könnten zum Beispiel ein Ergebnis wie dieses erhalten:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND
t1.unique2 = t2.unique2;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..327.02 rows=49 width=296)
                                (actual time=1.18..29.82 rows=50 loops=1)
   ->  Index Scan using tenk1_unique1 on tenk1 t1
                (cost=0.00..179.33 rows=49 width=148)
                                (actual time=0.63..8.91 rows=50 loops=1)
         Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2
                (cost=0.00..3.01 rows=1 width=148)
                                (actual time=0.29..0.32 rows=1 loops=50)
         Index Cond: ("outer".unique2 = t2.unique2)
 Total runtime: 31.60 msec
```

Beachten Sie, dass die tatsächliche Laufzeit in Millisekunden Echtzeit ausgegeben wird, wohingegen die Kostenschätzungen in willkürlichen Einheiten (Diskseiten-Fetch) sind. Es ist also unwahrscheinlich, dass die Zahlen übereinstimmen; worauf Sie achten sollten, sind die Verhältnisse.

In einigen Anfrageplänen ist es möglich, dass ein Subplan mehrmals ausgeführt wird. Im oben gesehenen Nested-Loop-Plan zum Beispiel, wird die innere Indexsuche einmal für jede Zeile in der äußeren Schleife ausgeführt. In solchen Fällen steht bei `loops`, wie oft der Planknoten insgesamt ausgeführt wurde, und die berichtete Ausführungszeit ist der Durchschnitt aus allen Läufen. Das wird gemacht, damit die Zahlen leicht mit den Kostenschätzungen verglichen werden können. Multiplizieren Sie diese Ausführungszeit mit der `loops`-Angabe, um die gesamte in dem Planknoten gebrauchte Zeit zu berechnen.

Die von `EXPLAIN ANALYZE` gezeigte Gesamtlaufzeit (`Total runtime`) enthält die Zeit, um den Executor zu starten und zu beenden, sowie die Zeit, um die Ergebniszeilen zu verarbeiten. Sie enthält nicht die Zeit zum Parsen, Umschreiben oder Planen. Bei einer Anfrage mit `SELECT` wird die Gesamtlaufzeit normalerweise nur etwas größer sein als die vom obersten Planknoten berichtete Zeit. Bei `INSERT`-, `UPDATE`- und `DELETE`-Befehlen kann die Gesamtlaufzeit aber beträchtlich größer sein, weil die Zeit zum Verarbeiten der Ergebniszeilen mitgerechnet wird. Bei diesen Befehlen ist die Zeit des obersten Planknoten, im Prinzip die Zeit, die gebraucht wurde, um die neuen Zeilen zu berechnen und/oder die alten zu finden, aber sie enthält nicht die Zeit, die gebraucht wurde, um die Änderungen zu vollziehen.

Es sollte auch erwähnt werden, dass die Ergebnisse von `EXPLAIN` nicht auf andere Situationen übertragen werden sollten. Man kann zum Beispiel nicht davon ausgehen, dass die Ergebnisse von kleinen Testtabellen auch für große Tabellen gelten. Die Kostenschätzungen des Planers sind nicht linear und so könnte er sehr wohl für große und kleine Tabellen unterschiedliche Pläne auswählen. Um ein extremes Beispiel zu nehmen: Bei einer Tabelle, die nur eine Diskseite belegt, erhalten Sie fast immer einen Plan mit sequenzieller Suche, egal ob Indexe vorhanden sind oder nicht. Der Planer erkennt, dass er so oder so eine Diskseite lesen muss, um die Tabelle zu verarbeiten, und dass es also keine Sinn macht, zusätzliche Seiten zu lesen, um in einen Index zu schauen.

13.2 Vom Planer verwendete Statistiken

Wie wir im vorigen Abschnitt gesehen haben, muss der Anfrageplaner die Zahl der von einer Anfrage zurückgegebenen Zeilen abschätzen können, um beim Anfrageplan eine gute Wahl treffen zu können. Dieser Abschnitt bietet einen kurzen Überblick über die Statistiken, die das System für diese Schätzungen verwendet.

Ein Bestandteil der Statistiken ist die Gesamtzahl der Einträge in jeder Tabelle und in jedem Index sowie die Zahl der Diskblöcke, die von jeder Tabelle und von jedem Index belegt werden. Diese Informationen werden in der Tabelle `pg_class` in den Spalten `reltuples` und `relpages` abgelegt. Wir können sie mit einer Anfrage wie dieser ansehen:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE
'tenk1%';

    relname    | relkind | reltuples | relpages
---------------+---------+-----------+----------
 tenk1         | r       |     10000 |      233
 tenk1_hundred | i       |     10000 |       30
 tenk1_unique1 | i       |     10000 |       30
 tenk1_unique2 | i       |     10000 |       30
(4 rows)
```

Hier sehen wir, dass die Tabelle `tenk1` 10000 Zeilen enthält, genauso wie ihre Indexe. Aber die Indexe sind (nicht überraschend) viel kleiner als die Tabelle.

Aus Effizienzgründen werden `reltuples` und `relpages` nicht laufend aktualisiert, sodass sie normalerweise nur annähernde Werte enthalten (was für den Planer ausreichend ist). Wenn eine Tabelle erzeugt wird, werden sie mit willkürlichen Vorgabewerten gefüllt (gegenwärtig 1000 bzw. 10). Aktualisiert werden sie von bestimmten Befehlen, gegenwärtig `VACUUM`, `ANALYZE` und `CREATE INDEX`. Ein alleinstehendes `ANALYZE`, das nicht Teil eines `VACUUM`-Befehls ist, erzeugt einen annähernden Wert für `reltuples`, da es nicht jede Zeile der Tabelle liest.

Die meisten Anfragen lesen nur einen Teil der Zeilen in einer Tabelle aus, weil `WHERE`-Klauseln die Menge der Zeilen einschränken. Der Planer muss daher eine Schätzung der **Selektivität** einer `WHERE`-Klausel abgeben, dass heißt, er muss den Bruchteil der Zeilen schätzen, die die einzelnen Bedingungen in der `WHERE`-Klausel erfüllen. Die Informationen, die für diese Aufgabe verwendet werden, sind in der Systemtabelle `pg_statistic` abgelegt. Die Einträge in `pg_statistic` werden von den Befehlen `ANALYZE` und `VACUUM ANALYZE` aktualisiert und sind auch direkt nach der Aktualisierung immer Annäherungswerte.

Wenn man die Statistiken selbst untersuchen will, ist es besser, die Sicht `pg_stats` statt `pg_statistic` direkt anzuschauen. `pg_stats` wurde entworfen, um leichter lesbar zu sein. Außerdem ist `pg_stats` von allen lesbar, wohingegen `pg_statistic` nur von einem Superuser lesbar ist. (Dadurch wird verhindert, dass nichtprivilegierte Benutzer durch die Statistiken etwas über den Inhalt der Tabellen anderer Benutzer erfahren. Die Sicht `pg_stats` zeigt nur Zeilen über Tabellen, die der aktuelle Benutzer lesen kann.) Wir könnten zum Beispiel Folgendes machen: Dieses Beispiel ist viel zu breit, aber ein besseres gibt's zur Zeit nicht.

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename =
'road';


attname|n_distinct|        most_common_vals
--------+-----------+--------------------------------------------------------
 name   | -0.467008 | {"I- 580       Ramp","I- 880      Ramp","Sp Railroad
","I- 580                  ","I- 680               Ramp","I- 80
Ramp","14th                         St  ","5th                            St
","Mission                        Blvd","I- 880                           "}
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
```

Tabelle 13.1 zeigt die Spalten, die es in `pg_stats` gibt.

Name	Datentyp	Beschreibung
`tablename`	`name`	Name der Tabelle, in die die Spalte gehört.
`attname`	`name`	Name der Spalte, die von dieser Zeile beschrieben wird.
`null_frac`	`real`	Anteil der Spalteneinträge, die NULL sind.
`avg_width`	`integer`	Durchschnittliche Breite (Größe) in Bytes der Spalteneinträge.
`n_distinct`	`real`	Wenn größer als null, die geschätzte Zahl der voneinander verschiedenen Werte in der Spalte. Wenn kleiner als null, das Negative der Zahl der verschiedenen Werte geteilt durch die Zahl der Zeilen. (Die negative Form wird verwendet, wenn `ANALYZE` glaubt, dass die Zahl der verschiedenen Werte wahrscheinlich mit der Tabelle wächst; die positive Form wird verwendet, wenn die Spalte eine feste Zahl von möglichen Werten zu haben scheint.) Zum Beispiel bedeutet -1 eine Spalte, wo alle Werte unterschiedlich sind und die Zahl der verschiedenen Werte genauso groß ist wie die Zahl der Zeilen.

Tabelle 13.1: Spalten von `pg_stats`

Name	Daten-typ	Beschreibung
most_common_vals	text[]	Eine Liste der häufigsten Werte in der Spalte. (Fällt weg, wenn keine Werte häufiger als andere zu sein scheinen.)
most_common_freqs	real[]	Eine Liste der Häufigkeiten der häufigsten Werte, d.h. die Zahl der Vorkommen geteilt durch die Zahl der Zeilen.
histogram_bounds	text[]	Eine Liste von Werten, die die Spaltenwerte in etwa gleich große Gruppen unterteilen (Histogrammgrenzen). Die Werte in most_common_vals, falls vorhanden, werden von dieser Histogrammberechnung ausgeschlossen. (Diese Spalte wird nicht gefüllt, wenn der Datentyp der Spalte keinen Operator < hat oder wenn die Liste in most_common_vals schon alle Spaltenwerte abdeckt.)
correlation	real	Die statistische Korrelation zwischen der physikalischen Reihenfolge der Zeilen und der logischen Reihenfolge der Spaltenwerte. Reicht von -1 bis +1. Wenn der Wert nahe -1 oder +1 ist, dann wird eine Indexsuche durch die Spalte als billiger eingeschätzt, als wenn er nahe null ist, weil die Verteilung der Festplattenzugriffe im ersten Fall weniger zufällig ist. (Diese Spalte wird nicht gefüllt, wenn der Datentyp der Spalte keinen Operator < hat.)

Tabelle 13.1: Spalten von pg_stats (Forts.)

Die Höchstzahl der Einträge in den Arrays most_common_vals und histogram_bounds kann mit dem Befehl ALTER TABLE SET STATISTICS für jede Spalte individuell eingestellt werden. Die Vorgabe ist gegenwärtig 10. Wenn Sie die Grenze erhöhen, kann der Planer unter Umständen genauere Schätzungen erstellen, besonders wenn die Daten sehr ungleichmäßig verteilt sind. Der Preis dafür wäre, dass etwas mehr Platz in pg_statistic gebraucht würde und das Berechnen der Schätzungen etwas länger dauern würde. Umgekehrt könnte eine niedrigere Grenze für Spalten mit einfacher Datenverteilung angebracht sein.

13.3 Den Planer mit expliziten JOIN-Klauseln kontrollieren

Es ist möglich, den Anfrageplaner in einem gewissen Maß durch die Verwendung des expliziten JOIN-Syntax zu kontrollieren. Um zu sehen, warum das wichtig sein kann, müssen wir erst etwas Hintergrundwissen besprechen.

In einer einfachen Verbundanfrage wie

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

kann der Planer die angegebenen Tabellen in beliebiger Reihenfolge verbinden. Er könnte zum Beispiel einen Anfrageplan erstellen, der A und B unter Verwendung der WHERE-Bedingung a.id = b.id verbindet und dann das Ergebnis unter Verwendung der anderen WHERE-Bedingung mit C verbindet. Oder er könnte B und C verbinden und dann das Ergebnis mit A. Oder er könnte A und C verbinden und dann das Ergebnis mit B, aber das wäre ineffizient, da in dem Fall das volle Kreuzprodukt aus A und C errechnet werden müsste, weil es keine passende WHERE-Bedingung gibt, um den Verbund optimieren zu können. (Alle Verbunde im PostgreSQLPostgreSQL-Executor geschehen zwischen zwei Eingabetabellen, also muss das Ergebnis auf eine dieser Arten aufgebaut werden.) Es ist wichtig anzumerken, dass diese unterschiedlichen Verbundmöglichkeiten alle semantisch äquivalente Ergebnisse liefern, aber enorm unterschiedliche Ausführungskosten haben können. Daher untersucht der Planer sie alle um den effizientesten Plan zu finden.

Wenn eine Anfrage nur zwei oder drei Tabellen verwendet, gibt es nicht viele Verbundreihenfolgen, um die es sich zu kümmern gilt. Aber die Zahl der möglichen Verbundreihenfolgen steigt exponentiell mit der Zahl der Tabellen. Jenseits von rund zehn Tabellen ist es nicht mehr praktikabel, alle Möglichkeiten zu durchsuchen und auch bei sechs oder sieben Tabellen kann das Planen lästig lange dauern. Wenn zu viele Tabellen angegeben wurden, wechselt PostgreSQL von der erschöpfenden Suche zu einer **genetischen** Suche auf Wahrscheinlichkeitsbasis durch eine begrenzte Zahl von Möglichkeiten. (Die Schwelle für diesen Wechsel wird durch den Konfigurationsparameter `geqo_threshold` eingestellt.) Die genetische Suche braucht weniger Zeit, findet aber nicht unbedingt den bestmöglichen Plan.

Wenn die Anfrage einen äußeren Verbund enthält, dann hat der Planer viel weniger Freiheit als bei einfachen, inneren Verbunden. Betrachten wir zum Beispiel

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Obwohl die Bedingungen in dieser Anfrage oberflächlich gesehen dem vorangegangenen Beispiel ähnlich sind, ist doch die Bedeutung unterschiedlich, weil für jede Zeile von A, die keine passende Reihe im Verbund von B und C hat, eine Ergebniszeile erzeugt werden muss. Daher hat der Planer hier keine Wahl bezüglich der Verbundreihenfolge: Er muss B und C verbinden und dann A mit dem Ergebnis. Dementsprechend dauert das Planen dieser Anfrage auch kürzer als bei der vorigen Anfrage.

Der PostgreSQL-Anfrageplaner behandelt alle ausdrücklichen `JOIN`-Syntaxen so, dass durch sie die Verbundreihenfolge festgelegt wird, obwohl dies bei inneren Verbunden mathematisch nicht notwendig wäre. Alle folgenden Anfragen ergeben daher das gleiche Ergebnis:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

aber die zweite und dritte kann schneller als die erste geplant werden. Dieser Effekt ist bei nur drei Tabellen nicht weiter beachtenswert, kann aber bei Anfragen mit vielen Tabellen ein Lebensretter sein.

Sie müssen die Verbundreihenfolge nicht vollständig einschränken um die Suchzeit zu verkürzen. Sie können nämlich `JOIN`-Operatoren in einer normalen `FROM`-Liste verwenden. Zum Beispiel:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Das zwingt den Planer, A und B zu verbinden, bevor sie mit anderen Tabellen verbunden werden, schränkt die Wahlmöglichkeiten aber nicht anderweitig ein. Dadurch wird in diesem Beispiel die Zahl der möglichen Verbundreihenfolgen um den Faktor 5 reduziert.

Wenn Sie in einer komplexen Anfrage eine Mischung aus inneren und äußeren Verbunden haben, wollen Sie vielleicht die Suche des Planers nach einer guten Reihenfolge für die inneren Verbunde innerhalb der äußeren Verbunde nicht einschränken. Mit der `JOIN`-Syntax können Sie das nicht direkt machen, aber Sie können diese syntaktische Beschränkung mit Unteranfragen umgehen; zum Beispiel:

```
SELECT * FROM d LEFT JOIN
              (SELECT * FROM a, b, c WHERE ...) AS ss
                ON (...);
```

Hier muss der Verbund mit D der letzte Schritt im Anfrageplan sein, aber der Planer kann alle möglichen Verbundreihenfolgen zwischen A, B und C in Betracht ziehen.

Das Beschränken der Suche des Planers auf diese Art ist eine nützliche Technik, um die Planzeit zu reduzieren auch auch den Planer zu einem guten Anfrageplan zu führen. Wenn der Planer in der Standardeinstellung eine schlechte Verbundreihenfolge wählt, können Sie eine bessere Reihenfolge mit der `JOIN`-Syntax erzwingen. Das heißt natürlich, wenn Sie eine bessere Reihenfolge kennen. Ausprobieren wird empfohlen.

13.4 Eine Datenbank füllen

Wenn Sie eine Datenbank am Anfang mit Daten füllen, müssen Sie möglicherweise eine große Zahl von Einfügeoperationen vornehmen. Hier sind einige Tipps und Techniken, um das so effizient wie möglich zu machen.

13.4.1 Autocommit ausschalten

Schalten Sie Autocommit, also das automatische Abschließen einer Transaktion nach jedem Befehl, aus und führen Sie nur ein `COMMIT` am Ende aus. (Im normalen SQL machen Sie das, indem Sie ein `BEGIN` am Anfang und ein `COMMIT` am Ende ausführen. Manche Clientbibliotheken machen das vielleicht automatisch hinter Ihrem Rücken; dann müssen Sie schauen, dass die Bibliothek es macht, wenn Sie es wollen.) Wenn Sie es zulassen, dass nach jeder Einfügeoperation eine Transaktion abgeschlossen wird, dann hat PostgreSQL bei jeder neuen Zeile eine Menge Arbeit. Ein zusätzlicher Vorteil, wenn man alle Einfügeoperationen in einer Transaktion ausführt, ist, dass wenn das Einfügen einer Zeile fehlschlagen sollte, dann alle bis dahin eingefügten Zeilen zurückgerollt werden und Sie nicht mit teilweise geladenen Daten sitzen bleiben.

13.4.2 COPY FROM verwenden

Verwenden Sie `COPY FROM STDIN`, um alle Zeilen mit einem Befehl zu laden, anstatt eine Reihe von `INSERT`-Befehlen zu verwenden. Dadurch wird die Zeit zum Parsen, Planen usw. sehr verringert. Wenn Sie das tun, müssen Sie Autocommit nicht ausschalten, weil es sowieso nur ein Befehl ist.

13.4.3 Indexe entfernen

Wenn Sie Daten in eine frisch erzeugte Tabelle laden, ist es der schnellste Weg, die Tabelle zu erzeugen, die Daten mit `COPY` zu laden und dann die benötigten Indexe zu erzeugen. Es ist schneller, einen Index für schon bestehende Daten zu erzeugen, als ihn schrittweise mit jeder neuen Zeile aufzubauen.

Wenn Sie eine bestehende Tabelle um viele Daten erweitern wollen, können Sie den Index entfernen, die Daten laden und den Index erneut erzeugen. Während der Index fehlt, kann dadurch natürlich die Datenbankleistung für andere Benutzer negativ beeinflusst werden. Man sollte sich das Entfernen eines Unique Index auch genau überlegen, da die Fehlerprüfung, die durch einen solchen Index geboten wird, nicht durchgeführt wird, wenn der Index fehlt.

13.4.4 Danach ANALYZE ausführen

Es ist empfehlenswert, `ANALYZE` oder `VACUUM ANALYZE` immer dann auszuführen, wenn man viele Daten eingefügt oder aktualisiert hat, einschließlich, nachdem man die Tabelle anfänglich mit Daten gefüllt hat. Das stellt sicher, dass der Planer aktuelle Statistiken über die Tabelle hat. Ohne Statistiken oder mit veralteten Statistiken könnte der Planer schlechte Anfragepläne auswählen, was zu schlechter Leistung bei Anfragen, die die Tabelle verwenden, führt.

Teil III

Server-Administration

Dieser Teil behandelt Themen, die für Administratoren von PostgreSQL-Datenbanken von Interesse sind. Das beinhaltet die Installation der Software, das Einrichten und Konfigurieren des Servers, die Verwaltung von Benutzern und Datenbanken sowie Wartungsaufgaben. Jeder, der einen PostgreSQL-Server betreibt, sei es für den persönlichen Gebrauch oder besonders im Produktionsbetrieb, sollte mit den in diesem Teil behandelten Themen vertraut sein.

Die Informationen in diesem Teil sind in etwa in der Reihenfolge angeordnet, in der ein neuer Anwender sie lesen sollte. Die Kapitel sind aber selbstständig und können, wenn gewünscht einzeln gelesen werden. In diesem Teil sind die Informationen erzählerisch in Themenbereichen aufbereitet. Leser, die eine vollständige Beschreibung eines bestimmten Befehls suchen, sollten in ***Teil VI*** nachschauen.

Die ersten paar Kapitel sind so geschrieben, dass sie ohne Vorkenntnisse verstanden werden können, damit neue Anwender, die ihren eigenen Server einrichten müssen, ihre Erkundungen mit diesem Teil beginnen können. Der restlichen Kapitel dieses Teils, die von der Feinabstimmung und der Wartung handeln, gehen davon aus, dass der Leser mit der allgemeinen Verwendung des PostgreSQL-Datenbanksystems vertraut ist. Zusätzliche Informationen können in ***Teil I*** und in ***Teil II*** gefunden werden.

14 Installationsanweisungen

Dieses Dokument Kapitel beschreibt die Installation von PostgreSQL von der Quellcodedistribution.

14.1 Kurzversion

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

Die lange Version ist der Rest dieses Dokuments. Kapitels.

14.2 Voraussetzungen

Generell sollte PostgreSQL auf allen modernen Unix-kompatiblen Plattformen laufen können. Die Plattformen, auf denen zum Zeitpunkt der Veröffentlichung getestet wurde, sind unten in Abschnitt 14.7 aufgezählt. Im Unterverzeichnis doc in der Distribution gibt es mehrere plattformspezifische FAQ-Dokumente, in denen Sie nachschauen sollten, wenn es Probleme gibt.

Die folgenden Softwarepakete sind erforderlich, um PostgreSQL zu bauen:

- GNU `make` ist erforderlich; andere `make`-Programme funktionieren *nicht*. GNU `make` ist oft unter dem Namen `gmake` installiert; dieses Dokument wird diesen Namen durchweg verwenden. (Auf einigen Systemen ist GNU `make` das Standardwerkzeug mit dem Namen `make`.) Um zu überprüfen, ob Sie GNU `make` haben, geben Sie ein:

  ```
  gmake --version
  ```

  Es wird empfohlen, mindestens Version 3.76.1 zu verwenden.

- Sie benötigen einen ISO/ANSI-C-Compiler. Neuere Versionen von GCC sind empfehlenswert, aber PostgreSQL kann mit vielen Compilern von verschiedenen Herstellern übersetzt werden.
- `gzip` wird benötigt, um die Distribution zu entpacken.
- Die GNU Readline-Bibliothek (zur bequemen Bearbeitung von Eingabezeilen) wird in der Voreinstellung verwendet. Wenn Sie sie nicht benutzen wollen, dann müssen Sie die Option `--without-readline` angeben, wenn Sie `configure` ausführen. (Auf NetBSD ist die Bibliothek `libedit` mit Readline kompatibel und wird verwendet, wenn `libreadline` gefunden wird.)
- Um den Quellcode auf Windows NT oder Windows 2000 zu übersetzen, benötigen Sie die Pakete Cygwin und cygipc. Einzelheiten finden Sie in der Datei `doc/FAQ_MSWIN`.

Die folgenden Pakete können wahlweise verwendet werden. Sie sind in der Standardeinstellung nicht erforderlich, aber Sie sind nötig, wenn bestimmte Optionen eingeschaltet werden, wie unten erklärt wird.

- Um die Serverprogrammiersprache PL/Perl zu bauen, benötigen Sie eine volle Perl-Installation, einschließlich der `libperl`-Bibliothek und der Headerdateien. Da PL/Perl eine dynamische Bibliothek sein wird, muss `libperl` auf den meisten Plattformen auch eine dynamische Bibliothek sein. Das scheint in den neueren Perl-Versionen die Voreinstellung zu sein, aber es war nicht so in früheren Versionen, und generell ist es die Wahl desjenigen, der Perl bei Ihnen installiert hat.

  Wenn Sie die dynamische Bibliothek nicht haben, aber eine brauchen, taucht wahrscheinlich während des Übersetzungsprozesses eine Meldung wie die Folgende auf, um Sie darauf hinzuweisen:

  ```
  *** Cannot build PL/Perl because libperl is not a shared library.
  *** You might have to rebuild your Perl installation.  Refer to
  *** the documentation for details.
  ```

  (Wenn Sie die Ausgaben auf dem Bildschirm nicht verfolgen, werden Sie nur feststellen, dass das Bibliotheksobjekt für PL/Perl, `plperl.so` oder ähnlich, nicht installiert sein wird.) Wenn Sie dies sehen, müssen Sie Perl neu übersetzen und installieren, um PL/Perl bauen zu können. Während des Konfigurationsvorgangs von Perl verlangen Sie dann eine dynamische Bibliothek (*shared library*).

- Um die Python-Schnittstelle oder die Serverprogrammiersprache PL/Python bauen zu können, benötigen Sie eine Python-Installation, einschließlich der Headerdateien. Da PL/Python eine dynamische Bibliothek sein wird, muss `libpython` auf den meisten Plattformen auch eine dynamische Bibliothek sein. Dies ist bei einer Standardinstallation von Python nicht der Fall.

  Wenn Sie nach dem Übersetzen und Installieren eine Datei namens `plpython.so` (möglicherweise eine andere Erweiterung) haben, dann ist alles in Ordnung. Ansonsten sollten Sie eine Hinweismeldung ähnlich dieser vorbeilaufen gesehen haben:

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  Das bedeutet, dass Sie Ihre Python-Installation (oder einen Teil davon) neu bauen müssen, um diese dynamische Bibliothek zur Verfügung zu stellen.

Der Haken ist, dass die Python-Distribution oder die Python-Entwickler dafür keine direkte Möglichkeit vorgesehen haben. Das beste, was wir Ihnen anbieten können, sind die Informationen in Python FAQ 3.30. Auf einigen Betriebssystemen müssen Sie eigentlich keine dynamische Bibliothek bauen, aber dann müssen Sie das Buildsystem von PostgreSQL davon überzeugen. Schauen Sie sich die `Makefile` im Verzeichnis `src/pl/plpython` wegen Einzelheiten an.

- Wenn Sie Tcl- oder Tk-Komponenten (Clients oder die PL/Tcl-Sprache) bauen wollen, benötigen Sie natürlich eine Tcl-Installation.
- Um den JDBC-Treiber zu bauen, benötigen Sie `Ant` 1.5 oder höher und ein JDK. `Ant` ist ein spezielles Werkzeug für das Bauen von Java-basierenden Paketen. Sie können es von der `Ant`-Website herunterladen.

  Wenn Sie mehrere Java-Compiler installiert haben, hängt es von der Ant-Konfiguration ab, welcher verwendet wird. Vorkompilierte `Ant`-Distributionen sind typischerweise so eingerichtet, dass sie eine Datei `.antrc` im Home-Verzeichnis des aktuellen Benutzers lesen, um Konfigurationseinstellungen vorzunehmen. Um zum Beispiel eine andere JDK als die voreingestellte zu verwenden, könnte dies hier funktionieren:

```
JAVA_HOME=/usr/local/sun-jdk1.3
JAVACMD=$JAVA_HOME/bin/java
```

Anmerkung

Versuchen Sie nicht, den Treiber direkt durch das Aufrufen von `ant` oder gar `javac` zu bauen. Das wird nicht funktionieren. Führen Sie `gmake` normal wie unten beschrieben aus.

- Um *Native Language Support* (NLS), das heißt, die Fähigkeit, die Programmmitteilungen in einer anderen Sprache als Englisch anzuzeigen, zu aktivieren, benötigen Sie eine Implementierung der `Gettext`-Schnittstelle. Einige Betriebssysteme haben dies eingebaut (z.B. `Linux`, `NetBSD`, `Solaris`), für andere Systeme können Sie sich hier ein Zusatzpaket herunterladen: . Wenn Sie die `Gettext`-Implementierung aus der GNU C-Bibliothek verwenden, benötigen Sie außerdem das Paket GNU Gettext für einige Hilfsprogramme. Bei den anderen Implementierungen brauchen Sie es nicht.
- `Kerberos`, `OpenSSL` oder `PAM`, wenn Sie Authentifizierung mit diesen Diensten unterstützen wollen.

Wenn Sie mit einem CVS-Baum anstatt der veröffentlichten Quellpakete arbeiten oder selbst Entwicklungen tätigen wollen, benötigen Sie außerdem die folgenden Pakete:

- `Flex` und `Bison` werden benötigt, um einen CVS-Checkout zu bauen oder wenn Sie die eigentlichen Definitionsdateien von Scanner und Parser verändert haben. Wenn Sie sie benötigen, dann sollten Sie mindestens `Flex` 2.5.4 und mindestens `Bison` 1.50 verwenden. Andere `Yacc`-Programme können manchmal verwendet werden, aber das erfordert zusätzliche Anstrengungen und ist nicht zu empfehlen. Andere `Lex`-Programme funktionieren definitiv nicht.

Wenn Sie ein GNU-Paket benötigen, können Sie es auf Ihrer örtlichen GNU Mirrorsite (siehe für eine Liste) oder auf finden.

Überprüfen Sie auch, dass sie ausreichend Platz auf der Festplatte haben. Sie brauchen etwa 65 MB im Quellbaum während der Übersetzung und etwa 15 MB im Installationsverzeichnis. Ein leerer Datenbankcluster belegt etwa 25 MB, Datenbanken belegen etwa fünfmal so viel Platz, wie eine einfache Textdatei mit denselben Daten belegen würde. Wenn Sie die Regressionstests durchführen wollen, dann werden Sie vorübergehen weitere 90 MB benötigen. Verwenden Sie den Befehl `df`, um den Festplattenplatz zu überprüfen.

14.3 Die Quellen erhalten

Die Quellen für PostgreSQL kann man durch anonymes FTP von erhalten. Verwenden Sie, wenn möglich, einen Mirror. Wenn Sie die Datei erhalten haben, entpacken Sie sie:

```
gunzip postgresql-.tar.gz
tar xf postgresql-.tar
```

Dadurch wird ein Verzeichnis `postgresql-` unter dem aktuellen Verzeichnis erzeugt, welches die PostgreSQL-Quellen enthält. Wechseln Sie für den Rest des Installationsvorgangs in dieses Verzeichnis.

14.4 Wenn Sie eine alte Version aktualisieren

Das interne Datenformat ändert sich bei jeder Hauptversion von PostgreSQL. Wenn Sie daher eine bestehende Installation, die nicht die Versionsnummer ".x" hat, aktualisieren wollen, müssen Sie Ihre Daten wie hier gezeigt sichern und wiederherstellen. Diese Anweisungen gehen davon aus, dass Ihre bestehende Installation unter dem Verzeichnis `/usr/local/pgsql` ist und das Datenverzeichnis `/usr/local/pgsql/data` ist. Ersetzen Sie Ihre Pfade entsprechend.

1. Versichern Sie sich, dass Ihre Datenbank während oder nach dem Sicherungsvorgang nicht verändert wird. Die Integrität der gesicherten Daten wird dadurch nicht beeinträchtigt, aber die veränderten Daten würden natürlich nicht enthalten sein. Bearbeiten Sie, wenn nötig, die Zugriffskontrolldatei `/usr/local/pgsql/data/pg_hba.conf` (oder das Äquivalent) um allen außer Ihnen den Zugriff zu verbieten.
2. Um Ihre Datenbankinstallation zu sichern, geben Sie ein:

   ```
   pg_dumpall > ausgabedatei
   ```

   Wenn Sie die OIDs erhalten müssen (wenn Sie sie zum Beispiel als Fremdschlüssel verwenden), müssen Sie die Option `-o` von `pg_dumpall` verwenden.

   `pg_dumpall` sichert keine Large Objects. Schauen Sie in der Dokumentation in Abschnitt 22.1.4 nach, wenn Sie das tun müssen.

   Für die Datensicherung können Sie den `pg_dumpall`-Befehl von der Version nehmen, die Sie aktuell verwenden. Für die besten Ergebnisse sollten Sie aber versuchen, `pg_dumpall` von PostgreSQL zu verwenden, da diese Version Berichtigungen und Verbesserungen gegenüber älteren Versionen enthält. Obwohl dieser Rat eigenartig zu sein scheint, weil Sie die neue Version ja noch nicht installiert haben, ist er aber zu empfehlen, wenn Sie die neue Version parallel zur alten installieren wollen. In dem Fall können Sie die Installation normal durchführen und die Daten später übertragen. Dadurch senken Sie auch die Ausfallzeit.
3. Wenn Sie die neue Version am gleichen Platz wie die alte installieren, dann fahren Sie den alten Server, spätestens bevor Sie die neuen Dateien installieren, herunter:

   ```
   kill -INT `cat /usr/local/pgsql/data/postmaster.pid`
   ```

   Versionen vor 7.0 haben diese Datei `postmaster.pid` nicht. Wenn Sie so eine Version verwenden, müssen Sie die Prozess-ID des Servers selbst herausfinden, zum Beispiel indem Sie `ps ax | grep postmaster` eingeben und sie an den Befehl `kill` übergeben.

   Auf Systemen, die PostgreSQL beim Booten starten, gibt es wahrscheinlich eine Startdatei, die auch dazu verwendet werden kann. Auf einem `Red-Hat-Linux`-System könnte zum Beispiel

```
/etc/rc.d/init.d/postgresql stop
```

funktionieren. Eine andere Möglichkeit ist `pg_ctl stop`.

4. Wenn Sie auf dem gleichen Platz wie die alte Version installieren, ist es auch zu empfehlen, die alte Installation aus dem Weg zu verschieben, falls Sie Probleme haben und die Installation rückgängig machen müssen. Verwenden Sie einen Befehl wie diesen:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

Nachdem Sie PostgreSQL installiert haben, erzeugen Sie ein neues Datenbankverzeichnis und starten den Server. Denken Sie daran, dass Sie diese Befehle ausführen müssen, während Sie mit dem speziellen Datenbankbenutzerzugang angemeldet sind (den Sie schon haben, wenn Sie bereits eine Installation haben).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
```

Schließlich stellen Sie Ihre Daten wieder her, indem Sie Folgendes ausführen:

```
/usr/local/pgsql/bin/psql -d template1 -f ausgabedatei
```

Verwenden Sie dazu das *neue* `psql`.

Diese Themen werden in der Dokumentation in Abschnitt 22.3 in weiteren Einzelheiten besprochen.

14.5 Installationsvorgang

1. Konfiguration

   Der erste Schritt des Installationsvorgangs ist es, den Quellbaum für Ihr System zu konfigurieren und die gewünschten Optionen auszuwählen. Dazu führen Sie das Skript `configure` aus. Um eine Installation mit allen Vorgabeeinstellungen durchzuführen, geben Sie

   ```
   ./configure
   ```

   ein.

   Dieses Skript führt eine Reihe von Tests aus, um Werte für verschiedene systemabhängige Variablen zu erraten und einige Macken Ihres Betriebssystems zu entdecken, und erzeugt abschließend einige Dateien unter dem aktuellen Verzeichnis, um aufzuzeichnen, was es gefunden hat. (Sie können `configure` auch in einem Verzeichnis außerhalb des Quellbaums ausführen, um das Buildverzeichnis getrennt zu halten.)

   Die voreingestellte Konfiguration baut den Server und Hilfsprogramme sowie alle Clientanwendungen und -schnittstellen, die nur einen C-Compiler erfordern. Alle Dateien werden in der Voreinstellung unter `/usr/local/pgsql` installiert.

   Sie können den Build- und Installationsprozess anpassen, indem Sie eine oder mehrere der folgenden Kommandozeilenoptionen für `configure` angeben:

   `--prefix=PRÄFIX`

   > Installiert alle Dateien unter dem Verzeichnis *PRÄFIX* anstelle von `/usr/local/pgsql`. Die eigentlichen Dateien werden in verschiedene Unterverzeichnisse installiert; keine Datei wird jemals direkt im Verzeichnis *PRÄFIX* installiert.

Wenn Sie besondere Anforderungen haben, können Sie auch die einzelnen Unterverzeichnisse mit den folgenden Optionen anpassen.

`--exec-prefix=EXEC-PRÄFIX`

Sie können die architekturabhängigen Dateien unter einem anderen Präfix, *EXEC-PRÄFIX*, installieren als *PRÄFIX*. Das kann nützlich sein, um architekturunabhängige Dateien zwischen mehreren Maschinen zu teilen. Wenn Sie diese Option weglassen, wird *EXEC-PRÄFIX* gleich mit *PRÄFIX* gesetzt und architekturabhängige und unabhängige Dateien werden unter demselben Baum installiert, was wahrscheinlich das ist, was Sie wollen.

`--bindir=`*`VERZEICHNIS`*

Gibt das Verzeichnis für ausführbare Programme an. Die Voreinstellung ist `EXEC-PRÄFIX/bin`, was normalerweise `/usr/local/pgsql/bin` bedeutet.

`--datadir=`*`VERZEICHNIS`*

Setzt das Verzeichnis für nur lesbare Datendateien, die von den installierten Programmen verwendet werden. Die Voreinstellung ist `PRÄFIX/share`. Beachten Sie, dass dies nichts damit zu tun hat, wo die Datenbankdateien abgelegt werden sollen.

`--sysconfdir=`*`VERZEICHNIS`*

Das Verzeichnis für diverse Konfigurationsdateien, in der Voreinstellung `PRÄFIX/etc`.

`--libdir=`*`VERZEICHNIS`*

Der Platz, um Bibliotheken und dynamisch ladbare Module zu installieren. Die Voreinstellung ist `EXEC-PRÄFIX/lib`.

`--includedir=`*`VERZEICHNIS`*

Das Verzeichnis für die Installation von Headerdateien für C und C++. Die Voreinstellung ist `PRÄFIX/include`.

`--docdir=`*`VERZEICHNIS`*

Dokumentationsdateien, außer “man”-Seiten, werden in dieses Verzeichnis installiert. Die Voreinstellung ist `PRÄFIX/doc`.

`--mandir=`*`VERZEICHNIS`*

Die `man`-Seiten, die mit PostgreSQL geliefert werden, werden unter diesem Verzeichnis, in ihre entsprechenden `manx`-Unterverzeichnisse, installiert. Die Voreinstellung ist `PREFIX/man`.

Anmerkung

Es wurde Sorge getragen, dass PostgreSQL in geteilte Installationsverzeichnisse (wie `/usr/local/include`) installiert werden kann, ohne den Namensraum des restlichen Systems zu stören. Erstens wird die Zeichenkette “`/postgresql`” automatisch an `datadir`, `sysconfdir` und `docdir` angehängt, wenn der voll ausgewertete Verzeichnisname nicht schon die Zeichenkette “`postgres`” oder “`pgsql`” enthält. Zum Beispiel, wenn Sie `/usr/local` als Präfix wählen, wird die Dokumentation in `/usr/local/doc/postgresql` installiert, aber wenn der Präfix `/opt/postgres` ist, wird sie in `/opt/postgres/doc` installiert. Die öffentlichen C-Headerdateien der Clientschnittstellen werden in `includedir` installiert und respektieren den Namensraum des Systems. Die internen Headerdateien und die Server-Headerdateien werde in private Verzeichnisse unter `includedir` installiert. In der Anleitung zu jeder Schnittstelle finden Sie Informationen, wie Sie die jeweils benötigten Headerdateien finden. Schließlich wird auch, wenn angebracht, ein privates Unterverzeichnis unter `libdir` für dynamisch ladbare Module erzeugt.

`--with-includes=`*`VERZEICHNISSE`*

VERZEICHNISSE ist eine Liste von Verzeichnissen, durch Doppelpunkte getrennt, die der Liste, die der Compiler nach Headerdateien durchsucht, hinzugefügt werden. Wenn Sie optio-

nale Pakete (wie GNU `Readline`) an nicht standardisierten Orten installiert haben, müssen Sie diese Option verwenden, und wahrscheinlich auch die entsprechende Option mit `--with-libraries`.

Beispiel: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`VERZEICHNISSE`*

VERZEICHNISSE ist eine Liste von Verzeichnissen, durch Doppelpunkte getrennt, die die nach Bibliotheken durchsucht werden sollen. Sie werden diese Option (und die entsprechenden Option mit `--with-includes`) wahrscheinlich verwenden müssen, wenn Sie Pakete an nicht standardisierten Orten installiert haben.

Beispiel: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-recode`

Schaltet Einzelbyte-Zeichensatzrekodierungs-Unterstützung ein. Weitere Informationen über diese Fähigkeit finden Sie in der Dokumentationin Abschnitt 20.3. Beachten Sie, dass eine allgemeinere Form von Zeichensatzkonvertierung in der Standardkonfiguration verfügbar ist; dieses Feature ist obsolet.

`--enable-nls`[=*SPRACHEN*]

Schaltet *Native Language Support* (NLS) ein, das heißt, die Fähigkeit, die Programmmitteilungen in einer anderen Sprache als Englisch anzuzeigen. *SPRACHEN* ist eine Liste von Kürzeln, durch Leerzeichen getrennt, für die Sprachen, die Sie unterstützt haben wollen, zum Beispiel `--enable-nls='de fr'`. (Die Schnittmenge zwischen Ihrer Liste und den tatsächlich vorhandenen Übersetzungen wird automatisch berechnet.) Wenn Sie keine Liste angeben, werden alle vorhandenen Übersetzungen installiert.

Um diese Option zu verwenden, benötigen Sie eine Implementierung der `Gettext` API; siehe oben.

`--with-pgport=NUMMER`

Bestimmt *NUMMER* als die voreingestellte Portnummer für Server und Clients. Die Standardeinstellung ist 5432. Der Port kann immer noch später geändert werden, aber wenn Sie ihn hier angeben, werden Server und Clients die gleiche Voreinstellung eingebaut bekommen, was sehr praktisch sein kann. Der einzige gute Grund, um einen anderen Wert als den Standard zu wählen, ist normalerweise, wenn Sie mehrere PostgreSQL-Server auf einer Maschine laufen haben wollen.

`--with-perl`

Baut die serverseitige Sprache PL/Perl.

`--with-python`

Baut das Python-Schnittstellenmodul und die serverseitige Sprache PL/Python. Sie müssen `root`-Zugang haben, um das Python-Modul an seinem normalen Platz (`/usr/lib/pythonx.y`) installieren zu können.

`--with-tcl`

Baut Komponenten, die Tcl/Tk erfordern, das heißt `libpgtcl`, `pgtclsh`, `pgtksh` und `PL/Tcl`. Siehe aber unten über `--without-tk`.

`--without-tk`

Wenn Sie `--with-tcl` und diese Option angeben, wird das Programm, das Tk benötigt (`pgtksh`), nicht mit installiert.

`--with-tclconfig=`*`VERZEICHNIS`*,

`--with-tkconfig=`*`VERZEICHNIS`*

Tcl/Tk installiert die Dateien `tclConfig.sh` und `tkConfig.sh`, welche Konfigurationsinformationen enthalten, die benötigt werden, um Tcl- oder Tk-Module zu bauen. Diese Dateien werden normalerweise automatisch an den bekannten Orten gefunden, aber wenn Sie eine andere Version von Tcl oder Tk verwenden wollen, können Sie die Verzeichnisse angeben, wo sie zu finden sind.

`--with-java`

Baut den JDBC-Treiber und zugehörige Java-Pakete.

`--with-krb4`[=*VERZEICHNIS*]

`--with-krb5`[=*VERZEICHNIS*]

Baut mit Unterstützung für Authentifizierung mit Kerberos. Sie können Kerberos Version 4 oder 5 verwenden, aber nicht beide. Das Argument *VERZEICHNIS* gibt das Wurzelverzeichnis der Kerberos-Installation an; `/usr/athena` ist die Voreinstellung. Wenn die relevanten Headerdateien und Bibliotheken nicht unter einem gemeinsamen Verzeichnis installiert sind, müssen Sie zusätzlich zu dieser Option die Optionen `--with-includes` und `--with-libraries` verwenden. Wenn andererseits die erforderlichen Dateien an einem Ort liegen, der automatisch durchsucht wird (z.B. `/usr/lib`), können Sie das Argument weglassen.

`configure` überprüft die erforderlichen Headerdateien und Bibliotheken, um sicherzustellen, dass Ihre Kerberos-Installation ausreichend ist.

`--with-krb-srvnam=NAME`

Der Name des Service-Principals für Kerberos. `postgres` ist der Standard. Es gibt wahrscheinlich keinen Grund, das zu ändern.

`--with-openssl`[=*VERZEICHNIS*]

Baut mit Unterstützung für (verschlüsselte) SSL-Verbindungen. Dafür muss das Paket OpenSSL installiert sein. Das Argument *VERZEICHNIS* gibt das Wurzelverzeichnis der OpenSSL-Installation an; der Standardwert ist `/usr/local/ssl`.

`configure` prüft die erforderlichen Headerdateien und Bibliotheken, um sicherzustellen, dass Ihre OpenSSL-Installation ausreichend ist.

`--with-pam`

Baut mit Unterstützung für PAM (*Pluggable Authentication Modules*).

`--without-readline`

Verhindert die Verwendung der `Readline`-Bibliothek. Das schaltet Kommandozeilenbearbeitung und -geschichte in `psql` ab, ist also nicht zu empfehlen.

`--without-zlib`

Verhindert die Verwendung der `Zlib`-Bibliothek. Das schaltet Komprimierungsunterstützung in `pg_dump` ab. Diese Option ist nur für jene seltenen Systeme gedacht, wo diese Bibliothek nicht verfügbar ist.

`--enable-debug`

Kompiliert alle Programme und Bibliotheken mit Debugsymbolen. Das bedeutet, dass Sie Ihre Programme durch einen Debugger schicken können, um Probleme zu analysieren. Das vergrößert die installierten Dateien beträchtlich, und bei Compilern außer GCC schaltet es gewöhnlich die Optimierung ab und verlangsamt dadurch die Ausführung. Die Debugsymbole sind allerdings äußerst hilfreich, um etwaige Probleme zu behandeln. Gegenwärtig wird diese Option für Produktionsinstallationen nur empfohlen, wenn Sie GCC verwenden. Aber wenn Sie Betaversionen verwenden oder selbst Entwicklungsarbeit tun, dann sollten Sie sie immer an haben.

`--enable-cassert`

Schaltet **Assertion**-Prüfungen im Server ein, welche Bedingungen testen, die eigentlich nicht auftreten können sollten. Das ist unschätzbar bei der Codeentwicklung, aber die Tests verlangsamen alles ein wenig. Durch die Tests wird auch die Stabilität des Servers nicht unbedingt verbessert! Die Tests sind nicht nach Schwere kategorisiert, also werden selbst relativ harmlose Fehler einen Serverneustart verursachen, wenn eine Assertion-Prüfung fehlschlägt. Gegenwärtig ist diese Option nicht für Produktionsanwendungen empfohlen, aber Sie sollten sie an haben, wenn Sie selbst entwickeln oder eine Betaversion verwenden.

`--enable-depend`

Schaltet die automatische Verfolgung von Dateiabhängigkeiten ein. Mit dieser Option werden die Makefiles so eingerichtet, dass alle betroffenen Objektdateien neu kompiliert werden, wenn eine Headerdatei geändert wird. Das ist bei der Entwicklungsarbeit nützlich, aber nur unnötiger Ballast, wenn Sie nur einmal kompilieren und installieren wollen. Zurzeit funktioniert diese Option nur mit GCC.

Wenn Sie einen anderen C-Compiler als den, den `configure` auswählt, bevorzugen, können Sie die Umgebungsvariable `CC` auf das Programm Ihrer Wahl setzen. Wenn nichts anderes angegeben wird, wählt `configure` `gcc`, außer wenn das für die Plattform unpassend ist. Ähnlich können Sie die Compileroptionen mit der Variable `CFLAGS` setzen.

Sie können Umgebungsvariablen auf der Kommandozeile von `configure` angeben, zum Beispiel:

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

2. Übersetzung
   Um die Übersetzung des Quelltextes zu starten, geben Sie

   ```
   gmake
   ```

   ein.

   (Denken Sie daran, GNU `make` zu verwenden.) Der Vorgang kann zwischen 5 Minuten und einer halben Stunde dauern, abhängig von Ihrer Hardware. Die letzte ausgegebene Zeile sollte sein:

   ```
   All of PostgreSQL is successfully made. Ready to install.
   ```

3. Regressionstests
   Wenn Sie den neu gebauten Server vor der Installation testen möchten, können Sie an diesem Punkt die Regressionstests ausführen. Die Regressionstests sind eine Testsammlung, die überprüfen, ob PostgreSQL auf Ihrer Maschine so funktioniert, wie die Entwickler es beabsichtigt hatten. Geben Sie

   ```
   gmake check
   ```

   ein.

   (Das funktioniert nicht als `root`; machen Sie es als unprivilegierter Benutzer.) Es kann vorkommen, dass einige Tests wegen unterschiedlichem Wortlaut von Fehlermeldungen oder Problemen mit Fließkommaberechnungen nicht erfolgreich sind. Die Datei `src/test/regress/README` und die Dokumentation enthalten Kapitel 26 enthält detaillierte Informationen, wie man die Testergebnisse zu verstehen hat. Sie können diesen Test auch später mit dem gleichen Befehl wiederholen.

4. Die Dateien installieren

   Um PostgreSQL zu installieren, geben Sie

   ```
   gmake install
   ```

   ein.

Anmerkung

Wenn Sie eine bestehende Installation aktualisieren und die neuen Dateien über die alten installieren, sollten Sie spätestens jetzt Ihre Daten gesichert und den alten Server heruntergefahren haben, wie oben in Abschnitt 14.4 beschrieben.

Dadurch werden die Dateien in die in Schritt 1. angegebenen Verzeichnisse installiert. Achten Sie darauf, dass Sie entsprechende Zugriffsrechte haben, um in diesen Bereich zu schreiben. Normalerweise müssen Sie diesen Schritt als `root` machen. Alternativ könnten Sie die Zielverzeichnisse vorher erzeugen und die passenden Rechte dafür erteilen.

Sie können `gmake install-strip` anstelle von `gmake install` verwenden, um die Programmdateien und Bibliotheken zu "strippen", während sie installiert werden. Das spart etwas Platz. Wenn Sie mit Debugunterstützung kompiliert haben, entfernt das Strippen faktisch die Debugunterstützung, sollte also nur durchgeführt werden, wenn das Debuggen nicht mehr benötigt wird. `install-strip` versucht, ein vernünftiges Maß an Platz zu sparen, aber es hat nicht das perfekte Wissen, um jedes unnötige Byte aus einer Datei zu entfernen. Wenn Sie also so viel Festplattenplatz wie möglich sparen wollen, dann müssen Sie von Hand nacharbeiten.

Wenn Sie die Python-Schnittstelle gebaut haben und Sie nicht der `root`-Benutzer waren, als Sie den obigen Befehl ausgeführt haben, ist der Teil der Installation wahrscheinlich fehlgeschlagen. In diesem Fall sollten Sie sich als `root` anmelden und dann

```
gmake -C src/interfaces/python install
```

ausführen. Wenn Sie keinen `root`-Zugang haben, dann sind Sie auf sich allein gestellt: Sie können die benötigten Dateien in anderen Verzeichnissen ablegen, wo Python sie finden kann, aber wie das zu tun ist, ist als Hausaufgabe belassen.

Die Standardinstallation enthält nur die Headerdateien für die Entwicklung von Clientanwendungen. Wenn Sie serverseitige Programmentwicklung planen (wie zum Beispiel eigene in C geschriebene Funktionen oder Datentypen), möchten Sie vielleicht komplett alle PostgreSQL-Headerdateien in das Zielverzeichnis installieren. Um das zu tun, geben Sie

```
gmake install-all-headers
```

ein.

Das vergrößert die Installation um ein oder zwei Megabyte und ist nur sinnvoll, wenn Sie nicht den Quellcodebaum behalten wollen. (Wenn ja, können Sie einfach die Headerdateien aus dem Quellbaum nehmen, wenn Sie Serversoftware bauen.)

Clientinstallation: Wenn Sie nur die Clientanwendungen und Schnittstellenbibliotheken installieren wollen, können Sie diese Befehle verwenden:

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
gmake -C doc install
```

Deinstallation: Sie können die Installation mit dem Befehl `gmake uninstall` rückgängig machen. Dadurch werden jedoch keine Verzeichnisse entfernt.

Säubern: Nach der Installation können Sie Platz schaffen, indem Sie die gebauten Dateien aus dem Quellbaum mit dem Befehl `gmake clean` entfernen. Dadurch behalten Sie die von `configure` erzeugten Dateien, sodass Sie später alles mit `gmake` neu bauen können. Um den Quellbaum in den Zustand zurückzusetzen, in dem Sie ihn erhalten hatten, verwenden Sie `gmake distclean`. Wenn Sie für mehrere Plattformen vom selben Quellbaum aus bauen wollen, müssen Sie das für jede Plattform tun und neu konfigurieren.

Wenn Sie einen Build durchgeführt haben und feststellen, dass die `configure`-Optionen falsch waren, oder wenn Sie etwas geändert haben, das von `configure` untersucht wird (zum Beispiel neue Software), dann ist es zu empfehlen, vor der Neukonfiguration `gmake distclean` auszuführen und dann neu zu bauen. Ansonsten kann es sein, dass Ihre Änderungen bei den Konfigurationsoptionen nicht überall ankommen.

14.6 Einrichtung nach der Installation

14.6.1 Dynamische Bibliotheken

Auf einigen Systemen, die dynamische Bibliotheken haben (was die meisten Systeme tun), müssen Sie Ihrem System mitteilen, wie es die neu installierten dynamischen Bibliotheken finden kann. Auf folgenden Systemen ist dies *nicht* notwendig: `BSD/OS`, `FreeBSD`, `HP-UX`, `IRIX`, `Linux`, `NetBSD`, `OpenBSD`, `Tru64 UNIX` (ehemals `Digital UNIX`) und `Solaris`.

Die Methoden, um den Suchpfad für dynamische Bibliotheken zu setzen, unterscheiden sich von Plattform zu Plattform, aber die am häufigsten brauchbare Methode ist es, die Umgebungsvariable `LD_LIBRARY_PATH` so zu setzen: in Bourne-Shells (`sh`, `ksh`, `bash`, `zsh`)

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

oder in `csh` oder `tcsh`

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Ersetzen Sie `/usr/local/pgsql/lib` mit dem, auf das Sie `--libdir` in Schritt 1. gesetzt haben. Sie sollten diese Befehle in eine Shell-Startdatei eintragen, wie zum Beispiel `/etc/profile` oder `~/.bash_profile`. Einige gute Informationen über mögliche Vorbehalte gegenüber dieser Methode können in gefunden werden.

Auf einigen Systemen mag es besser sein, die Umgebungsvariable `LD_RUN_PATH` *vor* der Compilierung zu setzen.

Auf `Cygwin` stellen Sie das Bibliotheksverzeichnis in den Pfad (`PATH`) oder verschieben die `.dll`-Dateien in das `bin`-Verzeichnis.

Wenn Sie sich nicht sicher sind, dann schauen Sie sich die Anleitung Ihres Systems an. (Die Manualseiten von `ld.so` oder `rld` könnten von besonderem Interesse sein). Wenn Sie später eine Mitteilung sehen wie

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

war dieser Schritt notwendig. Erledigen Sie ihn dann einfach später.

Wenn Sie auf `BSD/OS`, `Linux` oder `SunOS 4` sind und `root`-Zugang haben, dann können Sie

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(oder entsprechendes Verzeichnis) nach der Installation ausführen, um dem Laufzeitlinker das schnellere Finden der dynamischen Bibliotheken zu ermöglichen. Weitere Informationen finden Sie in der Anleitung zu `ldconfig`. Auf `FreeBSD`, `NetBSD` und `OpenBSD` ist der Befehl stattdessen

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Auf anderen Systemen ist ein entsprechender Befehl nicht bekannt.

14.6.2 Umgebungsvariablen

Wenn Sie in `/usr/local/pgsql` installiert haben oder an einem anderen Ort, der normalerweise nicht nach Programmen durchsucht wird, dann sollten Sie `/usr/local/pgsql/bin` (oder auf was Sie `--bindir` in Schritt 1. gesetzt haben) in Ihren Suchpfad (`PATH`) einfügen. Streng genommen ist das nicht notwendig, aber es macht die Verwendung von PostgreSQL viel bequemer.

Fügen Sie also Folgendes in eine Ihrer Shell-Startdateien, wie `~/.bash_profile` (oder `/etc/profile`, wenn es auf alle Benutzer Auswirkung haben soll), ein:

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

Wenn Sie `csh` oder `tcsh` verwenden, nehmen Sie diesen Befehl:

```
set path = ( /usr/local/pgsql/bin $path )
```

Um Ihrem System zu ermöglichen, die Dokumentation im `man`-Format zu finden, müssen Sie Zeilen wie die folgenden in eine Shell-Startdatei einfügen, außer wenn Sie an einem Ort installiert haben, der in der Standardeinstellung schon durchsucht wird.

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

Die Umgebungsvariablen `PGHOST` und `PGPORT` geben Clientanwendungen den Host und Port des Datenbankservers an und übergehen damit die eingebauten Vorgaben. Wenn Sie Clientanwendungen auf entfernten Hosts laufen lassen wollen, ist es praktisch, wenn jeder Benutzer, der die Datenbank verwenden will, `PGHOST` setzt. Das ist nicht erforderlich, aber wenn Sie es nicht tun, müssen Sie die Einstellungen mit Kommandozeilenoptionen oder Ähnlichem vornehmen.

14.7 Unterstützte Plattformen

PostgreSQL wurde von der Entwicklergemeinde auf den unten gelisteten Plattformen auf korrektes Funktionieren überprüft. Eine unterstützte Plattform bedeutet dabei in der Regel, dass PostgreSQL entsprechend dieser Anleitung gebaut und installiert werden kann und dass die Regressionstests erfolgreich waren.

Anmerkung

Wenn Sie Probleme mit der Installation auf unterstützten Plattformen haben, schreiben Sie bitte an pgsql-bugs@postgresql.org oder pgsql-ports@postgresql.org, nicht an die hier aufgelisteten Leute.

Betriebssystem	Prozessor	Version	Bericht	Bemerkungen
AIX	RS6000	7.3	2002-11-12, Andreas Zeugswetter (ZeugswetterA@spardat.at	siehe auch `doc/FAQ_AIX`
BSD/OS	x86	7.3	2002-10-25, Bruce Momjian (pgman@candle.pha.pa.us)	4.2
FreeBSD	Alpha	7.3	2002-11-13, Chris Kings-Lynne (chriskl@familyhealth.com.au)	
FreeBSD	x86	7.3	2002-10-29, 3.3, Nigel J. Andrews (nandrews@investsystems.co.uk), 4.7, Larry Rosenman (ler@lerctr.org), 5.0, Sean Chittenden (sean@chittenden.org)	
HP-UX	PA-RISC	7.3	2002-10-28, 10.20 Tom Lane (tgl@sss.pgh.pa.us), 11.00, 11.11, 32 und 64 Bits, Giles Lean (giles@nemeton.com.au	`gcc` und `cc`; siehe auch `doc/FAQ_HPUX`
IRIX	MIPS	7.3	2002-10-27, Ian Barwick (barwick@gmx.net	Irix64 Komma 6.5
Linux	Alpha	7.3	2002-10-28, Magnus Naeslund (mag@fbab.net)	2.4.19-pre6
Linux	PlayStation 2	7.3	2002-11-19, Permaine Cheung pcheung@redhat.com)	`#undef HAS_TEST_AND_SET`, entferne `slock_t typedef`
Linux	PPC74xx	7.3	2002-10-26, Tom Lane (tgl@sss.pgh.pa.us)	2.2.18; Apple G3
Linux	S/390	7.3	2002-11-22, Permaine Cheung pcheung@redhat.com)	s390 und s390x (32 und 64 Bits)
Linux	Sparc	7.3	2002-10-26, Doug McNaught (doug@mcnaught.org)	3.0
Linux	x86	7.3	2002-10-26, Alvaro Herrera (alvherre@dcc.uchile.cl)	2.4
MacOS X	PPC	7.3	2002-10-28, 10.1, Tom Lane (tgl@sss.pgh.pa.us), 10.2.1, Adam Witney (awitney@sghms.ac.uk)	
NetBSD	arm32	7.3	2002-11-19, Patrick Welche (prlw1@newn.cam.ac.uk)	1.6
NetBSD	x86	7.3	2002-11-14, Patrick Welche (prlw1@newn.cam.ac.uk)	1.6
OpenBSD	Sparc	7.3	2002-11-17, Christopher Kings-Lynne (chriskl@familyhealth.com.au)	3.2
OpenBSD	x86	7.3	2002-11-14, 3.1 Magnus Naeslund (mag@fbab.net), 3.2 Christopher Kings-Lynne (chriskl@familyhealth.com.au)	
SCO OpenServer 5	x86	7.3.1	2002-12-11, Shibashish Satpathy (shib@postmark.net)	5.0.4, `gcc`; siehe auch `doc/FAQ_SCO`
Solaris	Sparc	7.3	2002-10-28, Andrew Sullivan (andrew@libertyrms.info)	Solaris 7 und 8; siehe auch `doc/FAQ_Solaris`

Betriebssystem	Prozessor	Version	Bericht	Bemerkungen
Solaris	x86	7.3	2002-11-20, Martin Renters (martin@datafax.com)	5.8; siehe auch `doc/FAQ_Solaris`
Tru64 UNIX	Alpha	7.3	2002-11-05, Alessio Bragadini (alessio@albourne.com)	
UnixWare	x86	7.3	2002-11-01, 7.1.3 Larry Rosenman (ler@lerctr.org), 7.1.1 und 7.1.2 (8.0.0) Olivier Prenant (ohp@pyrenet.fr)	siehe auch `doc/FAQ_SCO`
Windows	x86	7.3	2002-10-29, Dave Page (dpage@vale-housing.co.uk), Jason Tishler (jason@tishler.net)	mit `Cygwin`; siehe `doc/FAQ_MSWIN`
Windows	x86	7.3	2002-11-05, Dave Page (dpage@vale-housing.co.uk)	nativ nur clientseitig; siehe Dokumentation siehe Kapitel 15

Nicht unterstützte Plattformen:

Die folgenden Plattformen funktionieren entweder dem letzten Kenntnisstand nach nicht, oder sie funktionierten in einer früheren Version einmal und wir haben keine ausdrückliche Bestätigung eines erfolgreichen Tests mit Version zum Zeitpunkt, als diese Liste zusammengestellt wurde, erhalten. Wir erwähnen sie hier, um Ihnen zu zeigen, dass diese Plattformen unterstützt werden *könnten*, wenn sie etwas Aufmerksamkeit erfahren würden.

Betriebssystem	**Prozessor**	**Version**	**Bericht**	**Bemerkungen**
BeOS	x86	7.2	2001-11-29, Cyril Velter (cyril.velter@libertysurf.fr)	benötigt neuen Semaphorcode
DG/UX 5.4R4.11	m88k	6.3	1998-03-01, Brian E Gallew (geek+@cmu.edu)	keine neueren Berichte
Linux	armv4l	7.2	2001-12-10, Mark Knox (segfault@hardline.org)	2.2.x
Linux	MIPS	7.2	2001-11-15, Hisao Shibuya (shibuya@alpha.or.jp)	2.0.x; Cobalt Qube2
MkLinux DR1	PPC750	7.0	2001-04-03, Tatsuo Ishii (t-ishii@sra.co.jp)	7.1 braucht neue OS-Version?
NetBSD	Alpha	7.2	2001-11-20, Thomas Thai (tom@minnesota.com)	1.5W
NetBSD	m68k	7.0	2000-04-10, Henry B. Hotz (hotz@jpl.nasa.gov)	Mac 8xx
NetBSD	MIPS	7.2.1	2002-06-13, Warwick Hunter (whunter@agile.tv)	1.5.3

Betriebssystem	Prozessor	Version	Bericht	Bemerkungen
NetBSD	PPC	7.2	2001-11-28, Bill Studenmund (wrstuden@netbsd.org)	1.5
NetBSD	Sparc	7.2	2001-12-03, Matthew Green (mrg@eterna.com.au)	32 und 64 Bits
NetBSD	VAX	7.1	2001-03-30, Tom I. Helbekkmo (tih@kpnQwest.no)	1.5
NeXTSTEP	x86	6.x	1998-03-01, David Wetzel (dave@turbocat.de)	Bitrot vermutet
QNX 4 RTOS	x86	7.2	2001-12-10, Bernd Tegge (tegge@repas-aeg.de)	benötigt neuen Semaphorcode; siehe auch `doc/FAQ_QNX4`
QNX RTOS v6	x86	7.2	2001-11-20, Igor Kovalenko (Igor.Kovalenko@motorola.com)	Patches im Archiv verfügbar, aber zu spät für 7.2
SunOS 4	Sparc	7.2	2001-12-04, Tatsuo Ishii (t-ishii@sra.co.jp)	
System V R4	m88k	6.2.1	1998-03-01, Doug Winterburn (dlw@seavme.xroads.com)	benötigt neuen TAS-Spinlockcode
System V R4	MIPS	6.4	1998-10-28, Frank Ridderbusch (ridderbusch.pad@sni.de)	keine neueren Berichte
Ultrix	MIPS	7.1	2001-03-26	TAS-Spinlockcode wird nicht entdeckt
Ultrix	VAX	6.x	1998-03-01	

15 Installation auf Windows

Trotzdem PostgreSQL für Unix-ähnliche Betriebssystem geschrieben wurde, können die C-Clientbibliothek (`libpq`) und das interaktive Terminal (`psql`) als Windows-Anwendungen kompiliert werden. Die `make`-Steuerdateien in der Quellendistribution sind für Microsoft Visual C++ gedacht und funktionieren wahrscheinlich nicht mit anderen Systemen. Es sollte aber möglich sein, in anderen Fällen die Anwendungen von Hand zu kompilieren.

Tipp

Wenn Sie Windows 98 oder neuer verwenden, dann können Sie das gesamte PostgreSQL-Paket auf "Unix-Art" kompilieren und anwenden, wenn Sie zuerst das Cygwin-Toolkit installieren. In diesem Fall finden Sie die Installationsanweisungen in Kapitel 14.

Um alles, was auf Windows möglich ist, zu bauen, wechseln Sie in das Verzeichnis `src` und geben Sie diesen Befehl ein:

```
nmake /f win32.mak
```

Hierbei wird davon ausgegangen, dass Visual C++ in Ihrem Pfad zu finden ist.

Die folgenden Dateien werden gebaut werden:

`interfaces\libpq\Release\libpq.dll`

: Die dynamisch linkbare Clientbibliothek

`interfaces\libpq\Release\libpqdll.lib`

: Importbibliothek, um Ihre Programme mit `libpq.dll` linken zu können

`interfaces\libpq\Release\libpq.lib`

: Statische Version der Clientbibliothek

`bin\psql\Release\psql.exe`

: Das interaktive Terminal von PostgreSQL

Die einzige Datei, die wirklich installiert werden muss, ist die Bibliothek `libpq.dll`. Diese Datei sollte in den meisten Fällen im Verzeichnis `WINNT\SYSTEM32` (oder `WINDOWS\SYSTEM` auf Windows 95/98/ME) abgelegt werden. Wenn diese Datei mit einem Setup-Programm installiert wird, dann sollte sie mit Versionsprüfung über die in der Datei enthaltene Ressource *VERSIONINFO* installiert werden, damit eine neuere Version der Bibliothek nicht überschrieben wird.

Wenn Sie Anwendung mit `libpq` auf diesem Rechner entwickeln wollen, müssen Sie die Verzeichnisse `src\include` und `src\interfaces\libpq` im Quellbaum zum Suchpfad für Headerdateien Ihres Compilers hinzufügen.

Um die Bibliothek zu verwenden, müssen Sie die Datei `libpqdll.lib` Ihrem Projekt hinzufügen. (In Visual C++ rechtsklicken Sie einfach auf das Projekt und wählen Sie den Menüpunkt zum Hinzufügen aus.)

16 Laufzeitverhalten des Servers

Dieses Kapitel bespricht, wie der Datenbankserver eingerichtet wird, was passiert, wenn er läuft, und welche Wechselwirkungen mit dem Betriebssystem dadurch entstehen.

16.1 Der PostgreSQL-Benutzerzugang

Wie bei jedem Serverprogramm, das mit der Welt verbunden ist, ist es auch bei PostgreSQL empfehlenswert, den Server unter einem getrennten Benutzerzugang laufen zu lassen. Dieser Benutzerzugang sollte nur die vom Server verwalteten Daten besitzen und sollte nicht mit anderen Daemons geteilt werden. (Die Verwendung des Benutzers `nobody` wäre zum Beispiel keine gute Idee.) Es ist nicht zu empfehlen, die Programmdateien unter diesem Benutzerzugang zu installieren, weil ein kompromittiertes System dann seine eigenen Programmdateien verändern könnte.

Um einen Unix-Benutzerzugang in Ihrem System zu erzeugen, schauen Sie, ob Sie einen Befehl `useradd` oder `adduser` haben. Der Benutzername `postgres` wird oft verwendet, aber es besteht dahingehend keinerlei Notwendigkeit.

16.2 Einen Datenbankcluster erzeugen

Bevor Sie irgendetwas tun können, müssen Sie einen Bereich zur Speicherung der Datenbank auf der Festplatte initialisieren. Wir nennen das einen **Datenbankcluster**. (SQL verwendet anstelle dessen den Begriff Katalogcluster.) Ein Datenbankcluster ist eine Sammlung von Datenbanken, auf die über einen einzigen laufenden Server zugegriffen werden kann. Nach der Initialisierung enthält ein Datenbankcluster eine Datenbank namens `template1`. Diese Datenbank wird als Vorlage (englisch *template*) für später erzeugte Datenbanken verwendet; sie sollte nicht für die eigentliche Arbeit verwendet werden. (Informationen über die Erzeugung von Datenbanken finden Sie in Kapitel 18.)

Im Dateisystem ist ein Datenbankcluster ein einzelnes Verzeichnis, unter dem alle Daten gespeichert werden. Wir nennen das das **Datenverzeichnis** oder den **Datenbereich**. Es ist vollkommen Ihnen überlassen, wo Sie Ihre Daten speichern. Es gibt keine Voreinstellung, aber Stellen wie `/usr/local/pgsql/data` oder `/var/lib/pgsql/data` werden häufig verwendet. Um einen Datenbankcluster zu initialisie-

ren, verwenden Sie den Befehl `initdb`, welcher mit PostgreSQL installiert wird. Das gewünschte Verzeichnis für Ihr Datenbanksystem wird mit der Option `-D` angegeben, zum Beispiel

```
$ initdb -D /usr/local/pgsql/data
```

Beachten Sie, dass Sie mit dem im vorigen Abschnitt beschriebenen PostgreSQL-Benutzer angemeldet sein müssen, wenn Sie diesen Befehl ausführen.

Tipp

Als Alternative zu der Option `-D` können Sie die Umgebungsvariable `PGDATA` setzen.

Wenn das von Ihnen angegebene Verzeichnis noch nicht existiert, wird `initdb` versuchen, es zu erzeugen. Dazu wird es wahrscheinlich nicht die Rechte haben (wenn Sie unserem Rat gefolgt sind und einen unprivilegierten Benutzer geschaffen haben). In diesem Fall sollten Sie das Verzeichnis selbst erstellen (als `root`) und dann den PostgreSQL-Benutzer zum Eigentümer machen. So könnte man das anstellen:

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

`initdb` wird abbrechen, wenn es so aussieht, als wäre das Datenverzeichnis schon initialisiert worden.

Da das Datenverzeichnis alle in der Datenbank gespeicherten Daten enthält, ist es von äußerster Wichtigkeit, dass es vor unerlaubtem Zugriff geschützt wird. `initdb` entzieht daher jedem, außer dem PostgreSQL-Benutzer, die Zugriffsrechte für dieses Verzeichnis.

Während der Verzeichnisinhalt sicher ist, erlaubt die Voreinstellung der Clientauthentifizierung jedoch, dass jeder lokale Benutzer mit der Datenbank verbinden kann und sogar ein Superuser werden kann. Wenn Sie Ihren lokalen Benutzern nicht vertrauen, empfehlen wir Ihnen, dass Sie die Option `-W` oder `--pwprompt` in `initdb` verwenden, um dem Datenbank-Superuser ein Passwort zuzuweisen. Verändern Sie nach `initdb` die Datei `pg_hba.conf` und stellen Sie die Authentifizierungsmethode auf `md5` oder `password` anstelle von `trust`, *bevor* Sie den Server das erste Mal starten. (Weitere Möglichkeiten, um Verbindungen zu beschränken, sind die Verwendung der Methode `ident` oder die Verwendung der Dateisystemzugriffsrechte. Siehe Kapitel 19 für weitere Informationen.)

`initdb` stellt auch die Locale des Datenbankclusters ein. Normalerweise nimmt es einfach die Locale-Einstellung aus der Umgebung und wendet Sie auf die initialisierte Datenbank an. Es ist auch möglich, eine andere Locale für die Datenbank anzugeben; weitere Informationen dazu finden Sie in Abschnitt 20.1. Eine kleine Überraschung während der Ausführung von `initdb` könnte eine Meldung wie diese sein:

```
The database cluster will be initialized with locale de_DE.
This locale setting will prevent the use of indexes for pattern matching
operations.  If that is a concern, rerun initdb with the collation order
set to "C".  For more information see the Administrator's Guide.
```

Das soll Sie darauf hinweisen, dass die aktuell eingestellte Locale Indexe so sortiert, dass sie nicht für Suchen mit `LIKE` oder regulären Ausdrücken verwendet werden können. Wenn Sie für solche Suchen eine guten Leistung haben wollen, dann sollten Sie die aktuelle Locale auf `C` setzen und `initdb` noch einmal ausführen, z.B. mit `initdb --lc-collate=C`. Die in einem bestimmten Datenbankcluster verwendete Sortierreihenfolge wird von `initdb` festgesetzt und kann später nicht mehr verändert werden, außer wenn Sie alle Daten sichern (mit `pg_dump`), `initdb` nochmal ausführen, und dann alle Daten wiederherstellen. Es ist also wichtig, jetzt die richtige Wahl zu treffen.

16.3 Den Datenbankserver starten

Bevor jemand auf die Datenbank zugreifen kann, müssen Sie den Datenbankserver starten. Das Datenbankserverprogramm heißt `postmaster`. Der `postmaster` muss wissen, wo er die Daten finden soll, die er verwenden soll. Das wird mit der Option `-D` getan. Die einfachste Art, den Server zu starten, ist also:

```
$ postmaster -D /usr/local/pgsql/data
```

Dadurch bleibt der Server im Vordergrund. Diesen Befehl müssen Sie ausführen, während Sie mit dem PostgreSQL-Benutzerzugang angemeldet sind. Wenn die Option `-D` nicht angegeben wurde, dann wird der Server versuchen, die Umgebungsvariable `PGDATA` zu verwenden. Wenn das auch nicht funktioniert, wird der Server nicht starten.

Um den `postmaster` im Hintergrund zu starten, verwenden Sie die übliche Shell-Syntax:

```
$ postmaster -D /usr/local/pgsql/data > logdatei 2>&1 &
```

Es ist wichtig, sowohl `stdout` als auch `stderr` vom Server irgendwo hinzuleiten, wie hier gezeigt. Das hilft bei der Kontrolle der Serveraktivität und um Probleme zu analysieren. (Eine gründlichere Besprechung der Verwaltung von Logdateien finden Sie in Abschnitt 21.3.)

Der `postmaster` hat auch eine Reihe von Kommandozeilenoptionen. Weitere Informationen darüber finden Sie auf der Referenzseite und unten in Abschnitt 16.4. Insbesondere müssen Sie, wenn der Server TCP/IP-Verbindungen akzeptieren soll (anstatt nur Unix-Domain-Socket-Verbindungen), die Option `-i` verwenden.

Die Shell-Syntax kann schnell ziemlich umständlich werden. Daher gibt es das Wrapperprogramm `pg_ctl`, das einige Aufgaben vereinfachen kann. Zum Beispiel:

```
pg_ctl start -l logdatei
```

Das startet den Server im Hintergrund und leitet die Ausgabe in die angegebene Logdatei um. Die Option `-D` hat hier die gleiche Bedeutung wie bei `postmaster`. Mit `pg_ctl` kann man den Server auch wieder anhalten.

Normalerweise wollen Sie es sicher so einrichten, dass der Datenbankserver gestartet wird, wenn der Computer bootet. Wie das gemacht wird, hängt stark vom Betriebssystem ab. Einige Autostartskripts finden Sie in der PostgreSQL-Distribution im Verzeichnis `contrib/start-scripts`. `root`-Zugang wird wahrscheinlich vonnöten sein.

Verschiedene Systeme haben verschiedene Methoden, um Daemons beim Booten zu starten. Viele Systeme haben eine Datei `/etc/rc.local` oder `/etc/rc.d/rc.local`. Andere haben `rc.d`-Verzeichnisse. Was auch immer Sie machen, der Server muss vom PostgreSQL-Benutzerzugang gestartet werden und *nicht als* `root` oder ein anderer Benutzer. Daher sollten Ihre Befehle wahrscheinlich so etwas wie `su -c '...' postgres` enthalten. Zum Beispiel:

```
su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog' postgres
```

Hier sind ein paar konkretere Vorschläge für verschiedene Betriebssysteme. (Ersetzen Sie immer die richtigen Installationsverzeichnisse und Benutzernamen.)

- Für FreeBSD schauen Sie in die Datei `contrib/start-scripts/freebsd` in der PostgreSQL-Distribution.
- Für OpenBSD fügen Sie folgende Zeilen der Datei `/etc/rc.local` an:

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postmaster ];
then
```

```
        su - -c '/usr/local/pgsql/bin/pg_ctl start -l /var/postgresql/log -s'
        postgres echo -n ' postgresql'
fi
```

- Für Linux-Systeme fügen Sie entweder

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

  in `/etc/rc.d/rc.local` ein oder schauen Sie in die Datei `contrib/start-scripts/linux` in der PostgreSQL-Distribution.

- Für NetBSD verwenden Sie entweder das Startskript von FreeBSD oder das von Linux, je nach Geschmack.
- Für Solaris erzeugen Sie eine Datei namens `/etc/init.d/postgresql` mit der folgenden Zeile:

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/
pgsql/data"
```

- Dann erzeugen Sie in `/etc/rc3.d` einen symbolischen Link dahin als `S99postgresql`.

Während der `postmaster` läuft, ist seine PID in der Datei `postmaster.pid` im Datenverzeichnis gespeichert. Diese Datei wird verwendet, um zu verhindern, dass mehrere `postmaster`-Prozesse im selben Datenverzeichnis laufen, und kann auch dazu verwendet werden, um den `postmaster` zu beenden.

16.3.1 Fehler beim Starten des Servers

Es gibt einige häufige Gründe, warum der Server einmal nicht starten könnte. Prüfen Sie die Serverlogdatei oder starten Sie ihn von Hand (ohne die Ausgabeströme umzuleiten) und sehen Sie, welche Fehlermeldungen ausgegeben werden. Unten gehen wir auf einige der häufigsten Fehlermeldungen etwas genauer ein.

```
FATAL: StreamServerPort: bind() failed: Address already in use
        Is another postmaster already running on port 5432?
        If not, wait a few seconds an retry.
```

Das bedeutet in der Regel genau das, was es aussagt: Sie haben versucht, einen `postmaster` an einem Port zu starten, wo schon einer läuft. Wenn die Fehlermeldung des Kernels jedoch nicht `Address already in use` oder etwas Ähnliches ist, könnte es ein anderes Problem sein. Wenn man zum Beispiel den `postmaster` an einer reservierten Portnummer laufen lassen will, könnte man Folgendes sehen:

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
        Is another postmaster already running on port 666?
        If not, wait a few seconds an retry.
```

Eine Meldung wie

```
IpcMemoryCreate: shmget(key=5440001, size=83918612, 01600) failed: Invalid
argument
FATAL 1:  ShmemCreate: cannot create region
```

bedeutet wahrscheinlich, dass die Kernelgrenze für Shared Memory kleiner ist als der Pufferbereich, den PostgreSQL anlegen wollte (in diesem Beispiel 83918612 Bytes). Oder es kann andeuten, dass Ihr Kernel System-V Shared Memory gar nicht unterstützt. Als vorübergehenden Ausweg können Sie den Server mit

weniger Puffern als normal starten (Option `-B`). Früher oder später werden Sie Ihren Kernel aber umkonfigurieren müssen, um die Shared-Memory-Größe zu erhöhen. Sie könnten diese Mitteilung auch sehen, wenn Sie mehrere Server auf derselben Maschine starten und diese zusammen die Kernelbeschränkungen ausreizen.

Ein Fehler wie

```
IpcSemaphoreCreate: semget(key=5440026, num=16, 01600) failed: No space left on
device
```

bedeutet *nicht*, dass Sie keinen Platz mehr auf der Festplatte haben. Er bedeutet, dass die Kernelgrenze für die Anzahl von System-V-Semaphoren kleiner ist als die Zahl, die PostgreSQL erzeugen will. Wie oben können Sie dieses Problem eventuell umgehen, indem Sie die den Server mit einer verkleinerten Anzahl an erlaubten Verbindungen (Option `-N`) starten, aber früher oder später sollten Sie Ihren Kernel umkonfigurieren.

Wenn die Fehlermeldung `illegal system call` erwähnt, dann ist es wahrscheinlich so, dass Ihr Kernel Shared Memory oder Semaphore gar nicht unterstützt. In diesem Fall ist es Ihre einzige Möglichkeit, den Kernel umzukonfigurieren und diese Einrichtungen einzuschalten.

Einzelheiten über die Konfiguration von `System V` IPC finden Sie in Abschnitt 16.5.1.

16.3.2 Probleme bei Client-Verbindungen

Obwohl die möglichen Fehler auf der Clientseite sehr verschieden sind und von der Anwendung abhängen, stehen einige davon doch direkt damit in Verbindung, wie der Server gestartet wurde. Fehlerumstände außer den hier gezeigten werden bei der entsprechenden Anwendung beschrieben.

```
psql: could not connect to server: Connection refused
        Is the server running on host server.joe.com and accepting
        TCP/IP connections on port 5432?
```

Das ist die allgemeine Fehlermeldung, die besagt: “Ich konnte keinen Server finden.” So wie hier sieht es aus, wenn eine TCP/IP-Verbindung versucht wurde. Ein häufiger Fehler ist es, zu vergessen, den Server so zu konfigurieren, dass er TCP/IP-Verbindungen annehmen kann.

Wenn Sie eine Verbindung zu einem lokalen Server über Unix-Domain-Sockets versuchen, dann würden Sie dies sehen:

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Die letzte Zeile ist nützlich, um zu überprüfen, ob der Client auch mit der richtigen Stelle verbinden will. Wenn da wirklich kein Server läuft, wird die Fehlermeldung des Kernels entweder `Connection refused` oder `No such file or directory` sein, wie gezeigt. (Wichtig ist, dass `Connection refused` in diesem Zusammenhang *nicht* bedeutet, dass der Server Ihre Verbindungsanfrage erhalten hat und sie abgelehnt hat. In dem Fall sieht die Fehlermeldung anders aus, wie in Abschnitt 19.2.4 gezeigt wird.) Andere Fehlermeldungen wie `Connection timed out` zeigen womöglich grundlegendere Probleme an, wie zum Beispiel das Fehlen einer Netzwerkverbindung.

16.4 Laufzeit-Konfiguration

Es gibt eine ganze Reihe von Konfigurationsparametern, die sich auf das Verhalten des Datenbanksystems auswirken. Hier beschreiben wir, wie sie gesetzt werden, und die folgenden Unterabschnitte beschreiben jeden Parameter im Einzelnen.

Bei den Namen der Parameter spielt die Groß- und Kleinschreibung keine Rolle. Jeder Parameter hat einen der folgenden Datentypen: Boolean (an/aus), ganze Zahl, Fließkommazahl oder Zeichenkette. Boolean-Werte sind `ON`, `TRUE`, `YES`, `1` für "an" sowie `OFF`, `FALSE`, `NO`, `0` für "aus" oder ein eindeutiger Präfix von einem dieser Werte. (Die Groß-/Kleinschreibung ist egal.)

Eine Möglichkeit, diese Optionen zu setzen, ist die Datei `postgresql.conf` im Datenverzeichnis zu bearbeiten. (Ein Beispieldatei mit Voreinstellungen wird dort installiert.) Ein einfaches Beispiel einer solchen Datei wäre:

```
# Das ist ein Kommentar.
log_connections = yes
syslog = 2
search_path = '$user, public'
```

Wie Sie sehen, steht eine Option pro Zeile. Das Gleichheitszeichen zwischen Name und Wert kann weggelassen werden. Leerzeichen können frei verwendet werden und leere Zeilen werden ignoriert. Kommentare können an jeder Stelle mit # anfangen. Parameterwerte, die keine einfachen Bezeichner sind, sollten zwischen Apostrophen stehen.

Die Konfigurationsdatei wird neu gelesen, wenn der `postmaster`-Prozess das Signal `SIGHUP` erhält (was man am einfachsten mit `pg_ctl reload` auslösen kann). Der `postmaster`-Prozess schickt dieses Signal dann auch an alle Serverkindprozesse weiter, damit alle aktiven Sitzungen den neuen Wert erhalten. Alternativ können Sie das Signal auch direkt an einen einzelnen Serverprozess schicken.

Die zweite Möglichkeit, diese Konfigurationsparameter zu setzen, ist, sie als Kommandozeilenoptionen beim Aufruf von `postmaster` anzugeben, zum Beispiel:

```
postmaster -c log_connections=yes -c syslog=2
```

Kommandozeilenoptionen heben etwaige widersprüchliche Einstellungen in `postgresql.conf` auf.

Manchmal kann es auch nützlich sein, eine Kommandozeilenoption nur für eine bestimmte Sitzung zu verwenden. Dazu kann die Umgebungsvariable `PGOPTIONS` auf der Clientseite verwendet werden:

```
env PGOPTIONS='-c geqo=off' psql
```

(Das funktioniert bei jeder Clientanwendung auf `libpq`-Basis, nicht nur bei `psql`.) Beachten Sie aber, dass das nicht für Optionen funktioniert, die feststehen, wenn der Server gestartet wird, wie zum Beispiel die Portnummer.

Einige Optionen können in einzelnen SQL-Sitzungen mit dem Befehl `SET` geändert werden, zum Beispiel:

```
SET enable_seqscan TO off;
```

Einzelheiten zur Syntax finden Sie in der SQL-Befehlsreferenz.

Außerdem ist es möglich, Optionseinstellungen für einzelne Benutzer oder Datenbanken zu machen. Immer wenn eine Sitzung gestartet wird, werden die Voreinstellungen für den betroffenen Benutzer und die Datenbank geladen. Die Befehle `ALTER DATABASE` bzw. `ALTER USER` werden verwendet, um diese Einstellungen vorzunehmen. Solche datenbankspezifischen Einstellungen haben Vorrang vor Einstellungen von der `postmaster`-Kommandozeile und der Konfigurationsdatei. Benutzerspezifische Einstellungen haben wiederum noch höheren Vorrang.

Die virtuelle Tabelle pg_settings ermöglicht das Anzeigen und Aktualisieren von Konfigurationsparametern. Sie enthält eine Zeile für jeden Parameter und die in Tabelle 16.1 gezeigten Spalten. Diese Form ermöglicht es Ihnen, die Konfigurationsdaten mit anderen Tabellen zu verbinden und Auswahlkriterien anzuwenden.

Ein UPDATE in pg_settings ist gleichbedeutend mit der Ausführung des Befehls SET für den benannten Parameter. Die Veränderung gilt nur für die aktuelle Sitzung. Wenn ein UPDATE in einer Transaktion ausgeführt wird, die später zurückgerollt wird, wird die Einstellung zurückgenommen, wenn die Tranaktion zurückgerollt wird. Wenn die umgebende Transaktion erfolgreich abgeschlossen wird, dann bleiben die Auswirkungen bis zum Ende der Sitzung bestehen, außer wenn ein weiterer UPDATE- oder SET-Befehl die Einstellung wieder ändert.

Name	Datentyp	Beschreibung
name	text	der Name des Konfigurationsparameters
setting	text	der aktuelle Wert des Konfigurationsparameters

Tabelle 16.1: pg_settings Spalten

16.4.1 Planer- und Optimierereinstellungen

CPU_INDEX_TUPLE_COST (Fließkommazahl)

Setzt den vom Anfrageplaner geschätzten Aufwand, um ein Indextupel während eines Indexscans zu verarbeiten. Das wird relativ zum Aufwand eines sequenziellen Zeilenlesevorgangs gemessen.

CPU_OPERATOR_COST (Fließkommazahl)

Setzt den vom Anfrageplaner geschätzten Aufwand, um einen Operator in einer WHERE-Klausel zu verarbeiten. Das wird relativ zum Aufwand eines sequenziellen Zeilenlesevorgangs gemessen.

CPU_TUPLE_COST (Fließkommazahl)

Setzt den vom Anfrageplaner geschätzten Aufwand, um ein Tupel während einer Anfrage zu verarbeiten. Das wird relativ zum Aufwand eines sequenziellen Zeilenlesevorgangs gemessen.

DEFAULT_STATISTICS_TARGET (ganze Zahl)

Setzt das voreingestellte Statistikziel für Tabellenspalten, bei denen das Statistikziel nicht mit ALTER TABLE SET STATISTICS geändert wurde. Größere Werte führen dazu, dass ANALYZE länger dauert, aber können auch zu besseren Planerschätzungen beitragen.

EFFECTIVE_CACHE_SIZE (Fließkommazahl)

Setzt die Annahme des Planers über die effektive Größe des Diskcaches (das heißt des Teils des Diskcaches vom Kernel, der für die Datendateien von PostgreSQL verwendet wird). Das wird in Diskseiten gemessen, welche normalerweise 8 kB groß sind.

ENABLE_HASHJOIN (Boolean)

Schaltet die Verwendung des Plantyps Hash-Verbund im Planer an oder aus. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_INDEXSCAN (Boolean)

Schaltet die Verwendung des Indexscan-Plantyps im Planer ein oder aus. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_MERGEJOIN (Boolean)

Schaltet die Verwendung des Plantyps Merge-Verbund im Planer an oder aus. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_NESTLOOP (Boolean)

Schaltet die Verwendung des Plantyps Nested-Loop-Verbund im Planer an oder aus. Man kann Nested-Loop-Verbunde nicht vollständig unterdrücken, aber wenn man diese Variable ausschaltet, hält das den Planer davon ab, einen zu verwenden, wenn andere Methoden zur Verfügung stehen. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_SEQSCAN (Boolean)

Schaltet die Verwendung von sequenziellen Scans im Planer ein oder aus. Man kann sequenzielle Scans nicht vollständig unterdrücken, aber wenn man diese Variable ausschaltet, hält das den Planer davon ab, einen zu verwenden, wenn andere Methoden zur Verfügung stehen. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_SORT (Boolean)

Schaltet die Verwendung von ausdrücklichen Sortierschritten im Planer an oder aus. Man kann ausdrückliche Sortierschritte nicht vollständig unterdrücken, aber wenn man diese Variable ausschaltet, dann hält das den Planer davon ab, einen zu verwenden, wenn andere Methoden zur Verfügung stehen. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

ENABLE_TIDSCAN (Boolean)

Schaltet die Verwendung des TID-Scan-Plantyps im Planer an oder aus. Die Voreinstellung ist an. Diese Option kann zum Debuggen des Anfrageplaners verwendet werden.

GEQO (Boolean)

Schaltet genetische Anfrageoptimierung (*genetic query optimization*), ein Algorithmus, der versucht, Anfragen ohne erschöpfende Suchen zu planen, an oder aus. Die Voreinstellung ist an. Siehe auch bei den verschiedenen anderen GEQO_-Einstellungen.

GEQO_EFFORT (ganze Zahl),

GEQO_GENERATIONS (ganze Zahl),

GEQO_POOL_SIZE (ganze Zahl),

GEQO_RANDOM_SEED (ganze Zahl),

GEQO_SELECTION_BIAS (Fließkommazahl)

Verschiedene Einstellungen für den genetischen Anfrageoptimierungsalgorithmus: GEQO_POOL_SIZE ist die Anzahl der Individien in der Bevölkerung. Gültige Werte sind zwischen 128 und 1024. Wenn es 0 ist (Voreinstellung), dann wird 2^{QS+1} verwendet, wobei QS die Anzahl der FROM-Elemente in der Anfrage ist. GEQO_EFFORT wird verwendet, um für GEQO_GENERATIONS einen Vorgabewert zu errechnen. Gültige Werte sind zwischen 1 und 80, wobei 40 die Voreinstellung ist. GEQO_GENERATIONS ist die Anzahl der Iterationen des Algorithmus. Die Zahl muss eine positive ganze Zahl sein. Wenn 0 angeben wird, dann wird `Effort * Log2(PoolSize)` verwendet. Die Laufzeit des Algorithmus ist in etwa proportional zur Summe von GEQO_POOL_SIZE und GEQO_GENERATIONS. GEQO_SELECTION_BIAS ist der Auswahldruck in der Bevölkerung. Werte können von 1,50 bis 2,00 gehen; Letzterer ist die Voreinstellung. GEQO_RANDOM_SEED kann gesetzt werden, um vom Algorithmus wiederholbare Ergebnisse zu erhalten. Wenn der Wert -1 ist, dann verhält sich der Algorithmus nicht deterministisch.

GEQO_THRESHOLD (ganze Zahl)

Gibt an, dass genetische Anfrageoptimierung verwendet werden soll, wenn eine Anfrage mindestens so viele Elemente in der FROM-Liste hat. (Eine JOIN-Konstruktion zählt als nur ein Element.) Die Voreinstellung ist 11. Für einfachere Anfragen ist es normalerweise besser, den normalen, deterministischen Planer zu verwenden. Dieser Parameter kontrolliert auch, wie sehr der Optimierer versuchen wird, FROM-Elemente mit Unteranfragen mit der übergeordneten Anfrage zusammenzuführen.

`RANDOM_PAGE_COST` (Fließkommazahl)

Setzt den vom Anfrageplaner geschätzten Aufwand, um eine Diskseite außer der Reihe von der Festplatte zu lesen. Das wird relativ zum Aufwand eines sequenziellen Zeilenlesevorgangs gemessen.

Anmerkung

Leider gibt es für die Familie der eben beschriebenen `COST`-Variablen keine richtig gute Methode, um ideale Werte zu ermitteln. Wir ermutigen Sie, zu experimentieren und Ihre Ergebnisse mit anderen zu teilen.

16.4.2 Log- und Debug-Ausgabe

`CLIENT_MIN_MESSAGES` (Zeichenkette)

Kontrolliert, welche Meldungstypen an den Client geschickt werden. Gültige Werte sind `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING` und `ERROR`. Jeder Typ schließt alle ihm folgenden Typen mit ein. Je weiter hinten der Typ steht, desto weniger Meldungen werden gesendet werden. Die Voreinstellung ist `NOTICE`. Beachten Sie, dass `LOG` hier anders eingeordnet ist als bei `SERVER_MIN_MESSAGES`. Sehen Sie auch dort nach, um eine Beschreibung der verschiedenen Werte zu erhalten.

`DEBUG_ASSERTIONS` (Boolean)

Schaltet diverse Assertion-Prüfungen ein. Das ist eine Debug-Hilfe. Wenn sonderbare Probleme auftreten oder der Server abstürzt, können Sie diese Option anschalten, weil dadurch Programmierfehler entlarvt werden könnten. Um diese Option benutzen zu können, muss das Makro `USE_ASSERT_CHECKING` bei der Compilierung definiert sein (am besten durch `configure --enable-cassert` erreicht). Beachten Sie, dass, wenn PostgreSQL damit kompiliert wurde, diese Option in der Voreinstellung an ist.

`DEBUG_PRINT_PARSE` (Boolean),

`DEBUG_PRINT_REWRITTEN` (Boolean),

`DEBUG_PRINT_PLAN` (Boolean),

`DEBUG_PRETTY_PRINT` (Boolean)

Diese Optionen sorgen dafür, dass bestimmte Debug-Ausgaben an den Serverlog geschickt werden. Für jede ausgeführte Anfrage wird entweder der resultierende Parsebaum, das Ergebnis des Umschreibers oder der Ausführungsplan ausgegeben. Durch `DEBUG_PRETTY_PRINT` wird die Ausgabe eingerückt, um besser lesbar, aber auch wesentlich länger zu sein.

`EXPLAIN_PRETTY_PRINT` (Boolean)

Bestimmt, ob `EXPLAIN VERBOSE` das eingerückte oder das nicht eingerückte Format zur Ausgabe der Anfragebäume verwendet.

`HOSTNAME_LOOKUP` (Boolean)

In der Voreinstellung zeigen die Verbindungslogs nur die IP-Adressen der Clienthosts. Wenn Sie den Hostnamen auch anzeigen wollen, dann können Sie diese Option anschalten, aber je nachdem, wie Ihr DNS eingerichtet ist, kann das die Leistung nicht unerheblich beeinträchtigen. Diese Option kann nur beim Start des Servers gesetzt werden.

`LOG_CONNECTIONS` (Boolean)

Schreibt für jede erfolgreiche Verbindung eine Zeile mit Einzelheiten in den Serverlog. Das ist in der Voreinstellung aus, ist aber wahrscheinlich sehr nützlich. Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden.

`LOG_DURATION` (Boolean)

Wenn an, dann wird die Dauer jedes abgeschlossenen Befehls geloggt. Wenn Sie diese Option verwenden, sollten Sie auch `LOG_STATEMENT` und `LOG_PID` anschalten, damit Sie die Befehle und die Dauer über die Prozess-ID in Verbindung setzen können.

`LOG_MIN_ERROR_STATEMENT` (Zeichenkette)

Kontrolliert, für welche Meldungstypen der SQL-Befehl, der diese Meldung verursacht hat, im Serverlog aufgezeichnet wird. Alle Befehle, die eine Meldung des gesetzten Typs oder höher verursachen, werden geloggt. Die Voreinstellung ist `PANIC` (wodurch diese Funktion im Prinzip ausgeschaltet wird). Gültige Werte sind `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `FATAL` und `PANIC`. Wenn Sie die Option zum Beispiel auf `ERROR` setzen, dann werden alle SQL-Befehle, die Meldungen der Typen `ERROR`, `FATAL` oder `PANIC` erzeugen, geloggt.

Wir empfehlen Ihnen außerdem, `LOG_PID` anzuschalten, damit Sie den fehlerhaften Befehl und die Fehlermeldung leichter in Verbindung bringen können.

`LOG_PID` (Boolean)

Schreibt vor jede Meldung im Serverlog die Prozess-ID des Serverprozesses. Das ist nützlich, um herauszufinden, welche Meldungen zu welcher Verbindung gehören. Die Voreinstellung ist aus. Dieser Parameter hat keine Auswirkung auf Meldungen, die mit `Syslog` geloggt werden, da dort die Prozess-ID immer enthalten ist.

`LOG_STATEMENT` (Boolean)

Schreibt jeden SQL-Befehl in den Serverlog.

`LOG_TIMESTAMP` (Boolean)

Schreibt vor jede Serverlogmeldung die aktuelle Zeit. Die Voreinstellung ist aus.

`SERVER_MIN_MESSAGES` (Zeichenkette)

Kontrolliert, welche Meldungstypen in den Serverlog geschrieben werden. Gültige Werte sind `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` und `PANIC`. Jeder Typ schließt alle ihm folgenden Typen mit ein. Je weiter hinten der Typ steht, desto weniger Meldungen werden in den Log geschrieben werden. Die Voreinstellung ist `NOTICE`. Beachten Sie, dass `LOG` hier anders eingeordnet ist als bei `CLIENT_MIN_MESSAGES`.

Hier ist eine Liste der verschiedenen Meldungstypen:

`DEBUG[1-5]`

Enthält Informationen für Entwickler.

`INFO`

Enthält Informationen, die implizit vom Benutzer angefordert wurden, z.B bei `VACUUM VERBOSE`.

`NOTICE`

Enthält Informationen, die für Benutzer hilfreich sein könnten, z.B. das Abschneiden langer Namen oder die Erzeugung eines Indexes als Teil eines Primärschlüssels.

`WARNING`

Enthält Warnhinweise für den Benutzer, z.B. `COMMIT` außerhalb eines Transaktionsblocks.

`ERROR`

Berichtet einen Fehler, der den Abbruch der aktuellen Transaktion verursachte.

`LOG`

Enthält Informationen für Administratoren, z.B. über Checkpoint-Aktivität.

FATAL

Berichtet einen Fehler, der den Abbruch der aktuellen Sitzung verursachte.

PANIC

Berichtet einen Fehler, der den Abbruch aller Sitzungen verursachte.

SHOW_STATEMENT_STATS (Boolean),

SHOW_PARSER_STATS (Boolean),

SHOW_PLANNER_STATS (Boolean),

SHOW_EXECUTOR_STATS (Boolean)

Schreibt für jede Anfrage Leistungsstatistiken des entsprechenden Moduls in den Serverlog. Das ist ein grobes Profiling-Instrument.

SHOW_SOURCE_PORT (Boolean)

Zeigt die Nummer des ausgehenden Ports auf dem verbundenen Client in den Verbindungslogmeldungen. Sie könnten diese Portnummer zurückverfolgen, um herauszufinden, welcher Benutzer die Verbindung eingeleitet hat. Ansonsten ist diese Option ziemlich nutzlos und daher in der Voreinstellung aus. Diese Option kann nur beim Start des Servers eingestellt werden.

STATS_COMMAND_STRING (Boolean),

STATS_BLOCK_LEVEL (Boolean),

STATS_ROW_LEVEL (Boolean)

Diese Optionen bestimmen, welche Informationen an den Statistikkollektor gesendet werden: aktueller Befehl, Statistiken über Aktivitäten auf Blockebene oder Statistiken über Aktivitäten auf Zeilenebene. In der Voreinstellung sind alle aus. Wenn die Sammlung der Statistiken eingeschaltet ist, kostet sie pro Anfrage ein wenig Zeit; sie ist aber unschätzbar zum Debuggen und um Feinabstimmungen zur Leistungsverbesserung vornehmen zu können.

STATS_RESET_ON_SERVER_START (Boolean)

Wenn an, werden die gesammelten Statistiken zurückgesetzt, wenn der Server neu gestartet wird. Wenn aus, werden die Statistiken nach einem Neustart weitergezählt. Die Voreinstellung ist aus. Diese Option kann nur beim Start des Servers eingestellt werden.

STATS_START_COLLECTOR (Boolean)

Kontrolliert, ob der Server den Statistikkollektor-Subprozess starten soll. Das ist in der Voreinstellung angeschaltet, kann aber ausgeschaltet werden, wenn Sie kein Interesse an Statistiken haben. Diese Option kann nur beim Start des Servers eingestellt werden.

SYSLOG (ganze Zahl)

PostgreSQL kann Syslog zum Loggen verwenden. Wenn diese Option 1 ist, gehen Meldungen an Syslog und die Standardausgabe. Die Einstellung 2 schickt die Ausgabe nur an Syslog. (Einige Meldungen gehen aber trotzdem an die Standardausgabe oder die Standardfehlerausgabe.) Die Voreinstellung ist 0, womit Syslog aus ist. Diese Option kann nur beim Start des Servers eingestellt werden.

SYSLOG_FACILITY (Zeichenkette)

Diese Option bestimmt die Syslog-“Facility”, die verwendet wird, wenn mit Syslog geloggt wird. Zus Auswahl stehen LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7; die Voreinstellung ist LOCAL0. Eine Erklärung dazu finden Sie in der Anleitung von Syslog auf Ihrem System.

SYSLOG_IDENT (Zeichenkette)

Wenn Loggen mit Syslog an ist, bestimmt diese Option den Programmnamen, der verwendet wird, um PostgreSQL-Meldungen in den Syslog-Meldungen zu identifizieren. Die Voreinstellung ist postgres.

TRACE_NOTIFY (Boolean)

Erzeugt eine Menge Debug-Ausgaben für die Befehle LISTEN und NOTIFY.

16.4.3 Allgemeine Operation

AUTOCOMMIT (Boolean)

Wenn auf wahr gesetzt, wird PostgreSQL nach jedem erfolgreichen Befehl, der nicht in einem ausdrücklichen Transaktionsblock ist (das heißt, ein BEGIN ohne zugehöriges COMMIT wurde ausgeführt), automatisch ein COMMIT ausführen. Wenn auf falsch gesetzt, dann schließt PostgreSQL Transaktionen nur dann ab, wenn ein COMMIT ausdrücklich ausgeführt wird. Man kann sich diesen Modus auch so vorstellen, dass implizit ein BEGIN ausgeführt wird, wenn ein Befehl empfangen wird, der nicht schon in einem Transaktionsblock ist. Die Voreinstellung ist wahr, um mit dem bisherigen Verhalten von PostgreSQL kompatibel zu sein. Um jedoch zu sehr wie möglich mit dem SQL-Standard kompatibel zu sein, setzen Sie die Option auf falsch.

Anmerkung

Selbst wenn autocommit falsch ist, wird durch SET, SHOW und RESET kein neuer Transaktionsblock gestartet; sie laufen in ihren eigenen Transaktionen. Wenn erst ein anderer Befehl ausgeführt wird, fängt ein Transaktionsblock an und etwaige folgende SET-, SHOW- oder RESET-Befehle sind Teil des Transaktionsblocks, das heißt, sie könnten eventuell zurückgerollt werden, wenn folgende Befehle Fehler verursachen. Um SET, SHOW oder RESET am Anfang eines Transaktionsblocks auszuführen, verwenden Sie zuerst BEGIN.

Anmerkung

In PostgreSQL 7.3 wird autocommit auf falsch zu setzen noch nicht sehr gut unterstützt. Es ist ein neues Feature, das noch nicht von allen Clientanwendungen und -bibliotheken unterstützt wird. Bevor Sie es in Ihrer Installation zur Voreinstellung machen, testen Sie sorgfältig.

AUSTRALIAN_TIMEZONES (Boolean)

Wenn wahr, dann werden ACST, CST, EST und SAT als australische Zeitzonen anstatt amerikanische Zeitzonen bzw. "Samstag" interpretiert. Die Voreinstellung ist falsch.

AUTHENTICATION_TIMEOUT (ganze Zahl)

Maximale Zeit, um die Clientauthentifizierung abzuschließen, in Sekunden. Wenn ein potenzieller Client die Authentifizierung nicht in dieser Zeit abgeschlossen hat, bricht der Server die Verbindung ab. Das verhindert, dass festgefahrene Clients eine Verbindung endlos besetzen können. Diese Option kann nur beim Start des Servers oder in der Datei postgresql.conf eingestellt werden.

CLIENT_ENCODING (Zeichenkette)

Setzt den auf dem Client verwendeten Zeichensatz. Die Voreinstellung ist, den Zeichensatz der Datenbank zu verwenden.

DATESTYLE (Zeichenkette)

Setzt das Ausgabeformat für Datums- und Zeitangaben sowie die Regeln um zweideutige Datumseingaben zu interpretieren. Einzelheiten finden Sie in Abschnitt 8.5. Die Voreinstellung ist ISO, US.

DB_USER_NAMESPACE (Boolean)

Hierdurch werden datenbank-lokale Benutzernamen ermöglicht. Die Voreinstellung ist aus.

Wenn die Option an ist, sollten Sie Ihre Benutzer als benutzername@dbname erzeugen. Wenn ein verbindender Client benutzername übergibt, wird @ und der Datenbankname angehängt und dieser datenbankspezifische Benutzername vom Server verwendet. Beachten Sie: Wenn Sie in der SQL-Umgebung Benutzernamen mit @ erzeugen, dann werden Sie den Benutzernamen in Anführungszeichen stellen müssen.

Wenn diese Option an ist, können Sie immer noch normale globale Benutzer erzeugen. Hängen Sie einfach @ an den Namen an, wenn Sie den Benutzernamen im Client angeben. Das @ wird entfernt, bevor der Server den Benutzernamen weiterverwendet.

Anmerkung

Dieses Feature ist als vorübergehende Maßnahme gedacht, bis eine komplette Lösung gefunden wird. Diese Option wird dann entfernt werden.

DEADLOCK_TIMEOUT (ganze Zahl)

Dies ist die Zeit, in Millisekunden, die auf eine Sperre gewartet wird, bis geprüft wird, ob eine Verklemmung vorliegt. DieÜberprüfung auf Verklemmungen ist relativ langsam, und daher möchte der Server sie nicht jedes Mal, wenn er auf eine Sperre wartet, durchführen. Wir nehmen (optimistischerweise?) an, dass Verklemmungen in normalen Anwendungen nicht häufig vorkommen, und warten einfach ein bisschen auf die Sperre, ehe wir die Verklemmungsprüfung starten. Wenn dieser Wert erhöht wird, verringert sich die Zeit, die in nutzlosen Verklemmungsprüfungen verwendet wird, aber es dauert länger, bis eine wirkliche Verklemmung erkannt wird. Die Voreinstellung ist 1000 (d.h. eine Sekunde), was wahrscheinlich in etwa der niedrigste Wert ist, den man in der Praxis verwenden sollte. Auf einem stark belasteten Server könnten Sie eine Erhöhung in Erwägung ziehen. Idealerweise sollte dieser Wert höher sein als die Dauer einer typischen Transaktion, damit die Chancen besser stehen, dass eine Sperre aufgegeben wird, ehe die wartende Sitzung auf Verklemmung prüft. Diese Option kann nur beim Start des Servers eingestellt werden.

DEFAULT_TRANSACTION_ISOLATION (Zeichenkette)

Jede SQL-Transaktion hat einen Isolationsgrad, der entweder read committed oder serializable ist. Dieser Parameter kontrolliert den Isolationsgrad, mit dem jede neue Transaktion anfängt. Die Voreinstellung ist read committed.

Weitere Informationen finden Sie in Kapitel 12 und auf Seite 776.

DYNAMIC_LIBRARY_PATH (Zeichenkette)

Wenn ein dynamisch ladbares Modul geöffnet werden muss und der angegebene Name keine Verzeichniskomponente hat (das heißt er enthält keinen Schrägstrich), sucht das System im DYNAMIC_LIBRARY_PATH-Pfad nach der angegebenen Datei. (Der betrachtete Name ist der Name, der in CREATE FUNCTION or LOAD angegeben wurde.)

Der Wert von DYNAMIC_LIBRARY_PATH muss eine Liste von absoluten Verzeichnisnamen, getrennt durch Doppelpunkte, sein. Wenn ein Verzeichnisname mit dem besonderen Wert $libdir anfängt, wird dafür der eingebaute Name des Verzeichnisses für PostgreSQL-Pakete eingesetzt. In diesem Verzeichnis sind die in der PostgreSQL-Distribution enthaltenen ladbaren Module installiert. (Verwenden Sie pg_config --pkglibdir, um den Namen dieses Verzeichnisses auszugeben.) Zum Beispiel:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/mein_projekt/
lib:$libdir'
```

Der voreingestellte Wert für diesen Parameter ist '$libdir'. Wenn der Wert eine leere Zeichenkette ist, wird die automatische Pfadsuche abgestellt.

Dieser Parameter kann zur Laufzeit nur von einem Superuser geändert werden, aber eine solche Änderung hält nur bis zum Ende der Clientverbindung. Daher sollte diese Methode nur bei der Entwicklung verwendet werden. Empfohlen wird, diesen Parameter in der Konfigurationsdatei `postgresql.conf` zu setzen.

`KRB_SERVER_KEYFILE` (Zeichenkette)

Setzt den Ort der Kerberos-Server-Schlüsseldatei. Siehe Abschnitt 19.2.3 wegen Einzelheiten.

`FSYNC` (Boolean)

Wenn diese Option an ist, verwendet der PostgreSQL-Server an mehreren Stellen den Systemaufruf `fsync()` um sicherzustellen, dass Änderungen physikalisch auf die Festplatte geschrieben werden. Das versichert, dass ein Datenbankcluster nach einem Betriebssystem- oder Hardwareabsturz in einen konsistenten Zustand wiederhergestellt werden kann. (Ein Absturz des Datenbankservers selbst hat damit *nichts* zu tun.)

Diese Operation verlangsamt PostgreSQL jedoch, weil es beim Abschluss einer Transaktion darauf warten muss, dass das Betriebssystem den Write-Ahead-Log auf die Festplatte zurückschreibt. Ohne `fsync` steht es dem Betriebssystem frei, die Schreibvorgänge durch Zwischenspeichern, Umsortieren und Verzögern zu optimieren, wodurch die Leistung erheblich verbessert werden kann. Wenn das System jedoch abstürzt, könnten die Ergebnisse der letzten paar abgeschlossenen Transaktionen ganz oder teilweise verloren gehen. Im schlimmsten Fall werden die Daten unwiederbringlich zerstört.

Aus den oben genannten Gründen kann jeder selbst entscheiden, wie er mit der Option `fsync` verfahren will. Einige Administratoren lassen sie immer ausgeschaltet; einige schalten sie nur aus, wenn sie viele Daten auf einmal laden wollen, weil es da einen klaren Punkt zum Neuanfang gibt, wenn etwas schief geht; einige lassen sie immer an, nur um auf der sicheren Seite zu sein. In der Voreinstellung ist sie an, damit Sie auf der sicheren Seite sind. Wenn Sie Ihrem Betriebssystem, Ihrer Hardware und Ihrem Stromversorger (oder besser Ihrem Notstromaggregat) vertrauen, können Sie sich überlegen, `fsync` abzustellen.

Wir sollten anmerken, dass die Leistungsbeeinträchtigung durch `fsync` seit PostgreSQL Version 7.1 erheblich geringer ist. Wenn Sie früher `fsync` aus Leistungsgründen ausgeschaltet haben, dann sollten Sie diese Wahl jetzt überdenken.

Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden.

`LC_MESSAGES` (Zeichenkette)

Setzt die Sprache, in der Mitteilungen ausgegeben werden. Gültige Werte sind systemabhängig; siehe Abschnitt 20.1 für weitere Informationen. Wenn der Wert eine leere Zeichenkette ist (was die Voreinstellung ist), wird der Wert auf eine systemabhängige Weise aus der Umgebung des Servers ermittelt.

Auf einigen Systemen gibt es diese Locale-Kategorie nicht. Man kann die Variable trotzdem setzen, aber es wird sich nichts ändern. Es besteht außerdem die Möglichkeit, dass für die gewünschte Sprache keine übersetzten Meldungen vorliegen. In dem Fall sehen Sie weiterhin die englischsprachigen Meldungen.

`LC_MONETARY` (Zeichenkette)

Setzt die Locale zur Formatierung von Geldbeträgen, zum Beispiel mit der Familie der `to_char`-Funktionen. Gültige Werte sind systemabhängig; siehe Abschnitt 20.1 für weitere Informationen. Wenn der Wert eine leere Zeichenkette ist (was die Voreinstellung ist), wird der Wert auf eine systemabhängige Weise aus der Umgebung des Servers ermittelt.

`LC_NUMERIC` (Zeichenkette)

Setzt die Locale zur Formatierung von Zahlen, zum Beispiel mit der Familie der `to_char`-Funktionen. Gültige Werte sind systemabhängig; siehe Abschnitt 20.1 für weitere Informationen. Wenn

der Wert eine leere Zeichenkette ist (was die Voreinstellung ist), wird der Wert auf eine systemabhängige Weise aus der Umgebung des Servers ermittelt.

`LC_TIME` (Zeichenkette)

Setzt die Locale zur Formatierung von Datums- und Zeitwerten. (Gegenwärtig macht diese Einstellung gar nichts, aber sie könnte in der Zukunft gebraucht werden.) Gültige Werte sind systemabhängig; siehe Abschnitt 20.1 für weitere Informationen. Wenn der Wert eine leere Zeichenkette ist (was die Voreinstellung ist), wird der Wert auf eine systemabhängige Weise aus der Umgebung des Servers ermittelt.

`MAX_CONNECTIONS` (ganze Zahl)

Bestimmt die Maximalzahl gleichzeitiger Verbindungen zum Server. Die Voreinstellung ist 32 (wenn nicht bei der Compilierung des Servers verändert). Dieser Parameter kann nur beim Start des Servers eingestellt werden.

`MAX_EXPR_DEPTH` (ganze Zahl)

Setzt die vom Parser maximal zugelassene Verschachtelungstiefe von Ausdrücken. Die Voreinstellung ist für alle normalen Anfragen hoch genug, aber Sie können sie erhöhen, wenn es notwendig ist. (Wenn Sie sie aber zu hoch setzen, riskieren Sie, dass Ihr Server abstürzt, weil der Stack übergelaufen ist.)

`MAX_FILES_PER_PROCESS` (ganze Zahl)

Setzt die maximale Anzahl an Dateien, die ein Serversubprozess gleichzeitig offen halten darf. Die Voreinstellung ist 1000. Die vom Code tatsächlich verwendete Grenze ist der kleinere Wert aus dieser Einstellung und dem Ergebnis von `sysconf(_SC_OPEN_MAX)`. Auf Systemen, wo `sysconf` vernünftige Werte liefert, müssen Sie sich um diese Einstellung daher keine Gedanken machen. Aber auf einigen Plattformen (besonders BSD-Systemen) gibt `sysconf` einen Wert zurück, der viel größer ist, als was das System in Wirklichkeit unterstützen kann, wenn viele Prozesse auf einmal alle so viele Dateien zu öffnen versuchen. Wenn Sie des Öfteren Fehlermeldungen über zu viele offene Dateien sehen, versuchen Sie, diesen Wert zu verkleinern. Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden. Wenn Sie in der Konfigurationsdatei gesetzt wird, wirkt sie sich nur auf Serverprozesse aus, die danach gestartet werden.

`MAX_FSM_RELATIONS` (ganze Zahl)

Setzt die maximale Anzahl der Relationen (Tabellen), für die die Free-Space-Map den freien Platz verfolgt. Die Voreinstellung ist 1000. Diese Option kann nur beim Start des Servers eingestellt werden.

`MAX_FSM_PAGES` (ganze Zahl)

Setzt die maximale Anzahl der Diskseiten, für die die Free-Space-Map den freien Platz verfolgt. Die Voreinstellung ist 10000. Diese Option kann nur beim Start des Servers eingestellt werden.

`MAX_LOCKS_PER_TRANSACTION` (ganze Zahl)

Die geteilte Sperrentabelle hat eine Große, die auf der Annahme basiert, dass höchstens `max_locks_per_transaction` mal `max_connections` verschiedene Objekte auf einmal gesperrt werden müssen. Die Voreinstellung ist 64, was sich im Verlauf der Zeit als ausreichend herausgestellt hat. Wenn Sie aber Clients haben, die viele verschiedene Tabellen in einer einzelnen Transaktion verwenden, kann es sein, dass Sie diesen Wert erhöhen müssen. Diese Option kann nur beim Start des Servers eingestellt werden.

`PASSWORD_ENCRYPTION` (Boolean)

Wenn in `CREATE USER` oder `ALTER USER` ein Passwort ohne `ENCRYPTED` or `UNENCRYPTED` angegeben wird, bestimmt diese Option, ob das Passwort verschlüsselt werden soll. Die Voreinstellung ist ja (das Passwort verschlüsseln).

PORT (ganze Zahl)

Der TCP-Port, auf dem der Server auf Verbindungen hört. Die Voreinstellung ist 5432. Diese Option kann nur bei Start des Servers eingestellt werden.

SEARCH_PATH (Zeichenkette)

Diese Variable gibt an, in welcher Reihenfolge Schemas durchsucht werden, wenn ein Objekt (Tabelle, Datentyp, Funktion usw.) mit einfachem Namen ohne Schemateil verwendet wird: der Suchpfad. Wenn es Objekte mit gleichen Namen in verschiedenen Schemas gibt, wird das zuerst im Suchpfad gefundene verwendet. Ein Objekt, das in keinem der Schemas im Suchpfad ist, kann nur durch Angabe eines Namens mit Schemaqualifikation verwendet werden.

Der Wert von search_path muss eine Liste von Schemas sein durch Kommas getrennt. Wenn ein Element der Liste der besondere Wert $user ist, dann steht das für das Schema mit dem durch SESSION_USER zurückgegebenen Namen (also der Sitzungsbenutzer), wenn es so ein Schema gibt. (Wenn nicht, dann wird $user ignoriert.)

Das Systemkatalogschema pg_catalog wird immer durchsucht, egal ob es im Pfad erwähnt wird oder nicht. Wenn es im Pfad steht, wird es an der angegebenen Position durchsucht. Wenn pg_catalog nicht im Pfad steht, dann wird es *vor* allen anderen Schemas im Pfad durchsucht. Außerdem wird das Schema für temporäre Tabellen, pg_temp_nnn, automatisch vor allen diesen Schemas durchsucht.

Wenn ein Objekt erzeugt wird, ohne dass ein bestimmtes Schema angegeben wird, dann wird es im ersten Schema im Suchpfad abgelegt. Ein Fehler wird ausgegeben, wenn der Suchpfad leer ist.

Der voreingestellte Wert für diesen Parameter ist '$user, public' (wobei der zweite Teil ignoriert wird, wenn es kein Schema namens public gibt). Das unterstützt die gemeinsame Verwendung einer Datenbank (wo keine Benutzer private Schemas haben und alle public verwenden), private Schemas für jeden Benutzer und Kombinationen davon. Andere Wirkungen können erzielt werden, indem der Pfad global oder für jeden Benutzer eingestellt wird.

Der aktuelle effektive Wert des Suchpfads kann mit der SQL-Funktion current_schemas() untersucht werden. Das ist nicht ganz dasselbe wie den Wert von search_path anzuschauen, da current_schemas() zeigt, wie die Elemente in search_path aufgelöst wurden.

Weitere Informationen über die Schemaverwaltung finden Sie in Abschnitt 5.7.

STATEMENT_TIMEOUT (ganze Zahl)

Bricht jeden Befehl, der länger als die angegebene Anzahl an Millisekunden dauert, ab. Wenn null, dann wird die Zeitmessung ausgeschaltet.

SHARED_BUFFERS (ganze Zahl)

Setzt die Anzahl der vom Datenbankserver verwendeten Puffer im Shared Memory. Die Voreinstellung ist 64. Jeder Puffer ist normalerweise 8192 Bytes groß. Diese Option kann beim Start des Servers eingestellt werden.

SILENT_MODE (Boolean)

Führt den Server im stillen Modus aus. Wenn diese Option gesetzt ist, wird der Server automatisch im Hintergrund ausgeführt und von allen kontrollierenden Terminals getrennt. Es werden also keine Meldungen auf den Standardausgabe- oder Standardfehlerstrom geschrieben (gleiche Wirkung wie die Option -S von postmaster). Die Verwendung dieser Option wird nicht empfohlen, wenn Sie kein anderes Logsystem, wie zum Beispiel Syslog, aktiviert haben, weil es sonst unmöglich ist, Fehlermeldungen zu sehen.

SORT_MEM (ganze Zahl)

Gibt an, wie viel Speicher interne Sortier- und Hash-Vorgänge verwenden, bevor sie auf temporäre Dateien ausweichen. Der Wert wird in Kilobyte angegeben und die Voreinstellung ist 1024 (also 1 MB). Beachten Sie, dass bei komplexen Anfragen mehrere Sortiervorgänge parallel ablaufen kön-

nen und dass dann jeder Einzelne so viel Speicher verwenden kann, wie dieser Wert angibt, bevor er anfängt, Daten in temporäre Dateien auszulagern. Außerdem könnten mehrere Sitzungen Sortiervorgänge gleichzeitig ausführen. Der tatsächlich verwendete Speicher könnte also ein Vielfaches von SORT_MEM sein. Sortiervorgänge werden von ORDER BY, Merge-Verbunden und CREATE INDEX durchgeführt.

SQL_INHERITANCE (Boolean)

Diese Option kontrolliert das Verhalten der Tabellenvererbung, insbesondere ob Untertabellen von verschiedenen Befehlen automatisch mit einbezogen werden. Vor Version 7.1 war das nicht der Fall. Wenn Sie das alte Verhalten benötigen, können Sie diese Option ausschalten, aber auf lange Sicht sollten Sie Ihre Anwendungen ändern und das Schlüsselwort ONLY verwenden, um Untertabellen auszuschließen. Weitere Informationen über Vererbung finden Sie in Teil II.

SSL (Boolean)

Ermöglicht SSL-Verbindungen. Bitte lesen Sie Abschnitt 16.7, bevor Sie dies verwenden. Die Voreinstellung ist aus.

SUPERUSER_RESERVED_CONNECTIONS (ganze Zahl)

Bestimmt die Anzahl der möglichen Verbindungen, die für PostgreSQL-Superuser reserviert werden sollen. Höchstens max_connections-Verbindungen können gleichzeitig aktiv sein. Wenn die Anzahl der aktiven Verbindungen max_connections minus superuser_reserved_connections erreicht oder überschreitet, werden neue Verbindungen nur für Superuser angenommen.

Die Voreinstellung ist 2. Der Wert muss kleiner als der Wert von max_connections sein. Dieser Parameter kann nur beim Start des Servers eingestellt werden.

TCPIP_SOCKET (Boolean)

Wenn dies wahr ist, dann nimmt der Server TCP/IP-Verbindungen entgegen. Ansonsten werden nur Verbindungen über Unix-Domain-Sockets angenommen. Die Voreinstellung ist aus. Diese Option kann nur beim Start des Servers eingestellt werden.

TIMEZONE (Zeichenkette)

Setzt die Zeitzone, die zur Ausgabe und Eingabe von Datums- und Zeitangaben verwendet wird. Einzelheiten finden Sie in Abschnitt 8.5. Die Voreinstellung ist die im Betriebssystem eingestellte Zeitzone.

TRANSFORM_NULL_EQUALS (Boolean)

Wenn an, dann werden Ausdrücke der Form ausdruck = NULL (oder NULL = ausdruck) wie ausdruck IS NULL behandelt, das heißt, sie ergeben wahr, wenn das Ergebnis von *ausdruck* der NULL-Wert ist, und ansonsten falsch. Das korrekte Verhalten von ausdruck = NULL ist immer den NULL-Wert (für unbekannt) zurückzugeben. Daher ist diese Option in der Voreinstellung aus.

Es scheint jedoch, dass gefilterte Formulare in Microsoft Access Anfragen erzeugen, die ausdruck = NULL verwenden, um nach NULL-Werten zu suchen. Wenn Sie also diese Anwendung verwenden, um auf Ihre Datenbank zuzugreifen, können Sie diese Option anschalten. Da Ausdrücke der Form expr = NULL immer den NULL-Wert ergeben (nach der korrekten Interpretation), sind sie nicht besonders nützlich und tauchen in normalen Anwendungen selten auf, und daher verursacht diese Option in der Praxis kaum Probleme. Aber weil neue Benutzer häufig Schwierigkeiten beim Verstehen von Ausdrücken mit NULL-Werten haben, ist diese Option in der Voreinstellung abgeschaltet, um die Verwirrung zu reduzieren.

Beachten Sie, dass diese Option sich nur auf den Operator = auswirkt, nicht aber auf andere Vergleichsoperatoren oder auf andere Ausdrücke, die irgendeinem Ausdruck mit dem Vergleichsoperator mathematisch gleichgestellt sind (wie zum Beispiel IN). Wenn Anfragen also grundsätzlich falsch programmiert worden sind, kann diese Option auch nicht weiterhelfen.

Weitere Informationen über verwandte Themen finden Sie in Abschnitt 9.2.

`UNIX_SOCKET_DIRECTORY` (Zeichenkette)

Gibt das Verzeichnis für die Unix-Domain-Socket an, auf der der Server auf Verbindungen von Clientanwendungen hören soll. Die Voreinstellung ist normalerweise `/tmp`, kann aber beim Compilieren geändert werden.

`UNIX_SOCKET_GROUP` (Zeichenkette)

Setzt die Gruppe, die Eigentümer der Unix-Domain-Socket sein soll. (Der Benutzer, der Eigentümer ist, ist immer der, der den Server gestartet hat.) Zusammen mit `UNIX_SOCKET_PERMISSIONS` kann das als zusätzlicher Zugriffskontrollmechanismus für diesen Sockettyp verwendet werden. Die Voreinstellung ist eine leere Zeichenkette, wodurch die Standardgruppe des aktuellen Benutzers verwendet wird. Diese Option kann nur beim Start des Servers eingestellt werden.

`UNIX_SOCKET_PERMISSIONS` (ganze Zahl)

Setzt die Zugriffsrechte der Unix-Domain-Sokket. Unix-Domain-Sockets verwenden die üblichen Zugriffsrechte für Unix-Dateien. Der Wert dieser Option muss ein numerischer Wert in der von den Systemaufrufen `chmod` und `umask` verwendeten Form sein. (Um das gebräuchliche Oktalformat zu verwenden, muss die Zahl mit `0` (einer Null) anfangen.)

Die Voreinstellung für die Zugriffsrechte ist `0777`, was bedeutet, dass jeder verbinden kann. Sinnvolle Alternativen sind `0770` (nur Benutzer und Gruppe, siehe auch unter `UNIX_SOCKET_GROUP`) und `0700` (nur Benutzer). (Beachten Sie, dass für eine Unix-Domain-Socket eigentlich nur die Schreibrechte von Bedeutung sind und das Setzen oder Entziehen der Lese- und Ausführrechte keinen Unterschied macht.)

Dieser Zugriffskontrollmechanismus ist unabhängig von dem in Kapitel 19 beschriebenen.

Diese Option kann nur beim Start des Servers eingestellt werden.

`VACUUM_MEM` (ganze Zahl)

Gibt die maximale Speichermenge an, die `VACUUM` verwenden darf, um sich freizugebende Zeilen zu merken. Der Wert ist in Kilobytes und die Voreinstellung ist 8192. Größere Werte können die `VACUUM`-Zeit bei großen Tabellen mit vielen gelöschten Zeilen verbessern.

`VIRTUAL_HOST` (Zeichenkette)

Gibt den Hostnamen oder die IP-Adresse an, auf der der Server auf Clientverbindungen hören soll. In der Voreinstellung wird auf allen konfigurierten Adressen (einschließlich `localhost`) gehört.

16.4.4 WAL

Einzelheiten zur Konfiguration von WAL finden Sie in Abschnitt 25.3.

`CHECKPOINT_SEGMENTS` (ganze Zahl)

Maximaler Abstand zwischen automatischen WAL-Checkpoints, in Logdateisegmenten (jedes normalerweise 16 Megabyte groß). Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden.

`CHECKPOINT_TIMEOUT` (ganze Zahl)

Maximale Zeit zwischen automatischen WAL-Checkpoints, in Sekunden. Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden.

`COMMIT_DELAY` (ganze Zahl)

Zeitverzögerung zwischen dem Schreiben eines Commit-Eintrags in den WAL-Puffer und dem Zurückschreiben des Puffers auf die Festplatte, in Mikrosekunden. Eine Verzögerung höher als null ermöglicht, dass mehrere Transaktionen mit nur einem `fsync`-Systemaufruf abgeschlossen werden

können, wenn die Systembelastung hoch genug ist, dass genug weitere Transaktionen in der angegebenen Zeitspanne zum Ende kommen könnten. Aber die Verzögerung ist Verschwendung, wenn keine weiteren Transaktion zum Abschluss bereit stehen. Daher wird die Verzögerung nur eingelegt, wenn mindestens `COMMIT_SIBLINGS` weitere Transaktionen in dem Moment, da der Serverprozess seinen Commit-Eintrag schreibt, aktiv sind.

`COMMIT_SIBLINGS` (ganze Zahl)

Anzahl gleichzeitiger Transaktionen, die aktiv sein müssen, bevor die `COMMIT_DELAY`-Verzögerung eingelegt wird. Ein größerer Wert erhöht die Wahrscheinlichkeit, dass zumindest eine weitere Transaktion während der Verzögerungsspanne zum Abschluss bereit wird.

`WAL_BUFFERS` (ganze Zahl)

Anzahl Diskseitenpuffer für WAL im Shared Memory. Diese Option kann nur beim Start des Servers eingestellt werden.

`WAL_DEBUG` (ganze Zahl)

Wenn verschieden von null, werden WAL-bezogenen Debuginformationen in den Serverlog geschrieben.

`WAL_SYNC_METHOD` (Zeichenkette)

Die Methode, um WAL-Änderung auf die Festplatte zu zwingen. Mögliche Werte sind `FSYNC` (nach jeder Transaktion `fsync()` aufrufen), `FDATASYNC` (nach jeder Transaktion `fdatasync()` aufrufen), `OPEN_SYNC` (WAL-Dateien mit der `open()`-Option *O_SYNC* schreiben) und `OPEN_DATASYNC` (WAL-Dateien mit der `open()`-Option *O_DSYNC* schreiben). Nicht alle diese Werte stehen auf allen Plattformen zur Verfügung. Diese Option kann nur beim Start des Servers oder in der Datei `postgresql.conf` eingestellt werden.

16.4.5 Kurze Optionen

Der Bequemlichkeit halber gibt es für einige Parameter auch äquivalente Kommandozeilenoptionen aus einem Buchstaben. Sie sind in Tabelle 16.2 beschrieben.

Kurze Option	Äquivalente Einstellung
`-B` *x*	`shared_buffers =` *x*
`-d` *x*	`server_min_messages = DEBUG`*x*
`-F`	`fsync = off`
`-h` *x*	`virtual_host =` *x*
`-i`	`tcpip_socket = on`
`-k` *x*	`unix_socket_directory =` *x*
`-l`	`ssl = on`
`-N` *x*	`max_connections =` *x*
`-p` *x*	`port =` *x*
`-fi`, `-fh`, `-fm`, `-fn`, `-fs`, `-ft`[a]	`enable_indexscan=off`, `enable_hashjoin=off`, `enable_mergejoin=off`, `enable_nestloop=off`, `enable_seqscan=off`, `enable_tidscan=off`
`-s`	`show_statement_stats = on`

Tabelle 16.2: Kurze Optionen

Kurze Option	Äquivalente Einstellung
`-S` *x*	`sort_mem` = *x*
`-tpa, -tpl, -te`	`show_parser_stats=on, show_planner_stats=on, show_executor_stats=on`

Tabelle 16.2: Kurze Optionen (Forts.)

a. Aus historischen Gründen müssen diese Optionen mit der `postmaster`-Option `-o` an die einzelnen Serverprozesse übergeben werden, zum Beispiel

```
$ postmaster -o '-S 1024 -s'
```

oder wie oben beschrieben mit `PGOPTIONS` von der Clientseite.

16.5 Verwaltung von Kernelressourcen

Eine große PostgreSQL-Installation kann schnell verschiedene Ressourcen des Betriebssystems ausreizen. (Auf einigen Systemen sind die Fabrikeinstellungen so niedrig, dass sich nicht einmal eine wirklich "große" Installation brauchen.) Wenn Sie auf ein derartiges Problem gestoßen sind, dann lesen Sie weiter.

16.5.1 Shared Memory und Semaphore

Shared Memory und Semaphore werden zusammenfassend als "`System V` IPC" bezeichnet (zusammen mit Message Queues, die für PostgreSQL keine Relevanz haben). Fast alle modernen Betriebssysteme bieten diese Fähigkeiten an, aber nicht alle haben sie in der Standardkonfiguration eingeschaltet oder groß genug eingestellt, besonders Systeme mit BSD-Erbe. (Für `QNX` und `BeOS` hat PostgreSQL eigene Ersatzimplementierungen dieser Schnittstellen.)

Das komplette Fehlen dieser Einrichtungen offenbart sich normalerweise durch einen `Illegal-system call`-Fehler, wenn der Server gestartet wird. In diesem Fall bleibt Ihnen nichts anderes übrig, als Ihren Kernel umzukonfigurieren. PostgreSQL kann ohne sie nicht arbeiten.

Wenn PostgreSQL eine der verschiedenen harten IPC-Grenzen überschreitet, wird der Server sich weigern zu starten und sollte eine detaillierte Fehlermeldung ausgeben, die das Problem beschreibt und erklärt, wie man es ausräumen kann. (Siehe auch Abschnitt 16.3.1.) Die betroffenen Kernelparameter werden auf den verschiedenen Systemen ziemlich einheitlich bezeichnet; Tabelle 16.3 gibt einen Überblick. Die Methoden, um sie einzustellen variieren jedoch sehr. Vorschläge für einige Plattformen finden Sie unten. Seien Sie aber gewarnt, dass es oft notwendig ist, Ihre Maschine neu zu booten oder eventuell sogar den Kernel neu zu kompilieren, um diese Einstellungen zu ändern.

Name	Beschreibung	Vernünftige Werte
`SHMMAX`	maximale Größe eines Shared-Memory-Segments (Bytes)	250 kB + 8.2 kB * `shared_buffers` + 14.2 kB * `max_connections` bis unendlich
`SHMMIN`	minimale Größe eines Shared-Memory-Segments (Bytes)	1
`SHMALL`	Gesamtgröße des verfügbaren Shared Memory (Bytes oder Seiten)	wenn Bytes, dann gleich `SHMMAX`; wenn Seiten, dann `ceil(SHMMAX/PAGE_SIZE)`
`SHMSEG`	maximale Anzahl von Shared-Memory-Segmenten pro Prozess	nur 1 Segment wird benötigt, aber die Voreinstellung ist viel höher
`SHMMNI`	maximale Anzahl von Shared-Memory-Segmenten im ganzen System	wie `SHMSEG` plus Platz für andere Anwendungen

Tabelle 16.3: `System V` IPC-Parameter

Name	Beschreibung	Vernünftige Werte
SEMMNI	maximale Anzahl von Semaphorbezeichnern (d.h. -gruppen)	mindestens `ceil(max_connections / 16)`
SEMMNS	maximale Anzahl von Semaphoren im ganzen System	`ceil(max_connections / 16) * 17` plus Platz für andere Anwendungen
SEMMSL	maximale Anzahl von Semaphoren pro Gruppe	mindestens 17
SEMMAP	Anzahl Einträge in Semaphorkarte	siehe Text
SEMVMX	Maximalwert eines Semaphors	mindestens 255 (Der Standard ist oft 32767; nicht ändern, wenn nicht nötig.)

Tabelle 16.3: `System V` IPC-Parameter (Forts.)

Der wichtigste Shared-Memory-Parameter ist SHMMAX, die Maximalgröße, in Bytes, eines Shared-Memory-Segments. Wenn Sie von `shmget` eine Fehlermeldung ähnlich `Invalid argument` sehen, dann ist es möglich, dass dieses Limit überschritten wurde. Die Größe des erforderlichen Segments hängt sowohl von der Zahl der gewünschten Puffer (Option `-B`) und der Zahl der erlaubten Verbindungen (Option `-N`) ab, obwohl Ersteres überwiegt. (Sie können als vorübergehende Maßnahme diese Einstellungen verringern, um die Fehler abzuschalten.) Als grobe Schätzung zur Größe des erforderlichen Segments können Sie die Zahl der Puffer mit der Blockgröße (Standard 8 kB) multiplizieren und reichlich Extraplatz (mindestens ein halbes Megabyte) dazu addieren. Wenn das Anlegen eines Segments scheitert, wird die Fehlermeldung auch die Größe des angeforderten Segments zur Information enthalten.

Ein weniger wahrscheinlicher Problemgrund ist die Minimalgröße eines Shared-Memory-Segments (SHMMIN), welche für PostgreSQL höchstens ca. 256 kB sein sollte (normalerweise ist sie einfach 1). Die Maximalzahlen für Segmente im ganzen System (SHMMNI) und pro Prozess (SHMSEG) sollten keine Probleme verursachen können, außer wenn Ihr System sie auch null gesetzt hat. Einige Systeme haben auch eine Grenze für die Gesamtgröße des Shared Memory in einem System; mehr dazu finden Sie in den plattformspezifischen Anweisungen unten.

PostgreSQL verwendet einen Semaphor pro erlaubte Verbindung (Option `-N`), in Gruppen von 16. Jede solche Gruppe enthält darüber hinaus einen 17. Semaphor, welcher eine "magische Zahl" enthält, die Konflikte mit von anderen Anwendungen verwendeten Semaphoren verhindern soll. Die Maximalzahl für Semaphore in einen System wird von SEMMNS gesetzt, was folglich mindestens so hoch wie die Verbindungseinstellung sein muss plus einen zusätzlichen pro 16 erlaubte Verbindungen (siehe Formel in Tabelle 16.3). Der Parameter SEMMNI bestimmt, wie viele Semaphorgruppen auf einmal im System existieren können. Daher muss dieser Parameter mindestens `ceil(max_connections / 16)` sein. Bei Fehlern, die meistens die verwirrende Formulierung `No space left on device` verwenden und von der Funktion `semget` kommen, kann als Ausweichlösung die Zahl der erlaubten Verbindungen heruntergesetzt werden.

In manchen Fällen kann es auch notwendig sein, SEMMAP so zu erhöhen, dass es etwa auf Höhe von SEMMNS ist. Dieser Parameter bestimmt die Größe der Semaphorressourcenkarte, in der jeder zusammenhängende Block aus freien Semaphoren einen Eintrag benötigt. Wenn eine Semaphorgruppe freigegeben wird, wird sie entweder zu einem bestehenden Eintrag, der an den freigegebenen Block angrenzt, hinzugefügt oder sie wird unter einem neuen Eintrag registriert. Wenn die Karte voll ist, sind die freigegebenen Semaphore verloren (bis zum Reboot). Die Fragmentierung des Semaphorraums kann also mit der Zeit dazu führen, dass weniger Semaphore zur Verfügung stehen als eigentlich sollten.

Der Parameter SEMMSL, der bestimmt, wie viele Semaphore in einer Gruppe sein können, muss für PostgreSQL mindestens 17 sein.

Verschiedene Einstellungen, die sich auf *Semaphore Undo* beziehen, wie SEMMNU und SEMUME, sind für PostgreSQL nicht von Bedeutung.

BSD/OS

Shared Memory. In der Voreinstellung werden nur 4 MB Shared Memory unterstützt. Bedenken Sie, dass Shared Memory nicht pagebar ist; es sitzt fest im RAM. Um die vom System unterstützte Shared-Memory-Größe zu erhöhen, fügen Sie Folgendes zu Ihrer Kernelkonfigurationsdatei hinzu. Ein SHMALL-Wert von 1024 ergibt 4 MB Shared Memory. Folgendes erhöht den maximalen Shared-Memory-Bereich auf 32 MB:

```
options "SHMALL=8192"
options "SHMMAX=\(SHMALL*PAGE_SIZE\)"
```

Wenn Sie Version 4.1 oder später haben, führen Sie obige Änderungen durch, kompilieren Sie den Kernel neu und booten Sie.

Wenn Sie frühere Versionen haben, verwenden Sie `bpatch,` um den `sysptsize`-Wert im aktuellen Kernel zu finden. Der wird beim Booten dynamisch berechnet.

```
$ bpatch -r sysptsize
0x9 = 9
```

Als Nächstes fügen Sie einen fixen `SYSPTSIZE`-Wert in die Kernelkonfigurationsdatei ein. Erhöhen Sie den mit `bpatch` gefundenen Wert. Wenn Sie 1 addieren, fügen Sie jeweils 4 MB Shared Memory hinzu.

```
options "SYSPTSIZE=16"
```

`sysptsize` kann nicht mit `sysctl` geändert werden.

Semaphore. Sie müssen eventuell die Zahl der Semaphore erhöhen. In der Voreinstellung benötigt PostgreSQL 34 Semaphore, was über die Hälfte des Systemvorrats von 60 ist. Setzen Sie die gewünschten Werte in Ihrer Kernelkonfigurationsdatei, zum Beispiel:

```
options "SEMMNI=40"
options "SEMMNS=240"
```

FreeBSD,

NetBSD,

OpenBSD

Die Optionen `SYSVSHM` und `SYSVSEM` müssen angeschaltet sein, wenn der Kernel kompiliert wird. (In der Voreinstellung ist das der Fall.) Die Maximalgröße des Shared Memory wird von der Option `SHMMAXPGS` (in Seiten) bestimmt. Folgendes ist ein Beispiel, wie die verschiedenen Parameter gesetzt werden:

```
options         SYSVSHM
options         SHMMAXPGS=4096
options         SHMSEG=256

options         SYSVSEM
options         SEMMNI=256
options         SEMMNS=512
options         SEMMNU=256
options         SEMMAP=256
```

(Auf `NetBSD` und `OpenBSD` ist das Schlüsselwort `option` in der Einzahl.)

Sie können auch `sysctl` verwenden, um das Shared Memory im RAM festzuhalten und zu verhindern, dass es ausgeswappt wird.

`HP-UX`

Die Voreinstellungen reichen für normale Installationen. Auf HP-UX 10 ist die Voreinstellung von `SEMMNS` 128, was für größere Datenbanken zu niedrig sein könnte.

IPC-Parameter können im `System Administration Manager` (SAM) unter Kernel ConfigurationConfigurable Parameters eingestellt werden. Wählen Sie `Create A New Kernel`, wenn Sie fertig sind.

`Linux`

Die voreingestellte Shared-Memory-Grenze (sowohl `SHMMAX` als auch `SHMALL`) ist 32 MB in Kernels der Serie 2.2, aber sie kann im `proc`-Dateisystem geändert werden (ohne Reboot). Um zum Beispiel 128 MB zu erlauben:

```
$ echo 134217728 >/proc/sys/kernel/shmall
$ echo 134217728 >/proc/sys/kernel/shmmax
```

Sie könnten diese Befehle in ein Skript stellen, das beim Booten ausgeführt wird.

Alternativ können Sie, wenn vorhanden, `sysctl` verwenden, um diese Parameter zu kontrollieren. Suchen Sie die Datei `/etc/sysctl.conf` und fügen Sie diese oder ähnliche Zeilen ein:

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

Diese Datei wird üblicherweise beim Booten verarbeiten, aber `sysctl` kann auch später noch aufgerufen werden.

Die anderen Parameter sind für alle Anwendungen ausreichend dimensioniert. Wenn Sie selbst nachsehen wollen, schauen Sie in `/usr/src/linux/include/asm-xxx/shmpara m.h` und `/usr/src/linux/include/linux/sem.h`.

`MacOS X`

Bearbeiten Sie die Datei `/System/Library/StartupItems/SystemTuning/SystemTuning` und ändern Sie die folgenden Werte:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

`SCO OpenServer`

In der Voreinstellung werden nur 512 kB Shared Memory unterstützt, was etwa für `-B 24 -N 12` reicht. Um die Einstellung zu ändern, wechseln Sie zuerst in das Verzeichnis `/etc/conf/cf.d`. Um den aktuellen Wert von `SHMMAX` anzuzeigen, führen Sie aus:

```
./configure -y SHMMAX
```

Um einen neuen Wert für `SHMMAX` zu setzen, führen Sie aus:

```
./configure SHMMAX=wert
```

wo *wert* der neue Wert, den Sie verwenden wollen, ist (in Bytes). Nachdem Sie `SHMMAX` gesetzt haben, bauen Sie den Kernel neu:

```
./link_unix
```

und booten Sie neu.

`Solaris`

Zumindest in Version 2.6 ist die voreingestellte Maximalgröße eines Shared-Memory-Segments zu niedrig für PostgreSQL. Die relevanten Einstellungen können in der Datei `/etc/system` geändert werden, zum Beispiel:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

Um die Änderungen zu aktivieren, müssen Sie neu booten.

In finden Sie Informationen über Shared Memory auf Solaris.

`UnixWare`

Auf UnixWare 7 ist die Maximalgröße für Shared-Memory-Segmente in der Standardeinstellung 512 kB. Das reicht etwa für `-B 24 -N 12`. Um den aktuellen Wert von `SHMMAX` anzuzeigen, führen Sie aus:

```
/etc/conf/bin/idtune -g SHMMAX
```

was den aktuellen, den voreingestellten, den minimalen und den maximalen Wert ausgibt. Um einen neuen Wert für `SHMMAX` zu setzen, führen Sie aus:

```
/etc/conf/bin/idtune SHMMAX wert
```

wo *wert* der neue Wert, den Sie verwenden wollen, ist (in Bytes). Nachdem Sie `SHMMAX` gesetzt haben, bauen Sie den Kernel neu:

```
/etc/conf/bin/idbuild -B
```

und booten Sie neu.

16.5.2 Ressourcenbegrenzungen

Unix-ähnliche Betriebssysteme haben verschiedene Arten von Ressourcenbegrenzungen, die den Betrieb Ihres PostgreSQL-Servers stören könnten. Besonders wichtig sind die Begrenzungen der Anzahl der Prozesse pro Benutzer, der Anzahl der offenen Dateien pro Prozess und des für jeden Prozess zur Verfügung stehenden Speichers. Jeder dieser Werte hat ein “hartes” und ein “weiches” Limit. Das weiche Limit ist das wirklich geltende Limit, aber es kann vom Benutzer bis zum harten Limit hoch gesetzt werden. Das harte Limit kann nur vom `root`-Benutzer verändert werden. Der Systemaufruf `setrlimit` ist für des Setzen

dieser Parameter verantwortlich. Der eingebaute Shell-Befehl `ulimit` (Bourne-Shells) bzw. `limit` (`csh`) wird zur Kontrolle dieser Ressourcenbegrenzungen auf der Kommandozeile verwendet. Auf Systemen aus der BSD-Familie kontrolliert die Datei `/etc/login.conf` die Ressourcenbegrenzungen beim Login. Einzelheiten finden Sie in der Betriebssystemdokumentation. Die relevanten Parameter sind `maxproc`, `openfiles` und `datasize`. Zum Beispiel:

```
default:\
...
            :datasize-cur=256M:\
            :maxproc-cur=256:\
            :openfiles-cur=256:\
...
```

(`-cur` ist das weiche Limit. Hängen Sie `-max` an, um das harte Limit zu setzen.)

Kernel können für einige Ressourcen auch systemweite Begrenzungen haben.

- Auf Linux bestimmt `/proc/sys/fs/file-max`, wie viele offene Dateien der Kernel maximal unterstützen kann. Man kann den Wert verändern, indem man einen neuen Wert in die Datei schreibt oder in dem man einen Eintrag in der Datei `/etc/sysctl.conf` macht. Die Höchstgrenze für Dateien pro Prozess wird beim Compilieren des Kernels festgelegt. Weitere Einzelheiten finden Sie in `/usr/src/linux/Documentation/proc.txt`.

Der PostgreSQL-Server verwendet einen Prozess pro Verbindung, also sollten Sie für mindestens so viele Prozesse Platz schaffen, wie sie Verbindungen zulassen, neben dem, was Sie für den Rest des Systems brauchen. Das ist normalerweise kein Problem, aber wenn Sie mehrere Server auf einer Maschine laufen haben, dann könnte es eng werden.

Die Voreinstellung für die Höchstzahl offener Dateien ist oft ein "sozialverträglicher" Wert, der es ermöglicht, dass viele Benutzer eine Maschine nebeneinander verwenden können, ohne einen ungebührlich hohen Anteil der Systemressourcen zu verbrauchen. Wenn Sie viele Server auf einer Maschine haben, dann wollen Sie das vielleicht so, aber bei dedizierten Servern sollten Sie die Begrenzung vielleicht erhöhen.

Die Kehrseite der Medaille ist, dass einige Systeme einzelnen Prozessen erlauben, eine große Anzahl von Dateien zu öffnen, und wenn mehr als ein paar Prozesse das tun, dann kann die systemweite Grenze leicht überschritten werden. Wenn Ihnen das passiert und Sie das systemweite Limit nicht ändern wollen, können Sie in PostgreSQL den Konfigurationsparameter `max_files_per_process` setzen, um den Verbrauch von offenen Dateien einzuschränken.

16.6 Den Server herunterfahren

Man kann den Datenbankserver auf verschiedene Arten herunterfahren. Sie kontrollieren die Art des Herunterfahrens, indem Sie unterschiedliche Signale an den Serverprozess senden.

`SIGTERM`

> Nach Erhalt von `SIGTERM` erlaubt der Server keine neuen Verbindungen, aber bestehenden Sitzungen, ihre Arbeit normal zu beenden. Er fährt erst herunter, nachdem alle Sitzungen normal beendet wurden. Das nennt sich **Smart Shutdown**.

`SIGINT`

> Der Server erlaubt keine neuen Verbindungen und sendet an alle bestehenden Serverprozesse das Signal `SIGTERM`, was dazu führt, dass diese die aktuelle Transaktionen abbrechen und zügig beenden. Dann wartet der Server, bis alle Serverprozesse beendet sind, und fährt dann herunter. Das nennt sich **Fast Shutdown**.

`SIGQUIT`

Dies ist der **Immediate Shutdown**, der daraus besteht, dass der `postmaster`-Prozess an alle Serverprozesse das Signal `SIGQUIT` sendet und sich dann selbst sofort beendet (ohne selbst ordentlich herunterzufahren). Die Kindprozesse beenden gleichermaßen sofort nach dem Erhalt von `SIGQUIT`. Das führt beim Neustart zu einem Wiederherstellungsdurchlauf (indem der WAL-Log abgespielt wird). Das ist nur in Notfällen zu empfehlen.

Wichtig

Man sollte am besten `SIGKILL` nicht zum Herunterfahren des Servers verwenden. Das verhindert, dass der Server Shared Memory und Semaphore freigibt, was man dann selbst machen müsste.

Die PID des `postmaster`-Prozesses kann mit dem Programm `ps` herausgefunden oder in der Datei `postmaster.pid` im Datenverzeichnis gefunden werden. Um also zum Beispiel einen *Fast Shutdown* zu machen:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

Das Programm `pg_ctl` ist ein Shell-Skript, dass das Herunterfahren des Servers vereinfacht.

16.7 Sichere TCP/IP-Verbindungen mit SSL

PostgreSQL hat eingebaute Unterstützung für SSL-Verbindungen, um die Client/Server-Kommunikation für verbesserte Sicherheit zu verschlüsseln. Das erfordert, dass OpenSSL auf den Client- und Serversystemen installiert ist und dass die Unterstützung in PostgreSQL beim Compilieren angeschaltet wurde (siehe Kapitel 14).

Wenn die SSL-Unterstützung vorhanden ist, kann der PostgreSQL-Server mit SSL-Unterstützung gestartet werden, indem der Parameter `ssl` in `postgresql.conf` auf an gesetzt wird. Wenn er im SSL-Modus gestartet wird, sucht der Server die Dateien `server.key` und `server.crt` im Datenverzeichnis, welche den privaten Schlüssel bzw. das Zertifikat des Servers enthalten sollten. Diese Dateien müssen eingerichtet werden, bevor ein Server im SSL-Modus gestartet werden kann. Wenn der private Schlüssel durch eine Passphrase geschützt ist, wird der Server um Eingabe der Passphrase bitten und erst starten, wenn sie eingegeben wurde.

Der Server wartet auf normale und auf SSL-Verbindungen auf demselben TCP-Port und verhandelt mit verbindenden Clients, ob SSL verwendet werden soll. In Kapitel 19 wird beschrieben, wie ein Server die Verwendung von SSL auf bestimmten Verbindungen erzwingen kann.

Einzelheiten, wie Sie für Ihren Server einen privaten Schlüssel und Zertifikate erzeugen, finden Sie in der Anleitung zu OpenSSL. Um zum Ausprobieren einen Einstieg zu erhalten, kann man ein einfaches selbstsigniertes Zertifikat verwenden, aber im richtigen Betrieb sollte ein von einer Certificate Authority (CA) (entweder eine der globalen CAs oder eine lokale) signiertes Zertifikat verwendet werden, damit der Client die Identität des Servers überprüfen kann. Um schnell ein selbstsigniertes Zertifikat zu erzeugen, verwenden Sie folgenden OpenSSL-Befehl:

```
openssl req -new -text -out server.req
```

Füllen Sie die Informationen, nach denen `openssl` fragt, aus. Achten Sie darauf, dass Sie den lokalen Hostnamen als “Common Name” eingeben; das Passwort kann leer gelassen werden. Das Programm erzeugt einen Schlüssel, der mit einer Passphrase geschützt is,t und akzeptiert keine Passphrase, die kürzer

als vier Zeichen ist. Um die Passphrase zu entfernen (was Sie müssen, wenn Sie den Server automatisch starten lassen wollen), führen Sie die Befehle

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

aus. Geben Sie die alte Passphrase ein, um den bestehenden Schlüssel zu entsperren. Jetzt machen Sie

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
chmod og-rwx server.key
```

um aus dem Zertifikat ein selbstsigniertes zu machen und Schlüssel und Zertifikat dorthin zu kopieren, wo der Server danach suchen wird.

16.8 Sichere TCP/IP-Verbindungen mit SSH-Tunneln

Man kann SSH verwenden, um die Netzwerkverbindung zwischen Clients und einem PostgreSQL-Server zu verschlüsseln. Wenn das richtig gemacht wird, führt es zu einer ausreichend sicheren Netzwerkverbindung.

Als Erstes sorgen Sie dafür, dass auf der Maschine mit dem PostgreSQL-Server ein SSH-Server läuft und dass Sie sich mit `ssh` als irgendein Benutzer einloggen können. Dann können Sie von der Clientmaschine mit einem Befehl wie diesem einen sicheren Tunnel erzeugen:

```
ssh -L 3333:foo.com:5432 joe@foo.com
```

Die erste Zahl im `-L`-Argument, 3333, ist die Portnummer von Ihrem Tunnelende; sie kann frei gewählt werden. Die zweite Zahl, 5432, ist das entfernte Ende des Tunnels: die Portnummer, die der Server verwendet. Der Name oder die Adresse zwischen den Portnummern ist der Host mit dem Datenbankserver, mit dem Sie verbinden wollen. Wenn Sie durch diesen Tunnel mit dem Datenbankserver verbinden wollen, dann verbinden Sie mit Port 3333 auf der lokalen Maschine:

```
psql -h localhost -p 3333 template1
```

Für den Datenbankserver sieht es dann so aus, als ob Sie in Wirklichkeit der Benutzer `joe@foo.com` sind, und er wird dann die für diesen Benutzer eingerichtete Authentifizierungsprozedur anwenden. Damit die Tunneleinrichtung erfolgreich sein kann, muss es Ihnen möglich sein, mit `ssh` als `joe@foo.com` zu verbinden, genauso, als ob Sie mit `ssh` versuchen würden, eine Terminalsitzung einzurichten.

Tipp

Es gibt auch andere Programme, mit denen sichere Tunnel auf eine ähnliche Art und Weise wie die eben beschriebene eingerichtet werden können.

17

Datenbankbenutzer und Privilegien

Jeder Datenbankcluster enthält Datenbankbenutzer. Diese Benutzer sind getrennt von den Benutzern, die das Betriebssystem, auf dem der Server läuft, verwaltet. Benutzer sind die Eigentümer von Datenbankobjekten (zum Beispiel Tabellen) und können anderen Benutzern Privilegien für diese Objekte zuweisen, um zu kontrollieren, wer auf welches Objekt Zugriff hat.

Dieses Kapitel beschreibt, wie man Benutzer erzeugt und verwaltet, und führt Sie in das Privilegiensystem ein. Weitere Informationen über die verschiedenen Arten von Datenbankobjekten und die Auswirkungen von Privilegien können Sie in Kapitel 5 finden.

17.1 Datenbankbenutzer

Datenbankbenutzer sind vollständig getrennt von Betriebssystembenutzern. Es mag praktisch sein, eine Übereinstimmung zu erhalten, aber das ist nicht erforderlich. Datenbankbenutzer gelten global in der gesamten Datenbankclusterinstallation (und nicht nur in einzelnen Datenbanken). Um einen Benutzer zu erzeugen, verwenden Sie den SQL-Befehl `CREATE USER`:

```
CREATE USER name;
```

name folgt den Namen für SQL-Bezeichner: entweder ohne Verzierung und Sonderzeichen oder in Anführungszeichen. Um einen bestehenden Benutzer zu entfernen, verwenden Sie den analogen Befehl `DROP USER`:

```
DROP USER name;
```

Der Bequemlichkeit halber gibt es die Programme `createuser` und `dropuser`, die vom Betriebssystem aus aufgerufen werden können und im Prinzip nur die entsprechenden SQL-Befehle ausführen:

```
createuser name
dropuser name
```

Um das Datenbanksystem überhaupt starten zu können, enthält ein frisch initialisiertes System immer einen vordefinierten Benutzer. Dieser Benutzer hat die festgelegte ID 1 und hat in der Voreinstellung (wenn nicht durch `initdb` geändert) den gleichen Namen wie der Betriebssystembenutzer, der den

Datenbankcluster initialisierte. Traditionell heißt dieser Benutzer `postgres`. Um weitere Benutzer zu erzeugen, müssen Sie sich erst mit diesem Benutzer anmelden.

Auf jeder Verbindung zum Datenbankserver ist genau eine Benutzeridentität aktiv. Der Benutzername, der für eine bestimmte Datenbankverbindung verwendet werden soll, wird von dem Client, der die Verbindungsanfrage einleitet, je nach Art der Anwendung angegeben. Zum Beispiel wird beim Programm `psql` der Benutzer mit der Option `-U` angegeben. Viele Anwendungen nehmen den Namen des aktuellen Betriebssystembenutzers als Vorgabe für den Namen des gewünschten Datenbankbenutzers (einschließlich `createuser` und `psql`). Daher kann es praktisch sein, wenn es eine Namensübereinstimmung zwischen diesen beiden Benutzerwelten gibt.

Welche Datenbankbenutzer ein bestimmter Client bei der Verbindung verwenden darf, wird von der Konfiguration der Client-Authentifizierung bestimmt, wie in Kapitel 19 beschrieben. (Ein Client muss also nicht mit dem gleichen Namen wie dem Betriebssystembenutzernamen verbinden, genauso wie der Betriebssystemname nicht mit dem richtigen Namen einer Person übereinstimmen muss.) Da die Benutzeridentität das Ausmaß der dem verbundenen Client zur Verfügung stehenden Privilegien bestimmt, ist es wichtig, dies in einer Mehrbenutzerumgebung richtig zu konfigurieren.

17.2 Benutzerattribute

Ein Datenbankbenutzer kann eine Anzahl von Attributen haben, die seine Privilegien bestimmen und Auswirkung auf das Clientauthentifizierungs-System haben.

Superuser

> Ein Datenbank-Superuser umgeht alle Rechteprüfungen. Außerdem kann nur ein Superuser neue Benutzer erzeugen. Um einen Datenbank-Superuser zu erzeugen, verwenden Sie `CREATE USER name CREATEUSER`.

Erzeugung von Datenbanken

> Einem Benutzer muss die Erlaubnis, Datenbanken erzeugen zu dürfen, ausdrücklich erteilt werden (außer bei Superusern, da diese alle Rechteprüfungen umgehen). Um einen solchen Benutzer zu erzeugen, verwenden Sie `CREATE USER name CREATEDB`.

Passwort

> Ein Passwort ist nur von Bedeutung, wenn die Clientauthentifizierungs-Methode verlangt, dass der Benutzer beim Verbinden zur Datenbank ein Passwort angibt. Die Authentifizierungsmethoden `password`, `md5` und `crypt` verwenden Passwörter. Datenbankpasswörter sind getrennt von Betriebssystempasswörtern. Um ein Passwort bei der Benutzererzeugung anzugeben, verwenden Sie `CREATE USER name PASSWORD 'zeichenkette'`.

Die Attribute eines Benutzers können nach der Erzeugung mit `ALTER USER` geändert werden. Sehen Sie sich die Referenzseiten von `CREATE USER` und `ALTER USER` wegen Einzelheiten an.

Ein Benutzer kann auch persönliche Voreinstellungen für die in Abschnitt 16.4 beschriebenen Konfigurationseinstellungen machen. Wenn Sie zum Beispiel aus irgendeinem Grund Indexscans immer abschalten wollen, wenn Sie verbinden (Tipp: keine gute Idee), dann können Sie folgenden Befehl verwenden:

```
ALTER USER meinname SET enable_indexscan TO off;
```

Das speichert die Einstellung (aber aktiviert sie nicht sofort) und bei den folgenden Verbindungen wird es den Anschein haben, dass `SET enable_indexscan TO off;` direkt vor Beginn der Sitzung ausgeführt wurde. Sie können die Einstellung auch während der Sitzung noch ändern; sie ist nur die Vorgabe. Um eine solche Einstellung rückgängig zu machen, verwenden Sie `ALTER USER benutzername RESET varname;`.

17.3 Gruppen

Wie in Unix können durch Gruppen Benutzer logisch zusammengefasst werden, um die Verwaltung von Privilegien zu vereinfachen: Privilegien können einer Gruppe als Ganzes gewährt oder entzogen werden. Um eine Gruppe zu erzeugen, verwenden Sie

```
CREATE GROUP name;
```

Um Benutzer einer Gruppe hinzuzufügen, verwenden Sie

```
ALTER GROUP name ADD USER bname1, ... ;
ALTER GROUP name DROP USER bname1, ... ;
```

17.4 Privilegien

Wenn ein Datenbankobjekt erzeugt wird, dann wird ihm ein Eigentümer zugewiesen. Der Eigentümer ist der Benutzer, der den Befehl zur Erzeugung ausgeführt hat. Um den Eigentümer einer Tabelle, eines Index, einer Sequenz oder einer Sicht zu ändern, verwenden Sie den Befehl `ALTER TABLE`. In der Voreinstellung kann nur der Eigentümer (oder ein Superuser) etwas mit dem Objekt anstellen. Um es anderen Benutzern zu erlauben, das Objekt zu benutzen, müssen **Privilegien** gewährt werden.

Es gibt mehrere verschiedene Privilegien: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE`, `USAGE` und `ALL PRIVILEGES`. Weitere Informationen über die verschiedenen von PostgreSQL unterstützen Privilegientypen finden Sie auf der Seite für `GRANT` in Teil VI. Das Recht, ein Objekt zu verändern oder zu löschen, ist immer nur das Privileg des Eigentümers. Um Privilegien zuzuweisen wird der Befehl `GRANT` benutzt. Wenn also `joe` ein bestehender Benutzer ist und `konten` eine bestehende Tabelle, kann das Privileg, die Tabelle aktualisieren zu dürfen (*update*), so gewährt werden:

```
GRANT UPDATE ON konten TO joe;
```

Der Benutzer, der diesen Befehl ausführt, muss Eigentümer der Tabelle sein. Um ein Privileg einer Gruppe zu gewähren, verwenden Sie

```
GRANT SELECT ON konten TO GROUP mitarbeiter;
```

Der spezielle “Benutzername” `PUBLIC` kann verwendet werden, um ein Privileg allen Benutzern im System zu gewähren. Wenn man `ALL` anstelle eines spezifischen Privilegs schreibt, werden alle Privilegien gewährt.

Um ein Privileg zu entziehen, verwenden Sie den Befehl `REVOKE`:

```
REVOKE ALL ON konten FROM PUBLIC;
```

Die besonderen Privilegien des Tabelleneigentümers (d.h. das Recht `DROP`, `GRANT`, `REVOKE` usw. auszuführen) liegt immer automatisch beim Eigentümer und können nicht weitergegeben oder entzogen werden. Aber der Tabelleneigentümer kann sich seine eigenen normalen Privilegien entziehen, um zum Beispiel zu verhindern, dass er selbst aus Versehen in die Tabelle schreibt.

17.5 Funktionen und Trigger

Durch Funktionen und Trigger können Benutzer in den Server Code einfügen, den andere Benutzer ausführen könnten, ohne es zu wissen. Folglich können Benutzer auf beide Arten relativ ungestraft “Trojanische Pferde” erzeugen. Der einzige richtige Schutz ist eine scharfe Kontrolle darüber, wer Funktionen erzeugen darf.

Funktionen, die in einer Sprache außer SQL geschrieben wurden, laufen im Serverprozess mit den Betriebssystemzugriffsrechten des Datenbankservers. Beliebige Funktionen könnten also die internen Datenstrukturen des Servers verändern. Sie könnten also unter anderem die Zugriffskontrollen des Systems umgehen. Dieses Problem liegt in der Natur von C-Funktionen.

18

Datenbanken verwalten

Jede Instanz eines laufenden PostgreSQL-Servers verwaltet eine oder mehrere Datenbanken. Datenbanken sind daher das oberste hierarchische Niveau für die Organisation von SQL-Objekten (“Datenbankobjekten”). Dieses Kapitel beschreibt die Eigenschaften von Datenbanken und wie sie erzeugt, verwaltet und zerstört werden.

18.1 Überblick

Eine Datenbank ist eine benannte Sammlung von SQL-Objekten (“Datenbankobjekten”). Generell gehört jedes Datenbankobjekt (Tabellen, Funktionen usw.) in genau eine Datenbank. (Es gibt aber einige Systemkataloge, zum Beispiel `pg_database`, die zum gesamten Cluster gehören und auf die man von jeder Datenbank des Clusters zugreifen kann.) Genauer gesagt ist eine Datenbank eine Sammlung von Schemas und diese enthalten die Tabellen, Funktionen usw. Die gesamte Hierarchie ist also: Server, Datenbank, Schema, Tabelle (oder etwas anderes anstatt Tabelle).

Eine Anwendung, die mit dem Datenbankserver verbindet, gibt in ihrem Verbindungsgesuch den Namen der Datenbank an, mit der sie verbinden will. Es ist nicht möglich, auf mehrere Datenbanken mit einer Verbindung zuzugreifen. (Aber eine Anwendung hat keine Beschränkung hinsichtlich der Zahl der Verbindungen, die sie zur selben oder zu anderen Datenbanken öffnen kann.) Es ist jedoch möglich, auf mehrere Schemas von derselben Verbindung aus zuzugreifen. Schemas sind eine rein logische Struktur, wo die Zugriffsrechte durch das Privilegiensystem geregelt werden. Datenbanken sind physikalisch getrennt und die Zugriffsrechte werden auf der Verbindungsebene geregelt. Wenn eine PostgreSQL-Serverinstanz Projekte oder Benutzer beherbergen soll, die getrennt sind und im Großen und Ganzen die anderen nicht sehen sollen, ist es daher empfehlenswert, ihnen getrennte Datenbanken zu geben. Wenn die Projekte oder Benutzer zusammenhängen und die Ressourcen der anderen mitbenutzen sollen, dann sollten Sie dieselbe Datenbank, aber möglicherweise getrennte Schemas verwenden. Weitere Informationen über die Verwaltung von Schemas finden Sie in Abschnitt 5.7.

Anmerkung

SQL nennt Datenbanken “Kataloge”, aber das macht in der Praxis keinen Unterschied.

18.2 Eine Datenbank erzeugen

Um eine Datenbank erzeugen zu können, muss der PostgreSQL-Server gestartet sein und laufen (siehe Abschnitt 16.3).

Datenbanken werden mit dem SQL-Befehl `CREATE DATABASE` erzeugt:

```
CREATE DATABASE name;
```

wo *name* den normalen Regeln für SQL-Bezeichner folgt. Der aktuelle Benutzer wird automatisch der Eigentümer der neuen Datenbank. Es ist das Privileg des Eigentümers, die Datenbank später wieder entfernen zu können (wodurch alle darin enthaltenen Objekte mit entfernt werden, auch wenn sie einen anderen Eigentümer haben).

Die Erzeugung von Datenbanken ist eine beschränkte Operation. Informationen, wie Sie die Erlaubnis dafür erteilen können, finden Sie in Abschnitt 17.2.

Da man mit dem Datenbankserver verbunden sein muss, um den Befehl `CREATE DATABASE` auszuführen, bleibt die Frage, wie die *erste* Datenbank eines Clusters erzeugt werden kann. Die erste Datenbank wird immer mit dem Befehl `initdb` erzeugt, wenn der Bereich zur Datenspeicherung initialisiert wird. (Siehe Kapitel 16.2.) Diese Datenbank heißt `template1`. Um also die erste "richtige" Datenbank zu erzeugen, können Sie mit `template1` verbinden.

Der Name `template1` ist kein Zufall (*template* = Vorlage): Wenn eine neue Datenbank erzeugt wird, dann wird die Templatedatenbank im Prinzip geklont. Das bedeutet, dass Änderungen, die Sie in `template1` vornehmen, an alle danach erzeugten Datenbanken weitergegeben werden. Daraus folgt, dass Sie die Templatedatenbank nicht für die eigentliche Arbeit verwenden sollten, aber bei vernünftiger Verwendung kann dieses Klonen ziemlich nützlich sein. Weitere Einzelheiten finden Sie in Abschnitt 18.3.

Als zusätzliche Annehmlichkeit gibt es auch ein Programm, das Sie von der Shell aus ausführen können, um neuen Datenbanken zu erzeugen, `createdb`.

```
createdb dbname
```

`createdb` kann auch nicht zaubern. Es verbindet mit der Datenbank `template1` und führt den Befehl `CREATE DATABASE` aus, genau wie oben beschrieben. Intern verwendet es dazu das Programm `psql`. Die Referenzseite zu `createdb` enthält Einzelheiten, wie man es aufruft. Beachten Sie, dass `createdb` ohne Argumente eine Datenbank mit dem Namen des aktuellen Benutzers erzeugt, was Sie vielleicht beabsichtigen oder auch nicht.

Anmerkung

Kapitel 19 enthält Informationen darüber, wie man einschränken kann, wer mit einer bestimmten Datenbank verbinden kann.

Manchmal wollen Sie eine Datenbank für jemanden anders erzeugen. Jener Benutzer sollte der Eigentümer der Datenbank werden, sodass er sie selbst konfigurieren und verwalten kann. Um das zu erreichen, verwenden Sie einen der folgenden Befehle:

```
CREATE DATABASE dbname OWNER benutzername;
```

in der SQL-Umgebung oder

```
createdb -O benutzername dbname
```

Sie müssen ein Superuser sein, um eine Datenbank für jemanden anders erzeugen zu dürfen.

18.3 Template-Datenbanken

`CREATE DATABASE` funktioniert eigentlich, indem es eine bestehende Datenbank kopiert. Wenn nichts anderes angegeben wurde, kopiert es die Systemdatenbank `template1`. Diese Datenbank ist also das "Template" (englisch für Vorlage), aus dem neue Datenbanken erzeugt werden. Wenn Sie in `template1` Objekte erzeugt haben, werden diese Objekte in anschließend erzeugte Benutzerdatenbanken kopiert. Durch dieses Verhalten können Datenbankadministratoren weitere Objekte zu den in allen Datenbanken vorhandenen Systemobjekten hinzufügen. Wenn Sie zum Beispiel die prozedurale Sprache `PL/pgSQL` in `template1` installieren, wird sie automatisch in allen Benutzerdatenbanken verfügbar sein, ohne dass man noch etwas tun muss, wenn diese Datenbanken erzeugt werden.

Es gibt eine zweite Systemdatenbank: `template0`. Diese Datenbank hat am Anfang den gleichen Inhalt wie `template1`, das heißt nur die von Ihrer PostgreSQL-Version vordefinierten Standardobjekte. `template0` sollte nach `initdb` nicht verändert werden. Wenn Sie `CREATE DATABASE` anweisen, `template0` statt `template1` zu kopieren, erzeugen sie quasi eine jungfräuliche Datenbank, die keine der lokalen Veränderungen in `template1` enthält. Das ist besonders hilfreich, wenn Sie eine mit `pg_dump` erzeugte Sicherungsdatei wiederherstellen: Die gesicherte Datenbank sollte in eine unveränderte "leere" Datenbank wiederhergestellt werden, damit man nicht mit eventuellen Modifikationen in `template1` Konflikte erzeugt.

Um eine Datenbank durch Kopieren von `template0` zu erzeugen, verwenden Sie

```
CREATE DATABASE dbname TEMPLATE template0;
```

in der SQL-Umgebung, oder

```
createdb -T template0 dbname
```

aus der Shell.

Man kann auch weitere Templatedatenbanken erzeugen, und in der Tat kann man auch eine beliebige Datenbank eines Clusters kopieren, indem man ihren Namen bei `CREATE DATABASE` als Template angibt. Man muss allerdings unbedingt verstehen, dass das (noch) keine generell anwendbare Methode zum Kopieren von Datenbanken ist. Insbesondere ist es unerlässlich, dass die Quelldatenbank während des gesamten Kopiervorgangs unbenutzt ist (keine datenverändernden Transaktionen aktiv). `CREATE DATABASE` prüft, ob am Anfang des Vorgangs keine andere Sitzung (außer die eigene) mit der Quelldatenbank verbunden ist, aber das kann nicht ausschließen, dass während des Kopiervorgangs irgendwelche Änderungen getätigt werden, wodurch die kopierte Datenbank inkonsistent würde. Daher empfehlen wir, dass die Datenbanken, die als Template verwendet werden sollen, als nicht beschreibbar behandelt werden.

Es gibt für jede Datenbank zwei nützliche Einstellmöglichkeiten in `pg_database`: die Spalten `datistemplate` und `datallowconn`. `datistemplate` können gesetzt werden, um anzuzeigen, dass diese Datenbank als Template für `CREATE DATABASE` vorgesehen ist. Wenn es gesetzt ist, dann kann die Datenbank von jedem Benutzer mit `CREATEDB`-Privileg geklont werden; wenn nicht, können nur Superuser und der Datenbankeigentümer die Datenbank klonen. Wenn `datallowconn` falsch ist, werden keine neuen Verbindungen zu dieser Datenbank erlaubt (aber bestehende Sitzungen werden durch das einfache Ändern der Spalte nicht angebrochen). Bei der Datenbank `template0` ist normalerweise `datallowconn = false`, um zu verhindern, dass sie verändert wird. Bei sowohl `template0` als auch `template1` sollte immer `datistemplate = true` sein.

Nachdem Sie eine Templatedatenbank vorbereitet haben oder nachdem Sie eine verändert haben, ist es empfehlenswert, `VACUUM FREEZE` oder `VACUUM FULL FREEZE` in dieser Datenbank auszuführen. Wenn Sie dies tun, während es keine weiteren offenen Transaktionen in der selben Datenbank gibt, wird garantiert, dass alle Zeilen in der Datenbank "eingefroren" sind und keine Probleme mit dem Überlauf von Transaktionsnummern haben werden. Dies ist besonders wichtig in Datenbanken, bei denen Sie `datallowconn` auf falsch setzen wollen, da es dann unmöglich sein wird, die Routinewartung mit `VACUUM` in dieser Datenbank durchzuführen. Siehe Abschnitt 21.1.3 für weitere Informationen.

Anmerkung

`template1` und `template0` haben keinen Sonderstatus im System, außer dass der Name `template1` der voreingestellte Quelldatenbankname für `CREATE DATABASE` und bei einigen Programmen, wie `createdb`, die voreingestellte Datenbank zum Verbinden ist. Man könnte zum Beispiel `template1` ohne Probleme löschen und aus `template0` neu erzeugen. Das kann in der Tat ratsam sein, wenn man zu sorglos war und einen Haufen Unsinn in `template1` erzeugt hat.

18.4 Datenbankkonfiguration

In Abschnitt 16.4 haben wir beschrieben, dass der PostgreSQL-Server eine große Zahl von Konfigurationsparametern bietet. Sie können für viele dieser Parameter datenbankspezifische Vorgabewerte setzen.

Wenn Sie zum Beispiel aus irgendwelchen Gründen in einer bestimmten Datenbank den GEQO-Optimierer abschalten wollen, müssten Sie ihn entweder für alle Datenbanken abschalten oder Sorge tragen, dass jeder verbundene Client `SET geqo TO off;` ausführt. Um diese Voreinstellung zu machen, können Sie folgenden Befehl ausführen:

```
ALTER DATABASE meinedb SET geqo TO off;
```

Das speichert die Einstellung (aber aktiviert sie nicht sofort), und bei den folgenden Verbindungen wird es den Anschein haben, dass `SET geqo TO off;` direkt vor Beginn der Sitzung ausgeführt wurde. Beachten Sie, dass die Benutzer diese Einstellung während der Sitzung ändern können; sie ist nur die Vorgabe. Um eine solche Einstellung rückgängig zu machen, verwenden Sie `ALTER DATABASE dbname RESET varname;`.

18.5 Alternativer Speicherplatz

Man kann eine Datenbank an einer anderen Stelle auf der Festplatte als der von der Installation vorgegebenen erzeugen. Bedenken Sie aber, dass alle Datenbankzugriffe durch den Server gehen und jeder Speicherplatz also vom Server erreichbar sein muss.

Alternative Speicherplätze werden durch eine Umgebungsvariable, die den absoluten Pfad der gewünschten Stelle enthält, angegeben. Diese Umgebungsvariable muss in der Umgebung des Servers stehen, muss also gesetzt gewesen sein, als der Server gestartet wurde. (Die Menge der verfügbaren alternativen Plätze steht damit unter der Kontrolle des Administrators und kann von einfachen Benutzern nicht verändert werden.) Jede beliebige Umgebungsvariable kann verwendet werden, um alternative Speicherplätze zu bezeichnen, aber der Verwendung von Variablennamen mit dem Präfix `PGDATA` wird empfohlen, um Verwirrung und Konflikte mit anderen Variablen zu vermeiden.

Um eine Variable in der Umgebung des Serverprozesses zu erzeugen, müssen Sie erst den Server herunterfahren, die Variable definieren, den Datenbereich initialisieren und schließlich den Server neu starten. (Siehe auch Abschnitt 16.6 und Abschnitt 16.3.) Um eine Umgebungsvariable zu setzen, geben Sie ein

```
PGDATA2=/home/postgres/data
export PGDATA2
```

in Bourne-Shells oder

```
setenv PGDATA2 /home/postgres/data
```

in csh oder tcsh. Sie müssen sicherstellen, dass diese Umgebungsvariable immer in der Serverumgebung definiert ist, ansonsten werden Sie auf die Datenbank nicht zugreifen können. Daher sollten Sie sie vielleicht in einer Startdatei der Shell oder dem Startskript des Servers setzen.

Um den Datenbereich in PGDATA2 zu erzeugen, achten Sie darauf, dass das übergeordnete Verzeichnis (hier /home/postgres) existiert und von dem Benutzerzugang, unter dem der Server läuft (siehe Abschnitt 16.1), beschreibbar ist. Dann geben Sie auf der Kommandozeile ein:

```
initlocation PGDATA2
```

(*nicht* initlocation $PGDATA2). Danach können Sie der Server neu starten.

Um eine Datenbank an dem neuen Speicherplatz zu erzeugen, verwenden Sie den Befehl

```
CREATE DATABASE name WITH LOCATION 'platz';
```

wo *platz* die verwendete Umgebungsvariable, in diesem Beispiel PGDATA2, ist. Der Befehl createdb hat zu diesem Zweck die Option -D.

Datenbanken an alternativen Speicherplätzen können wie andere Datenbanken verwendet und entfernt werden.

Anmerkung

Es kann auch ermöglicht werden, absolute Pfade direkt in CREATE DATABASE anzugeben, ohne Umgebungsvariablen definieren zu müssen. Das ist in der Voreinstellung verboten, weil es eine Sicherheitslücke wäre. Um es zu erlauben, müssen Sie, wenn Sie PostgreSQL compilieren, das C-Präprozessormakro ALLOW_ABSOLUTE_DBPATHS definieren. Ein Möglichkeit, dies zu tun, ist, den Compilierungsschritt so auszuführen:

```
gmake CPPFLAGS=-DALLOW_ABSOLUTE_DBPATHS all
```

18.6 Eine Datenbank zerstören

Datenbanken werden mit dem Befehl DROP DATABASE zerstört:

```
DROP DATABASE name;
```

Nur der Eigentümer einer Datenbank (d.h. der Benutzer, der sie erzeugt hat) oder ein Superuser kann eine Datenbank löschen. Das Löschen einer Datenbank entfernt alle Objekte, die in der Datenbank enthalten waren. Die Zerstörung einer Datenbank kann nicht rückgängig gemacht werden.

Sie können DROP DATABASE nicht ausführen, während Sie mit der Opferdatenbank verbunden sind. Sie können jedoch mit einer beliebigen anderen Datenbank, einschließlich template1, verbunden sein. template1 wäre die einzige Wahl, um die letzte Benutzerdatenbank eines Clusters zu löschen.

Der Bequemlichkeit halber gibt es auch ein Programm, das Sie in der Shell ausführen können, um Datenbanken zu entfernen:

```
dropdb dbname
```

(Im Gegensatz zu createdb entfernt dropdb nicht als Vorgabe die Datenbank mit den Namen des aktuellen Benutzers.)

19 Clientauthentifizierung

Wenn sich eine Clientanwendung mit dem Datenbankserver verbindet, gibt sie an, unter welchem PostgreSQL-Benutzernamen sie verbinden will, ähnlich wie man sich in einem Unix-Computer als ein bestimmter Benutzer anmeldet. Innerhalb der SQL-Umgebung bestimmt der aktive Datenbankbenutzername die Zugriffsprivilegien für Datenbankobjekte; siehe Kapitel 17 für weitere Informationen. Daher ist es wichtig, die Menge der Benutzer, die verbinden können, einzuschränken.

Authentifizierung ist der Vorgang, in dem der Datenbankserver die Identität des Clients feststellt und im erweiterten Sinne bestimmt, ob die Clientanwendung (oder der Benutzer, der sie ausführt) unter dem angegebenen Benutzernamen verbinden darf.

PostgreSQL bietet eine Reihe verschiedener Clientauthentifizierungs-Methoden. Die Methode, die zur Authentifizierung einer bestimmten Clientverbindung verwendet wird, kann auf Basis der Hostadresse des Clients, der Datenbank und des Benutzernamens ausgewählt werden.

PostgreSQL-Benutzernamen sind von den Benutzernamen des Betriebssystems, auf dem der Server läuft, logisch getrennt. Wenn alle Benutzer eines bestimmten Servers auch Zugänge auf der Servermaschine haben, macht es Sinn, die Datenbankbenutzernamen so zu wählen, dass sie mit den Betriebssystembenutzernamen übereinstimmen. Ein Server, der Verbindungen von anderen Maschinen akzeptiert, kann allerdings viele Datenbankbenutzer haben, die keinen Betriebssystemzugang auf dem Server haben, und dann muss keine Verbindung zwischen den Benutzernamen auf Datenbank- und auf Betriebssystemebene bestehen.

19.1 Die Datei pg_hba.conf

Clientauthentifizierung wird durch die Datei `pg_hba.conf` im Datenverzeichnis, z.B. `/usr/local/pgsql/data/pg_hba.conf`, kontrolliert. (HBA steht für *host-based authentication*, also Authentifizierung auf Host-Basis.) Eine voreingestellte `pg_hba.conf`-Datei wird von `initdb` im Datenverzeichnis eingerichtet, wenn der Datenbereich initialisiert wird.

Generell besteht die Datei `pg_hba.conf` aus einer Reihe von Datensätzen, einem pro Zeile. Leere Zeilen werden ignoriert, genauso wie gesamte Text nach dem Kommentarzeichen #. Ein Datensatz besteht aus einer Anzahl von Feldern, die durch Leerzeichen oder Tabs getrennt sind. Felder können Leerzeichen enthalten, wenn der Wert in Anführungszeichen steht. Ein Datensatz kann sich nicht über mehrere Zeilen erstrecken.

Jeder Satz besteht aus einem Verbindungstyp, einem IP-Adressbereich für Clients (falls für den Verbindungstyp von Bedeutung), einem Datenbanknamen, einem Benutzernamen und einer Authentifizierungs-

methode, die für Verbindungen, die mit diesen Parametern übereinstimmen, verwendet wird. Der erste Satz, bei dem der Verbindungstyp, die Clientadresse, die gewünschte Datenbank und der Benutzername übereinstimmen, wird für die Authentifizierung angewendet. Wenn ein Satz ausgewählt wurde und die Authentifizierung nicht erfolgreich ist, werden die folgenden Sätze trotzdem nicht mehr in Erwägung gezogen. Wenn kein Datensatz mit den Verbindungsparametern übereinstimmt, wird der Zugriff verweigert.

Ein Datensatz kann eines dieser drei Formate haben:

```
local datenbank benutzer authentifizierungs-methode [authentifizierungs-option]
host datenbank benutzer IP-adresse IP-maske authentifizierungs-methode
[authentifizierungs-option]
hostssl datenbank benutzer IP-adresse IP-maske authentifizierungs-methode
[authentifizierungs-option]
```

Die Bedeutung der Felder ist wie folgt:

`local`

Dieser Satz gilt für Verbindungsversuche über Unix-Domain-Sockets. Ohne Sätze dieses Typs werden keine Verbindungen über Unix-Domain-Sockets zugelassen.

`host`

Dieser Satz gilt für Verbindungsversuche über TCP/IP-Netzwerke. Beachten Sie, dass TCP/IP-Verbindungen nur möglich sind, wenn der Server mit der Option `-i` gestartet wurde oder der Konfigurationsparameter `tcpip_socket` in `postgresql.conf` eingeschaltet ist.

`hostssl`

Dieser Satz gilt für Verbindungen mit SSL über TCP/IP. `host`-Sätze gelten für SSL- und Nicht-SSL-Verbindungen, aber `hostssl` erfordert eine SSL-Verbindung.

Um diese Option verwenden zu können, muss der Server mit SSL-Unterstützung kompiliert worden sein. Außerdem muss SSL mit der Option `ssl` in `postgresql.conf` angeschaltet sein (siehe Abschnitt 16.4).

`datenbank`

Gibt an, für welche Datenbank der Satz gilt. Der Wert `all` passt auf alle Datenbanken. Der Wert `sameuser` gibt an, dass der Satz gilt, wenn die gewünschte Datenbank den gleichen Namen hat wie der gewünschte Benutzer. Der Wert `samegroup` gibt an, dass der gewünschte Benutzer ein Mitglied der Gruppe mit dem gleichen Namen wie die gewünschte Datenbank sein muss. Ansonsten ist dies der Name einer bestimmten PostgreSQL-Datenbank. Man kann mehrere Datenbanknamen angeben, indem man sie durch Kommas trennt. Eine Datei mit Namen von Datenbanken kann angegeben werden, indem vor den Dateinamen ein `@` gestellt wird. Die Datei muss im gleichen Verzeichnis wie `pg_hba.conf` liegen.

`benutzer`

Gibt an, für welche PostgreSQL-Benutzer der Eintrag gilt. Der Wert `all` passt auf alle Benutzer. Ansonsten ist dies der Name eines bestimmten PostgreSQL-Benutzers. Man kann mehrere Benutzernamen angeben, indem man sie durch Kommas trennt. Gruppennamen können angegeben werden, indem man vor den Namen ein `+` stellt. Eine Datei mit Namen von Benutzern kann angegeben werden, indem vor den Dateinamen ein `@` gestellt wird. Die Datei muss im gleichen Verzeichnis wie `pg_hba.conf` liegen.

`IP-adresse`

`IP-maske`

Diese zwei Felder enthalten IP-Adressen und -Masken in der normalen Schreibweise aus Dezimalzahlen und Punkten. (IP-Adressen können nur numerisch angegeben werden, nicht als Domain-

oder Hostnamen.) In Kombination geben sie an, für welche Client-IP-Adressen dieser Satz gilt. Genau gesagt ist es so, dass

```
(tatsächliche-IP-adresse xor IP-adresse-feld) and IP-maske-feld
```

null sein muss, damit der Satz gilt. (Natürlich kann man IP-Adressen verfälschen, aber das liegt nicht im Einflussbereich von PostgreSQL.)

Diese Felder können nur in `host`- und `hostssl`-Sätzen verwendet werden.

`authentifizierungs-methode`

Gibt an, welche Authentifizierungsmethode verwendet werden soll, wenn dieser Satz auf die Verbindungsparameter passt. Die Wahlmöglichkeiten sind hier kurz zusammengefasst; Einzelheiten finden Sie in Abschnitt 19.2.

`trust`

Die Verbindung wird bedingungslos zugelassen. Diese Methode erlaubt jedem, der mit dem PostgreSQL-Datenbankserver verbinden kann, sich mit jedem beliebigen PostgreSQL-Benutzernamen anzumelden, ohne ein Passwort zu benötigen. Einzelheiten stehen in Abschnitt 19.2.1.

`reject`

Die Verbindung wird bedingungslos angelehnt. Das ist nützlich, um bestimmte Hosts aus einer Gruppe “herauszufiltern”.

`md5`

Verlangt, dass der Client zur Authentifizierung ein mit MD5 verschlüsseltes Passwort schickt. Dies ist die einzige Methode, die verschlüsselte Passwörter in `pg_shadow` ermöglicht. Einzelheiten stehen in Abschnitt 19.2.2.

`crypt`

Wie die `md5`-Methode, aber verwendet die ältere Verschlüsselung mit `crypt()`, welche für Clients vor Version 7.2 benötigt wird. `md5` sollte für Clients aus Version 7.2 und später bevorzugt werden. Einzelheiten stehen in Abschnitt 19.2.2.

`password`

Wie `md5`, aber das Passwort wird unverschlüsselt über das Netzwerk geschickt. Dies sollte in unvertraulichen Netzwerken nicht verwendet werden. Einzelheiten stehen in Abschnitt 19.2.2.

`krb4`

Verwendet Kerberos V4, um den Benutzer zu authentifizieren. Dies ist nur bei TCP/IP-Verbindungen möglich. Einzelheiten stehen in Abschnitt 19.2.3,

`krb5`

Verwendet Kerberos V5, um den Benutzer zu authentifizieren. Dies ist nur bei TCP/IP-Verbindungen möglich. Einzelheiten stehen in Abschnitt 19.2.3.

`ident`

Ermittelt den Betriebssystembenutzernamen des Clients (bei TCP/IP-Verbindungen, indem der Ident-Server auf dem Client kontaktiert wird, bei lokalen Verbindungen vom Betriebssystem selbst) und prüft, ob der Benutzer unter dem gewünschten Datenbankbenutzernamen verbinden darf. Dazu wird die hinter dem Schlüsselwort `ident` angegebene *Map* verwendet.

Wenn Sie den Map-Namen `sameuser` verwenden, müssen die Benutzernamen gleich sein. Wenn nicht, wird der Map-Name in der Datei `pg_ident.conf` im selben Verzeichnis wie `pg_hba.conf` gesucht. Die Verbindung wird akzeptiert, wenn diese Datei einen Eintrag für den Map-Namen enthält, der den Betriebssystembenutzernamen und den gewünschten PostgreSQL-Benutzernamen verbindet.

Bei lokalen Verbindungen funktioniert dies nur auf Maschinen, die für Unix-Domain-Sockets *Credentials* unterstützen (gegenwärtig `Linux`, `FreeBSD`, `NetBSD` und `BSD/OS`).

Einzelheiten stehen in Abschnitt 19.2.4.

`pam`

Verwendet den vom Betriebssystem angebotenen PAM-Dienst (*Pluggable Authentication Modules*) zur Authentifizierung. Einzelheiten stehen in Abschnitt 19.2.5.

`authentifizierungs-option`

Die Bedeutung dieses Felds hängt von der gewählten Authentifizierungsmethode ab und wird im nächsten Abschnitt beschrieben.

Da die Datensätze in `pg_hba.conf` bei jedem Verbindungsversuch hintereinander geprüft werden, ist die Reihenfolge der Sätze von Bedeutung. Normalerweise haben die oberen Einträge enger gefasste Verbindungsparameter und schwächere Authentifizierungsmethoden, während unteren die Einträge weiter gefasste Verbindungsparameter und strengere Authentifizierungsmethoden verwenden. Man könnte zum Beispiel für lokale TCP/IP-Verbindungen `trust` verwenden, aber für andere TCP/IP-Verbindungen Passwörter verlangen. In dem Fall würde der Eintrag, der `trust` für Verbindungen von 127.0.0.1 angibt, vor dem Eintrag, der Passwortauthentifizierung für einen größeren Bereich von IP-Adressen wählt, stehen.

Wichtig

Verhindern Sie nicht, dass der Superuser auf die Datenbank `template1` zugreifen kann. Diverse Hilfsprogramme benötigen den Zugriff auf `template1`.

Die Datei `pg_hba.conf` wird beim Start des Servers und wenn der Hauptserverprozess (`postmaster`) das Signal `SIGHUP` erhält gelesen. Wenn Sie die Datei in einem aktiven System bearbeiten, müssen Sie das dem `postmaster`-Prozess signalisieren (mit `pg_ctl reload` oder `kill -HUP`), damit er die Datei neu liest.

Ein Beispiel für eine `pg_hba.conf`-Datei wird in Beispiel 19.1 gezeigt. Einzelheiten über die verschiedenen Authentifizierungsmethoden finden Sie im nächsten Abschnitt.

Beispiel 19.1: Ein Beispiel für pg_hba.conf

```
# Erlaube jedem Benutzer auf dem lokalen System, über
# Unix-Domain-Sockets (Standard für lokale Verbindungen) mit jeder
# Datenbank unter jedem Benutzernamen zu verbinden.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
local   all         all                                             trust

# Das gleiche für lokale Verbindungen über TCP/IP-Loopback.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
host    all         all         127.0.0.1         255.255.255.255   trust

# Erlaube jedem Benutzer von einem Host mit IP-Adresse 192.168.93.x mit
```

```
# der Datenbank "template1" unter dem Benutzernamen, den Ident für die
# Verbindung ermittelt hat (normalerweise der Unix-Benutzername), zu
# verbinden.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
host    template1   all         192.168.93.0      255.255.255.0     ident sameuser

# Erlaube einem Benutzer von Host 192.168.12.10 mit der Datenbank
# "template1" zu verbinden, wenn das richtige Passwort angegeben wird.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
host    template1   all         192.168.12.10     255.255.255.255   md5

# Ohne die vorangegangenen "host"-Zeilen, verhindern diese beiden Zeilen
# alle Verbindungen von 192.168.54.1 (da dieser Eintrag zuerst kommt),
# aber erlauben Kerberos-V-Verbindungen vom Rest des Internets.  Die
# Nullmaske bedeutet, dass keine Bits der Host-IP-Adresse betrachtet
# werden, so dass sie auf alle Hosts passt.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
host    all         all         192.168.54.1      255.255.255.255   reject
host    all         all         0.0.0.0           0.0.0.0           krb5

# Erlaube Benutzern von 192.168.x.x-Hosts zu verbinden, wenn sie die
# Ident-Prüfung bestehen.  Wenn zum Beispiel Ident sagt, dass der
# Benutzer "bryanh" ist und er als PostgreSQL-Benutzer "guest1"
# verbinden will, dann wird die Verbindung erlaubt, wenn in
# pg_ident.conf ein Eintrag für ein Map "omicron" ist, der sagt, dass
# "bryanh" als "guest1" verbinden darf.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
host    all         all         192.168.0.0       255.255.0.0       ident omicron

# Wenn dies die einzigen drei Zeilen für lokale Verbindunge sind, dann
# erlauben Sie lokalen Benutzern, nur mit ihren eigenen Datenbanken
# (Datenbanken mit den gleichen Namen wie der Benutzername) zu
# verbinden, außer Administratoren und Mitglieder der Gruppe
# "support", die mit allen Datenbanken verbinden dürfen.  Die Datei
# $PGDATA/admins enthält eine Liste mit Benutzernamen.  In allen
# Fällen werden Passwörter verlangt.
#
# TYP   DATENBANK   BENUTZER    IP-ADRESSE        IP-MASKE          METHODE
local   sameuser    all                                             md5
local   all         @admins                                         md5
local   all         +support                                        md5
```

```
# Die letzten zwei Zeilen können zu einer einzigen Zeile
# zusammengefasst werden:
local   all         @admins,+support                              md5

# Die Datenbankspalte kann auch Listen und Dateinamen enthalten, aber
# keine Gruppen:
local   db1,db2,@demodbs  all                                     md5
```

19.2 Authentifizierungsmethoden

Im Folgenden werden die Authentifizierungsmethoden im Einzelnen beschrieben.

19.2.1 Freier Zugang

Wenn `trust`-Authentifizierung angegeben ist, dann nimmt PostgreSQL an, dass jeder, der mit dem Server verbinden kann, als jeder beliebiger Benutzer auf die Datenbank zugreifen darf (einschließlich als Datenbank-Superuser). Diese Methode sollte nur verwendet werden, wenn Verbindungen zum Server auf Betriebssystemebene ausreichend geschützt werden.

`trust`-Authentifizierung ist für lokale Verbindungen auf Einzelbenutzerarbeitsplätzen angebracht und bequem. Für Mehrbenutzermaschinen ist sie alleine in der Regel *nicht* angebracht. Sie können jedoch `trust` auch auf einer Mehrbenutzermaschine verwenden, wenn Sie den Zugang zur Unix-Domain-Socket-Datei des Servers auf Dateisystemebene einschränken. Um das zu tun, setzen Sie den Parameter `unix_socket_permissions` (und eventuell `unix_socket_group`) in `postgresql.conf`, wie in Abschnitt 16.4.3 beschrieben. Oder Sie könnten `unix_socket_directory` verändern, um die Socketdatei in ein entsprechend beschränktes Verzeichnis zu verlegen.

Die Dateisystemzugriffsrechte helfen nur bei Verbindungen über Unix-Domain-Sockets. Auf lokale TCP/IP-Verbindungen haben Sie keine Auswirkung. Wenn Sie also die Dateisystemzugriffsrechte verwenden wollen, um lokale Verbindungen zu kontrollieren, müssen Sie die Zeile `host ... 127.0.0.1 ...` aus `pg_hba.conf` entfernen oder das `trust` in eine andere Methode ändern.

`trust`-Authentifizierung ist für TCP/IP-Verbindungen nur sinnvoll, wenn Sie allen Benutzern auf allen Maschinen, die mit dem Server über `trust`-Zeilen in `pg_hba.conf` verbinden können, vertrauen. Es ist selten vernünftig, `trust` für TCP/IP-Verbindungen außer denen von `localhost` (127.0.0.1) zu verwenden.

19.2.2 Passwort-Authentifizierung

Die Authentifizierungsmethoden mit Passwörtern sind `md5`, `crypt` und `password`. Die Methoden funktionieren ganz ähnlich, unterscheiden sich aber durch die Art, wie das Passwort über die Verbindung geschickt wird. Wenn Sie sich Sorgen um Passwort-"Sniffing" machen, sollten Sie `md5` verwenden, wobei `crypt` die zweite Wahl wäre, wenn sie Clients aus Versionen vor 7.2 unterstützen müssen. Die Methode `password` sollte insbesondere bei Verbindungen über das offene Internet vermieden werden (es sei denn, Sie verwenden SSL, SSH oder andere Vorrichtungen um die Verbindung zu sichern).

PostgreSQL-Datenbankpasswörter sind getrennt von den Passwörtern im Betriebssystem. Das Passwort jedes Datenbankbenutzers wird in der Systemtabelle `pg_shadow` gespeichert. Passwörter können mit den SQL-Befehlen `CREATE USER` und `ALTER USER` verwaltet werden, zum Beispiel `CREATE USER foo WITH PASSWORD 'geheim';`. In der Voreinstellung, das heißt, wenn kein Passwort eingerichtet wurde, ist das gespeicherte Passwort der NULL-Wert, wodurch die Passwortauthentifizierung für den Benutzer nie erfolgreich sein kann.

Um zu beschränken, welche Benutzer mit bestimmten Datenbanken verbinden können, können Sie die Benutzer in der Spalte *benutzer* in der Datei `pg_hba.conf` wie im vorigen Abschnitt beschrieben aufführen.

19.2.3 Kerberos-Authentifizierung

Kerberos ist ein etabliertes, sicheres Authentifizierungssystem für verteilte Dienste in öffentlichen Netzwerken. Eine Beschreibung des Kerberos-Systems würde den Rahmen dieses Dokuments sprengen; das ganze System kann ziemlich komplex (aber mächtig) sein. Die Kerberos FAQ oder MIT Project Athena können als Ausgangspunkte für Erforschungen dienen. Es gibt verschiedene Quellen für Kerberos-Produkte.

Um Kerberos benutzen zu können, muss die Unterstützung beim Compilieren eingeschaltet worden sein. Weitere Informationen darüber finden Sie in Kapitel 14. Kerberos 4 und 5 werden angeboten, aber nur eine Version kann von jeder Installation unterstützt werden. PostgreSQL funktioniert wie ein normaler Kerberos-Dienst. Der Name des Service-Principals ist `servicename/hostname@realm`, wobei *servicename* gleich `postgres` ist (außer wenn ein anderer Servicename konfiguriert wurde, mit `./configure --with-krb-srvnam=irgendwas`). *hostname* ist der voll qualifizierte Hostname der Servermaschine. Das Realm des Service-Principals ist das bevorzugte Realm der Servermaschine.

Client-Principals haben den PostgreSQL-Benutzernamen als ersten Teil, zum Beispiel `pgbenutzer/andereszeug@realm`. Das Realm wird gegenwärtig von PostgreSQL nicht überprüft; wenn Sie also Cross-Realm-Authentifizierung eingeschaltet haben, werden alle Principals in allen Realms, die mit Ihrem kommunizieren können, akzeptiert.

Achten Sie darauf, dass die Serverschlüsseldatei vom PostgreSQL-Server lesbar (und bevorzugt nur lesbar) ist. (Siehe auch Abschnitt 16.1.) Der Ort der Schlüsseldatei wird vom Konfigurationsparameter `krb_server_keyfile` bestimmt. (Siehe auch Abschnitt 16.4.) Die Voreinstellung ist `/etc/srvtab` für Kerberos 4 und `FILE:/usr/local/pgsql/etc/krb5.keytab` (je nachdem, welches Verzeichnis bei der Compilierung als `sysconfdir` angegeben wurde) für Kerberos 5.

Um eine Keytab-Datei zu erzeugen, verwenden Sie zum Beispiel (mit Version 5):

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Einzelheiten finden Sie in der Dokumentation zu Kerberos.

Wenn Sie mit der Datenbank verbinden, müssen Sie ein Ticket haben, dass dem gewünschten Datenbankbenutzernamen entspricht. Beispiel: Für den Datenbankbenutzernamen `fred` können die Principals `fred@EXAMPLE.COM` oder `fred/users.example.com@EXAMPLE.COM` zur Authentifizierung mit dem Datenbankserver verwendet werden.

Wenn Sie `mod_auth_krb` und `mod_perl` auf Ihrem Apache-Webserver verwenden, können Sie `AuthType KerberosV5SaveCredentials` mit einem `mod_perl`-Skript verwenden. Dadurch erhalten Sie sicheren Datenbankzugriff über das Web ohne zusätzliche Passwörter.

19.2.4 Ident-Authentifizierung

Das Ident-Authentifizierungssystem prüft den Betriebssystembenutzernamen des Clients und ermittelt dann die erlaubten Datenbankbenutzernamen, indem es eine Map-Datei verwendet, die die erlaubten Benutzernamenpaare auflistet. Der sicherheitskritische Aspekt ist die Bestimmung des Benutzernamens des Clients, und wie das funktioniert, hängt vom Verbindungstyp ab.

Ident-Authentifizierung über TCP/IP

Das *Identification Protocol* ist in *RFC 1413* (siehe Literaturverzeichnis) beschrieben. Praktisch jedes Unix-ähnliche Betriebssystem hat einen Ident-Server, der in der Voreinstellung auf TCP-Port 113 hört. Die grundsätzliche Funktion des Ident-Servers ist es, Fragen wie diese zu beantworten: "Welcher Benutzer startete die Verbindung, die aus deinem Port *X* kommt und bei meinen Port *Y* ankommt?" Da PostgreSQL sowohl *X* als auch *Y* kennt, wenn die Verbindung physikalisch eingerichtet ist, kann es den Ident-Server auf dem Host des verbindenden Client befragen und könnte so theoretisch den Betriebssystembenutzer zu jeder beliebigen Verbindung ermitteln.

Der Nachteil dieses Vorgehens ist, dass es sich auf die Integrität des Clients verlässt: Wenn die Clientmaschine nicht vertrauenswürdig ist oder kompromittiert wurde, könnte ein Angreifer so ziemlich jedes Programm an Port 113 laufen lassen und Benutzernamen nach Belieben zurückgeben. Diese Authentifizierungsmethode ist daher nur in geschlossenen Netzwerken angebracht, wo jede Clientmaschine streng kontrolliert wird und die Verwalter von Datenbank und System eng zusammenarbeiten. Anders ausgedrückt müssen Sie der Maschine mit dem Ident-Server vertrauen können. Beachten Sie die Warnung:

> RFC 1413
>
> The Identification Protocol is not intended as an authorization or access control protocol.

Ident-Authentifizierung über lokale Sockets

Auf Systemen, die auf Unix-Domain-Sockets *SO_PEERCRED*-Anfragen unterstützen (gegenwärtig `Linux`, `FreeBSD`, `NetBSD` und `BSD/OS`), kann Ident-Authentifizierung auch bei lokalen Verbindungen angewendet werden. In diesem Fall erzeugt die Verwendung von Ident keine weiteren Sicherheitsprobleme; in der Tat ist Ident auf solchen Systemen die bevorzugte Methode für lokale Verbindungen.

Auf Systemen ohne *SO_PEERCRED*-Anfragen steht Ident-Authentifizierung nur für TCP/IP-Verbindungen zur Verfügung. Als Ausweg kann man als Verbindungsziel die `localhost`-Adresse `127.0.0.1` angeben.

Ident-Maps

Wenn Ident-basierte Authentifizierung verwendet wird und der Name des Betriebssystembenutzers, der die Verbindung eingeleitet hat, ermittelt wurde, prüft PostgreSQL, ob dieser Benutzer als der Datenbankbenutzer, den er gewünscht hat, verbinden darf. Das wird durch das Ident-Map-Argument kontrolliert, das in der Datei `pg_hba.conf` nach dem Schlüsselwort `ident` steht. Es gibt eine vordefinierte Ident-Map namens `sameuser`, welche es jedem Betriebssystembenutzer erlaubt, als der Datenbankbenutzer mit dem gleichen Namen (falls dieser existiert) zu verbinden. Andere Maps müssen selbst erzeugt werden.

Andere Ident-Maps neben `sameuser` werden in der Datei `pg_ident.conf` im Datenverzeichnis definiert, die Zeilen mit folgender allgemeinen Form hat:

```
map-name ident-benutzername database-benutzername
```

Kommentare und Leerzeichen werden auf die übliche Art behandelt. Der *map-name* ist ein beliebiger Name, der verwendet wird, um in `pg_hba.conf` auf diese Map zu verweisen. Die anderen beiden Felder geben an, welcher Betriebssystembenutzer als welcher Datenbankbenutzer verbinden darf. Der gleiche *map-name* kann mehrfach verwendet werden, um mehrere Benutzerpaare in einer Map zu definieren. Es gibt keine Einschränkungen darüber, wie viele Datenbankbenutzer einem Betriebssystembenutzer oder umgekehrt entsprechen dürfen.

Die Datei `pg_ident.conf` wird beim Start des Servers und wenn der Hauptserverprozess (`postmaster`) das Signal `SIGHUP` erhält gelesen. Wenn Sie die Datei in einem aktiven System bearbeiten, müssen Sie das dem `postmaster`-Prozess signalisieren (mit `pg_ctl reload` oder `kill -HUP`), damit er die Datei neu liest.

Eine `pg_ident.conf`-Datei, die zusammen mit der `pg_hba.conf`-Datei aus Beispiel 19.1 verwendet werden könnte, wird in Beispiel 19.2 gezeigt. In dieser Beispielkonfiguration würde jemandem aus dem

192.168-Netzwerk nur der Zugriff gewährt, wenn er den Unix-Benutzernamen `bryanh`, `ann` oder `robert` hat. Darüber hinaus würde dem Unix-Benutzer `robert` der Zugriff nur gewährt, wenn er als PostgreSQL-Benutzer `bob`, nicht als `robert` oder jemand anders, verbindet. `ann` dürfte nur als `ann` verbinden. Der Benutzer `bryanh` könnte entweder als `bryanh` selbst oder als `guest1` verbinden.

Beispiel 19.2: Ein Beispiel für pg_ident.conf

```
# MAPNAME     IDENT-BENUTZER    PG-BENUTZER

omicron       bryanh            bryanh
omicron       ann               ann
# bob heißt auf dieser Maschine robert
omicron       robert            bob
# bryanh kann auch als guest1 verbinden
omicron       bryanh            guest1
```

19.2.5 PAM-Authentifizierung

Diese Authentifizierungsmethode funktioniert ähnlich wie `password`, verwendet aber PAM (*Pluggable Authentication Modules*) als Mechanismus. Der voreingestellte PAM-Servicename ist `postgresql`. Wahlweise können Sie selbst einen Servicenamen nach dem Schlüsselwort `pam` in der Datei `pg_hba.conf` eingeben. Für weitere Informationen über PAM lesen Sie bitte die Linux-PAM Page und die `Solaris` PAM Page.

19.3 Authentifizierungsprobleme

Echte Authentifizierungsprobleme äußern sich im Allgemeinen durch Fehlermeldungen wie die folgende.

```
No pg_hba.conf entry for host 123.123.123.123, user andym, database testdb
```

Das sehen Sie am wahrscheinlichsten, wenn Sie den Server erfolgreich kontaktiert haben, aber er mit Ihnen nicht reden will. Wie die Meldung andeutet, hat der Server die Verbindung abgelehnt, weil er keinen passenden Eintrag in seiner Konfigurationsdatei `pg_hba.conf` gefunden hat.

```
Password authentication failed for user 'andym'
```

Meldungen wie diese zeigen an, dass Sie den Server kontaktiert haben und er mit Ihnen reden will, aber erst, wenn Sie die in der Datei `pg_hba.conf` angegebene Authentifizierungsmethode durchlaufen haben. Überprüfen Sie das angegebene Passwort oder prüfen Sie Ihre Kerberos- oder Ident-Software, wenn sich die Meldung auf diese Authentifizierungstypen bezieht.

```
FATAL 1: user "andym" does not exist
```

Der angegebene Benutzer wurde nicht gefunden.

```
FATAL 1: Database "testdb" does not exist in the system catalog.
```

Die Datenbank, mit der Sie zu verbinden versuchen, existiert nicht. Beachten Sie, das wenn Sie keinen Benutzernamen angeben, der Benutzername als Datenbankname verwendet wird, was Sie vielleicht nicht beabsichtigt hatten.

Der Serverlog kann unter Umständen mehr Informationen über einen gescheiterten Authentifizierungsversuch enthalten als das, was an den Client geschickt wird. Wenn Sie also bei Authentifizierungsproblemen verwirrt sind, prüfen Sie den Log.

20 Lokalisierung

Dieses Kapitel beschreibt die verfügbaren Features zur Lokalisierung aus Sicht des Administrators. PostgreSQL unterstützt Lokalisierung mit drei Ansätzen:

- mit den Locale-Fähigkeiten des Betriebssystems, um sprachspezifische Sortierreihenfolgen, Zahlenformatierung, übersetzte Meldungen und andere Aspekte anzubieten
- durch eine Reihe verschiedener im PostgreSQL-Server definierter Zeichensätze, einschließlich Mehrbyte-Zeichensätze, um Text in allen möglichen Sprachen abspeichern zu können, und durch Konvertierung zwischen Zeichensätzen zwischen Client und Server
- Zeichensatzkonvertierung zwischen Einbyte-Zeichensätzen als effizientere Lösung für Benutzer, für die das ausreicht.

20.1 Locale-Unterstützung

Locale-Unterstützung bezieht sich darauf, dass Anwendungen die von einer bestimmten Kultur bevorzugten Bräuche für das Alphabet, die Sortierreihenfolge, Zahlenformate usw. respektieren. PostgreSQL verwendet die vom Serverbetriebssystem bereitgestellten Locale-Vorrichtungen entsprechend dem ISO-C- bzw. POSIX-Standard. Weitere Informationen dazu finden Sie auch in der Dokumentation Ihres Betriebssystems.

20.1.1 Überblick

Die Locale-Unterstützung wird automatisch initialisiert, wenn ein Datenbankcluster mit `initdb` erzeugt wird. `initdb` initialisiert den Datenbankcluster mit den Locale-Einstellungen aus seiner Umgebung; wenn also Ihr System schon die Locale verwendet, die Sie in Ihrem Datenbankcluster verwenden wollen, müssen Sie nichts weiter tun. Wenn Sie eine andere Locale verwenden wollen (oder Sie sich nicht sicher sind, welche Ihr System verwendet), können Sie `initdb` mit der Option `--locale` genau sagen, welche Locale Sie verwenden wollen. Zum Beispiel:

```
initdb --locale=sv_SE
```

Dieses Beispiel setzt die Locale auf Schwedisch (`sv`) wie es in Schweden (`SE`) gesprochen wird. Weitere Beispiele sind `de_DE` (Deutsch, Deutschland), `en_US` (Englisch, USA) oder `fr_CA` (Französisch, Kanada). Wenn mehr als ein Zeichensatz für eine Locale verwendet werden kann, sieht die Angabe so aus:

cs_CZ.ISO8859-2. Welche Locales auf Ihrem System vorhanden sind, hängt davon ab, was vom Hersteller zur Verfügung gestellt und installiert wurde.

Gelegentlich kann es nützlich sein, die Regeln aus mehreren Locales zu kombinieren, zum Beispiel englische Sortierreihenfolge aber spanische Meldungen. Um das zu ermöglichen, gibt es mehrere Locale-Subkategorien, die nur bestimmte Aspekte der Locale-Regeln kontrollieren.

LC_COLLATE	Zeichenketten-Sortierreihenfolge
LC_CTYPE	Zeichenklassifizierung (Was ist ein Buchstabe? Was ist der entsprechende Großbuchstabe?)
LC_MESSAGES	Sprache der Meldungen
LC_MONETARY	Format für Geldbeträge
LC_NUMERIC	Format für Zahlen
LC_TIME	Format für Datum und Zeit

Aus den Kategorienamen ergeben sich die Namen von Kommandozeilenoptionen für initdb, die die Locale-Wahl für eine bestimmte Kategorie einstellen. Um zum Beispiel die Locale auf Kanada/Französisch zu setzen, aber die Regeln von USA/Englisch zur Formatierung von Geldbeträgen auszuwählen, verwenden Sie initdb --locale=fr_CA --lc-monetary=en_US.

Wenn Sie wollen, dass sich Ihr System so verhält, als hätte es keine Locale-Unterstützung, dann setzen Sie die Locale auf C oder POSIX.

Bei einigen Locale-Kategorien steht der Wert für die gesamte Lebensdauer des Datenbankclusters fest. Das heißt, nachdem initdb ausgeführt wurde, können Sie ihn nicht mehr verändern. Diese Kategorien sind LC_COLLATE und LC_CTYPE. Sie beeinflussen die Sortierreihenfolge der Indexe und müssen daher konstant gehalten werden oder Indexe für Textspalten werden verfälscht und ungültig werden. PostgreSQL sorgt dafür, dass die Locale-Werte konstant bleiben, indem die von initdb gesehenen Werte von LC_COLLATE und LC_CTYPE gespeichert werden. Der Server stellt diese beiden Werte dann automatisch ein, wenn er gestartet wird.

Die anderen Locale-Kategorien können, während der Server läuft, immer geändert werden, indem die Konfigurationsparameter mit den gleichen Namen wie die Locale-Kategorien eingestellt werden (siehe Abschnitt 16.4 wegen Einzelheiten). Die durch initdb gewählten Voreinstellungen werden nur in die Konfigurationsdatei postgresql.conf geschrieben, wo sie als Voreinstellungen dienen, wenn der Server gestartet wird. Wenn Sie die Zuweisungen aus postgresql.conf löschen, dann übernimmt der Server die Einstellungen aus der Umgebung.

Beachten Sie, dass das Locale-Verhalten des Servers von den vom Server gesehenen Umgebungsvariablen abhängt und nicht von der Umgebung irgendeines Clients. Daher müssen Sie darauf achten, dass Sie die Locale-Einstellungen tätigen, bevor Sie den Server starten. Daraus folgt auch, wenn der Server und der Client verschiedene Locales verwenden, können Meldungen in verschiedenen Sprachen erscheinen, je nachdem, woher sie stammen.

Wenn wir davon reden, dass die Locale aus der Umgebung übernommen wird, bedeutet das auf den meisten Betriebssystemen Folgendes: Für eine gegebene Locale-Kategorie, sagen wir die Sortierreihenfolge, werden die folgenden Umgebungsvariablen in dieser Reihenfolge untersucht, bis eine gefunden wird, die gesetzt ist: LC_ALL, LC_COLLATE (die Variable, die der Kategorie entspricht) und LANG. Wenn keine dieser Umgebungsvariablen gesetzt ist, ist die voreingestellte Locale C.

Manche Bibliotheken zur Übersetzung von Programmmeldungen betrachten auch die Umgebungsvariable LANGUAGE, welche alle anderen Locale-Einstellungen für die Sprache von Programmmeldungen übergeht. Wenn Sie sich nicht sicher sind, dann schauen Sie bitte in der Dokumentation Ihres Betriebssystems nach, insbesondere in der Anleitung zu Gettext.

Um Mitteilungen in der vom Benutzer bevorzugten Sprache anzeigen zu können, muss NLS beim Compilieren angeschaltet worden sein. Diese Wahl ist unabhängig von der anderen Locale-Unterstützung.

20.1.2 Nutzen

Locale-Unterstützung hat insbesondere Auswirkung auf die folgenden Aspekte:

- Sortierreihenfolge in Anfragen mit `ORDER BY`
- die Funktionsfamilie `to_char`
- die Operatoren `LIKE` und ~ für Mustervergleiche

Der einzige schwerwiegende Nachteil der Locale-Unterstützung in PostgreSQL ist die Geschwindigkeit. Also verwenden Sie Locales nur, wenn Sie sie wirklich benötigen. Man sollte anmerken, dass die Auswahl einer anderen Locale außer C die Verwendung eines Index für die Operatoren `LIKE` und ~ verhindert, was für die Geschwindigkeit von Anfragen, die diese Operatoren verwenden, einen gewaltigen Unterschied machen kann.

20.1.3 Probleme

Wenn die Locale-Unterstützung trotz der Erklärung oben nicht funktioniert, überprüfen Sie, ob die Locale-Unterstützung in Ihrem Betriebssystem richtig läuft. Um zu sehen, welche Locales auf Ihrem System installiert sind, können Sie den Befehl `locale -a` verwenden, wenn Ihr Betriebssystem ihn zur Verfügung stellt.

Überprüfen Sie, ob PostgreSQL tatsächlich die Locale verwendet, die Sie denken. Die Einstellungen von `LC_COLLATE` und `LC_CTYPE` werden von `initdb` gemacht und können später nicht geändert werden ohne `initdb` zu wiederholen. Die anderen Locale-Kategorien werden anfänglich von der Umgebung, in der der Server gestartet wird, bestimmt. Die Einstellungen von `LC_COLLATE` und `LC_CTYPE` können Sie mit dem Hilfsprogramm `pg_controldata` einsehen.

Das Verzeichnis `src/test/locale` in der Quellendistribution enthält einige Tests für die Locale-Unterstützung in PostgreSQL.

Clientanwendungen, die Serverfehler verarbeiten, indem Sie den Text der Meldung analysieren, werden natürlich Probleme haben, wenn die Meldungen des Servers in einer anderen Sprache sind. Wenn Sie so eine Anwendung entwickeln, müssen Sie einen Plan aufstellen, um mit dieser Situation fertig zu werden. Die Schnittstelle `ECPG` (eingebettetes SQL) ist von diesem Problem auch betroffen. Wenn ein Server mit `ECPG`-Anwendungen kommunizieren muss, wird gegenwärtig empfohlen, dass er so konfiguriert wird, dass er Meldungen auf Englisch schickt.

Die Pflege von Katalogen mit Übersetzungen der Programmmeldungen erfordert die andauernden Bemühungen vieler Freiwilliger, die wollen, dass PostgreSQL ihre bevorzugte Sprache gut sprechen kann. Wenn die Meldungen in einer Sprache gegenwärtig nicht angeboten werden oder nicht vollständig übersetzt sind, würde Ihre Hilfe sehr geschätzt werden. Wenn Sie helfen wollen, schreiben Sie an die Entwicklermailingliste.

20.2 Zeichensatz-Unterstützung

Die Zeichensatzunterstützung in PostgreSQL ermöglicht Ihnen, Text in vielfältigen Zeichensätzen zu speichern, einschließlich Einbyte-Zeichensätze wie die ISO-8859-Serie und Mehrbyte-Zeichensätze wie EUC (Extended Unix Code), Unicode und Mule Internal Code. Alle Zeichensätze können transparent im ganzen Server verwendet werden. (Wenn Sie Erweiterungsfunktionen aus anderen Quellen verwenden, hängt es davon ab, ob die Autoren ihren Code richtig geschrieben haben.) Der Standardzeichensatz wird

gewählt, wenn der PostgreSQL-Datenbankcluster mit `initdb` initialisiert wird. Er kann übergangen werden, wenn eine Datenbank mit `createdb` oder dem SQL-Befehl `CREATE DATABASE` erzeugt wird. Sie können also mehrere Datenbanken mit jeweils einem anderen Zeichensatz haben.

20.2.1 Unterstützte Zeichensätze

Tabelle 20.1 zeigt die im Server zur Verfügung stehenden Zeichensätze. Server-Zeichensätze

Name	Beschreibung
SQL_ASCII	ASCII
EUC_JP	Japanischer EUC
EUC_CN	Chinesischer EUC
EUC_KR	Koreanischer EUC
JOHAB	Koreanischer EUC (Basis-Hangul)
EUC_TW	Taiwanischer EUC
UNICODE	Unicode (UTF-8)
MULE_INTERNAL	Mule Internal Code
LATIN1	ISO 8859-1/ECMA 94 (Lateinisches Alphabet Nr.1)
LATIN2	ISO 8859-2/ECMA 94 (Lateinisches Alphabet Nr.2)
LATIN3	ISO 8859-3/ECMA 94 (Lateinisches Alphabet Nr.3)
LATIN4	ISO 8859-4/ECMA 94 (Lateinisches Alphabet Nr.4)
LATIN5	ISO 8859-9/ECMA 128 (Lateinisches Alphabet Nr.5)
LATIN6	ISO 8859-10/ECMA 144 (Lateinisches Alphabet Nr.6)
LATIN7	ISO 8859-13 (Lateinisches Alphabet Nr.7)
LATIN8	ISO 8859-14 (Lateinisches Alphabet Nr.8)
LATIN9	ISO 8859-15 (Lateinisches Alphabet Nr.9)
LATIN10	ISO 8859-16/ASRO SR 14111 (Lateinisches Alphabet Nr.10)
ISO-8859-5	ISO 8859-5/ECMA 113 (Lateinisch/Kyrillisch)
ISO-8859-6	ISO 8859-6/ECMA 114 (Lateinisch/Arabisch)
ISO-8859-7	ISO 8859-7/ECMA 118 (Lateinisch/Griechisch)
ISO-8859-8	ISO 8859-8/ECMA 121 (Lateinisch/Hebräisch)
KOI8	KOI8-R(U)
WIN	Windows CP1251
ALT	Windows CP866
WIN1256	Windows CP1256 (Arabisch)
TCVN	TCVN-5712/Windows CP1258 (Vietnamesisch)
WIN874	Windows CP874 (Thai)

Tabelle 20.1: Server-Zeichensätze

Wichtig

Vor PostgreSQL 7.2 stand `LATIN5` fälschlicherweise für ISO 8859-5. Seit 7.2 bedeutet `LATIN5` richtig ISO 8859-9. Wenn Sie eine mit Version 7.1 oder früher erzeugte Datenbank in `LATIN5` haben und auf 7.2 oder später umsteigen wollen, sollten Sie auf diese Änderung achten.

Nicht alle Schnittstellen unterstützen alle aufgelisteten Zeichensätze. Der JDBC-Treiber von PostgreSQL unterstützt zum Beispiel nicht `MULE_INTERNAL`, `LATIN6`, `LATIN8` oder `LATIN10`.

20.2.2 Den Zeichensatz einstellen

`initdb` bestimmt den Standardzeichensatz eines PostgreSQL-Datenbankclusters. Zum Beispiel:

```
initdb -E EUC_JP
```

Das setzt den Standardzeichensatz (die Kodierung) auf `EUC_JP` (Extenden Unix Code für Japanisch). Sie können `--encoding` anstatt `-E` verwenden, wenn Sie lieber die langen Optionen eingeben. Wenn weder `-E` noch `--encoding` angegeben wird, dann wird `SQL_ASCII` verwendet.

Sie können eine Datenbank mit einem anderen Zeichensatz erzeugen:

```
createdb -E EUC_KR korean
```

Das erzeugt eine Datenbank namens `korean`, die den Zeichensatz `EUC_KR` verwendet. Das Gleiche kann man auch mit diesem SQL-Befehl erreichen:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

Die Kodierung der Datenbank wird im Systemkatalog `pg_database` gespeichert. Sie können sich diese Informationen zum Beispiel mit der Option `-l` oder dem Befehl `\l` in `psql` ansehen.

```
$ psql -l
            List of databases
   Database    |  Owner  |   Encoding
---------------+---------+---------------
 euc_cn        | t-ishii | EUC_CN
 euc_jp        | t-ishii | EUC_JP
 euc_kr        | t-ishii | EUC_KR
 euc_tw        | t-ishii | EUC_TW
 mule_internal | t-ishii | MULE_INTERNAL
 regression    | t-ishii | SQL_ASCII
 template1     | t-ishii | EUC_JP
 test          | t-ishii | EUC_JP
 unicode       | t-ishii | UNICODE
(9 rows)
```

20.2.3 Automatische Zeichensatzkonvertierung zwischen Client und Server

PostgreSQL unterstützt die automatische Zeichensatzkonvertierung zwischen Client und Server für bestimmte Zeichensätze. Die Konvertierungsinformationen werden im Systemkatalog `pg_conversion` gespeichert. Neuen Konversionen können Sie mit dem SQL-Befehl `CREATE CONVERSION` erzeugen. PostgreSQL enthält einige vordefinierte Konversionen. Diese sind in Tabelle 20.2 aufgeführt.

Server-Zeichensatz	Verfügbare Client-Zeichensätze
SQL_ASCII	SQL_ASCII, UNICODE, MULE_INTERNAL
EUC_JP	EUC_JP, SJIS, UNICODE, MULE_INTERNAL
EUC_CN	EUC_CN, UNICODE, MULE_INTERNAL
EUC_KR	EUC_KR, UNICODE, MULE_INTERNAL
JOHAB	JOHAB, UNICODE
EUC_TW	EUC_TW, BIG5, UNICODE, MULE_INTERNAL
LATIN1	LATIN1, UNICODE MULE_INTERNAL
LATIN2	LATIN2, WIN1250, UNICODE, MULE_INTERNAL
LATIN3	LATIN3, UNICODE, MULE_INTERNAL
LATIN4	LATIN4, UNICODE, MULE_INTERNAL
LATIN5	LATIN5, UNICODE
LATIN6	LATIN6, UNICODE, MULE_INTERNAL
LATIN7	LATIN7, UNICODE, MULE_INTERNAL
LATIN8	LATIN8, UNICODE, MULE_INTERNAL
LATIN9	LATIN9, UNICODE, MULE_INTERNAL
LATIN10	LATIN10, UNICODE, MULE_INTERNAL
ISO_8859_5	ISO_8859_5, UNICODE, MULE_INTERNAL, WIN, ALT, KOI8
ISO_8859_6	ISO_8859_6, UNICODE
ISO_8859_7	ISO_8859_7, UNICODE
ISO_8859_8	ISO_8859_8, UNICODE
UNICODE	EUC_JP, SJIS, EUC_KR, UHC, JOHAB, EUC_CN, GBK, EUC_TW, BIG5, LATIN1 to LATIN10, ISO_8859_5, ISO_8859_6, ISO_8859_7, ISO_8859_8, WIN, ALT, KOI8, WIN1256, TCVN, WIN874, GB18030, WIN1250
MULE_INTERNAL	EUC_JP, SJIS, EUC_KR, EUC_CN, EUC_TW, BIG5, LATIN1 to LATIN5, WIN, ALT, WIN1250, BIG5, ISO_8859_5, KOI8
KOI8	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
ALT	ISO_8859_5, WIN, ALT, KOI8, UNICODE, MULE_INTERNAL
WIN1256	WIN1256, UNICODE
TCVN	TCVN, UNICODE
WIN874	WIN874, UNICODE

Tabelle 20.2: Client/Server-Zeichensatzkonversionen

Um die automatische Zeichensatzkonvertierung zu ermöglichen, müssen Sie PostgreSQL mitteilen, welchen Zeichensatz (welche Kodierung) Sie im Client verwenden wollen. Es gibt mehrere Möglichkeiten, dies zu erreichen:

- Mit dem Befehl `\encoding` in `psql`. Mit `\encoding` können Sie die Clientkodierung jederzeit ändern. Um zum Beispiel die Kodierung auf `SJIS` zu setzen, geben Sie ein:

```
\encoding SJIS
```

- Mit `libpq`-Funktionen. `\encoding` selbst ruft intern `PQsetClientEncoding()` auf.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

  wobei *conn* eine Verbindung zum Server ist und *encoding* der gewünschte Zeichensatz. Wenn die Funktion den Zeichensatz erfolgreich gesetzt hat, ergibt sie 0, ansonsten -1. Der aktuelle Zeichensatz einer Verbindung kann mit der Funktion ermittelt werden:

```
int PQclientEncoding(const PGconn *conn);
```

  Beachten Sie, dass diese Funktion die interne Nummer des Zeichensatzes zurückgibt, nicht einen Namen wie `EUC_JP`. Um die Nummer in einen Zeichensatznamen umzuwandeln, können Sie diese Funktion verwenden:

```
char *pg_encoding_to_char(int encoding_id);
```

- Mit `SET CLIENT_ENCODING TO`. Die Clientkodierung kann mit diesem SQL-Befehl gesetzt werden:

```
SET CLIENT_ENCODING TO 'wert';
```

  Sie können dafür auch die etwas SQL-konformere Syntax `SET NAMES` verwenden:

```
SET NAMES 'wert';
```

  Um die aktuelle Kodierung anzuzeigen, verwenden Sie:

```
SHOW CLIENT_ENCODING;
```

  Um zur Standardkodierung zurückzukehren, verwenden Sie:

```
RESET CLIENT_ENCODING;
```

- Mit `PGCLIENTENCODING`. Wenn die Umgebungsvariable `PGCLIENTENCODING` in der Umgebung des Clients gesetzt ist, wird diese Kodierung automatisch ausgewählt, wenn eine Verbindung zum Server aufgebaut wird. (Nachher kann Sie mit einer der oben genannten Methoden geändert werden.)
- Mit dem Konfigurationsparameter `client_encoding`. Wenn der Parameter `client_encoding` in `postgresql.conf` gesetzt ist, wird diese Clientkodierung automatisch ausgewählt, wenn eine Verbindung zum Server aufgebaut wird. (Nachher kann Sie mit einer der oben genannten Methoden geändert werden.)

Wenn die Konvertierung eines bestimmten Zeichens nicht möglich ist – nehmen wir an, Sie wählen `EUC_JP` für den Server und `LATIN1` für den Client, können einige japanische Zeichen nicht in `LATIN1` übertragen werden, wird das Zeichen durch seine hexadezimalen Bytewerte in Klammern dargestellt, zum Beispiel `(826C)`.

20.2.4 Weitere Informationsquellen

Hier sind ein paar gute Quellen, um über die verschiedenen Zeichensatzsysteme etwas zu erfahren.

```
ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf
```

In Abschnitt 3.2 gibt es detaillierte Beschreibungen von EUC_JP, EUC_CN, EUC_KR und EUC_TW.

```
http://www.unicode.org/
```

Die Website des Unicode Consortium RFC 2044. Hier wird UTF-8 definiert.

20.3 Einbyte-Zeichensatz-Konvertierung

Dieses Feature können Sie mit der `configure`-Option `--enable-recode` einschalten. Diese Option wurde früher als "Kyrillische Rekodierungs-Unterstützung" bezeichnet, was aber nicht ihre vollen Fähigkeiten beschreibt. Man kann damit *alle* Einbyte-Zeichensätze konvertieren lassen.

Diese Methode verwendet eine Datei `charset.conf` im Datenverzeichnis zur Konfiguration. Diese ist eine übliche Text-Konfigurationsdatei, in der Leerzeichen und Newlines Felder und Datensätze trennen und mit # ein Kommentar anfängt. Drei Schlüsselwörter mit folgender Syntax werden erkannt:

```
BaseCharset server_zeichensatz
RecodeTable quell_zeichensatz ziel_zeichensatz dateiname
HostCharset host_angabe host_zeichensatz
```

BaseCharset definiert den Zeichensatz des Datenbankservers. Alle Zeichensatznamen werden nur intern in `charset.conf` verwendet; Sie können also tippfreundliche Namen wählen.

`RecodeTable`-Sätze bestimmen Konvertierungstabellen zwischen Server und Client. Der Dateiname ist relativ zum Datenverzeichnis. Das Format der Tabellendatei ist sehr einfach. Es gibt keine Schlüsselwörter, und Zeichenkonversionen werden durch ein Paar dezimaler oder hexadezimaler (mit vorgestelltem `0x`) Werte auf einzelnen Zeilen dargestellt:

```
zeichenwert konvertierter_zeichenwert
```

Im Verzeichnis `src/data/` in der Quellendistribution können Sie ein Beispiel für die Datei `charset.conf` und ein paar Konvertierungstabellen finden.

HostCharset-Sätze legen die Client-Zeichensätze nach IP-Adresse fest. Sie können eine einzelne IP-Adresse, eine IP-Maske über eine Adresse oder eine Spanne von IP-Adressen verwenden (z.B. `127.0.0.1`, `192.168.1.100/24`, `192.168.1.20-192.168.1.40`).

Die Datei `charset.conf` wird immer bis zum Ende verarbeitet, so dass Sie einfach Ausnahmen von vorangehenden Regeln bestimmen können.

Da diese Lösung auf den IP-Adressen der Clients aufbaut, gibt es offensichtlich auch einige Einschränkungen. Sie können nicht auf einem Host zur gleichen Zeit mehrere Zeichensätze verwenden. Es wird auch unbequem, wenn Sie auf einem Client mehrere Betriebssysteme booten. Wenn diese Einschränkungen Sie aber nicht stören und Sie keine Mehrbyte-Zeichen benötigen, dann ist dies eine einfache und effektive Lösung.

21

Routinemäßige Datenbankwartungsaufgaben

Es gibt einige wenige routinemäßige Wartungsaufgaben, die durchgeführt werden müssen, damit ein PostgreSQL-Server problemlos laufen kann. Die hier beschriebenen Aufgaben müssen regelmäßig wiederholt werden und können leicht mit normalen Unix-Werkzeugen wie `cron` automatisiert werden. Aber es liegt in der Verantwortung des Datenbankadministrators, die passenden Skripts einzurichten und zu überwachen, dass sie erfolgreich ausgeführt werden.

Eine offensichtliche Wartungsaufgabe ist die regelmäßige Erstellung von Sicherungskopien der Daten. Ohne eine Sicherungskopie, die auf dem letzten Stand ist, haben Sie keine Chance, nach einer Katastrophe (Festplattenausfall, Feuer, aus Versehen eine wichtige Tabelle gelöscht usw.) Ihre Daten wiederherstellen zu können. Die in PostgreSQL verfügbaren Datensicherungs- und wiederherstellungsmechanismen werden ausführlich in Kapitel 22 besprochen.

Die andere Hauptaufgabe der Wartung ist das regelmäßige *Vacuum* (zu Deutsch “Staubsaugen”) der Datenbank. Dieses Thema wird in Abschnitt 21.1 besprochen.

Noch etwas, das regelmäßige Aufmerksamkeit erfordert, ist die Verwaltung der Logdateien. Das wird in Abschnitt 21.3 besprochen.

Verglichen mit anderen Datenbankprodukten hat PostgreSQL einen geringen Wartungsaufwand. Wenn Sie den sich stellenden Aufgaben nichtsdestoweniger die angebrachte Aufmerksamkeit widmen, dann werden Sie dadurch Ihre Arbeit mit dem System angenehmer und produktiver gestalten.

21.1 Routinemäßiges Vacuum

In PostgreSQL muss aus mehreren Gründen regelmäßig der Befehl `VACUUM` ausgeführt werden:

1. um den von aktualisierten oder gelöschten Zeilen belegten Speicherplatz wiederzugewinnen
2. um die vom PostgreSQL-Anfrageplaner verwendeten Datenstatistiken zu aktualisieren
3. um sich gegen den Verlust sehr alter Daten durch Überlauf der Transaktionsnummern zu schützen.

Die Häufigkeit und das Ausmaß der aus jedem dieser Gründe ausgeführten `VACUUM`-Operationen hängt von der individuellen Situation ab. Datenbankadministratoren müssen daher diese Angelegenheiten verstehen und eine passende Wartungsstrategie entwickeln. Dieser Abschnitt konzentriert sich auf das Erklären der allgemeinen Problematik; Einzelheiten der Befehlssyntax und so weiter finden Sie auf der `VACUUM`-Referenzseite.

Seit PostgreSQL 7.2 kann die Standardform von VACUUM parallel mit normalen Datenbankoperationen (Anfragen, Einfügen, Aktualisieren, Löschen, aber nicht Änderungen an der Tabellendefinition) laufen. Routinemäßiges Vacuum ist daher kein so großer Einschnitt in die Datenbankoperation mehr, wie es in früheren Versionen war, und es ist auch nicht mehr so wichtig, es an Tageszeiten mit niedriger Belastung auszuführen.

21.1.1 Speicherplatz wiedergewinnen

In PostgreSQL wird bei einem UPDATE oder DELETE einer Zeile die alte Zeilenversion nicht sofort entfernt. Dieses Verhalten ist notwendig um das Multiversionssystem für den Mehrbenutzerbetrieb zu ermöglichen (siehe Kapitel 12): Die Zeilenversion darf nicht gelöscht werden, solange sie potenziell noch in anderen Transaktionen sichtbar ist. Aber irgendwann wird sich keine Transaktion mehr für eine alte oder gelöschte Zeilenversion interessieren. Der davon belegte Platz muss für die Wiederverwendung durch neue Zeilen wiedergewonnen werden, um zu verhindern, dass die Speicherplatzanforderungen ins Unendliche wachsen. Das wird von VACUUM erledigt.

Eine Tabelle, in der oft aktualisiert oder gelöscht wird, muss also offensichtlich öfter “gevacuumt” werden als Tabellen, die selten verändert werden. Es könnte hilfreich sein, einen cron-Auftrag zu registrieren, der nur in ausgewählten Tabellen VACUUM ausführt und Tabellen, von denen bekannt ist, dass sie sich nicht oft ändern, auslässt. Aber das ist wahrscheinlich nur sinnvoll, wenn Sie große oft aktualisierte und große selten aktualisierte Tabellen haben, denn der zusätzliche Aufwand, um VACUUM in einer kleinen Tabelle auszuführen, ist nicht groß genug, um sich darüber Gedanken zu machen.

Die Standardform vom VACUUM wird am besten mit dem Ziel angewendet, einen relativ stabilen Festplattenplatzverbrauch zu erhalten. Die Standardform findet alte Zeilenversionen und macht deren Platz für die Wiederverwendung in der Tabelle verfügbar, aber sie versucht nicht zu sehr die Tabellendatei zu verkürzen und den Platz an das Betriebssystem zurückzugeben. Wenn Sie Platz an das Betriebssystem zurückgeben wollen, können Sie VACUUM FULL verwenden – aber welchen Sinn hat es, Festplattenplatz an das Betriebssystem zurückzugeben, der sowieso bald wieder verwendet werden muss? Einigermaßen häufige VACUUM-Läufe in der Standardform sind für oft aktualisierte Tabellen ein besserer Ansatz als seltene VACUUM FULL-Läufe.

Für die meisten Anwender wird empfohlen, einmal am Tag, wenn wenig Betrieb ist, VACUUM für die ganze Datenbank auszuführen und als Ergänzung, wenn nötig, häufigeres Vacuum von oft aktualisierten Tabellen. (Wenn Sie mehrere Datenbanken in einem Cluster haben, vergessen Sie nicht, VACUUM in jeder einzelnen auszuführen; das Programm vacuumdb kann da hilfreich sein.) Verwenden Sie das einfache VACUUM, nicht VACUUM FULL, für routinemäßiges Vacuum zur Platzwiedergewinnung.

VACUUM FULL ist zu empfehlen, wenn Sie wissen, dass Sie die Mehrzahl der Zeilen in einer Tabelle gelöscht haben, sodass die stabile Größe der Tabelle mit dem aggressiveren Ansatz von VACUUM FULL erheblich verringert werden kann.

Wenn Sie eine Tabelle haben, deren Inhalt ab und zu komplett gelöscht wird, sollten Sie in Erwägung ziehen, das mit TRUNCATE zu tun, anstatt DELETE gefolgt von VACUUM zu verwenden.

21.1.2 Planerstatistiken aktualisieren

Der PostgreSQL-Anfrageplaner benötigt statistische Informationen über den Inhalt der Tabellen, um gute Pläne für Anfragen erzeugen zu können. Diese Statistiken werden vom Befehl ANALYZE zusammengetragen, der alleine oder als zusätzlicher Schritt in VACUUM aufgerufen werden kann. Es ist wichtig, einigermaßen genaue Statistiken zu haben, ansonsten kann die schlechte Wahl von Plänen die Datenbankleistung beeinträchtigen.

Häufiges Aktualisieren der Statistiken ist, wie beim Vacuum zur Platzgewinnung, für oft aktualisierte Tabellen nützlicher als für selten aktualisierte. Aber selbst bei oft aktualisierten Tabellen muss es nicht unbedingt notwendig sein, die Statistiken zu aktualisieren, wenn sich die statistische Verteilung kaum ver-

ändert. Am einfachsten ist es, sich zu überlegen, wie sehr sich die Minimal- und Maximalwerte in den Tabellenspalten ändern. Eine timestamp-Spalte, zum Beispiel, die die Zeit der letzten Zeilenaktualisierung enthält, hat einen ständig steigenden Maximalwert; so eine Spalte benötigt wahrscheinlich häufigeres Aktualisieren der Statistik als, sagen wir, eine Spalte, die speichert, welche URLs einer Website abgerufen wurden. Die URL-Spalte mag sich vielleicht genauso oft ändern, aber die statistische Verteilung der Werte ändert sich wahrscheinlich nur sehr langsam.

`ANALYZE` kann man auch für bestimmte Tabellen und sogar nur für bestimmte Spalten einer Tabelle ausgeführt werden, sodass man einige Statistiken öfter als andere aktualisieren kann, wenn es die Anwendung erfordert. In der Praxis ist der Nutzen dieser Fähigkeit allerdings zweifelhaft. Seit PostgreSQL 7.2 ist `ANALYZE` sogar bei großen Tabellen ziemlich schnell, weil es nur eine zufällige Auswahl der Tabellenzeilen verwendet, anstatt jede einzelne Zeile zu lesen. Es ist also wahrscheinlich viel einfacher, es einfach ab und zu für die ganze Datenbank auszuführen.

Tipp

Obwohl ein verschieden häufiges `ANALYZE` für verschiedene Spalten in der Regel nicht sehr produktiv ist, kann es sehr wohl sinnvoll sein, das Ausmaß der gesammelten Statistiken einzustellen. Spalten, die oft in `WHERE`-Klauseln verwendet werden und sehr unregelmäßige Datenverteilungen haben, könnten ein detaillierteres Datenhistogramm gebrauchen als andere. Sehen Sie bei `ALTER TABLE SET STATISTICS` nach.

Für die meisten Anwender wird empfohlen, `ANALYZE` einmal am Tag, wenn wenig Betrieb ist, für die ganze Datenbank auszuführen; das kann am besten mit einem nächtlichen `VACUUM` verbunden werden. Anwender mit Tabellenstatistiken, die sich relativ langsam verändern, könnten jedoch zu der Feststellung gelangen, dass das übertrieben ist und dass weniger häufige `ANALYZE`-Läufe ausreichend sind.

21.1.3 Transaktionsnummernüberlauf verhindern

Im MVCC-System von PostgreSQL spielt das Vergleichen von Transactionsnummern (XID) eine wichtige Rolle: Eine Zeilenversion mit einer Einfügungs-XID, die größer als die XID der aktuellen Transaktion ist, ist "in der Zukunft" und sollte für die aktuelle Transaktion nicht sichtbar sein. Da aber Transaktionsnummern eine begrenzte Größe haben (gegenwärtig 32 Bits), kann es in einem Cluster, der eine lange Zeit läuft (mehr als 4 Milliarden Transaktionen), zum **Transaktionsnummernüberlauf** kommen: Der XID-Zähler fängt wieder bei null an und plötzlich werden Transaktionen aus der Vergangenheit so aussehen, als ob sie in der Zukunft wären, was bedeutet, dass deren Ergebnisse unsichtbar werden. Kurz gesagt würden Sie die totale Datenverlustkatastrophe erleben. (Die Daten sind eigentlich noch da, aber das ist sicherlich nicht sehr beruhigend, wenn man nicht auf sie zugreifen kann.)

Vor PostgreSQL 7.2 war die einzige Verteidigung gegen Transaktionsnummernüberlauf, dass man mindestens alle 4 Milliarden Transaktionen `initdb` neu ausführt. Dieser Weg war für Anwendungen mit viel Betrieb nicht zufriedenstellend und daher wurde eine bessere Lösung erdacht. Der neue Ansatz erlaubt es, dass der Server endlos laufen kann, ohne `initdb` oder irgendeinen Neustart. Der Preis dafür ist diese Wartungsanforderung: *Jede Tabelle in der Datenbank muss mindestens einmal alle eine Milliarde Transaktionen gevacuumt werden*.

In der Praxis ist das keine besonders umständliche Forderung, aber da die Konsequenz, wenn man sich nicht daran hält, kompletter Datenverlust sein kein (nicht nur verschwendeter Speicherplatz oder schlechte Leistung), wurden einige besondere Vorkehrungen errichtet, um den Datenbankadministratoren zu helfen, die Zeit seit dem letzten `VACUUM` zu verfolgen. Der Rest dieses Abschnitts gibt dazu Einzelheiten.

Der neue Ansatz beim XID-Vergleich kennt zwei besondere XIDs, Nummer 1 und 2 (`BootstrapXID` und `FrozenXID`). Diese beiden XIDs werden immer als älter als alle anderen XIDs betrachtet. Normale XIDs (die größer als 2) werden mit Arithmetik modulo 2^{31} verglichen. Das bedeutet, dass es für jede normale XID zwei Milliarden XIDs, die "älter", und zwei Milliarden, die "neuer" sind, gibt; anders ausge-

drückt ist der Raum der normalen XIDs kreisförmig ohne Endpunkte. Wenn daher eine Zeilenversion mit einer bestimmten XID erzeugt worden ist, scheint sie für die nächsten zwei Milliarden Transaktionen als "in der Vergangenheit", egal welche normale XID es ist. Wenn die Zeilenversion nach mehr als zwei Milliarden Transaktionen noch existiert, dann erscheint sie plötzlich in der Zukunft zu sein. Um Datenverlust zu vermeiden, muss alten Zeilenversionen irgendwann die XID `FrozenXID` zugewiesen werden (sie müssen also quasi "eingefroren" werden), bevor Sie zwei Milliarden Transaktionen alt werden. Wenn ihnen einmal diese besondere XID zugewiesen wurde, dann werden Sie für alle normalen Transaktionen als "in der Vergangenheit" erscheinen, ungeachtet der Überlaufproblematik, und daher werden sie bis zum Löschen gültig bleiben, egal, wie lange das sein wird. Diese Neuzuweisung der XID wird von `VACUUM` erledigt.

Die normale Strategie von `VACUUM` ist es, die `FrozenXID` jeder Zeilenversion zuzuweisen, deren XID mehr als eine Milliarde Transaktionen in der Vergangenheit liegt. Dadurch wird die originale Einfügungs-XID so lange erhalten, bis sie wahrscheinlich nicht mehr von Interesse ist. (Die meisten Zeilenversionen werden wahrscheinlich entstehen und verschwinden, ohne jemals "eingefroren" zu werden.) Mit dieser Strategie ist der höchste sichere Zeitraum zwischen `VACUUM`-Läufen für eine Tabelle genau eine Milliarde Transaktionen: Wenn Sie länger warten, kann es sein, dass eine Zeilenversion, die beim letzten Mal noch nicht alt genug war, um umnummeriert zu werden, jetzt zu alt ist und in der Zukunft liegt. Das hieße, sie ist für Sie verloren. (Natürlich wird sie nach weiteren zwei Milliarden Transaktionen wieder auftauchen, aber das hilft Ihnen sicher nicht weiter.)

Da regelmäßige `VACUUM`-Läufe aus den schon beschriebenen Gründen sowieso nötig sind, ist es unwahrscheinlich, dass eine Tabelle eine ganze Milliarde Transaktionen in diese nicht mit einbezogen würde. Um aber Administratoren zu helfen, dass diese Beschränkung eingehalten wird, speichert `VACUUM` Transaktionsnummernstatistiken in der Systemtabelle `pg_database`. Insbesondere wird die Spalte `datfrozenxid` der zu einer Datenbank gehörigen `pg_database`-Zeile am Ende jeder datenbankweiten `VACUUM`-Operation (d.h. `VACUUM`, das keine bestimmte Tabelle benennt) aktualisiert. Der in diesem Feld gespeicherte Wert ist quasi die vom letzten `VACUUM` verwendete Einfriergrenze: Alle normalen XIDs, die älter als der Grenzwert sind, sind in jener Datenbank garantiert durch `FrozenXID` ersetzt worden. Ein bequemer Weg, sich diese Information anzusehen, ist, folgende Anfrage auszuführen:

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

Die Spalte `age` misst die Anzahl der Transaktionen von jenem Grenzwert bis zur XID der aktuellen Transaktion.

Mit der Standardstrategie fängt die Spalte `age` in einer Datenbank direkt nach einem `VACUUM` bei einer Milliarde an. Wenn `age` sich zwei Milliarden nähert, muss `VACUUM` bald wieder ausgeführt werden, um Überlaufprobleme zu vermeiden. Wir empfehlen, in jeder Datenbank mindestens alle halbe Milliarde (500 Millionen) Transaktionen `VACUUM` auszuführen, damit reichlich Sicherheitsspielraum bleibt. Um Ihnen zu helfen, dieser Regel zu folgen, gibt jedes datenbankweite `VACUUM` automatisch eine Warnung aus, wenn irgendein `pg_database`-Eintrag für `age` mehr als 1,5 Milliarden Transaktionen zeigt, zum Beispiel:

```
play=# VACUUM;
WARNING:  Some databases have not been vacuumed in 1613770184 transactions.
        Better vacuum them within 533713463 transactions,
        or you may have a wraparound failure.
VACUUM
```

`VACUUM` mit der Option `FREEZE` verfolgt eine aggressivere Strategie beim Einfrieren: Zeilenversionen werden eingefroren, wenn sie alt genug sind, um in allen offenen Transaktionen gültig zu sein. Insbesondere gilt, wenn `VACUUM FREEZE` in einer ansonsten gerade unbenutzten Datenbank durchgeführt wird, ist garantiert, dass *alle* Zeilenversionen in dieser Datenbank eingefroren werden. Wenn dann die Datenbank nicht weiter verändert wird, muss sie auch nicht mehr gevacuumt werden, um Problemen mit Transaktionsnummernüberlauf zu begegnen. Diese Technik wird von `initdb` verwendet, um die Datenbank

template0 vorzubereiten. Sie sollte auch für Datenbanken verwendet werden, bei denen in pg_database datallowconn = false gesetzt wird, da man in Datenbanken, mit denen man nicht verbinden kann, auch kein VACUUM ausführen kann. Beachten Sie, dass VACUUM für Datenbanken mit datallowconn = false keine automatischen Warnungen über einen zu langen Zeitraum seit dem letzten VACUUM ausgibt, damit für solche Datenbanken keine falschen Warnungen ausgegeben werden; daher müssen Sie selbst darauf achten, dass solche Datenbanken ordnungsgemäß eingefroren werden.

21.2 Routinemäßiges Reindizieren

In bestimmten Fällen ist PostgreSQL unfähig, B-Tree-Indexseiten wiederzuverwenden. Das Problem ist, wenn indizierte Zeilen gelöscht werden, können diese Indexseiten nur von Zeilen mit ähnlichen Werten wiederverwendet werden. Wenn zum Beispiel indizierte Zeilen gelöscht werden und die neu eingefügten/aktualisierten Zeilen viel höhere Werte haben, können die neuen Zeilen den von den gelöschten Zeilen freigegebenen Platz im Index nicht verwenden. Solche neuen Zeilen müssen vielmehr in neuen Indexseiten abgelegt werden. In solchen Fällen wächst der von dem Index belegte Platz auf der Festplatte ins Endlose, selbst wenn häufig VACUUM ausgeführt wird.

Als Lösung können Sie regelmäßig den Befehl REINDEX ausführen, um von gelöschten Zeilen belegte Seiten freizugeben. Außerdem gibt es contrib/reindexdb, was die ganze Datenbank reindizieren kann.

21.3 Logdateiverwaltung

Es ist ratsam, die Logausgabe des Servers irgendwo zu speichern, anstatt sie einfach an /dev/null zu schicken. Der Log ist unschätzbar, wenn man Probleme diagnostizieren muss. Die Logausgabe kann jedoch ziemlich umfangreich werden (besonders bei einer Einstellung mit viel Debugausgaben) und man will sie in der Regel nicht endlos lange aufbewahren. Daher müssen Sie die Logdateien "rotieren", damit neue Logdateien angefangen und alte ab und zu weggeworfen werden.

Wenn Sie einfach den stderr von postmaster in eine Datei umleiten, können Sie die Logdatei nur verkürzen, indem Sie den Server anhalten und wieder neu starten. Das mag für Entwicklungsumgebungen ausreichend sein, aber einen Produktionsserver sollten Sie so nicht laufen lassen.

Die einfachste Lösung zur Verwaltung der Logausgabe ist, sie an Syslog zu schicken und die Dateirotierung Syslog zu überlassen. Um das zu veranlassen, setzen Sie den Konfigurationsparameter Syslog in postgresql.conf auf 2 (um nur mit Syslog zu loggen). Dann können Sie an den Syslog-Daemon das Signal SIGHUP schicken, wenn Sie wollen, dass der eine neue Logdatei anfängt.

Auf vielen Systemen ist Syslog jedoch nicht sehr zuverlässig, besonders bei großen Logmeldungen; Meldungen könnten gerade dann abgeschnitten oder fallen gelassen werden, wenn man Sie sie am meisten benötigt. Dann ist es vielleicht angebrachter, wenn Sie stderr von postmaster per Pipe mit einem Logrotierprogramm verbinden. Wenn Sie den Server mit pg_ctl starten, wird stderr von postmaster automatisch an stdout umgeleitet, sodass Sie nur einen einfachen Pipe-Befehl benötigen:

```
pg_ctl start | logrotate
```

Die PostgreSQL-Distribution enthält kein passendes Logrotierprogramm, aber im Internet werden Sie eine Auswahl finden; zum Beispiel gibt es in der Apache-Distribution eins.

22

Datensicherung und wiederherstellung

PostgreSQL-Datenbanken sollten, wie alles, was wertvolle Daten enthält, regelmäßig gesichert werden. Obwohl das Verfahren eigentlich ganz einfach ist, es es wichtig, dass Sie die zugrunde liegenden Techniken und Annahmen verstehen.

Es gibt zwei grundlegend verschiedene Ansätze, um eine PostgreSQL-Datenbank zu sichern:

- SQL-Dump
- Archivierung auf Dateisystemebene

22.1 SQL-Dump

Die Idee der SQL-Dump-Methode ist es, eine Textdatei zu erzeugen, die SQL-Befehle enthält, die, wenn man Sie an den Server zurückschickt, die Datenbank wieder so erzeugen, wie sie zum Zeitpunkt der Sicherung war. PostgreSQL bietet zu diesem Zweck das Hilfsprogramm `pg_dump` an. Dieser Befehl wird generell so verwendet:

```
pg_dump dbname > ausgabedatei
```

Wie Sie sehen, schreibt `pg_dump` seine Ergebnisse auf den Standardausgabestrom. Unten werden wir sehen, wie nützlich das sein kann.

`pg_dump` ist eine normale PostgreSQL-Clientanwendung (wenn auch eine ziemlich schlaue). Das bedeutet, dass Sie die Datensicherung von einer beliebigen Maschine im Netzwerk ausführen können, wenn diese Zugriff auf die Datenbank hat. Aber bedenken Sie, dass `pg_dump` keine besonderen Zugriffsrechte hat. Sie müssen insbesondere Leseerlaubnis für alle Tabellen haben, die Sie sichern wollen; in der Praxis müssen Sie fast immer ein Datenbank-Superuser sein.

Um anzugeben, welchen Datenbankserver `pg_dump` kontaktieren soll, verwenden Sie die Kommandozeilenoptionen `-h` *`host`* und `-p` *`port`*. Wenn nichts angegeben wird, dann wird der lokale Host oder der, den die Umgebungsvariable `PGHOST` angibt, verwendet. Ähnlich wird der Port durch die Umgebungsvariable `PGPORT` oder den eingebauten Vorgabewert bestimmt, wenn kein anderer angegeben wird. (Günstigerweise hat der Server normalerweise den gleichen eingebauten Vorgabewert.)

Wie jede andere PostgreSQL-Clientanwendung verbindet `pg_dump`, wenn nichts anderes angegeben wird, mit der Datenbank, die den gleichen Namen wie der aktuelle Betriebssystembenutzer hat. Um das zu übergehen, geben Sie entweder die Option `-U` an oder setzen Sie die Umgebungsvariable `PGUSER`.

Bedenken Sie, dass Verbindungen mit pg_dump den normalen Clientauthentifizierungsmechanismen unterworfen sind (welche in Kapitel 19 beschrieben sind).

Mit pg_dump erzeugte Dumps sind intern konsistent, das heißt, dass Aktualisierungen, die in der Datenbank vorgenommen werden, während pg_dump läuft, nicht im Dump sein werden. pg_dump blockiert keine anderen Operationen in der Datenbank, während es läuft. (Ausnahmen sind Operationen, die zur Ausführung eine exklusive Sperre benötigen, wie zum Beispiel VACUUM FULL.)

Wichtig

Wenn Ihr Datenbankschema OIDs verwendet (zum Beispiel als Fremdschlüssel), müssen Sie pg_dump anweisen, die OIDs mit zu sichern. Dazu verwenden Sie die Kommandozeilenoption -o. "Large Objects" werden auch ohne weiteres Zutun nicht ausgegeben. Schauen Sie auf die Referenzseite von pg_dump, wenn Sie Large Objects verwenden.

22.1.1 Den Dump wiederherstellen

Die von pg_dump erzeugten Textdateien sind dafür gedacht, mit dem Programm psql gelesen zu werden. Die generelle Form eines Befehls, um einen Dump zurückzuspielen, ist

```
psql dbname < eingabedatei
```

wobei *eingabedatei* die Datei ist, die Sie oben bei pg_dump als *ausgabedatei* angegeben hatten. Die Datenbank *dbname* wird von diesem Befehl nicht erzeugt, Sie müssen sie aus template0 selbst erzeugen, bevor Sie psql ausführen (zum Beispiel mit createdb -T template0 *dbname*). psql hat ähnliche Optionen wie pg_dump, um den Namen des Datenbankservers und den Benutzernamen zu kontrollieren. Auf der Referenzseite finden Sie weitere Informationen.

Wenn die Objekte in der ursprünglichen Datenbank verschiedene Eigentümer hatten, wird der Dump psql anweisen, der Reihe nach als jeden Benutzer zu verbinden und dann die entsprechenden Objekte zu erzeugen. Dadurch werden die Eigentumsverhältnisse richtig wiederhergestellt. Das bedeutet jedoch auch, dass all diese Benutzer schon vorhanden sein müssen, und ferner, dass Sie die Erlaubnis haben müssen, als jeder dieser Benutzer zu verbinden. Es kann daher notwendig sein, dass Sie die Clientauthentifizierungseinstellungen vorübergehend etwas lockern.

Da pg_dump und psql in Pipes schreiben bzw. aus Pipes lesen können, kann man eine Datenbank direkt von einem Server zu einem anderen übertragen; zum Beispiel:

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

Wichtig

Die von pg_dump erzeugten Dumps sind auf template0 basierend. Das bedeutet, dass Sprachen, Prozeduren usw., die zu template1 hinzugefügt wurden, von pg_dump mit ausgegeben werden. Daher gilt, wenn Sie bei der Wiederherstellung eine selbst angepasste template1-Datenbank haben, müssen Sie die leere Datenbank aus template0 erzeugen, wie im Beispiel oben.

22.1.2 pg_dumpall verwenden

Der oben beschriebene Mechanismus ist umständlich und unangebracht, wenn man einen ganzen Datenbankcluster sichern will. Aus diesem Grund steht das Programm pg_dumpall zur Verfügung. pg_dumpall sichert alle Datenbanken in einem Cluster und sorgt auch dafür, dass globale Daten, wie Benutzer und Gruppen, erhalten werden. Aufgerufen wird pg_dumpall einfach so:

```
pg_dumpall > ausgabedatei
```

Die sich daraus ergebenden Dumps können mit `psql` wie oben beschrieben wiederhergestellt werden. Aber in diesem Fall müssen Sie auf jeden Fall Superuser-Rechte haben, da das zur Wiederherstellung der Benutzer- und Gruppeninformationen erforderlich ist.

22.1.3 Große Datenbanken

Da PostgreSQL Tabellen zulässt, die größer als die maximale Dateigröße auf Ihrem System sind, kann es problematisch sein, so eine Tabelle in eine Datei zu sichern, da die daraus resultierende Datei wahrscheinlich größer als die von Ihrem System erlaubte Dateigröße sein würde. Da `pg_dump` in den Standardausgabestrom schreiben kann, können Sie Unix-Standardwerkzeuge verwenden, um dieses mögliche Problem zu umgehen.

Komprimierte Dumps verwenden. Sie können Ihr Lieblingskomprimierungsprogramm verwenden, zum Beispiel `gzip`.

```
pg_dump dbname | gzip > dateiname.gz
```

Die Wiederherstellung geht mit

```
createdb dbname
gunzip -c dateiname.gz | psql dbname
```

oder

```
cat dateiname.gz | gunzip | psql dbname
```

`split` verwenden. Das Programm `split` ermöglicht Ihnen, die Ausgabe in Stücke zu teilen, die eine für das Dateisystem akzeptable Größe haben. Zum Beispiel, um 1 Megabyte große Stücke zu erzeugen:

```
pg_dump dbname | split -b 1m - dateiname
```

Die Wiederherstellung geht mit

```
createdb dbname
cat dateiname* | psql dbname
```

Das Dump-Format “custom” verwenden. Wenn PostgreSQL auf einem System mit der Komprimierungsbibliothek `zlib` kompiliert wurde, kann das Dump-Format “custom” die Daten beim Schreiben in die Ausgabedatei komprimieren. Für große Datenbanken ergibt das eine Ausgabedatei mit etwa der Größe, die mit `gzip` erreicht wird, die aber den zusätzlichen Vorteil hat, dass die Tabellen selektiv wiederhergestellt werden können. Der folgende Befehl sichert eine Datenbank in diesem speziellen Format:

```
pg_dump -Fc dbname > dateiname
```

Einzelheiten finden Sie auf den Referenzseiten zu `pg_dump` und `pg_restore`.

22.1.4 Vorbehalte

`pg_dump` (und als Konsequenz auch `pg_dumpall`) hat ein paar Beschränkungen, welche daher rühren, dass es schwierig ist, bestimmte Informationen aus den Systemkatalogen wieder zusammenzusetzen.

Insbesondere ist die Reihenfolge, in der `pg_dump` die Objekte ausgibt, nicht sonderlich intelligent. Das kann zum Beispiel zu Problemen führen, wenn Funktionen in Spaltenvorgabewerten verwendet werden. Die einzige Lösung ist, den Dump von Hand richtig zu sortieren. Wenn Sie zwischen Objekten gegenseitige Abhängigkeiten erzeugt haben, wird noch mehr Arbeit auf Sie zukommen.

Wegen der Rückwärtskompatibilität gibt pg_dump in der normalen Einstellung keine Large Objects aus. Um Large Objects mit zu sichern, müssen Sie entweder das "custom" oder das TAR-Dumpformat und die Option -b von pg_dump verwenden. Einzelheiten finden Sie auf der Referenzseite. Das Verzeichnis contrib/pg_dumplo im PostgreSQL-Quellbaum enthält ein Programm, das Large Objects auch sichern kann.

Bitte machen Sie sich mit der Referenzseite zu pg_dump vertraut.

22.2 Archivierung auf Dateisystemebene

Eine alternative Datensicherungsstrategie ist es, die Dateien, die PostgreSQL zum Speichern der Daten in der Datenbank verwendet, direkt zu kopieren. In Abschnitt 16.2 wurde beschrieben, wo diese Dateien sich befinden, aber wenn Sie sich für diese Methode interessieren, haben Sie sie sicher schon selbst gefunden. Um diese Dateien zu sichern, können Sie jede Methode verwenden, mit der Sie auch sonst Ihr Dateisystem sichern, zum Beispiel:

```
tar -cf backup.tar /usr/local/pgsql/data
```

Es gibt allerdings zwei Einschränkungen, die diese Methode unpraktisch oder zumindest der pg_dump-Methode unterlegen machen:

1. Um eine brauchbare Sicherungsdatei zu erhalten, *muss* der Datenbankserver gestoppt werden. Teilmaßnahmen, wie alle Verbindungen zu unterbinden, funktionieren nicht, da immer irgendwelche Daten in einem Zwischenspeicher sind. Daher wird auch nicht empfohlen, Dateisystemen zu vertrauen, die behaupten, "konsistente Schnappschüsse" zu ermöglichen. Informationen, wie Sie den Server anhalten können, finden Sie in Abschnitt 16.6.

   Es bedarf keiner Erwähnung, dass Sie den Server auch herunterfahren müssen, bevor Sie die Daten wiederherstellen.

2. Wenn Sie findig waren und Einzelheiten über die Dateisystemstruktur der Daten herausgefunden haben, sind Sie vielleicht versucht, nur einzelne Tabellen oder Datenbanken zu sichern oder wiederherzustellen, indem Sie nur die entsprechenden Dateien oder Verzeichnisse kopieren. Das funktioniert *nicht*, weil die in diesen Dateien enthaltenen Informationen nur die halbe Wahrheit darstellen. Die andere Hälfte ist in den Commit-Log-Dateien pg_clog/*, welche den Status aller Transaktionen (abgeschlossen/abgebrochen/weder noch) enthalten. Eine Tabellendatei ist nur mit diesen Informationen brauchbar. Natürlich ist es auch nicht möglich, nur eine Tabelle und die zugehörigen pg_clog-Daten wiederherzustellen, weil das alle anderen Tabellen im Cluster unbrauchbar machen würde.

Bedenken Sie auch, dass eine Kopie der Dateistruktur auch nicht unbedingt kleiner als ein SQL-Dump sein wird. Im Gegenteil, sie wird höchstwahrscheinlich größer sein. (pg_dump muss zum Beispiel den Inhalt der Indexe nicht sichern, sondern nur die Befehle, um sie neu zu erzeugen.)

22.3 Umstieg zwischen PostgreSQL-Versionen

Generell ändert sich das interne Datenformat bei jeder neuen Hauptversion von PostgreSQL. Das betrifft nicht die unterschiedlichen "Patch-Level" innerhalb einer Hauptversion, diese haben immer kompatible Speicherformate. Zum Beispiel sind die Versionen 7.0.1, 7.1.2 und 7.2 nicht kompatibel, aber 7.1.1 und 7.1.2 sind es. Wenn Sie zwischen kompatiblen Versionen wechseln, können Sie einfach den Datenbereich auf der Festplatte mit den neuen Programmdateien verwenden. Ansonsten müssen Sie Ihre Daten mit pg_dump sichern und dann auf dem neuen Server wiederherstellen. (Es gibt mehrere Kontrollmechanismen, die dafür sorgen, dass Sie nichts falsch machen. Sie können also nichts kaputt machen, wenn Sie

etwas durcheinander bringen.) Der genaue Installationsvorgang wird in diesem Abschnitt nicht beschrieben; Einzelheiten dazu finden Sie in Kapitel 14.

Die geringste Ausfallzeit können Sie erreichen, indem Sie den neuen Server in ein anderes Verzeichnis installieren und den alten und den neuen Server parallel, mit verschiedenen Portnummern, ausführen. Dann können Sie einen Befehl wie

```
pg_dumpall -p 5432 | psql -d template1 -p 6543
```

verwenden, um Ihre Daten zu übertragen. Oder wenn Sie wollen, können Sie eine Zwischendatei verwenden. In jedem Fall können Sie dann den alten Server herunterfahren und den neuen am Port des alten starten. Sie sollten sich versichern, dass die Datenbank nicht mehr verändert wird, nachdem Sie `pg_dumpall` ausgeführt haben, ansonsten verlieren Sie logischerweise diese Daten. Schauen Sie in Kapitel 19 nach, wie Sie den Zugriff verbieten können. In der Praxis sollten Sie Ihre Clientanwendungen mit der neuen Serverversion testen, bevor Sie den Wechsel vollziehen.

Wenn Sie keine zwei Server parallel laufen haben wollen oder können, dann können Sie den Datensicherungsschritt ausführen, bevor die neue Version installieren, dann halten Sie den Server an, verschieben die alte Version woandershin, installieren Sie die neue Version und starten den neuen Server. Zum Beispiel:

```
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql- gmake install
initdb -D /usr/local/pgsql/data
postmaster -D /usr/local/pgsql/data
psql template1 < backup
```

In Kapitel 16 finden Sie Informationen, wie Sie den Server starten und stoppen können und andere Einzelheiten. Die Installationsanweisungen sagen Ihnen die richtigen Stellen, um diese Schritte auszuführen.

Anmerkung

Wenn Sie die alte Version “woandershin” verschieben, ist sie nicht mehr voll gebrauchsfähig. Einige Teile der Installation enthalten Informationen darüber, wo die anderen Teile sich befinden. Das ist normalerweise kein großes Problem, aber wenn Sie beide Installationen noch eine Weile verwenden wollen, dann sollten Sie ihnen wie vorgesehen bei der Installation unterschiedliche Verzeichnisse zuweisen.

23

Überwachung der Datenbankaktivität

Ein Datenbankadministrator fragt sich häufig: “Was macht das System gerade?” Dieses Kapitel bespricht, wie Sie das herausfinden können.

Es gibt mehrere Werkzeuge, um die Datenbankaktivität zu überwachen und die Datenbankleistung zu analysieren. Der Großteil dieses Kapitels ist der Beschreibung des Statistikkollektors von PostgreSQL gewidmet, aber man sollte normale Unix-Befehl zur Überwachung, wie `ps` und `top`, nicht vernachlässigen. Wenn man eine zu langsame Anfrage gefunden hat, muss man eventuell weitere Nachforschungen mit dem PostgreSQL-Befehl `EXPLAIN` anstellen. In Abschnitt 13.1 gibt es eine ausführliche Beschreibung von `EXPLAIN` und anderer Methoden, um das Verhalten einzelner Anfragen zu verstehen.

23.1 Unix-Standardwerkzeuge

Auf den meisten Plattformen verändert PostgreSQL den von `ps` angezeigten Prozesstitel, damit einzelne Serverprozesse leicht identifiziert werden können. Hier ist eine Beispielausgabe:

```
$ ps auxww | grep ^postgres
postgres   960  0.0  1.1  6104 1480 pts/1    SN   13:17   0:00 postmaster -i
postgres   963  0.0  1.1  7084 1472 pts/1    SN   13:17   0:00 postgres: stats buffer
process
postgres   965  0.0  1.1  6152 1512 pts/1    SN   13:17   0:00 postgres: stats collector
process
postgres   998  0.0  2.3  6532 2992 pts/1    SN   13:18   0:00 postgres: tgl runbug
127.0.0.1 idle
postgres  1003  0.0  2.4  6532 3128 pts/1    SN   13:19   0:00 postgres: tgl regression
[local] SELECT waiting
postgres  1016  0.1  2.4  6532 3080 pts/1    SN   13:19   0:00 postgres: tgl regression
[local] idle in transaction
```

(Die Syntax von `ps` ist auf verschiedenen Plattformen unterschiedlich, ebenso wie das Format der ausgegebenen Informationen. Dieses Beispiel ist von einem neueren Linux-System.) Der erste hier aufgeführte Prozess ist der `Postmaster`, der Hauptserverprozess. Die dahinter gezeigten Argumente sind dieselben, die angegeben wurden, als der Befehl ausgeführt wurde. Die nächsten zwei Prozesse implementieren den Statistikkollektor, welcher im nächsten Abschnitt beschrieben wird. (Wenn Sie das System so eingestellt haben, dass der Statistikkollektor nicht gestartet wird, werden diese Prozesse nicht vorhanden sein.) Jeder

der übrigen Prozesse ist ein Serverprozess, der eine Clientverbindung bearbeitet. Jeder dieser Prozesse setzt die angezeigte Kommandozeile auf das Format

```
postgres: benutzer datenbank host aktivität
```

Der Benutzer, die Datenbank und der Host der Verbindungsquelle bleiben für die gesamte Lebensdauer der Clientverbindung gleich, aber die Aktivitätsanzeige verändert sich. Sie ist `idle`, wenn die Sitzung auf einen Befehl vom Client wartet, `idle in transaction`, wenn der Client in einem `BEGIN`-Block ist, ansonsten der Name eines Befehlstyps, wie etwa `SELECT`. Außerdem wird `waiting` angehängt, wenn der Serverprozess augenblicklich auf eine Sperre wartet, die von einem anderen Serverprozess gehalten wird. Im obigen Beispiel können wir ableiten, dass Prozess 1003 darauf wartet, dass Prozess 1016 seine Transaktion abschließt und dadurch irgendeine Sperre aufgehoben wird.

Tipp

Solaris erfordert eine besondere Behandlung. Sie müssen `/usr/ucb/ps` anstatt `/bin/ps` verwenden und zweimal die Option `w` angeben, nicht nur einmal. Außerdem muss die ursprüngliche Befehlszeile von `postmaster` eine kürzere `ps`-Anzeige haben als die von jedem Serverprozess eingestellte. Wenn Sie eine dieser Voraussetzungen nicht erfüllen, ist die Ausgabe für jeden Serverprozess die ursprüngliche Befehlszeile von `postmaster`.

23.2 Der Statistikkollektor

Der **Statistikkollektor** in PostgreSQL ist ein Subsystem, das Informationen über die Serveraktivität sammelt und Schnittstellen bietet, um über diese zu berichten. Gegenwärtig kann der Kollektor Tabellen- und Indexzugriff in Diskblöcken und Zeilen zählen. Außerdem kann er den genauen Befehl, der gerade von einem anderen Serverprozess ausgeführt wird, bestimmen.

23.2.1 Konfiguration des Statistikkollektors

Da die Sammlung von Statistiken die Leistung der Anfrageausführung leicht beeinträchtigt, kann die Sammlung von Informationen ein- und ausgeschaltet werden. Das wird von Konfigurationsparametern, die normalerweise in `postgresql.conf` gesetzt werden, kontrolliert. (Siehe Abschnitt 16.4 für Einzelheiten, wie man Konfigurationsparameter setzt.)

Der Parameter `stats_start_collector` muss auf `true` gesetzt sein, damit der Statistikkollektor überhaupt gestartet wird. Das ist die Voreinstellung und auch empfohlen, aber Sie können diesen Parameter ausschalten, wenn Sie kein Interesse an Statistiken haben und jedes Quäntchen Leistung aus Ihrem Server herauspressen wollen. (Die Verbesserung wird allerdings gering sein.) Beachten Sie, dass diese Option nicht geändert werden kann, während der Server läuft.

Die Parameter `stats_command_string`, `stats_block_level` und `stats_row_level` kontrollieren, welche Informationen tatsächlich an den Kollektor geschickt werden, und damit auch, wie sehr die Leistung beeinträchtigt wird. Die Parameter bestimmen, ob ein Serverprozess seinen aktuellen Befehlstext, Zugriffsstatistiken auf Diskblock-Ebene bzw. Zugriffsstatistiken auf Zeilenebene an den Kollektor schickt. Normalerweise werden diese Parameter in `postgresql.conf` gesetzt, damit Sie für alle Serverprozesse gelten, aber es ist möglich, sie mit dem Befehl `SET` für einzelne Sitzungen an- oder auszuschalten. (Damit gewöhnliche Benutzer ihre Aktivitäten nicht vor dem Administrator verstecken können, können nur Superuser diese Parameter mit `SET` ändern.)

Anmerkung

Da die Parameter `stats_command_string`, `stats_block_level` und `stats_row_level` in der Voreinstellung falsch sind, werden in der Ausgangskonfiguration keine Statistiken gesammelt. Sie müssen einen oder mehrere dieser Parameter anschalten, damit Sie brauchbare Ergebnisse von den Auswertungsfunktionen erhalten.

23.2.2 Ansehen der gesammelten Statistiken

Um die Ergebnisse der Statistiksammlung zu zeigen, gibt es einige vordefinierte Sichten, welche in Tabelle 23.1 gelistet sind. Alternativ kann man sich mit den zugrunde liegenden Statistikfunktionen eigene Sichten bauen.

Wenn Sie die Statistiken verwenden, um die aktuellen Aktivitäten zu überwachen, müssen Sie bedenken, dass diese Informationen nicht sofort aktualisiert werden. Jeder Serverprozess übermittelt neue Zugriffszählungen an den Kollektor, bevor er auf einen neuen Befehl vom Client wartet; eine noch in der Ausführung befindliche Anfrage hat also keine Auswirkungen auf die angezeigten Ergebnisse. Außerdem ermittelt der Kollektor selbst höchstens alle `pgstat_stat_interval` Millisekunden (500 in der Voreinstellung) neue Gesamtwerte. Die angezeigten Ergebnisse hinken der tatsächlichen Aktivität also hinterher.

Ein weiterer wichtiger Punkt ist, dass, wenn ein Serverprozess eine dieser Statistiken anzeigen soll, er zuerst die zuletzt vom Kollektorprozess ermittelten Gesamtwerte holt und diesen Schnappschuss dann für alle Statistiksichten und -funktionen bis zum Ende seiner aktuellen Transaktion verwendet. Die Statistiken werden sich also nicht ändern, solange Sie die aktuelle Transaktion fortsetzen. Das ist Absicht und kein Fehler, denn dadurch können Sie mehrere Anfragen über die Statistiken ausführen und die Ergebnisse zueinander in Beziehung setzen, ohne sich Sorgen zu machen, dass sich die Zahlen gleichzeitig ändern könnten. Wenn Sie mit jeder Anfrage neue Ergebnisse sehen wollen, führen Sie die Anfragen außerhalb eines Transaktionsblocks aus.

Sichtname	Beschreibung
`pg_stat_activity`	Eine Zeile pro Serverprozess, mit Prozessnummer, Datenbankbenutzer und aktueller Anfrage. Die aktuelle Anfrage ist nur für Superuser zugänglich; für andere ist sie der NULL-Wert. (Beachten Sie, dass die aktuelle Anfrage wegen der Verzögerung der Kollektorergebnisse nur bei lange laufenden Anfragen auf dem richtigen Stand ist.)
`pg_stat_database`	Eine Zeile pro Datenbank, mit der Anzahl aktiver Serverprozesse, Gesamtzahl abgeschlossener und zurückgerollter Transaktionen in der Datenbank, Gesamtzahl gelesener Diskblöcke und Gesamtzahl der Puffertreffer (d.h. Blockleseversuche, die vermieden wurden, weil der Block im Cache war).
`pg_stat_all_tables`	Für jede Tabelle in der aktuellen Datenbank, Gesamtzahl der sequenziellen und Indexscans, Gesamtzahl der von jeder Scan-Art zurückgegebenen Tupel und die Gesamtzahl eingefügter, aktualisierter und gelöschter Tupel.
`pg_stat_sys_tables`	Wie `pg_stat_all_tables`, aber nur für Systemtabellen.
`pg_stat_user_tables`	Wie `pg_stat_all_tables`, aber nur für Benutzertabellen.
`pg_stat_all_indexes`	Für jeden Index in der aktuellen Datenbank, die Gesamtzahl der Indexscans mit diesem Index, die Gesamtzahl gelesener Indextupel und die Gesamtzahl erfolgreich gelesener Heap-Tupel. (Das kann weniger sein, wenn Indexeinträge auf abgelaufene Heap-Tupel zeigen.)
`pg_stat_sys_indexes`	Wie `pg_stat_all_indexes`, aber nur für Indexe von Systemtabellen.
`pg_stat_user_indexes`	Wie `pg_stat_all_indexes`, aber nur für Indexe von Benutzertabellen.

Tabelle 23.1: Standard-Statistiksichten

Sichtname	Beschreibung
pg_statio_all_tables	Für jede Tabelle in der aktuellen Datenbank, die Gesamtzahl der aus dieser Tabelle gelesenen Diskblöcke, die Anzahl der Puffertreffer, die Anzahl der gelesenen Diskblöcke und Puffertreffer für alle Indexe dieser Tabelle, die Anzahl der gelesenen Diskblöcke und Puffertreffer für die zugehörige TOAST-Tabelle dieser Tabelle (falls vorhanden) und die Anzahl der gelesenen Diskblöcke und Puffertreffer für den Index der TOAST-Tabelle.
pg_statio_sys_tables	Wie pg_statio_all_tables, aber nur für Systemtabellen.
pg_statio_user_tables	Wie pg_statio_all_tables, aber nur für Benutzertabellen.
pg_statio_all_indexes	Für jeden Index in der aktuellen Datenbank, die Anzahl der gelesenen Diskblöcke und Puffertreffer für diesen Index.
pg_statio_sys_indexes	Wie pg_statio_all_indexes, aber nur für Indexe von Systemtabellen.
pg_statio_user_indexes	Wie pg_statio_all_indexes, aber nur für Indexe von Benutzertabellen.
pg_statio_all_sequences	Für jedes Sequenzobjekt in der aktuellen Datenbank, die Anzahl gelesener Diskblöcke und Puffertreffer in dieser Sequenz.
pg_statio_sys_sequences	Wie pg_statio_all_sequences, aber nur mit Systemsequenzen. (Gegenwärtig sind keine Systemsequenzen definiert, also ist diese Sicht immer leer.)
pg_statio_user_sequences	Wie pg_statio_all_sequences, aber nur mit Benutzersequenzen.

Tabelle 23.1: Standard-Statistiksichten (Forts.)

Die indexspezifischen Statistiken sind nützlich, um zu ermitteln, welche Indexe verwendet werden und wie effektiv sie sind.

Die pg_statio_-Sichten sind hauptsächlich nützlich, um die Effektivität des Puffercache zu ermitteln. Wenn die Anzahl der tatsächlich gelesenen Diskblöcke wesentlich kleiner ist als die Anzahl der Puffertreffer, dann erfüllt der Cache die meisten Leseanfragen ohne einen Kernelaufruf zu benötigen.

Wenn Sie die Statistiken auf andere Weise betrachten wollen, können Sie selbst Anfragen schreiben, die die selben Statistikfunktionen verwenden, die den Standardsichten zugrunde liegen. Diese Funktionen sind in Tabelle 23.2 aufgelistet. Die Funktionen, die Datenbankstatistiken berichten, haben die Datenbank-OID als Argument. Die Funktionen für Tabellen- und Indexstatistiken verlangen die OID einer Tabelle oder eines Index. (Beachten Sie, dass diese Funktionen nur über Tabellen und Indexe in der aktuellen Datenbank berichten können.) Die Funktionen, die über Serverprozesse berichten, verlangen die laufende Nummer eines Serverprozesses als Argument, welche von eins bis zur Anzahl der aktiven Serverprozesse zählen.

Funktion	Ergebnistyp	Beschreibung
pg_stat_get_db_numbackends(oid)	integer	Anzahl aktiver Serverprozesse für die Datenbank
pg_stat_get_db_xact_commit(oid)	bigint	Anzahl erfolgreich abgeschlossener Transaktionen in der Datenbank
pg_stat_get_db_xact_rollback(oid)	bigint	Anzahl zurückgerollter Transaktionen in der Datenbank
pg_stat_get_db_blocks_fetched(oid)	bigint	Anzahl angeforderter Diskblock-Lesevorgänge für die Datenbank
pg_stat_get_db_blocks_hit(oid)	bigint	Anzahl im Cache gefundener Diskblöcke für die Datenbank

Tabelle 23.2: Statistikfunktionen

Funktion	Ergebnistyp	Beschreibung
pg_stat_get_numscans(oid)	bigint	Wenn das Argument eine Tabelle ist, die Anzahl ausgeführter sequentieller Scans; wenn es ein Index ist, die Anzahl ausgeführter Indexscans
pg_stat_get_tuples_returned(oid)	bigint	Wenn das Argument eine Tabelle ist, die Anzahl der von sequenziellen Scans gelesenen Tupel; wenn es ein Index ist, die Anzahl der von Indexscans gelesenen Indextupel
pg_stat_get_tuples_fetched(oid)	bigint	Wenn das Argument eine Tabelle ist, die Anzahl der von sequenziellen Scans gelesenen gültigen (nicht abgelaufenen) Tabellentupel; wenn es ein Index ist, die Anzahl der von Indexscans gelesenen gültigen Tabellentupel
pg_stat_get_tuples_inserted(oid)	bigint	Anzahl der in die Tabelle eingefügten Tupel
pg_stat_get_tuples_updated(oid)	bigint	Anzahl der in der Tabelle aktualisierten Tupel
pg_stat_get_tuples_deleted(oid)	bigint	Anzahl der aus der Tabelle gelöschten Tupel
pg_stat_get_blocks_fetched(oid)	bigint	Anzahl angeforderter Diskblock-Lesevorgänge für die Tabelle oder den Index
pg_stat_get_blocks_hit(oid)	bigint	Anzahl im Cache gefundener Diskblöcke für die Tabelle oder den Index
pg_stat_get_backend_idset()	set of integer	Menge der aktuell aktiven Serverprozessnummern (von 1 bis zur Anzahl der aktiven Serverprozesse). Siehe Verwendungsbeispiel im Text.
pg_backend_pid()	integer	Prozessnummer des zu dieser Sitzung gehörenden Serverprozesses
pg_stat_get_backend_pid(integer)	integer	Prozessnummer des angegebenen Serverprozesses
pg_stat_get_backend_dbid(integer)	oid	Datenbank-OID des angegebenen Serverprozesses
pg_stat_get_backend_userid(integer)	oid	Benutzer-ID des angegebenen Serverprozesses
pg_stat_get_backend_activity(integer)	text	Aktiver Befehl des angegebenen Serverprozesses (NULL-Wert, wenn der aktuelle Benutzer kein Superuser ist)
pg_stat_reset()	boolean	Setzt alle gesammelten Statistiken zurück.

Tabelle 23.2: Statistikfunktionen (Forts.)

Anmerkung

`pg_stat_get_db_blocks_fetched` minus `pg_stat_get_db_blocks_hit` ergibt die Anzahl der vom Kernel ausgeführten `read()`-Aufrufe für die Tabelle, den Index oder die Datenbank; aber die Anzahl der eigentlichen physikalischen Lesevorgänge ist normalerweise niedriger, weil der Kernel auch noch einen Cache hat.

Die Funktion `pg_stat_get_backend_idset` bietet einen praktische Möglichkeit, eine Zeile für jeden aktiven Serverprozess zu erzeugen. Um zum Beispiel die PID und die aktuelle Anfrage für jeden Serverprozess anzuzeigen, können Sie folgende Anfrage verwenden:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

23.3 Ansehen der Sperren

Ein weiteres nützliches Werkzeug, um die Datenbankaktivität zu überwachen, ist die Systemtabelle `pg_locks`. Mit ihr können Datenbankadministratoren Informationen über die aktuell ausstehenden Sperren ansehen. Diese Fähigkeit kann zum Beispiel verwendet werden, um:

- alle aktuell ausstehenden Sperren, alle Sperren in einer bestimmten Datenbank, alle Sperren für eine bestimmte Tabelle oder alle von einer bestimmten PostgreSQL-Sitzung gehaltenen Sperren zu sehen,
- die Tabelle in der aktuellen Datenbank mit den meisten nicht gewährten Sperren (welche eine Quelle möglicher Konflikte zwischen Clients sein könnten) zu ermitteln,
- Die Auswirkung von Sperrkonflikten auf die Gesamtleistung der Datenbank sowie das Ausmaß des Zusammenhangs zwischen der Konflikthäufigkeit und dem Datenbankverkehr zu ermitteln.

Weitere Informationen über Sperren und Mehrbenutzerbetrieb in PostgreSQL finden Sie in Kapitel 12.

Anmerkung

Wenn auf die Sicht `pg_locks` zugegriffen wird, werden die internen Datenstrukturen der Sperrenverwaltung einen Augenblick lang gesperrt und es wird davon für die Sicht eine Kopie gemacht. Das versichert, dass die Sicht konsistente Ergebnisse liefert, ohne die Sperrenverwaltung länger als notwendig zu blockieren. Nichtsdestotrotz könnte es sich auf die Leistung der Datenbank auswirken, wenn diese Sicht oft angeschaut wird.

Tabelle 23.3 zeigt die Definition der Spalten von `pg_locks`. Die Sicht `pg_locks` enthält eine Zeile pro sperrbares Objekt und angeforderten Sperrmodus. Ein sperrbares Objekt kann also mehrfach auftauchen, wenn mehrere Transaktionen Sperren halten oder auf solche warten. Ein sperrbares Objekt ist entweder eine Relation (z.B. eine Tabelle) oder eine Transaktionsnummer. (Beachten Sie, dass diese Sicht nur Sperren auf Tabellenebene, nicht aber auf Zeilenebene enthält. Wenn eine Transaktion auf eine Zeilensperre wartet, erscheint in der Sicht, dass diese Transaktion auf die Transaktionnummer der Transaktion wartet, die die Sperre gerade hält.)

Spaltenname	Datentyp	Beschreibung
`relation`	`oid`	Die OID der gesperrten Relation oder der NULL-Wert, wenn das sperrbare Objekt eine Transaktionsnummer ist. Diese Spalte kann mit der Spalte `oid` der Systemtabelle `pg_class` verbunden werden, um weitere Informationen über die Tabelle zu erhalten. Beachten Sie jedoch, dass das nur für Relationen in der aktuellen Datenbank (jene, bei denen die Spalte `database` gleich der OID der aktuellen Datenbank oder null ist) funktioniert.
`database`	`oid`	Die OID der Datenbank, in der das sperrbare Objekt liegt, oder der NULL-Wert, wenn das sperrbare Objekt eine Transaktionsnummer ist. Wenn die Sperre für eine globale Tabelle ist, dann ist dieses Feld 0. Dieses Spalte kann mit der Spalte `oid` der Systemtabelle `pg_database` verbunden werden, um weitere Informationen über die Datenbank des sperrbaren Objekts zu erhalten.
`transaction`	`xid`	Die Nummer der Transaktion oder der NULL-Wert, wenn das sperrbare Objekt eine Relation ist. Jede Transaktion hält für ihre gesamte Dauer eine exklusive Sperre für ihre Transaktionsnummer. Wenn es sich ergibt, dass eine Transaktion speziell auf eine andere warten muss, tut sie das, in dem sie versucht, eine geteilte Sperre für die Nummer der anderen Transaktion zu erhalten. Das kann erst erfolgreich sein, wenn die andere Transaktion zu Ende geht und ihre Sperren freigibt.

Tabelle 23.3: Spalten von `pg_locks`

Spaltenname	Datentyp	Beschreibung
pid	integer	Die Prozess-ID des PostgreSQL-Serverprozesses, der zu der Sitzung gehört, die die Sperre gesetzt hat oder versucht, sie zu setzen. Wenn Sie den Statistikkollektor angeschaltet haben, kann diese Spalte mit der Sicht pg_stat_activity verbunden werden, um weitere Informationen über die Sitzung, die die Sperre hält oder darauf wartet, zu erhalten.
mode	text	Der Modus der gewünschten oder gehaltenen Sperre. Weitere Informationen über die unterschiedlichen Sperrmodi in PostgreSQL finden Sie in Kapitel 12.
isgranted	boolean	Ist wahr, wenn die Sperre von dieser Sitzung gehalten wird. Falsch zeigt an, dass diese Sitzung gegenwärtig darauf wartet, diese Sperre setzen zu können, woraus folgt, dass eine andere Sitzung für dasselbe sperrbare Objekt eine Sperre hält, die mit der gewünschten in Konflikt steht. Die wartende Sitzung schläft, bis die andere Sperre aufgehoben wird (oder eine Verklemmung entdeckt wird). Eine einzelne Sitzung kann zu jeder Zeit nur auf höchstens eine Sperre warten.

Tabelle 23.3: Spalten von pg_locks (Forts.)

24

Überwachung des Festplattenplatzverbrauchs

Dieses Kapitel bespricht, wie Sie den Festplattenplatzverbrauch eines PostgreSQL-Datenbanksystems überwachen können. In der aktuellen Version hat der Datenbankadministrator nicht viel Kontrolle über die Struktur und die Verteilung der Datenbankdateien und daher ist dieses Kapitel eher informativ und kann Ihnen einige Ideen geben, wie Sie den Speicherplatzverbrauch mit Betriebssystemwerkzeugen verwalten können.

24.1 Ermittlung des Festplattenplatzverbrauchs

Zu jeder Tabelle gehört eine **Heap**-Datei, wo die meisten Daten gespeichert sind. Um lange Spaltenwerte zu speichern gibt es außerdem eine zur Tabelle gehörende TOAST-Tabelle, deren, Name aus der OID der Tabelle (genau gesagt eigentlich `pg_class.relfilenode`) gebildet wird, und einen Index für die TOAST-Tabelle. Außerdem kann es Indexe für die eigentliche Tabelle geben.

Sie können den Festplattenplatzverbrauch auf dreierlei Arten überwachen: mit `psql` auf Basis von `VACUUM`-Informationen, mit `psql` und den Werkzeugen in `contrib/dbsize` sowie von der Kommandozeile mit den Werkzeugen in `contrib/oid2name`. In einer Datenbank, in der neulich `VACUUM` oder `ANALYZE` ausgeführt wurde, können Sie mit `psql` durch Anfragen den Platzverbrauch einer jeden Tabelle sehen:

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'customer';
 relfilenode | relpages
-------------+----------
       16806 |       60
(1 row)
```

Jede Seite (*page*) ist normalerweise 8 kB groß. (Erinnern Sie sich, dass `relpages` nur von `VACUUM` und `ANALYZE` aktualisiert wird.)

Um den von TOAST-Tabellen belegten Platz anzuzeigen, verwenden Sie eine Anfrage wie die folgende, wobei Sie die `relfilenode`-Nummer des Heaps einsetzen müssen (welche mit obiger Anfrage ermittelt wird):

```
SELECT relname, relpages
    FROM pg_class
    WHERE relname = 'pg_toast_16806' OR relname = 'pg_toast_16806_index'
    ORDER BY relname;
```

```
        relname        | relpages
-----------------------+----------
 pg_toast_16806        |        0
 pg_toast_16806_index  |        1
```

Sie können ganz leicht auch Indexgrößen anzeigen:

```
SELECT c2.relname, c2.relpages
    FROM pg_class c, pg_class c2, pg_index i
    WHERE c.relname = 'customer'
        AND c.oid = i.indrelid
        AND c2.oid = i.indexrelid
        ORDER BY c2.relname;
              relname | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Mit diesen Informationen können Sie leicht Ihre größten Tabellen und Indexe finden:

```
SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;
              relname | relpages
----------------------+----------
             bigtable |    3290
             customer |    3144
```

`contrib/dbsize` lädt Funktionen in Ihre Datenbank, die es Ihnen erlauben, die Größe einer Tabelle oder Datenbank aus `psql` zu ermitteln, ohne `VACUUM` oder `ANALYZE` ausführen zu müssen.

Sie können auch `contrib/oid2name` verwenden, um den Festplattenplatzverbrauch anzuzeigen. In der Datei `README.oid2name` in diesem Verzeichnis finden Sie Beispiele. Ein Skript, das für jede Datenbank den Platzverbrauch anzeigt, ist dort enthalten.

24.2 Verhalten bei voller Festplatte

Die wichtigste Aufgabe für den Datenbankadministrator bei der Überwachung des Festplattenplatzverbrauchs ist, sicherzustellen, dass die Festplatte nicht voll wird. Wenn die Festplatte voll ist, kann das zur Verfälschung der Datenbankindexe, nicht aber der Tabellen selbst, führen. Wenn die WAL-Dateien auf der gleichen Festplatte sind (was in der Standardeinstellung der Fall ist), dann kann eine volle Festplatte bei der Datenbankinitialisierung zu verfälschten oder unvollständigen WAL-Dateien führen. Dieser fehlerhafte Zustand würde entdeckt werden und der Datenbankserver würde sich weigern zu starten.

Wenn Sie keinen zusätzlichen Platz auf der Festplatte schaffen können, indem Sie andere Sachen löschen, können Sie einige der Datenbankdateien auf andere Dateisysteme verschieben und von der ursprünglichen Stelle einen symbolischen Link erzeugen. Aber beachten Sie, dass `pg_dump` in solchen Situationen die Verteilung der Dateien nicht sichern kann; eine Wiederherstellung würde alles wieder an einer einzigen Stelle ablegen. Um zu verhindern, dass Ihnen der Festplattenplatz ausgeht, können Sie WAL-Dateien und Datenbanken bei der Erzeugung gleich an andere Stellen verlegen. Weitere Informationen dazu finden Sie in der Dokumentation zu `initdb` und in Abschnitt 18.5.

Tipp

Einige Dateisysteme haben eine enorm verschlechterte Leistungsfähigkeit, wenn sie fast voll sind. Also warten Sie nicht, bis die Festplatte voll ist, bevor Sie etwas unternehmen.

25

Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) ist eine Standardtechnik für das Aufzeichnen (Loggen) von Transaktionen. Eine detaillierte Beschreibung davon kann in den meisten (wenn nicht gar allen) Büchern über die Transaktionsverarbeitung gefunden werden. Zusammengefasst ist das zentrale Konzept von WAL, dass Änderungen in Datendateien (wo Tabellen und Indexe abgelegt sind) erst geschrieben werden dürfen, wenn diese Änderungen geloggt wurden, das heißt, nachdem Sie im Log aufgezeichnet wurden und die Logeinträge auf ein permanentes Speichermedium geschrieben wurden. Wenn man so vorgeht, muss man die Datenseiten nicht nach jeder Transaktion auf die Festplatte zurückschreiben, weil man weiß, dass man im Falle eines Absturzes die Datenbank mit dem Log wiederherstellen kann: Änderungen, die noch nicht auf die Datenseiten übertragen wurden, werden zuerst aus den Logaufzeichnungen wiederhergestellt (das ist Vorwärts-Wiederherstellung, auch REDO genannt) und dann werden Änderungen aus nicht abgeschlossenen Transaktionen aus den Datenseiten entfernt (Rückwärts-Wiederherstellung, UNDO).

25.1 Nutzen aus WAL

Der erste offensichtliche Nutzen aus WAL ist, dass die Anzahl der Schreibvorgänge auf die Festplatte erheblich verringert wird, da nur die Logdatei am Ende jeder Transaktion auf die Festplatte zurückgeschrieben werden muss; in Mehrbenutzerumgebungen können die Commits vieler Transaktionen mit einem einzigen `fsync()`-Aufruf für die Logdatei erledigt werden. Außerdem wird die Logdatei sequenziell geschrieben, wodurch das Synchronisieren der Logdatei viel effizienter wird als das Rückschreiben der Datenseiten.

Der nächste Nutzen ist die Konsistenz der Datenseiten. Die Wahrheit ist, dass PostgreSQL vor WAL die Konsistenz im Falle eines Absturzes nicht garantieren konnte. Vor WAL konnte ein Absturz beim Schreiben folgende Konsequenzen haben:

1. Indexzeilen, die auf nicht existierende Tabellenzeilen zeigen
2. bei Splitoperationen verlorene Indexzeilen
3. total zerstörter Inhalt von Tabellen- oder Indexseiten wegen unvollständig geschriebener Datenseiten

Probleme mit den Indexen (Probleme 1 und 2) könnten möglicherweise durch zusätzliche `fsync()`-Aufrufe behoben werden, aber es ist nicht klar, wie der letzte Fall ohne WAL zu ausgeräumt werden kann. WAL sichert den gesamten Datenseiteninhalt im Log, wenn das notwendig ist, um die Konsistenz der Seite für den Fall einer Wiederherstellung nach einem Absturz zu garantieren.

25.2 Zukünfiger Nutzen

Die UNDO-Operation ist nicht implementiert. Das bedeutet, dass von abgebrochenen Transaktionen verursachte Änderungen immer noch Speicherplatz verbrauchen und dass immer noch eine permanente `pg_clog`-Datei benötigt wird, um den Transaktionsstatus aufzuzeichnen, da Transaktionsnummern nicht wiederverwendet werden können. Wenn UNDO implementiert ist, muss `pg_clog` nicht mehr permanent sein; es wird beim Herunterfahren des Servers gelöscht werden können. (Die Dringlichkeit dieser Angelegenheit wurde aber durch die Einführung einer segmentierten Speichermethode für `pg_clog` enorm verringert: Es ist nicht mehr notwendig, alte `pg_clog`-Einträge ewig zu behalten.)

Mit UNDO wird es auch möglich sein, *Sicherungspunkte* zu implementieren, um ungültige Transaktionsoperationen (Syntaxfehler durch Tippfehler in Befehlen, Einfügen von doppelten Primärschlüsseln usw.) teilweise zurückrollen zu können und gleichzeitig die in der Transaktion vor dem Fehler durchgeführten Operationen fortführen oder die Transaktion abschließen zu können. Gegenwärtig macht jeder Fehler die ganze Transaktionen ungültig und erfordert, dass die Transaktion abgebrochen wird.

WAL bietet die Möglichkeit für eine neue Methode für online Datenbanksicherung und wiederherstellung. Um diese Methode zu verwenden, müsste man in regelmäßigen Abständen die Datendateien auf eine andere Festplatte, ein Bandlaufwerk oder einen anderen Rechner kopieren und außerdem die WAL-Logdateien archivieren. Die Kopie der Datenbankdateien und die archivierten Logdateien könnten zur Wiederherstellung verwendet werden, wie bei einer Wiederherstellung nach einem Absturz. Jedes Mal, wenn die Datenbankdateien kopiert würden, könnten die alten Logdateien entfernt werden. Um diese Methode zu implementieren müssten das Erzeugen und das Löschen von Datendateien und Indexen geloggt werden; man müsste auch eine Methode zum Kopieren der Datendateien entwickeln (die Kopierbefehle im Betriebssystem sind nicht ausreichend).

Eine Schwierigkeit, die der Umsetzung dieser möglichen Nutzen im Weg steht, ist, dass sie erfordern, dass WAL-Einträge für beträchtliche Zeiträume gespeichert werden (z.B. so lang wie die längstmögliche Transaktion, wenn Transaktions-UNDO gewünscht wird). Das gegenwärtige Format der WAL-Einträge ist ziemlich umfangreich, da es viele Schnappschüsse von Diskseiten enthält. Das ist gegenwärtig kein ernsthaftes Problem, da die Einträge nur für ein oder zwei Checkpoint-Intervalle aufbewahrt werden müssen; aber um diese zukünftigen Nutzen zu erreichen, wird irgendeine Form von komprimiertem WAL-Format benötigt werden.

25.3 WAL-Konfiguration

Es gibt mehrere WAL-Konfigurationsparameter, die auf die Leistung der Datenbank Auswirkung haben. Dieser Abschnitt beschreibt die Verwendung. Schauen Sie in Abschnitt 16.4 wegen Einzelheiten, wie man Konfigurationsparameter setzt, nach.

Checkpoints sind Punkte in der Folge von Transaktionen, an denen es garantiert ist, dass alle Datendateien mit den vor dem Checkpoint geloggten Informationen aktualisiert wurden. Zum Zeitpunkt des Checkpoints werden alle schmutzigen Datenseiten auf die Festplatte zurückgeschrieben, und es wird ein besonderer Checkpoint-Eintrag in den Log geschrieben. Im Fall eines Absturzes kann man daher wissen, bei welchem Eintrag im Log (der so genannte Redo-Eintrag) die REDO-Operation starten soll, da alle Änderungen, die davor an Datendateien gemacht wurden, schon auf der Festplatte sind. Nachdem ein Checkpoint erzeugt wurde, werden alle Segmente vor dem Redo-Eintrag nicht mehr benötigt und können wiederverwendet oder entfernt werden. (Wenn Datensicherung und -wiederherstellung auf WAL-Basis implementiert ist, dann würden die Logsegmente vor dem Wiederverwenden oder Entfernen archiviert werden.)

Der Server startet in regelmäßigen Abständen einen besonderen Prozess, um den nächsten Checkpoint zu erzeugen. Ein Checkpoint wird alle `checkpoint_segments` Logsegmente oder alle `checkpoint_`

`timeout` Sekunden, je nachdem, was eher kommt, erzeugt. Die Standardeinstellung ist 3 Segmente bzw. 300 Sekunden. Man kann auch mit dem SQL-Befehl `CHECKPOINT` einen Checkpoint direkt erzwingen.

Wenn man `checkpoint_segments` und/oder `checkpoint_timeout` verringert, werden Checkpoints häufiger gemacht. Das ermöglicht schnellere Wiederherstellung nach einem Absturz (da weniger Arbeit wiederhergestellt werden muss). Man muss das jedoch abwägen gegen die erhöhten Kosten, um die schmutzigen Datenseiten häufiger zurückzuschreiben. Außerdem wird, um die Konsistenz der Datenseiten sicherzustellen, bei der ersten Modifikation einer Datenseite nach einem Checkpoint der gesamte Seiteninhalt geloggt. Ein kleineres Checkpoint-Intervall erhöht folglich die in den Log geschriebene Datenmenge, wodurch teilweise das Ziel eines kleineren Intervalls zunichte gemacht wird und auf jeden Fall die Eingabe- und Ausgabeaktivität auf der Festplatte erhöht wird.

Es gibt immer mindestens eine Segmentdatei zu 16 MB und normalerweise gibt es höchstens 2 * `checkpoint_segments` + 1 Dateien. Damit können Sie die Speicherplatzanforderungen von WAL abschätzen. Gewöhnlich werden alte, nicht mehr benötigte Logsegmente wiederverwendet (umbenannt, um die nächsten Segmente in der durchnummerierten Reihe zu werden). Wenn es, durch eine kurzzeitige Spitze bei der Logausgabe, mehr als 2 * `checkpoint_segments` + 1 Segmentdateien gibt, werden die unbenötigten Segmentdateien nicht wiederverwendet, sondern gelöscht, bis das System wieder unter die Grenze kommt.

Es gibt zwei häufig verwendete WAL-Funktionen: `LogInsert` und `LogFlush`. `LogInsert` wird verwendet, um einen neuen Eintrag in die WAL-Puffer im Shared Memory zu stellen. Wenn es keinen Platz für einen neuen Eintrag gibt, muss `LogInsert` einige gefüllte WAL-Puffer zurückschreiben (bzw. in den Kernelcache verschieben). Das ist aber nicht wünschenswert, weil `LogInsert` bei jeder Datenbankoperation auf unterster Ebene (zum Beispiel das Einfügen einer Zeile) verwendet wird, während auf den betroffenen Datenseiten eine exklusive Sperre liegt; daher muss diese Operation so schnell wie möglich sein. Und noch schlimmer ist, dass, wenn WAL-Puffer geschrieben werden, das dann auch dazu führen kann, dass ein neues Logsegment erzeugt werden muss, was noch mehr Zeit in Anspruch nimmt. Normalerweise sollten WAL-Puffer von einer `LogFlush`-Operation zurückgeschrieben werden, welche im Großen und Ganzen jeweils beim erfolgreichen Abschluss einer Transaktion ausgeführt wird, damit sichergestellt wird, dass die Daten der Transaktion auf einem permanenten Speichermedium angekommen sind. Auf Systemen mit großem Logausgabevolumen kann es sein, dass `LogFlush` nicht oft genug ausgeführt wird, um zu verhindern, dass WAL-Puffer von `LogInsert` geschrieben werden müssen. Auf solchen Systemen sollte man die Anzahl der WAL-Puffer erhöhen, indem man den Konfigurationsparameter `wal_buffers` verändert. Die voreingestellte Zahl der WAL-Puffer ist 8. Ein höherer Wert erfordert entsprechend mehr Shared Memory.

Der Parameter `commit_delay` bestimmt, wie viele Mikrosekunden ein Serverprozess nach dem Schreiben eines Commit-Eintrags mit `LogInsert`, aber vor der `LogFlush`-Operation schlafen wird. Diese Verzögerung erlaubt es anderen Serverprozessen, ihre Commit-Einträge zum Log hinzuzufügen, damit sie alle mit einer einzigen Operation zurückgeschrieben werden können. Dieses Schlafen wird nicht durchgeführt, wenn `fsync` nicht angeschaltet ist oder wenn weniger als `commit_siblings` andere Sitzungen gegenwärtig in aktiven Transaktionen sind; das vermeidet die Verzögerung, wenn es unwahrscheinlich ist, dass weitere Sitzungen ihre Transaktionen in Kürze abschließen werden. Beachten Sie, dass die Genauigkeit einer Schlafoperation auf den meisten Plattformen zehn Millisekunden ist, was hieße, dass jeder Wert für `commit_delay` zwischen 1 und 10000 Mikrosekunden die gleiche Wirkung haben würde. Gute Werte für diese Parameter sind noch nicht ganz klar; wir ermutigen Sie, zu experimentieren.

Der Parameter `wal_sync_method` bestimmt, wie PostgreSQL den Kernel darum bittet, WAL-Änderungen auf die Festplatte zurückzuschreiben. Die Optionen sollten in Sachen Zuverlässigkeit gleichwertig sein, aber es ist ziemlich plattformabhängig, welche die schnellste ist. Beachten Sie, dass dieser Parameter irrelevant ist, wenn `fsync` ausgeschaltet wurde.

Wenn man den Parameter `wal_debug` auf einen Wert verschieden von null setzt, wird jede `LogInsert`- und `LogFlush`-Operation im Serverlog vermerkt. Gegenwärtig macht es keinen Unterschied, was genau der Wert ist. Diese Einstellung könnte in der Zukunft durch einen allgemeineren Mechanismus ersetzt werden.

25.4 Interna

WAL ist automatisch angeschaltet; der Administrator muss nichts weiter unternehmen, außer für zusätzlichen Festplattenplatz für die WAL-Logs zu sorgen und die nötigen Feinabstimmungen vorzunehmen (siehe Abschnitt 25.3).

WAL-Logs werden im Verzeichnis *$PGDATA*/`pg_xlog` gespeichert, als ein Satz Segmentdateien, jede 16 MB groß. Jedes Segment ist in 8 kB große Seiten aufgeteilt. Der Kopf eines Logeintrags ist in `access/xlog.h` beschrieben; der Inhalt des Eintrags hängt vom Typ des geloggten Ereignisses ab. Segmentdateien erhalten stetig steigende Nummern als Namen, beginnend mit `0000000000000000`. Die Nummern haben keine Überlaufvorrichtung, aber es sollte sehr lange dauern, bis die verfügbaren Nummern aufgebraucht sind.

Die WAL-Puffer und -Kontrollstrukturen sind im Shared Memory und werden von den Serverkindprozessen verwaltet; sie werden durch leichtgewichtige Sperren geschützt. Der Verbrauch an Shared Memory hängt von der Anzahl der Puffer ab. Die voreingestellte Größe der WAL-Puffer ist 8 Puffer zu je 8 kB, also insgesamt 64 kB.

Es ist von Vorteil, wenn der Log auf einer anderen Festplatte als die anderen Datenbankdateien ist. Das kann man erreichen, indem man das Verzeichnis `pg_xlog` an eine andere Stelle verschiebt (natürlich während der Server nicht läuft) und einen symbolischen Link von der ursprünglichen Stelle in *$PGDATA* an die neue Stelle erzeugt.

Das Ziel von WAL, sicherzustellen, dass der Log geschrieben wird, bevor Datenbanksätze geändert werden, kann von Festplattenlaufwerken untergraben werden, die fälschlicherweise Schreibvorgänge an den Kernel als erfolgreich melden, obwohl sie tatsächlich die Daten nur in den Cache gestellt und nicht wirklich auf die Platte geschrieben haben. Ein Stromausfall kann in solchen Fällen immer noch zu Datenzerstörung ohne Chance auf Wiederherstellung führen. Administratoren sollten sich versichern, dass die Festplatten, die für PostgreSQL die WAL-Logdateien speichern, keine solchen falschen Ergebnisse liefern.

Nachdem ein Checkpoint erzeugt und der Log zurückgeschrieben wurde, wird die Position des Checkpoints in der Datei `pg_control` gesichert. Wenn eine Wiederherstellung ansteht, liest der Server daher zuerst `pg_control` und dann den Checkpoint-Eintrag; dann führt er die REDO-Operation aus, indem er von der im Checkpoint-Eintrag angegebenen Logposition vorwärts verfährt. Da bei der ersten Seitenveränderung nach einem Checkpoint der gesamte Inhalt einer Datenseite im Log gesichert wird, können alle Änderungen nach dem Checkpoint in einen konsistenten Zustand wiederhergestellt werden.

Die Verwendung von `pg_control`, um die Checkpoint-Position zu ermitteln, beschleunigt den Wiederherstellungsvorgang, aber um mit einer möglichen Verfälschung von `pg_control` fertig zu werden, sollten wir eigentlich die existierenden Logsegmente in umgekehrter Reihenfolge – vom neusten zum ältesten – lesen und so den letzten Checkpoint finden. Das ist noch nicht implementiert.

26 Regressionstests

Die Regressionstests sind eine umfassende Testsammlung für die SQL-Implementierung in PostgreSQL. Sie testen standardisierte Operationen und auch die erweiterten Fähigkeiten von PostgreSQL. Seit PostgreSQL 6.1 sind die Regressionstests für jede offizielle Version aktuell.

26.1 Die Tests ausführen

Die Regressionstests können mit einem schon installierten Server ausgeführt werden oder mit einer temporären Installation im Quelltextverzeichnis. Außerdem gibt es eine "parallele" und eine "sequenzielle" Methode, um die Tests durchzuführen. Die sequenzielle Methode führt jedes Testskript nacheinander aus, wohingegen die parallele Methode mehrere Serverprozesse startet, um Tests in Gruppen gleichzeitig laufen zu lassen. Das parallele Testen gibt Vertrauen, dass die Interprozesskommunikation und die Sperren richtig arbeiten. Aus historischen Gründen werden die sequenziellen Tests normalerweise mit einer bestehenden Installation ausgeführt und die parallele Methode mit einer temporären Installation, aber dafür gibt es keine technischen Gründe.

Um die Regressionstests nach dem Compilieren aber vor der Installation auszuführen, geben Sie ein

```
gmake check
```

während Sie sich im obersten Verzeichnis (oder in `src/test/regress`) befinden. Dadurch werden erst ein paar Hilfsdateien gebaut, wie einige Triggerfunktionen als Beispiel, und dann das Testskript ausgeführt. Am Ende sollten Sie etwas wie

```
======================
 All 77 tests passed.
======================
```

sehen oder ansonsten eine Mitteilung darüber, welchen Tests fehlgeschlagen sind. Siehe Abschnitt 26.3 unten für mehr.

(Das einzige mögliche "Sicherheitsproblem" hier ist, dass jemand anders hinter Ihrem Rücken die Testergebnisse verändern könnte. Verwalten Sie also Ihre Zugriffsrechte weise.)

Alternativ können Sie die Tests nach der Installation ausführen.

Anmerkung

Weil diese Testmethode einen vorübergehenden Testserver startet, funktioniert Sie nicht als `root` (der Server startet nicht als `root`). Wenn Sie schon den Rest des Builds als `root` gemacht haben, dann müssen Sie nicht nochmal ganz von vorne anfangen. Machen Sie stattdessen das Testverzeichnis durch einen anderen Benutzer schreibbar, melden Sie sich als der Benutzer an und starten Sie die Tests. Zum Beispiel:

```
root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd oberstes build verzeichnis
joeuser$ gmake check
```

Tipp

Die parallelen Tests starten eine ganze Zahl von Prozessen unter Ihrem Benutzerzugang. Gegenwärtig ist das Höchste 20 parallele Tests, was 60 Prozesse bedeutet: ein Serverprozess, ein `psql` und einen Shell-Elternprozess für `psql` für jedes Testskript. Wenn Ihr System eine Begrenzung der Prozesse pro Benutzer hat, dann achten Sie darauf, dass diese Grenze mindestens um 75 liegt, ansonsten erhalten Sie in den Tests willkürlich scheinende Misserfolge. Wenn Sie die Grenze nicht erhöhen können, dann können Sie die Datei `src/test/regress/parallel_schedule` bearbeiten, um die großen Paralleltestgruppen in kleinere Gruppen aufzuteilen.

Tipp

Auf einigen Systemen wird die Bourne-kompatible Shell verwirrt, wenn sie zu viele Kindprozesse gleichzeitig zu verwalten hat. Dadurch kann es vorkommen, dass die Tests sich festfahren oder abbrechen. In diesem Fall sollten Sie eine andere Bourne-kompatible Shell auf der Befehlszeile angeben, zum Beispiel:

```
gmake SHELL=/bin/ksh check
```

Wenn keine funktionierende Shell zur Verfügung steht, können Sie wie oben beschrieben den Testplan der parallelen Tests verändern.

Um die Tests nach der Installation (siehe Kapitel 14) auszuführen, initialisieren Sie das Datenverzeichnis und starten den Server, wie in Kapitel 16 erklärt, und geben Sie dann ein

```
gmake installcheck
```

Die Tests erwarten, dass sie den Server auf der lokalen Maschine mit der Standardportnummer kontaktieren können, falls nicht die Umgebungsvariablen `PGHOST` und `PGPORT` anderes aussagen.

26.2 Testauswertung

Auf einigen ordnungsgemäß installierten und voll funktionsfähigen PostgreSQL-Installationen könnten trotzdem einige Tests nicht “erfolgreich” sein, wegen plattformabhängigen Umständen wie die Darstellung von Fließkommazahlen oder Zeitzonenunterstützung. Die Tests werden gegenwärtig mit einem einfachen `diff`-Vergleich mit den auf einem Bezugssystem erzeugten Ausgaben ausgewertet, wodurch die Ergebnisse empfindlich für kleine Unterschiede sind. Wenn ein Test als nicht bestanden berichtet wird, sollten Sie immer die Unterschiede zwischen den erwarteten und den tatsächlichen Ergebnissen betrachten; Sie

könnten feststellen, dass die Unterschiede nicht erheblich sind. Nichtsdestotrotz versuchen wir, für alle Plattformen genaue Bezugsdateien zur Verfügung zu stellen, also können Sie erwarten, dass alle Tests erfolgreich sind.

Die eigentlichen Ausgabedateien der Tests sind im Verzeichnis `src/test/regress/results`. Das Testskript verwendet `diff`, um jede Ausgabedatei mit den erwarteten Ergebnissen in Verzeichnis `src/test/regress/expected` zu vergleichen. Irgendwelche Unterschiede werden für Sie in der Datei `src/test/regress/regression.diffs` zum Anschauen gespeichert. (Oder wenn Sie wollen, können Sie `diff` auch selbst ausführen.)

26.2.1 Unterschiede bei Fehlermeldungen

Einige der Regressionstests beinhalten absichtlich ungültige Eingabewerte. Fehlermeldungen kommen entweder vom PostgreSQL-Code oder aus dem Betriebssystem. Im letzteren Fall könnten die Fehlertexte zwischen verschiedenen Plattformen variieren, sollten aber ähnliche Informationen darstellen. Diese Unterschiede verursachen scheinbar gescheiterte Tests, die aber durch manuelle Inspektion abgetan werden können.

26.2.2 Unterschiede durch Locales

Wenn Sie die Tests auf einem bestehenden Server ausführen, der mit einer anderen Sortierreihenfolge als C initialisiert wurde, kann es Unterschiede wegen der Sortierreihenfolge und eventuelle Folgefehler geben. Die Regressionstests sind darauf vorbereitet, indem sie alternative Ausgabedateien haben, die zusammen eine große Zahl von Locales abdecken. Beim Test `char` zum Beispiel, ist die erwartete Ausgabedatei `char.out` für die Locales `C` und `POSIX` und die Datei `char_1.out` bewältigt viele andere Locales. Der Testtreiber wählt automatisch die am besten passende Datei aus, wenn er auf Erfolg prüft oder bei Misserfolg die Unterschiede berechnet. (Das bedeutet, dass die Regressionstests nicht entdecken können, ob das Ergebnis für die eingestellte Locale angebracht ist. Die Tests suchen sich einfach die Ergebnisdatei, die am besten passt.)

Wenn die bestehenden erwarteten Dateien eine Locale aus irgendwelchen Gründen nicht verarbeiten können, können Sie eine neue Datei hinzufügen. Das Namensschema ist *`testname_ziffer`*`.out`. Die eigentliche Ziffer ist unerheblich. Denken Sie aber daran, dass der Testtreiber alle solche Dateien als gleichermaßen gültiges Testergebnis betrachtet. Wenn die Testergebnisse plattformabhängig sind, sollten Sie stattdessen die in Abschnitt 26.3 beschriebene Technik verwenden.

26.2.3 Unterschiede bei Datum und Zeit

Einige Anfragen im Test `horology` schlagen fehl, wenn Sie die Tests an einem Tag der Sommer/Winterzeitumstellung oder einen Tag davor oder danach ausführen. Diese Anfragen gehen davon aus, dass der Zeitabstand zwischen gestern Mitternacht, heute Mitternacht und morgen Mitternacht jeweils genau 24 Stunden ist, was nicht der Fall ist, wenn zwischendurch die Sommerzeit angefangen oder aufgehört hat.

Die meisten Datums- und Zeitergebnisse hängen von der Zeitzone ab. Die Bezugsdateien wurden für die Zeitzone `PST8PDT` (Kalifornien) erzeugt und wenn die Tests nicht mit dieser Zeitzoneneinstellung durchgeführt werden, wird es zu Abweichungen kommen. Der Testtreiber setzt die Umgebungsvariable `PGTZ` auf `PST8PDT`, was normalerweise ausreicht. Aber Ihr Betriebssystem muss die Zeitzone `PST8PDT` unterstützen oder die zeitzonenabhängigen Tests werden nicht bestehen. Um zu überprüfen, ob Ihr System diese Unterstützung hat, geben Sie Folgendes ein:

```
env TZ=PST8PDT date
```

Dieser Befehl sollte die aktuelle Systemzeit in der Zeitzone PST8PDT ausgeben. Wenn die Zeitzone PST8PDT nicht verfügbar ist, hat Ihr System die Zeit vielleicht in GMT ausgegeben. Wenn die Zeitzone PST8PDT fehlt, können Sie die Zeitzonenregeln selbst einstellen:

```
PGTZ='PST8PDT7,M04.01.0,M10.05.03'; export PGTZ
```

Es scheint einige Systeme zu geben, die die empfohlene Syntax, um die Zeitzonenregeln selbst einzustellen, nicht akzeptieren; da müssen Sie womöglich einen anderen Wert für PGTZ verwenden.

Einige Systeme mit älteren Zeitzonenbibliotheken wenden die Sommerzeitumstellung nicht auf Daten vor 1970 an, wodurch Zeiten in der Zeitzone PDT (entspricht Sommerzeit) vor 1970 als PST (Normalzeit) ausgegeben werden. Dadurch gibt es stellenweise Unterschiede in den Testergebnissen.

26.2.4 Unterschiede bei Fließkommaberechnungen

Einige Tests beinhalten die Berechnung von 64-bittigen Fließkommazahlen (double precision) aus Tabellenspalten. Es sind Unterschiede bei den Ergebnissen mathematischer Funktionen mit Spalten vom Typ double precision beobachtet worden. Die Tests `float8` und `geometry` sind besonders anfällig für kleine Unterschiede zwischen Plattformen, oder selbst bei unterschiedlichen Compiler-Optimierungsoptionen. Ein Vergleich mit dem menschlichen Auge ist vonnöten, um die tatsächliche Erheblichkeit dieser Unterschiede, welche meistens 10 Stellen nach dem Komma sind, einzuschätzen.

Einige Systeme signalisieren Fehler bei den Funktionen `pow()` und `exp()`, anders als von PostgreSQL gegenwärtig erwartet.

26.2.5 Unterschiede bei Polygonen

Einige Tests beinhalten geografische Daten über die Straßenkarte von Oakland/Berkeley, Kalifornien. Die Kartendaten sind als Polygone dargestellt, deren Eckpunkte ein Paar Zahlen vom Typ double precision (geografische Länge und Breite) sind. Am Anfang werden einige Tabellen erzeugt und mit geografischen Daten geladen, dann werden einige Sichten erzeugt, die zwei Tabellen mit dem Polygonschnittoperator (##) verbinden, und dann wird eine Anfrage mit der Sicht ausgeführt.

Wenn man die Ergebnisse unterschiedlicher Plattformen vergleicht, ergeben sich Unterschiede an der 2. oder 3. Stelle nach dem Komma. Die SQL-Befehle, wo diese Probleme auftreten, sind die folgenden:

```
SELECT * from street;
SELECT * from iexit;
```

26.2.6 Unterschiede bei der Zeilenreihenfolge

Die könnte einige Unterschiede sehen, bei denen die gleichen Zeilen in anderer Reihenfolge als in der erwarteten Datei ausgegeben werden. In den meisten Fällen ist das streng genommen kein Programmfehler. Die meisten Regressionstests sind nicht so pedantisch und verwenden `ORDER BY` nicht bei jedem einzelnen `SELECT`, und daher ist die Reihenfolge der Ergebniszeilen nach dem SQL-Standard nicht definiert. Da wird aber in der Praxis die gleichen Anfragen ausgeführt mit den gleichen Daten auf der gleichen Software sehen, erhalten wird normalerweise die gleiche Ergebnisreihenfolge auf allen Plattformen und daher ist das fehlende `ORDER BY` kein Problem. Einige Anfragen zeigen aber doch Unterschiede bei der Sortierreihenfolge auf unterschiedlichen Plattformen. (Sortierunterschiede können auch von Locale-Einstellungen verursacht werden.)

Wenn Sie also einen Unterschied bei der Sortierreihenfolge sehen, dann ist das kein Anlass zur Sorge, außer wenn die Anfrage ein `ORDER BY` hat, das die Anfrage nicht beachtet. Aber sagen Sie uns trotzdem

Bescheid, damit wir das `ORDER BY` hinzufügen können und damit die falschen Fehlerberichte in der Zukunft verhindern können.

Sie fragen sich vielleicht, warum wir die Testanfragen nicht alle ausdrücklich sortieren lassen und damit das Problem ein für alle Mal aus dem Weg räumen. Der Grund ist, dass das die Regressionstests weniger nützlich machen würde, da sie dann hauptsächlich Anfrageplantypen, die sortierte Ergebnisse liefern, testen würden und solche, die keine sortierten Ergebnisse liefern, vernachlässigen würden.

26.2.7 Der Test random

Es gibt mindestens einen Testfall im Test `random`, der zufällige Ergebnisse liefern soll. Dadurch fällt der Test ab und zu (vielleicht einmal alle fünf bis zehn Versuche) durch. Wenn Sie eingeben

```
diff results/random.out expected/random.out
```

sollten Sie nur einige wenige Zeilen Unterschied sehen. Sie müssen sich keine Sorgen machen, es sei denn, der Test ist auch nach wiederholten Versuchen nie erfolgreich. (Wenn der Test andererseits auch in vielen Versuchen *immer* erfolgreich ist, sollten Sie sich vielleicht doch Sorgen machen.)

26.3 Plattformspezifische Vergleichsdateien

Da einige Tests der Natur der Sache wegen plattformspezifische Ergebnisse produzieren, bieten wir eine Methode, um plattformspezifische Ergebnisvergleichsdateien anzugeben. Häufig passt die gleiche Variation für mehrere Plattformen; anstatt eine getrennte Vergleichsdatei für jede Plattform anzugeben, gibt es eine Datei, die angibt, welche Vergleichsdatei für jede Plattform zu verwenden ist. Um also fälschlicherweise fehlgeschlagene Tests zu eliminieren, müssen Sie eine alternative Ergebnisdatei wählen oder erstellen und dann eine Zeile zur Steuerdatei `resultmap` hinzufügen.

Jede Zeile der Steuerdatei hat die Form

```
testname/plattformmuster=vergleichsdateiname
```

Der Testname ist einfach der Name des bestimmten Regressionstestmoduls. Das Plattformmuster ist ein Muster im Stile des Unix-Werkzeugs `expr` (das heißt ein regulärer Ausdruck mit einem impliziten `^` am Anfang). Es wird verglichen mit dem von `config.guess` ausgegebenen Plattformnamen, gefolgt von `:gcc` oder `:cc`, je nach dem, ob Sie den GNU-Compiler oder den Systemcompiler verwenden (auf den Systemen, wo es einen Unterschied gibt). Der Vergleichsdateiname ist der Name der Ersatzvergleichsdatei.

Zum Beispiel: Einige Systeme mit älteren Zeitzonenbibliotheken wenden die Sommerzeitumstellung nicht auf Daten vor 1970 an, wodurch Zeiten vor 1970 in der Zeitzone PDT als PST erscheinen. Das verursacht ein paar Unterschiede im Regressionstest `horology`. Daher haben wir eine alternative Vergleichsdatei `horology-no-DST-before-1970.out` erstellt, die die auf diesen Systemen zu erwartenden Ergebnisse enthält. Um die falschen Fehlermeldungen auf `HPPA`-Plattformen auszuschalten, enthält `resultmap`

```
horology/hppa=horology-no-DST-before-1970
```

was für alle Maschinen angewendet wird, bei denen die Ausgabe von `config.guess` mit `hppa` anfängt. Andere Zeilen in `resultmap` wählen alternative Ergebnisdateie für andere Plattformen, wenn nötig.

Teil IV

Client-Schnittstellen

Dieser Teil beschreibt die mit PostgreSQL gelieferten Client-Programmierschnittstellen. Jedes dieser Kapitel kann einzeln gelesen werden. Beachten Sie, dass es viele andere Programmierschnittstellen für Clientprogramme gibt, die getrennt erhältlich sind und ihre eigene Dokumentation enthalten. Leser dieses Teils sollten wissen, wie man SQL-Befehle verwendet, um die Datenbank zu bearbeiten und abzufragen (siehe ***Teil II***), und natürlich sollten sie die von der Schnittstelle verwendete Programmiersprache kennen.

27

libpq: Die C-Bibliothek

libpq ist die Schnittstelle zu PostgreSQL für Anwendungsprogrammierung in C. libpq ist eine Sammlung von Bibliotheksfunktionen, die es Clientprogrammen erlauben, Anfragen an den PostgreSQL-Server zu schicken und die Ergebnisse dieser Anfragen zu erhalten. libpq wird auch als Fundament für mehrere andere PostgreSQL-Anwendungsschnittstellen verwendet, einschließlich libpq++ (C++), libpgtcl (Tcl), Perl und ECPG. Einige Aspekte über das Verhalten von libpq sind daher auch für Sie von Bedeutung, wenn Sie eines dieser anderen Pakete verwenden.

Am Ende dieses Kapitels (Abschnitt 27.13) finden Sie drei kurze Programme, die zeigen, wie man Programme schreibt, die libpq verwenden. Mehrere komplette Beispiele von libpq-Anwendungen finden sich außerdem im Verzeichnis src/test/examples in der Quelltext-Distribution.

Clientprogramme, die libpq verwenden, müssen die Headerdatei libpq-fe.h einbinden und die libpq-Bibliothek einlinken.

27.1 Funktionen zur Datenbankverbindung

Die folgenden Funktionen dienen zum Aufbau einer Verbindung mit dem PostgreSQL-Server. Ein Anwendungsprogramm kann mehrere Serververbindungen zur gleichen Zeit offen halten. (Ein Grund, das zu tun, ist, auf mehrere Datenbanken zuzugreifen.) Jede Verbindung wird durch ein Objekt vom Typ PGconn repräsentiert, welches von der Funktion PQconnectdb oder PQsetdbLogin erhalten wurde. Beachten Sie, dass diese Funktionen nie einen Null-Zeiger ergeben, außer wenn womöglich selbst für das PGconn-Objekt zu wenig Speicher vorhanden ist. Die Funktion PQstatus sollte verwendet werden, um zu prüfen, ob die Verbindung erfolgreich aufgebaut wurde, bevor Anfragen durch das Verbindungsobjekt geschickt werden.

PQconnectdb

Baut eine neue Verbindung mit dem Datenbankserver auf.

```
PGconn *PQconnectdb(const char *conninfo);
```

Diese Funktion öffnet eine neue Datenbankverbindung mit den Parametern, die in der Zeichenkette conninfo stehen. Im Gegensatz zu PQsetdbLogin unten kann die Parametersammlung erweitert werden, ohne die Signatur der Funktion zu ändern. Daher ist die Verwendung dieser Routine oder der nicht blockierenden Analoga PQconnectStart und PQconnectPoll bei neuen Anwendungen zu bevorzugen.

Die angegebene Zeichenkette kann leer sein, um alle Vorgabeparameter zu verwenden, oder sie kann mehrere durch Whitespace getrennte Parametereinstellungen enthalten. Jede Parametereinstellung hat die Form `schlüssel = wert`. (Um einen leeren Wert oder einen Wert mit Leerzeichen zu schreiben, setzen Sie Apostrophe darum, zum Beispiel `schlüssel = 'ein wert'`. Apostrophe und Backslashes müssen von einem Backslash eingeleitet werden, das heißt `\'` bzw. `\\`.) Leerzeichen um das Gleichheitszeichen sind wahlfrei.

Die gegenwärtig erkannten Parameterschlüsselwörter sind:

`host`

Name des Servers, mit dem verbunden werden soll. Wenn der Wert mit einem Schrägstrich anfängt, dann wird eine Verbindung über Unix-Domain-Sockets statt TCP/IP verwendet; der Wert ist das Verzeichnis, in dem sich die Socketdatei befindet. Der Vorgabewert ist eine Unix-Domain-Socket in `/tmp`.

`hostaddr`

Die IP-Adresse der Servers. Der Wert sollte in der Standardform aus Zahlen und Punkten bestehen. Wenn ein nicht leerer Wert angegeben wird, dann wird eine TCP/IP-Verbindung verwendet.

Durch die Verwendung von `hostaddr` statt `host` können Anwendungen die Auflösung von Hostnamen vermeiden, was in Anwendungen mit Zeitvorgaben wichtig sein könnte. Authentifizierung mit Kerberos erfordert allerdings einen Hostnamen. Folgendes gilt daher: Wenn `host` ohne `hostaddr` angegeben wird, dann wird der Hostname aufgelöst. Wenn `hostaddr` ohne `host` angegeben wird, bestimmt der Wert für `hostaddr` die Serveradresse; wenn Kerberos verwendet wird, dann wird dadurch eine umgekehrte Namensauflösung verursacht. Wenn sowohl `host` als auch `hostaddr` angegeben werden, bestimmt der Wert für `hostaddr` die Serveradresse; der Wert für `host` wird ignoriert, außer wenn Kerberos verwendet wird, dann wird der Wert für die Authentifizierung mit Kerberos verwendet. Beachten Sie, dass die Authentifizierung wahrscheinlich fehlschlagen wird, wenn Sie `libpq` einen Hostnamen übergeben, der nicht der wirkliche Name der Maschine mit der Adresse `hostaddr` ist.

Wenn weder Hostname noch Hostadresse angegeben sind, verbindet `libpq` über eine lokale Unix-Domain-Socket.

`port`

Die Portnummer, mit der auf dem Server verbunden werden soll, bzw. die Dateierweiterung bei Verbindungen über Unix-Domain-Sockets.

`dbname`

Der Datenbankname.

`user`

Der Benutzername für die Verbindung.

`password`

Das Passwort, für den Fall, dass der Server Authentifizierung mit Passwörtern fordert.

`connect_timeout`

Das Zeitfenster in Sekunden, das der Verbindungsfunktion gegeben wird. Null oder nicht gesetzt heißt unendlich.

`options`

Konfigurationsoptionen, die an den Server geschickt werden.

`tty`

Eine Datei oder ein TTY für Debug-Ausgaben vom Server.

`requiressl`

Wenn 1, wird eine SSL-Verbindung mit dem Server verlangt. `libpq` verweigert dann die Verbindung, wenn der Server kein SSL anbietet. Wenn 0 (Vorgabe), wird `libpq` den Verbindungstyp mit dem Server aushandeln.

Wenn ein Parameter nicht angegeben wurde, wird die entsprechende Umgebungsvariable (siehe Abschnitt 27.9) untersucht. Wenn die Umgebungsvariable auch nicht gesetzt ist, werden die eingebauten Vorgaben verwendet.

`PQsetdbLogin`

Baut eine neue Verbindung mit dem Datenbankserver auf.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

Dies ist der Vorgänger von `PQconnectdb` mit einer festen Zahl von Parametern, aber mit der gleichen Funktionalität.

`PQsetdb`

Baut eine neue Verbindung mit dem Datenbankserver auf.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

Das ist ein Makro, das `PQsetdbLogin` mit Null-Zeigern für die Parameter `login` und `pwd` aufruft. Es ist hauptsächlich wegen der Kompatibilität mit alten Programmen vorhanden.

`PQconnectStart`,
`PQconnectPoll`

Baut eine neue Verbindung mit dem Datenbankserver auf, ohne zu blockieren.

```
PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Diese zwei Funktionen werden verwendet, um eine Verbindung mit dem Datenbankserver zu öffnen, ohne dass die Anwendung während der Eingabe- oder Ausgabeoperationen blockiert wird.

Die Datenbankverbindung wird mit den Parametern in der Zeichenkette `conninfo`, die `PQconnectStart` übergeben wird, erstellt. Die Zeichenkette hat das gleiche Format wie unter `PQconnectdb` beschrieben.

Weder `PQconnectStart` noch `PQconnectPoll` blockieren, wenn bestimmte Voraussetzungen erfüllt sind:

- Die Parameter `hostaddr` und `host` werden entsprechend verwendet um Hostnamenauflösungen zu vermeiden. Schauen Sie in der Dokumentation dieser Parameter oben unter `PQconnectdb` wegen Einzelheiten nach.
- Wenn Sie `PQtrace` aufrufen, achten Sie darauf, dass das Objekt, in das der Datenstrom geschrieben wird, nicht blockieren kann.
- Bringen Sie die Socket selbst in den passenden Zustand, bevor Sie `PQconnectPoll` wie unten beschrieben aufrufen.

Um eine nicht blockierende Verbindung aufzubauen, rufen Sie zuerst `conn = PQconnectStart("verbindungs_parameter")` auf. Wenn `conn` dann ein Null-Zeiger ist, konnte `libpq` nicht genug Speicher für eine neue `PGconn`-Struktur vergeben. Ansonsten erhalten Sie einen gültigen `PGconn`-Zeiger zurück (der aber noch keine gültige Verbindung zur Datenbank darstellt). Nachdem `PQconnectStart` zurückkehrt, rufen Sie `status = PQstatus(conn)` auf. Wenn der Status gleich *CONNECTION_BAD* ist, dann ist `PQconnectStart` fehlgeschlagen.

Wenn `PQconnectStart` erfolgreich war, fahren Sie mit der Verbindungsprozedur fort und achten dabei auf die Statusmeldungen. Führen Sie folgende Schleife aus: Betrachten Sie eine Verbindung anfangs als “inaktiv”. Wenn `PQconnectPoll` zuletzt *PGRES_POLLING_ACTIVE* ergab, betrachten Sie sie stattdessen als “aktiv”. Wenn `PQconnectPoll(conn)` zuletzt *PGRES_POLLING_READING* ergab, warten Sie mit `select()` auf lesbare Daten auf der durch `PQsokket(conn)` ermittelten Socket. Wenn es zuletzt *PGRES_POLLING_WRITING* ergab, warten Sie mit `select()` auf Schreibfreigabe auf der gleichen Socket. Wenn Sie `PQconnectPoll` noch nicht ausgeführt haben, d.h. nach dem Aufruf von `PQconnectStart`, verhalten Sie sich, als ob das letzte Ergebnis *PGRES_POLLING_WRITING* gewesen wäre. Wenn `select()` anzeigt, dass die Socket bereit ist, betrachten Sie sie als “aktiv”. Wenn entschieden wurde, dass die Verbindung “aktiv” ist, rufen Sie `PQconnectPoll(conn)` noch einmal auf. Wenn dieser Aufruf *PGRES_POLLING_FAILED* ergibt, ist die Verbindungsprozedur fehlgeschlagen. Wenn der Aufruf *PGRES_POLLING_OK* ergibt, wurde die Verbindung erfolgreich aufgebaut.

Es sei angemerkt, dass die Verwendung von `select()`, um die Socket zu prüfen, nur ein (wahrscheinliches) Beispiel ist. Wenn andere Möglichkeiten, zum Beispiel mit `poll()`, zur Verfügung stehen, können diese natürlich auch verwendet werden.

Zu jedem beliebigen Zeitpunkt während des Verbindungsaufbaus kann der Zustand der Verbindung mit `PQstatus` geprüft werden. Wenn dies *CONNECTION_BAD* ergibt, ist die Verbindungsprozedur fehlgeschlagen, wenn es `CONNECTION_OK` ergibt, ist die Verbindung bereit. Beide dieser Zustände können genauso durch das Ergebnis der Funktion `PQconnectPoll` erkannt werden, wie oben beschrieben. Auch andere Zustände können während einer asynchronen Verbindungsprozedur (und nur) auftreten. Diese geben dann den aktuellen Fortschritt der Verbindungsprozedur an und können zum Beispiel nützlich sein, um Rückmeldungen an den Benutzer zu erzeugen. Diese Zustände sind:

CONNECTION_STARTED

Wartet auf die Erstellung der Verbindung.

CONNECTION_MADE

Verbindung in Ordnung; wartet auf Senden.

CONNECTION_AWAITING_RESPONSE

Wartet auf Antwort vom Server.

CONNECTION_AUTH_OK

Authentifizierung erhalten; wartet auf Fortsetzung des Verbindungsaufbaus.

CONNECTION_SETENV

Aushandlung der Umgebung (Teil des Verbindungsaufbaus).

Beachten Sie, dass obwohl diese Konstanten bleiben werden (um Kompatibilität zu erhalten), eine Anwendung sich niemals darauf verlassen sollte, dass sie in einer bestimmten Reihenfolge oder überhaupt auftreten oder dass der Zustand immer einer dieser beschriebenen Werte sein wird. Eine Anwendung kann Folgendes machen:

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Verbinde...";
        break;

    case CONNECTION_MADE:
        feedback = "Mit Server verbunden...";
        break;
.
.
.
    default:
         feedback = "Verbinde...";
}
```

Wenn `PQconnectStart` einen gültigen Zeiger (nicht Null) zurückgibt, dann beachten Sie, dass Sie `PQfinish` aufrufen müssen, wenn Sie fertig sind, um die Verbindungsstruktur und zugehörige Speicherblöcke freizugeben. Das muss auch getan werden, wenn `PQconnectStart` oder `PQconnectPoll` nicht erfolgreich waren.

`PQconnectPoll` blockiert gegenwärtig, wenn `libpq` mit SSL-Unterstützung compiliert wurde. Diese Einschränkung wird möglicherweise in der Zukunft entfernt werden.

Schließlich belassen diese Funktionen die Socket im nicht blockierenden Zustand, als ob `PQsetnonblocking` aufgerufen worden wäre.

`PQconndefaults`

Gibt die Vorgabeverbindungseinstellungen zurück.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
   char *keyword;   /* Schlüselwort */
   char *envvar;    /* Name der Umgebungsvariable */
   char *compiled;  /* eingebauter Vorgabewert */
```

```
    char *val;       /* aktueller Vorgabewert, oder NULL */
    char *label;     /* Feldbeschriftung in Dialogfenster */
    char *dispchar;  /* Darstellung des Feldes in einem Dialogfenster:
                        "" normale Darstellung
                        "*" Passwort - nicht darstellen
                        "D" Debugoption - normalerweise nicht anzeigen
    */ int dispsize; /* Feldgröße in Zeichen für Dialogfenster */
} PQconninfoOption;
```

Ergibt ein Array mit Verbindungsparametern vom Typ PQconninfoOption. Das kann verwendet werden, um alle möglichen Parameter für `PQconnectdb` und ihre aktuellen Vorgabewerte zu ermitteln. Der letzte Eintrag des Arrays hat einen Null-Zeiger im Feld `keyword`. Beachten Sie, dass die aktuellen Vorgabewerte (in `val`) von Umgebungsvariablen und anderen Faktoren abhängen. In das Ergebnis sollte nicht geschrieben werden.

Wenn Sie mit dem Parameterarray fertig sind, sollten Sie es mit `PQconninfoFree` freigeben. Wenn Sie das nicht tun, verursachen Sie mit jedem Aufruf von `PQconndefaults` ein kleines Speicherleck.

In PostgreSQL-Versionen vor 7.0 gab `PQconndefaults` einen Zeiger in ein statisches Array zurück, anstelle eines dynamisch angelegten Arrays. Das war nicht Thread-sicher und wurde deswegen geändert.

`PQfinish`

Schließt die Verbindung mit dem Server. Gibt außerdem den vom `PGconn`-Objekt belegten Speicher frei.

```
void PQfinish(PGconn *conn);
```

Beachten Sie, dass die Anwendung `PQfinish` auch dann aufrufen sollten, wenn der Verbindungsversuch mit dem Server fehlgeschlagen ist (durch `PQstatus` erkennbar), um den vom `PGconn`-Objekt belegten Speicher freizugeben. Der `PGconn`-Zeiger sollte nach dem Aufruf von `PQfinish` nicht mehr verwendet werden.

`PQreset`

Setzt den Kommunikationskanal zum Server zurück.

```
void PQreset(PGconn *conn);
```

Diese Funktion schließt die Verbindung zum Server und versucht, eine neue Verbindung zum selben Server mit den gleichen Parametern aufzubauen. Das kann zur Fehlerbehandlung nützlich sein, wenn eine funktionierende Verbindung verloren gegangen ist.

`PQresetStart,`
`PQresetPoll`

Setzen den Kommunikationskanal zum Server zurück, ohne zu blockieren.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Diese Funktionen schließen die Verbindung zum Server und versuchen, eine neue Verbindung zum selben Server mit den gleichen Parametern aufzubauen. Das kann zur Fehlerbehandlung nützlich sein,

wenn eine funktionierende Verbindung verloren gegangen ist. Sie unterscheiden sich von `PQreset` (siehe oben) dadurch, dass sie auf eine nicht blockierende Art und Weise vorgehen. Sie haben dabei die gleichen Einschränkungen wie `PQconnectStart` und `PQconnectPoll`.

Um einen Verbindungs-Reset einzuleiten, rufen Sie `PQresetStart` auf. Wenn es 0 ergibt, ist er fehlgeschlagen. Wenn es 1 ergibt, überwachen Sie den Reset mit `PQresetPoll` auf genau die gleiche Art, als ob Sie die Verbindung über `PQconnectPoll` aufbauen.

`libpq`-Anwendungsprogrammierer sollten darauf achten, die Abstraktion von `PGconn` zu bewahren. Verwenden Sie die folgenden Zugriffsfunktionen, um auf den Inhalt von `PGconn` zuzugreifen. Vermeiden Sie es, direkt auf die Felder der `PGconn`-Struktur zuzugreifen, weil diese sich in der Zukunft ändern könnten. (Seit PostgreSQL 6.4 ist die Definition des struct hinter `PGconn` nicht mehr in der Datei `libpq-fe.h` vorhanden. Wenn Sie alten Code haben, der auf die Felder von `PGconn` direkt zugreift, können Sie ihn weiter verwenden, wenn Sie zusätzlich die Datei `libpq-int.h` einbinden, aber Sie sollten Ihren Code sehr bald ausbessern.)

`PQdb`

Ergibt den Datenbanknamen der Verbindung.

```
char *PQdb(const PGconn *conn);
```

`PQdb` und die nächsten paar Funktionen ergeben die Werte, die bei der Verbindung festgestellt werden. Die Werte sind während der Lebenszeit des `PGconn`-Objekts unveränderlich.

`PQuser`

Ergibt den Benutzernamen der Verbindung.

```
char *PQuser(const PGconn *conn);
```

`PQpass`

Ergibt das Passwort der Verbindung.

```
char *PQpass(const PGconn *conn);
```

`PQhost`

Ergibt den Serverhostnamen der Verbindung.

```
char *PQhost(const PGconn *conn);
```

`PQport`

Ergibt den Port der Verbindung.

```
char *PQport(const PGconn *conn);
```

`PQtty`

Ergibt das TTY für Debugausgaben der Verbindung.

```
char *PQtty(const PGconn *conn);
```

`PQoptions`

Ergibt die Konfigurationsoptionen, die in der Verbindungsanfrage übergeben wurden.

```
char *PQoptions(const PGconn *conn);
```

`PQstatus`

Ergibt den Status der Verbindung.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Der Status kann einer aus einer Reihe von Werten sein. Nur zwei davon werden allerdings außerhalb einer asynchronen Verbindungsprozedur auftreten: `CONNECTION_OK` und `CONNECTION_BAD`. Eine Verbindung, die in Ordnung ist, hat den Status `CONNECTION_OK`. Ein gescheiterter Verbindungsversuch wird vom Status `CONNECTION_BAD` signalisiert. Wenn der Status in Ordnung ist, bleibt er normalerweise so bis zu `PQfinish`, aber ein Kommunkationsfehler kann dazu führen, dass der Status vorzeitig auf `CONNECTION_BAD` wechselt. In diesem Fall könnte die Anwendung versuchen, die Verbindung mit `PQreset` zu erneuern.

Schauen Sie im Eintrag für `PQconnectStart` und `PQconnectPoll` bezüglich der anderen möglichen Statuscodes nach.

`PQerrorMessage`

Ergibt die Fehlermeldung, die zuletzt durch eine Operation auf dieser Verbindung verursacht wurde.

```
char *PQerrorMessage(const PGconn* conn);
```

Fast alle `libpq`-Funktionen setzen eine Meldung für `PQerrorMessage`, wenn ein Fehler auftritt. Beachten Sie, dass ein nicht leeres `PQerrorMessage`-Ergebnis ein abschließendes Newline-Zeichen enthält.

`PQsocket`

Ermittelt die Dateideskriptor-Nummer der mit dem Server verbundenen Socket. Ein gültiger Deskriptor ist größer oder gleich 0; das Ergebnis -1 sagt aus, dass aktuell keine Verbindung offen ist.

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Ergibt die Prozesskennung (PID) des Serverprozesses, der diese Verbindung bearbeitet.

```
int PQbackendPID(const PGconn *conn);
```

Die PID kann zum Debuggen nützlich sein und um `NOTIFY`-Mitteilungen zuzuordnen (da diese die PID des Ursprungsprozesses enthalten). Beachten Sie, dass die PID zu einem Prozess auf der Datenbankservermaschine und nicht auf der lokalen Maschine gehört!

`PQgetssl`

Ergibt die SSL-Struktur für die Verbindung oder einen Null-Zeiger, wenn SSL nicht verwendet wird.

```
SSL *PQgetssl(const PGconn *conn);
```

Diese Struktur kann verwendet werden, um Verschlüsselungsgrade zu ermitteln, Serverzertifikate zu prüfen und mehr. Informationen über diese Struktur finden Sie in der Anleitung zu OpenSSL.

Um den Prototyp für diese Funktion sichtbar zu machen, müssen Sie *USE_SSL* definieren. Dadurch wird auch automatisch `ssl.h` von OpenSSL eingebunden.

27.2 Funktionen zur Ausführung von Befehlen

Wenn eine Verbindung mit dem Datenbankserver erfolgreich aufgebaut wurde, dann werden die hier beschriebenen Funktionen verwendet, um SQL-Anfragen und -Befehle auszuführen.

27.2.1 Hauptfunktionen

`PQexec`

Schickt einen Befehl an den Server und wartet auf das Ergebnis.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Die Funktion ergibt einen Zeiger auf ein `PGresult` oder möglicherweise einen Null-Zeiger. Normalerweise wird ein gültiger Zeiger zurückgegeben, außer wenn der Speicher knapp ist oder bei ernsthaften Fehlern, wie zum Beispiel, wenn der Befehl nicht an den Server geschickt werden konnte. Wenn das Ergebnis ein Null-Zeiger ist, sollte es wie ein Ergebnis mit Status *PGRES_FATAL_ERROR* behandelt werden. Verwenden Sie `PQerrorMessage`, um mehr Informationen über den Fehler zu erhalten.

Die Struktur `PGresult` enthält das vom Server geschickte Ergebnis. `libpq`-Anwendungsprogrammierer sollten darauf achten, die Abstraktion von `PGresult` zu bewahren. Verwenden Sie die folgenden Zugriffsfunktionen, um auf den Inhalt von `PGresult` zuzugreifen. Vermeiden Sie es, direkt auf die Felder der `PGresult`-Struktur zuzugreifen, weil diese sich in der Zukunft ändern könnten. (Seit PostgreSQL 6.4 ist die Definition des struct hinter `PGresult` nicht mehr in der Datei `libpq-fe.h` vorhanden. Wenn Sie alten Code haben, der auf die Felder von `PGresult` direkt zugreift, können Sie ihn weiter verwenden, wenn Sie zusätzlich die Datei `libpq-int.h` einbinden, aber Sie sollten Ihren Code sehr bald ausbessern.)

`PQresultStatus`

Ergibt den Status des Befehls.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

Das Ergebnis von `PQresultStatus` ist einer der folgenden Werte:

`PGRES_EMPTY_QUERY`

Die an den Server gesendete Zeichenkette war leer.

`PGRES_COMMAND_OK`

Erfolgreicher Abschluss eines Befehls, der keine Daten liefert.

`PGRES_TUPLES_OK`

Die Anfrage wurde erfolgreich ausgeführt.

`PGRES_COPY_OUT`

Ausgehender `COPY`-Datentransfer (vom Server) gestartet.

`PGRES_COPY_IN`

Eingehender COPY-Datentransfer (zum Server) gestartet.

PGRES_BAD_RESPONSE

Die Antwort des Servers wurde nicht verstanden.

PGRES_NONFATAL_ERROR

Ein nicht fataler Fehler trat auf.

PGRES_FATAL_ERROR

Ein fataler Fehler trat auf.

Wenn das Ergebnis PGRES_TUPLES_OK ist, können die unten beschriebenen Funktionen verwendet werden, um die von der Anfrage gelieferten Zeilen auszulesen. Beachten Sie, dass ein SELECT-Befehl, der zufällig null Zeilen im Ergebnis hat, trotzdem PGRES_TUPLES_OK anzeigt. PGRES_COMMAND_OK ist für Befehle, die niemals Zeilen als Ergebnis haben (INSERT, UPDATE usw.). Wenn die Antwort PGRES_EMPTY_QUERY ist, dann deutet das oft auf einen Fehler in der Clientsoftware hin.

PQresStatus

Wandelt den von PQresultStatus ermittelten Statuscode in eine Textkonstante um, die den Statuscode beschreibt.

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

Ergibt die von dem Befehl verursachte Fehlermeldung oder eine leere Zeichenkette, wenn kein Fehler vorliegt.

```
char *PQresultErrorMessage(const PGresult *res);
```

Direkt nach einem Aufruf von PQexec oder PQgetResult ergibt PQerrorMessage (aufgerufen mit der Verbindung als Argument) das gleiche Resultat wie PQresultErrorMessage (aufgerufen mit dem Ergebnis als Argument). Ein PGresult behält allerdings die Fehlermeldung, bis es zerstört wird, wohingegen die Fehlermeldung im Verbindungsobjekt bei der nächsten Operation geändert wird. Verwenden Sie PQresultErrorMessage, wenn Sie den Status eines bestimmten PGresult-Objekts wissen wollen, und verwenden Sie PQerrorMessage, wenn Sie den Status der letzten Operation auf einer Verbindung wissen wollen.

PQclear

Gibt den zu einem PGresult-Objekt gehörenden Speicherplatz frei. Jedes Befehlsergebnis sollte mit PQclear freigegeben werden, wenn es nicht mehr benötigt wird.

```
void PQclear(PQresult *res);
```

Sie können ein PGresult-Objekt so lange behalten, wie sie wollen; es bleibt auch gültig, wenn ein neuer Befehl ausgeführt wird oder gar wenn die Verbindung geschlossen wird. Um es zu entfernen, müssen Sie PQclear aufrufen. Wenn Sie das nicht tun, verursachen Sie Speicherlecks in ihrer Clientanwendung.

PQmakeEmptyPGresult

Konstruiert eine leeres PGresult-Objekt mit dem angegebenen Status.

```
PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Das ist die Funktion, die von libpq intern verwendet wird, um ein leeres PGresult-Objekt anzulegen. Sie ist öffentlich, weil manche Anwendungen es praktisch finden, Ergebnisobjekte (besonders solche mit Fehlerstatus) selbst zu erzeugen. Wenn conn kein Null-Zeiger ist und status einen Fehler anzeigt, wird die aktuelle Fehlermeldung der angegebenen Verbindung in das PGresult-Objekt kopiert. Beachten Sie, dass das Ergebnis mit PQclear freigegeben werden muss, genauso wie bei einem PGresult, dass von libpq selbst erzeugt wurde.

27.2.2 Zeichenketten für SQL-Befehle vorbereiten

PQescapeString bereitet Zeichenketten zur Verwendung in SQL-Befehlen vor, indem bestimmte Zeichen durch Fluchtfolgen ersetzt werden.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

Wenn Sie Zeichenketten verwenden wollen, die aus nicht vertrauenswürdigen Quellen stammen (zum Beispiel weil irgendein wahlloser Benutzer sie eingegeben hat), sollten Sie diese aus Sicherheitsgründen nicht direkt in einen SQL-Befehl einbauen. Vielmehr sollten Sie bestimmte Zeichen, die vom SQL-Parser besonders behandelt werden, durch Fluchtfolgen ersetzen. PQescapeString führt diese Operation aus.

Der Parameter from zeigt auf das erste Zeichen des Textes, der bearbeitet werden soll, und der Parameter length zählt die Zeichen im Text. (Ein Null-Byte am Ende ist weder notwendig, noch wird es mitgezählt.) to muss auf einen Puffer zeigen, der mindestens ein Zeichen mehr als zweimal der Wert von length aufnehmen, ansonsten ist das Verhalten undefiniert. Ein Aufruf von PQescapeString schreibt eine Version der Zeichenkette from in den Puffer to, in der Sonderzeichen durch Fluchtfolgen ersetzt worden sind, sodass sie kein Unheil anrichten können, und fügt ein abschließendes Null-Byte an. Die Apostrophe, die um eine Zeichenkettenkonstante in PostgreSQL gehören, sind nicht Teil des Ergebnisses.

Das Ergebnis von PQescapeString ist die Anzahl der nach to geschriebenen Zeichen, ohne das abschließende Null-Byte. Das Verhalten ist nicht definiert, wenn die Parameter to und from sich überschneiden.

27.2.3 Binäre Daten für SQL-Befehle vorbereiten

```
PQescapeBytea
```

Bereitet binäre Daten zur Verwendung in SQL-Befehlen mit dem Typ bytea vor.

```
unsigned char *PQescapeBytea(unsigned char *from,
size_t from_length,
size_t *to_length);
```

Bestimmte Bytewerte *müssen* durch Fluchtfolgen ersetzt werden (aber alle Bytewerte *dürfen* durch Fluchtfolgen ersetzt werden), wenn Sie in einer bytea-Konstante in einem SQL-Befehl verwendet werden. Generell wird bei einer solchen Umwandlung ein Byte durch eine dreistellige Oktalzahl, die dem Bytewert entspricht, und einen vorangestellten Backslash ersetzt. Der Apostroph (') und der Backslash (\) haben besondere Fluchtfolgen. Weitere Informationen finden Sie in Abschnitt 8.4. PQescapeBytea führt diese Operation aus und wandelt nur die Bytes um, bei denen es nötig ist.

Der Parameter from zeigt auf das erste Byte der Kette, die bearbeitet werden soll, und der Parameter from_length gibt die Anzahl der Bytes in dieser Datenkette an. (Ein abschließendes Null-Byte ist we-

der notwendig noch wird es mitgezählt.) Der Parameter `to_length` zeigt auf eine Variable, die die Länge der umgewandelten Datenkette aufnimmt. Die Länge des Ergebnis schließt das abschließende Null-Byte im Ergebnisses mit ein.

`PQescapeBytea` ergibt eine Version der Binärdaten in `from`, in der alle Sonderzeichen durch Fluchtfolgen ersetzt wurden, sodass sie vom Zeichenkettenparser und von der bytea-Eingabefunktion ordnungsgemäß verarbeitet werden können. Der Ausgabepuffer wird mit `malloc()` angelegt. Ein abschließendes Null-Byte wird angehängt. Die Apostrophe, die um eine Zeichenkettenkonstante in PostgreSQL gehören, sind nicht Teil des Ergebnisses.

`PQunescapeBytea`

Wandelt eine Zeichenkettendarstellung mit Fluchtfolgen von binären Daten in binäre Daten um – die Umkehrung von `PQescapeBytea`.

```
unsigned char *PQunescapeBytea(unsigned char *from, size_t *to_length);
```

Der Parameter `from` zeigt auf eine Zeichenkette mit bytea-Fluchtfolgen, wie sie zum Beispiel von `PQgetvalue` zurückgegeben werden, wenn es auf eine bytea-Spalte angewendet wird. `PQunescapeBytea` wandelt diese Zeichendarstellung in die entsprechende binäre Darstellung um. Das Ergebnis ist ein Zeiger in einem mit `malloc()` angelegten Puffer, oder bei einem Fehler ein Null-Zeiger; die Länge des Puffers wird in `to_length` abgelegt.

27.2.4 Ermittlung von Informationen über Anfrageergebnisse

`PQntuples`

Ergibt die Anzahl der Zeilen (Tupel) im Anfrageergebnis.

```
int PQntuples(const PGresult *res);
```

`PQnfields`

Ergibt die Anzahl der Spalten (Felder) in jeder Zeile des Anfrageergebnisses.

```
int PQnfields(const PGresult *res);
```

`PQfname`

Ergibt den Spaltennamen der Spalte mit der angegebenen Nummer. Spaltennummern beginnen bei 0.

```
char *PQfname(const PGresult *res,
              int spalten_nummer);
```

`PQfnumber`

Ergibt die Spaltennummer zum angegebenen Spaltennamen.

```
int PQfnumber(const PGresult *res,
              const char *field_name);
```

Das Ergebnis ist -1, wenn keine Spalte den angegebenen Namen hat.

`PQftype`

Ergibt den Datentyp der Spalte mit der angegebenen Nummer. Das Ergebnis ist ein ganzzahliger Typ, der die interne OID-Nummer des Typs darstellt. Spaltennummern beginnen bei 0.

```
Oid PQftype(const PGresult *res,
            int spalten_nummer);
```

Sie können die Systemtabelle `pg_type` abfragen, um Namen und Eigenschaften der verschiedenen Datentypen zu erfahren. Die OIDs der eingebauten Datentypen sind in der Datei `src/include/catalog/pg_type.h` im Quelltextbaum definiert.

`PQfmod`

Ergibt die typenspezifischen Modifikationsdaten der Spalte mit der angegebenen Nummer. Spaltennummern beginnen bei 0.

```
int PQfmod(const PGresult *res,
           int spalten_nummer);
```

`PQfsize`

Ergibt die Größe in Bytes der Spalte mit der angegebenen Nummer. Spaltennummern beginnen bei 0.

```
int PQfsize(const PGresult *res,
            int spalten_nummer);
```

`PQfsize` ergibt, wie viel Speicher die Spalte in einer Datenbankzeile belegt, also die Größe in der internen Darstellung des Servers. Das Ergebnis ist -1, wenn die Spalte eine variable Größe hat.

`PQbinaryTuples`

Ergibt 1, wenn das `PGresult` binäre Zeilendaten enthält, und 0, wenn es Textdaten enthält.

```
int PQbinaryTuples(const PGresult *res);
```

Gegenwärtig können binäre Zeilendaten nur von einer Anfrage erzeugt werden, die Daten aus einem binären Cursor ausliest.

27.2.5 Auslesen von Anfrageergebnissen

`PQgetvalue`

Liest einen einzelnen Spaltenwert aus einer Zeile aus einem `PGresult`. Zeilen- und Spaltennummern beginnen bei 0.

```
char* PQgetvalue(const PGresult *res,
                 int zeilen_nummer,
                 int spalten_nummer);
```

Bei den meisten Anfragen ist der von `PQgetvalue` zurückgegebene Wert eine Zeichenkettendarstellung, mit einem abschließenden Null-Byte. Aber wenn `PQbinaryTuples` 1 ergibt, dann ist der von `PQgetvalue` zurückgegebene Wert in der internen binären Darstellung des Typs wie im Server (aber ohne das Größenwort, wenn der Typ eine variable Länge hat). Es liegt dann in der Verantwortung des Programmierers, die Daten in den korrekten C-Typ umzuwandeln.

Der von `PQgetvalue` erhaltene Zeiger zeigt in einen Speicherbereich, der zur `PGresult`-Struktur gehört. Man sollte die Daten, auf die er zeigt, nicht verändern, und man muss die Daten selbst in einen anderen Speicher kopieren, wenn man sie über die Lebensdauer der `PGresult`-Struktur hinaus verwenden will.

`PQgetisnull`

Testet, ob eine Spalte den NULL-Wert enthält. Zeilen- und Spaltennummern beginnen bei 0.

```
int PQgetisnull(const PGresult *res,
                int zeilen_nummer,
                int spalten_nummer);
```

Diese Funktion ergibt 1, wenn die Spalte den NULL-Wert enthält, und 0, wenn nicht. (Beachten Sie, dass `PQgetvalue` bei NULL-Werten eine leere Zeichenkette, keinen Null-Zeiger, zurückgibt.)

`PQgetlength`

Ergibt die Länge eines Spaltenwerts in Bytes. Zeilen- und Spaltennummern beginnen bei 0.

```
int PQgetlength(const PGresult *res,
                int zeilen_nummer,
                int spalten_nummer);
```

Dies ist die wirkliche Länge des Datenwerts, das heißt die Größe des durch `PQgetvalue` erhaltenen Objekts. Beachten Sie, dass bei Daten in Textformat dies wenig mit der von `PQfsize` ermittelten binären Größe zu tun hat.

`PQprint`

Gibt alle Zeilen und wahlweise die Spaltennamen im angegebenen Ausgabestrom aus.

```
void PQprint(FILE* fout, /* Ausgabestrom */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool header;    /* Kopfzeilen und Zeilenzahl ausgeben */
    pqbool align;     /* Felder mit Leerzeichen ausrichten */
    pqbool standard;  /* altes blödes Format */
    pqbool html3;     /* HTML-Tabellen ausgeben */
    pqbool expanded;  /* erweitertes Format */
    pqbool pager;     /* Pager wenn nötig verwenden */
    char *fieldSep;   /* Feldtrennzeichen */
    char *tableOpt;   /* Attribute für table-Element in HTML */
    char *caption;    /* Caption in HTML-Tabellen */
    char **fieldName; /* Ersatzfeldnamen (Array mit NULL abgeschlossen) */
} PQprintOpt;
```

Diese Funktion wurde früher von `psql` verwendet, um Anfrageergebnisse auszugeben, aber das ist nicht mehr der Fall und die Funktion wird nicht mehr aktiv betreut.

27.2.6 Ermittlung von Ergebnissen anderer Befehle

`PQcmdStatus`

Ergibt die Statuszeichenkette des SQL-Befehls, der das `PGresult` geschaffen hat.

```
char * PQcmdStatus(PGresult *res);
```

`PQcmdTuples`

Ergibt die Anzahl der Zeilen, auf die der SQL-Befehl Auswirkung hatte.

```
char * PQcmdTuples(PGresult *res);
```

Wenn der SQL-Befehl, der dieses `PGresult` geschaffen hat, `INSERT`, `UPDATE` oder `DELETE` war, ergibt dies eine Zeichenkette, die eine Zahl enthält, die die Anzahl der betroffenen Zeilen angibt. Wenn der Befehl ein anderer war, ergibt dies eine leere Zeichenkette.

`PQoidValue`

Ergibt die OID der eingefügten Zeile, wenn die SQL-Anweisung ein `INSERT` war, das genau eine Zeile eingefügt hat, und die Zieltabelle OIDs hat. Ansonsten ergibt es `InvalidOid`.

```
Oid PQoidValue(const PGresult *res);
```

Der Typ Oid und die Konstante `InvalidOid` sind definiert, wenn Sie die `libpq`-Headerdatei eingebunden haben. Beide sind ein ganzzahliger Typ.

`PQoidStatus`

Ergibt eine Zeichenkette mit der OID der eingefügten Zeile, wenn die SQL-Anweisung ein `INSERT` war. (Die Zeichenkette ist `0`, wenn das `INSERT` nicht genau eine Zeile eingefügt oder die Zieltabelle keine OIDs hat.) Wenn der Befehl kein `INSERT` war, dann ergibt die Funktion eine leere Zeichenkette.

```
char * PQoidStatus(const PGresult *res);
```

Dieser Funktion sollte `PQoidValue` vorgezogen werden. Sie ist auch nicht Thread-sicher.

27.3 Asynchrone Befehlsverarbeitung

Die Funktion ist angemessen für das Absenden von Befehlen in normalen, synchronen Anwendungen. Sie hat allerdings ein paar Mängel, die für einige Benutzer von Bedeutung sein können:

- `PQexec` wartet, bis der Befehl abgeschlossen ist. Die Anwendung hat möglicherweise andere Dinge zu tun (zum Beispiel eine Benutzerschnittstelle zu bedienen) und kann nicht blockieren, während sie auf das Ergebnis wartet.
- Da die Ausführung der Clientanwendung angehalten ist, während sie auf das Ergebnis wartet, ist es schwer für die Anwendung zu entscheiden, dass sie den aktiven Befehl abbrechen möchte. (Es geht mit einem Signalhandler, aber nicht anders.)
- `PQexec` kann nur eine `PGresult`-Struktur zurückgeben. Wenn die gesendete Befehlszeichenkette mehrere SQL-Befehle enthält, dann werden alle `PGresult`-Ergebnisse außer dem letzten von `PQexec` verworfen.

Anwendungen, die diese Beschränkungen nicht mögen, können die Funktionen verwenden, die auch `PQexec` zugrunde liegen: `PQsendQuery` und `PQgetResult`.

Ältere Programme, die diese Funktionalität sowie `PQputline` und `PQputnbytes` benutzen, können blockieren, während sie darauf warten, Daten an den Server zu schicken. Um diesem Problem zu begegnen, wurde die Funktion `PQsetnonblocking` hinzugefügt. Ältere Anwendungen können `PQsetnonblokking` weiterhin vernachlässigen und erhalten eine möglicherweise blockierende Verbindung. Neuere Programme können `PQsetnonblocking` verwenden, um eine vollständig blockierungsfreie Verbindung mit dem Server zu erreichen.

`PQsetnonblocking`

Stellt den Blockierzustand der Verbindung ein.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Der Zustand wird auf nicht blockierend gesetzt, wenn `arg` 1 ist, und blokkierend, wenn `arg` 0 ist. Gibt 0 zurück, wenn alles in Ordnung ist, und -1 bei einem Fehler.

Im nicht blockierenden Zustand werden Aufrufe von `PQputline`, `PQputnbytes`, `PQsendQuery` und `PQendcopy` nicht blockieren, sondern einen Fehler zurückgeben, wenn sie nochmal aufgerufen werden müssen.

Wenn eine Datenbankverbindung sich im nicht blockierenden Modus befindet und `PQexec` aufgerufen wird, dann wird der Zustand der Verbindung vorübergehend auf blockierend gestellt, bis der `PQexec`-Aufruf vorbei ist.

Es ist zu erwarten, dass in der Zukunft weitere Teile von `libpq` für den nicht blockierenden Modus bereitgemacht werden.

`PQisnonblocking`

Ergibt den Blockierzustand der Datenbankverbindung.

```
int PQisnonblocking(const PGconn *conn);
```

Ergibt 1, wenn die Verbindung im nicht blockierenden Modus ist, und 0 im blockierenden Modus.

`PQsendQuery`

Schickt einen Befehl an den Server, ohne auf das Ergebnis zu warten. Der Rückgabewert ist 1, wenn der Befehl erfolgreich gesendet wurde, und 0, wenn nicht (in dem Fall können Sie mit `PQerrorMessage` Informationen über den Fehler erhalten).

```
int PQsendQuery(PGconn *conn,
                const char *query);
```

Nach einem erfolgreichen Aufruf von `PQsendQuery` rufen Sie `PQgetResult` einmal oder mehrmals auf, um die Ergebnisse zu erhalten. `PQsendQuery` kann nicht erneut (auf derselben Verbindung) aufgerufen werden, ehe `PQgetResult` einen Null-Zeiger zurückgegeben hat, welcher anzeigt, dass der Befehl beendet ist.

`PQgetResult`

Wartet auf das nächste Ergebnis eines vorangegangenen `PQsendQuery`-Aufrufs und gibt es zurück. Wenn der Befehl beendet ist und es keine weiteren Ergebnisse gibt, wird ein Null-Zeiger zurückgegeben.

```
PGresult *PQgetResult(PGconn *conn);
```

`PQgetResult` muss so lange wiederholt aufgerufen werden, bis es einen Null-Zeiger zurückgibt, welcher anzeigt, dass der Befehl beendet ist. (Wenn es aufgerufen wird und kein Befehl aktiv ist, ergibt `PQgetResult` sofort einen Null-Zeiger.) Jedes Ergebnis von `PQgetResult` (außer Null-Zeiger) sollte mit denselben, oben beschriebenen Zugriffsfunktionen für `PGresult` verarbeitet werden. Vergessen Sie auch nicht, jedes Ergebnisobjekt mit `PQclear` freizugeben, wenn Sie damit fertig sind. Beachten Sie, dass `PQgetResult` nur blockiert, wenn ein Befehl aktiv ist und die notwendigen Antwortdaten noch nicht von `PQconsumeInput` gelesen wurden.

Die Verwendung von `PQsendQuery` und `PQgetResult` löst eins der Probleme von `PQexec`: Wenn eine Befehlszeichenkette mehrere SQL-Befehle enthält, können die Ergebnisse dieser Befehle einzeln ausgelesen werden. (Das ermöglicht übrigens eine einfache Form der überschneidenden Verarbeitung: Der Client kann die Ergebnisse eines Befehls verarbeiten, während der Server noch an den folgenden Befehlen in derselben Befehlszeichenkette arbeitet.) Durch einen Aufruf von `PQgetResult` wird die Clientanwendung allerdings nach wie vor blockieren, bis der Server den nächsten SQL-Befehl abgeschlossen hat. Das kann durch die richtige Verwendung dreier weiterer Funktionen vermieden werden:

`PQconsumeInput`

Konsumiert Daten vom Server, falls vorhanden.

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` ergibt normalerweise 1, wenn kein Fehler vorliegt, aber 0, wenn irgendein Problem auftrat (dann kann `PQerrorMessage` verwendet werden). Beachten Sie, dass das Ergebnis nicht aussagt, ob irgendwelche Daten angesammelt wurden. Nach dem Aufruf von `PQconsumeInput` kann die Anwendung `PQisBusy` und/oder `PQnotifies` aufrufen, um zu prüfen, ob ihr Status sich geändert hat.

`PQconsumeInput` kann auch aufgerufen werden, wenn die Anwendung noch nicht bereit ist, ein Ergebnis oder eine Nachricht zu verarbeiten. Die Funktion liest die verfügbaren Daten und speichert sie in einem Puffer, wodurch die Anzeige von lesbaren Daten in `select()` verschwindet. Die Anwendung kann also `PQconsumeInput` verwenden, um den `select()`-Zustand sofort zu löschen, und kann die Ergebnisse dann untersuchen, wenn es passt.

`PQisBusy`

Ergibt 1, wenn ein Befehl beschäftigt ist, das heißt, wenn `PQgetResult` beim Warten auf Daten blockieren würde. Wenn 0 zurückgegeben wird, kann `PQgetResult` mit der Versicherung aufgerufen werden, dass es nicht blockieren wird.

```
int PQisBusy(PGconn *conn);
```

`PQisBusy` liest selbst keine Daten vom Server; daher muss zuerst `PQconsumeInput` aufgerufen werden, oder der beschäftigte Zustand wird niemals enden.

`PQflush`

Versucht etwaige zwischengespeicherte Daten an den Server zu schicken (sog. *flush*). Ergibt 0, wenn es erfolgreich war (oder keine Daten zwischengespeichert waren), oder *EOF*, wenn ein Fehler auftrat.

```
int PQflush(PGconn *conn);
```

`PQflush` muss auf einer nicht blockierenden Verbindung aufgerufen werden, bevor `select()` verwendet wird, um zu überprüfen, ob eine Antwort angekommen ist. Wenn 0 zurückgegeben wird, dann ist sichergestellt, dass keine Daten zwischengespeichert sind, die noch an den Server geschickt werden müssen. Nur Anwendungen, die `PQsetnonblocking` verwendet haben, haben dies nötig.

Eine typische Anwendung, die diese Funktionen verwendet, hat eine Hauptschleife, die `select()` aufruft, um auf alle Ereignisse zu warten, auf die sie reagieren muss. Eins dieser Ereignisse wird sein, dass vom Server Daten zum Lesen vorhanden sind, was für `select()` bedeutet, dass auf dem durch `PQsokket` ermittelten Dateideskriptor Daten zum Lesen bereitstehen. Wenn die Hauptschleife das entdeckt, dann sollte sie `PQconsumeInput` aufrufen, um die Daten zu lesen. Sie kann dann `PQisBusy` aufrufen, gefolgt von `PQgetResult`, wenn `PQisBusy` 0 ergab. Sie kann außerdem `PQnotifies` aufrufen, um `NOTIFY`-Nachrichten zu entdecken (siehe Abschnitt 27.5).

Nicht blockierende Verbindungen (solche, die `PQsetnonblocking` verwendet haben), sollten `select()` erst aufrufen, wenn `PQflush` 0 zurückgegeben hat, was anzeigt, dass keine zwischengespeicherten Daten noch darauf warten, an den Server geschickt zu werden.

Ein Client, der `PQsendQuery`/`PQgetResult` verwendet, kann auch versuchen, einen Befehl, der noch vom Server bearbeitet wird, abzubrechen.

```
PQrequestCancel
```

Bittet den Server, die Bearbeitung des aktuellen Befehls abzubrechen.

```
int PQrequestCancel(PGconn *conn);
```

Der Rückgabewert ist 1, wenn die Abbruchanfrage erfolgreich abgeschickt wurde, und 0, wenn nicht. (Wenn nicht, dann gibt `PQerrorMessage` Auskunft, warum nicht.) Das erfolgreiche Abschicken gibt allerdings keine Gewähr dafür, dass die Anfrage irgendeine Auswirkung haben wird. Ungeachtet des Rückgabewerts von `PQrequestCancel` muss die Anwendung mit der normalen Folge von `PQgetResult`-Aufrufen fortfahren. Wenn der Abbruch Erfolg hatte, wird der aktuelle Befehl vorzeitig beendet und liefert ein Ergebnis mit einer Fehlermeldung. Wenn der Abbruchversuch scheitert (vielleicht weil die Bearbeitung des Befehls schon beendet war), wird im Client keine Auswirkung zu erkennen sein.

Beachten Sie, dass, wenn der aktuelle Befehl Teil eines Transaktionsblocks ist, die ganze Transaktion abgebrochen wird.

`PQrequestCancel` kann ohne Probleme von einem Signalhandler aus aufgerufen werden. Man kann die Funktion also auch in Verbindung mit dem normalen `PQexec` verwenden, wenn der Abbruchversuch in einem Signalhandler ausgelöst wird. `psql` zum Beispiel,ruft `PQrequestCancel` im Signalhandler für `SIGINT` auf und ermöglicht dadurch das interaktive Abbrechen von Befehlen, die es mit `PQexec` ausführt.

27.4 Die Fastpath-Schnittstelle

PostgreSQL bietet eine besondere Schnittstelle, Fastpath, um Funktionsaufrufe an den Server zu schicken. Dadurch wird direkter Zugriff auf Systeminterna gegeben, was ein mögliches Sicherheitsloch darstellen kann. Die meisten Benutzer werden diese Fähigkeit nicht benötigen.

Die Funktion `PQfn` fordert die Ausführung einer Serverfunktion über die Fast-Path-Schnittstelle an:

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);
               typedef struct {
```

```
                    int len;
                    int isint;
                    union {
                        int *ptr;
                        int integer;
                    } u;
                } PQArgBlock;
```

Das Argument `fnid` ist die OID der auszuführenden Funktion. `result_buf` ist der Puffer, in dem das Ergebnis abgespeichert wird. Sie müssen darin ausreichend Platz für das Ergebnis vorgesehen haben. (Es gibt keine Überprüfung!) Die Länge des tatsächlichen Ergebnisses wird in der Variablen, auf die `result_len` zeigt, abgelegt. Wenn eine 4-Byte-Ganzzahl als Ergebnis erwartet wird, setzen Sie `result_is_int` auf 1, ansonsten auf 0. (Wenn Sie `result_is_int` auf 1 setzen, wird `libpq` den Wert automatisch an die richtige Bytereihenfolge anpassen, sodass er als für die Clientmaschine passender int-Wert ankommt. Wenn `result_is_int` 0 ist, wird die vom Server kommende Bytefolge unverändert zurückgegeben.) `args` und `nargs` geben die der Funktion zu übergebenden Argumente an.

`PQfn` ergibt immer einen gültigen `PGresult`-Zeiger. Der Status des Ergebnisses sollte überprüft werden, bevor das Ergebnis verwendet wird. Es liegt in der Verantwortung der Anwendung, die `PGresult`-Struktur mit `PQclear` freizugeben, wenn sie nicht mehr benötigt wird.

27.5 Asynchrone Benachrichtigung

PostgreSQL bietet asynchrone Benachrichtigungen mit den Befehlen `LISTEN` und `NOTIFY`. Eine serverseitige Sitzung meldet ihr Interesse an einer bestimmten Benachrichtigung mit dem Befehl `LISTEN` an (und kann sich mit `UNLISTEN` wieder abmelden). Alle Sitzungen, die auf eine bestimmte Benachrichtigung warten, werden asynchron informiert, wenn ein `NOTIFY`-Befehl mit dem Namen der Benachrichtigung von irgendeiner Sitzung ausgeführt wird. Keine weiteren Informationen werden vom Sender zum Empfänger übertragen. Die Daten, die eigentlich übermittelt werden sollen, werden daher typischerweise über eine Datenbanktabelle verbreitet. Der Name der Benachrichtigung ist oft der Name der zugehörigen Tabelle, aber eine zugehörige Tabelle muss es nicht geben.

`libpq`-Anwendungen schicken die Befehle `LISTEN` und `UNLISTEN` als normale SQL-Befehle. Das Eintreffen einer `NOTIFY`-Nachricht kann danach durch Aufrufen von `PQnotifies` entdeckt werden.

Die Funktion `PQnotifies` gibt die nächste unbearbeitete Benachrichtigung aus der Liste der vom Server empfangenen Benachrichtigungen zurück. Wenn es keine anstehenden Benachrichtigungen gibt, dann gibt sie einen Null-Zeiger zurück. Wenn eine Benachrichtigung von `PQnotifies` einmal zurückgegeben wurde, wird sie als bearbeitet erachtet und aus der Liste der Benachrichtigungen entfernt.

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify {
    char *relname; /* Benachrichtigungsname */
    int be_pid; /* PID des Serverprozesses */
} PGnotify;
```

Nachdem ein von `PQnotifies` zurückgegebenes `PGnotify`-Objekt verarbeitet wurde, sollte es mit `free()` freigegeben werden, um Speicherlecks zu vermeiden.

In Beispiel 27.2 finden Sie ein Beispielprogramm, dass die Verwendung der asynchronen Benachrichtigung erläutert.

Anmerkung

Seit PostgreSQL 6.4 ist be_pid die PID des Prozesses, der die Benachrichtigung ausführte, aber in früheren Versionen war es die PID des eigenen Serverprozesses.

PQnotifies() empfängt keine Daten vom Server, sondern liefert nur Benachrichtigungen, die vorher von einer anderen libpq-Funktion empfangen wurden. In früheren Versionen vom libpq war die einzige Möglichkeit, den rechtzeitigen Empfang einer Benachrichtung sicherzustellen, andauernd Befehle abzuschicken, selbst leere, und nach jedem PQexec()-Aufruf mit PQnotifies() nachzuschauen. Das funktioniert immer noch, ist aber nicht mehr empfohlen, weil es Rechnerleistung verschwendet.

Eine bessere Möglichkeit, um nach NOTIFY-Nachrichten Ausschau zu halten, wenn Sie keine sinnvollen Befehl zum Ausführen haben, ist PQconsumeInput() aufzurufen und dann PQnotifies() zu prüfen. Sie können select() verwenden, um auf Daten vom Server zu warten, und verbrauchen dabei keine Rechenleistung, wenn nichts zu tun ist. (Den Dateideskriptor für select() erhalten Sie von PQsocket().) Das funktioniert, wenn Sie Befehle mit PQsendQuery/PQgetResult verschicken, und auch, wenn Sie einfach PQexec verwenden. Sie sollten sich aber merken, nach jedem PQgetResult oder PQexec PQnotifies() aufzurufen, um Benachrichtigungen zu erhalten, die während der Verarbeitung des Befehls hereinkamen.

27.6 Funktionen für den COPY-Befehl

Der COPY-Befehl in PostgreSQL hat die Möglichkeit, aus der von libpq erstellten Netzwerkverbindung zu lesen bzw. in sie zu schreiben. Daher gibt es Funktionen, die auf die Netzwerkverbindung direkt zugreifen können, um von dieser Möglichkeit Gebrauch zu machen.

Diese Funktionen sollten nur ausgeführt werden, nachdem von PQexec oder PQgetResult ein Ergebnisstatus PGRES_COPY_OUT oder PGRES_COPY_IN erhalten wurde.

PQgetline

Liest eine vom Server geschickte Zeile, mit Newline abgeschlossen, in den Puffer der Größe length.

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

Diese Funktion kopiert bis zu length-1 Zeichen in den Puffer und wandelt die abschließende Newline in ein Null-Byte um. Der Rückgabewert von PQgetline ist *EOF* am Ende des Eingabestroms, 0, wenn die ganze Zeile gelesen wurde, und 1, wenn der Puffer voll ist, aber das Zeilenende noch nicht gelesen wurde.

Beachten Sie, dass die Anwendung überprüfen muss, ob eine neue Zeile aus den zwei Zeichen \. besteht, was bedeutet, dass der Server mit der Übertragung des Ergebnisses des COPY-Befehls fertig ist. Wenn eine Anwendung Zeilen empfangen könnte, die länger als length-1 Zeichen sind, dann muss besondere Sorgfalt angewendet werden, um sicherzustellen, dass sie die Zeile mit \. richtig erkennt (und sie zum Beispiel nicht mit dem Ende einer langen Datenzeile verwechselt). Der Code in der Datei src/bin/psql/copy.c enthält Funktionen, die als Beispiel einer korrekten Behandlung des COPY-Protokolls dienen können.

PQgetlineAsync

Liest eine vom Server geschickte Zeile, mit Newline abgeschlossen, in einen Puffer, ohne zu blockieren.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int length);
```

Diese Funktion ist der Funktion `PQgetline` ähnlich, aber sie kann von Anwendungen verwendet werden, die `COPY`-Daten asynchron, das heißt ohne zu blockieren, lesen müssen. Nachdem der Befehl `COPY` ausgeführt und das Ergebnis `PGRES_COPY_OUT` erhalten wurde, sollte die Anwendung `PQconsumeInput` und `PQgetlineAsync` aufrufen, bis das Ende der Daten signalisiert wird.

Im Gegensatz zu `PQgetline` entdeckt diese Funktion das Ende der Daten selbst. Bei jedem Aufruf gibt `PQgetlineAsync` Daten zurück, wenn eine vollständige Datenzeile mit Newline im Zwischenspeicher von `libpq` vorhanden ist, oder wenn die Datenzeile zu lang ist, um in den angebotenen Puffer zu passen. Ansonsten werden keine Daten zurückgegeben, bis der Rest der Zeile angekommen ist. Die Funktion ergibt -1, wenn die Markierung am Datenende erkannt wurde, oder 0, wenn keine Daten verfügbar sind oder eine positive Zahl, die die Anzahl der Bytes in den zurückgegebenen Daten angibt. Wenn das Ergebnis -1 war, muss als nächstes `PQendcopy` aufgerufen werden und dann kann normal fortgefahren werden.

Die zurückgegebenen Daten enden spätestens mit dem Newline-Zeichen. Wenn möglich, wird die ganze Zeile auf einmal zurückgegeben. Wenn der angebotene Puffer zu klein ist, um die vom Server geschickte Zeile aufzunehmen, wird ein Teil der Zeile zurückgegeben. Das kann erkannt werden, indem man prüft, ob das letzte Byte im Ergebnis `\n` ist. Die Ergebniszeichenkette hat kein Null-Byte am Ende. (Wenn Sie eins anfügen wollen, achten Sie darauf, dass der Parameter `length`, den Sie übergeben, um eins kleiner ist als der Platz, den Sie tatsächlich haben.)

`PQputline`

Sendet eine Zeichenkette, mit Null-Byte abgeschlossen, an den Server. Ergibt 0, wenn alles in Ordnung ist, und *EOF*, wenn ein Fehler auftrat.

```
int PQputline(PGconn *conn,
              const char *string);
```

Beachten Sie, dass Anwendungen die zwei Zeichen `\.` in der letzten Zeile senden müssen, um anzuzeigen, dass sie mit dem Senden von Daten fertig sind.

`PQputnbytes`

Sendet eine Zeichenkette, nicht mit Null-Byte abgeschlossen, an den Server. Ergibt 0, wenn alles in Ordnung ist, und *EOF* ,wenn ein Fehler auftrat.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

Das ist das Gleiche wie `PQputline`, außer dass der Puffer nicht durch ein Null-Byte abgeschlossen sein muss, da die Anzahl der zu sendenden Bytes direkt angegeben wird.

`PQendcopy`

Synchronisiert den Client mit dem Server.

```
int PQendcopy(PGconn *conn);
```

Diese Funktion wartet, bis der Server den COPY-Vorgang abgeschlossen hat. Sie sollte aufgerufen werden, nachdem die letzte Zeichenkette mit PQputline an den Server geschickt wurde bzw. nachdem die letzten Daten vom Server mit PGgetline empfangen wurden. Sie muss ausgeführt werden, ansonsten verlieren Client und Server die Synchronisation. Nach der Ausführung dieses Befehls ist der Server bereit, den nächsten SQL-Befehl zu empfangen. Der Rückgabewert ist 0 bei Erfolg und ansonsten verschieden von null.

Wenn PQgetResult verwendet wird, sollte eine Anwendung auf ein PGRES_COPY_OUT-Ergebnis reagieren, indem sie PQgetline wiederholt ausführt, bis die Abschlusszeile erkannt wird und danach PQendcopy aufruft. Danach sollte sie zur PQgetResult-Schleife zurückkehren, bis PQgetResult einen Null-Zeiger ergibt. Ähnlich wird ein PGRES_COPY_IN-Ergebnis verarbeitet, indem man eine Reihe von PQputline-Aufrufen, gefolgt von PQendcopy, tätigt und dann zur PQgetResult-Schleife zurückkehrt. Dadurch wird sichergestellt, dass ein COPY-Befehl, der mitten in einer Reihe von SQL-Befehlen steht, ordnungsgemäß ausgeführt wird.

Ältere Anwendungen schicken den COPY-Befehl wahrscheinlich mit PQexec ab und gehen davon aus, dass die Transaktion nach PQendcopy vorbei ist. Das funktioniert aber nur, wenn COPY der einzige SQL-Befehl in der Befehlszeichenkette ist.

Ein Beispiel:

```
PQexec(conn, "CREATE TABLE foo (a integer, b varchar(16), d double precision);");
PQexec(conn, "COPY foo FROM STDIN;");
PQputline(conn, "3\tHallo, Welt\t4.5\n");
PQputline(conn, "4\tTschüss, Welt\t7.11\n");
...
PQputline(conn, "\\.\n");
PQendcopy(conn);
```

27.7 Funktionen zur Nachverfolgung

```
PQtrace
```

Schaltet die Nachverfolgung (englisch *tracing*) des Client/Server-Kommunikationsverkehrs in einen Datenstrom ein.

```
void PQtrace(PGconn *conn
             FILE *stream);
```

```
PQuntrace
```

Schaltet die durch PQtrace eingeschaltete Nachverfolgung wieder aus.

```
void PQuntrace(PGconn *conn);
```

27.8 Verarbeitung von Hinweismeldungen

Die Funktion PQsetNoticeProcessor kontrolliert, wie vom Server erzeugte Hinweis- und Warnmeldungen verarbeitet werden.

```
typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

In der Voreinstellung gibt `libpq` Hinweismeldungen vom Server sowie einige Fehlermeldungen, die es selbst erzeugt, auf `stderr` aus. Dieses Verhalten kann geändert werden, indem eine Rückruffunktion eingetragen wird, die mit den Meldungen etwas anderes macht. Diese Funktion erhält den Text der Hinweismeldung (welcher ein abschließendes Newline-Zeichen enthält) sowie einen typlosen Zeiger, der der gleiche ist, wie der, der `PQsetNoticeProcessor` übergeben wurde. (Dieser Zeiger kann, wenn nötig, verwendet werden, um auf anwendungsspezifische Daten zuzugreifen.) Die voreingestellte Rückruffunktion ist einfach

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

Um eine selbstgeschriebene Funktion zur Verarbeitung von Hinweismeldungen zu verwenden, rufen Sie `PQsetNoticeProcessor` gleich nach der Erzeugung eines neuen `PGconn`-Objekts auf.

Der Rückgabewert ist der Zeiger auf die vorher registrierte Rückruffunktion. Wenn der übergebene Zeiger ein Null-Zeiger ist, wird nichts gemacht, aber der Zeiger auf die aktuelle Funktion wird zurückgegeben.

Wenn Sie eine Funktion zur Verarbeitung von Hinweismeldungen eingesetzt haben, müssen Sie davon ausgehen, dass sie aufgerufen werden kann, solange das `PGconn`-Objekt oder daraus erzeugte `PGresult`-Objekte existieren. Bei der Erzeugung eines `PGresult`-Objekts wird die aktuelle Rückruffunktion des `PGconn`-Objekts kopiert, um Funktionen wie `PQgetvalue` zur Verfügung zu stehen.

27.9 Umgebungsvariablen

Die folgenden Umgebungsvariablen können verwendet werden, um Vorgabewerte für Verbindungsparameter zu setzen, welche von `PQconnectdb` oder `PQsetdbLogin` verwendet werden, wenn kein anderer Wert im Funktionsaufruf angegeben wird. Sie sind zum Beispiel nützlich, um zu vermeiden, dass in einfachen Anwendungen der Datenbankname direkt in den Code geschrieben werden muss.

- `PGHOST` setzt den vorgegebenen Servernamen. Wenn er mit einem Schrägstrich anfängt, dann wird eine Unix-Domain-Socket statt TCP/IP verwendet; der Wert ist das Verzeichnis, in dem sich die Socketdatei befindet (Vorgabe `/tmp`).
- `PGPORT` setzt die vorgegebene TCP-Portnummer bzw. die Erweiterung des Unix-Domain-Socket-Namen für die Kommunikation mit dem PostgreSQL-Server.
- `PGDATABASE` setzt den vorgegebenen PostgreSQL-Datenbanknamen.
- `PGUSER` setzt den vorgegebenen Benutzernamen für Datenbankverbindungen.
- `PGPASSWORD` setzt das Passwort, falls der Server Authentifizierung mit Passwörtern fordert. Diese Umgebungsvariable wird aus Sicherheitsgründen nicht empfohlen; verwenden Sie stattdessen die Datei `$HOME/.pgpass` (siehe Abschnitt 27.10).
- `PGREALM` setzt das Kerberos-Realm für PostgreSQL, wenn es nicht das lokale Realm ist. Wenn `PGREALM` gesetzt ist, versuchen `libpq`-Anwendungen die Authentifizierung mit Servern dieses Realms

und verwenden getrennte Ticketdateien, um Konflikte mit lokalen Ticketdateien zu vermeiden. Diese Umgebungsvariable wird nur verwendet, wenn der Server Authentifizierung mit Kerberos fordert.

- ❏ `PGOPTIONS` setzt zusätzliche Konfigurationsoptionen, die an den PostgreSQL-Server geschickt werden.
- ❏ `PGTTY` setzt die Datei oder das TTY für Debug-Ausgaben vom Server.

Die folgenden Umgebungsvariablen können verwendet werden, um Vorgabewerte für jede PostgreSQL-Sitzung einzustellen.

- ❏ `PGDATESTYLE` setzt den vorgegebenen Ausgabestil für Datums- und Zeitangaben. (Entspricht `SET datestyle TO ...`.)
- ❏ `PGTZ` setzt die Zeitzone. (Entspricht `SET timezone TO ...`.)
- ❏ `PGCLIENTENCODING` setzt die vorgegebene Zeichensatzkodierung im Client. (Entspricht `SET client_encoding TO ...`.)
- ❏ `PGGEQO` setzt den Vorgabemodus des genetischen Anfrageoptimierers. (Entspricht `SET geqo TO ...`.)

Informationen über die korrekten Werte für diese Umgebungsvariablen finden Sie beim SQL-Befehl `SET`.

27.10 Die Passwortdatei

Die Datei `.pgpass` im jeweiligen Home-Verzeichnis ist eine Datei, die Passwörter enthält, die verwendet werden, wenn die Verbindung ein Passwort erfordert (und anderweitig kein Passwort angegeben wurde). Die Datei sollte Zeilen mit folgendem Format haben:

```
hostname:port:datenbank:benutzername:passwort
```

Jedes dieser Felder kann eine Konstante sein oder `*`, was auf alles passt. Die erste Übereinstimmung wird verwendet, also stellen Sie spezifischere Einträge an den Anfang. Einträge mit `:` oder \ müssen ein vorangestelltes Fluchtzeichen \ haben.

Die Zugriffsrechte für `.pgpass` müssen Gruppen- und Weltzugriff verbieten. Das kann mit dem Befehl `chmod 0600 .pgpass` erreicht werden. Wenn die Zugriffsrechte weniger strikt sind, wird die Datei ignoriert.

27.11 Thread-Verhalten

Seit PostgreSQL 7.0 ist `libpq` Thread-sicher, solange nicht zwei Threads dasselbe `PGconn`-Objekt zu gleichen Zeit bearbeiten. Insbesondere kann man nicht von zwei Threads gleichzeitig Befehle durch dasselbe Verbindungsobjekt schicken. (Wenn Sie Befehle gleichzeitig ausführen müssen, öffnen sie mehrere Verbindungen.)

`PGresult`-Objekte werden nach der Erstellung nicht mehr verändert und können daher beliebig zwischen Threads umhergereicht werden.

Die nicht mehr empfohlenen Funktionen `PQoidStatus` und `fe_setauthsvc` sind nicht Thread-sicher und sollten nicht in Multithread-Programmen verwendet werden. `PQoidStatus` kann durch `PQoidValue` ersetzt werden. Für `fe_setauthsvc` gibt es überhaupt keinen guten Grund, es aufzurufen.

`libpq`-Anwendungen, die die Authentifizierungsmethode `crypt` verwenden, verlassen sich auf die Betriebssystemfunktion `crypt()`, welche oft nicht Thread-sicher ist. Es ist besser, die Methode `md5` zu verwenden, welche auf allen Plattformen Thread-sicher ist.

27.12 libpq-Programme bauen

Um Ihre `libpq`-Programme zu bauen (d.h. compilieren und linken), müssen Sie sämtliche hier genannte Schritte befolgen:

- Binden Sie die Headerdatei `libpq-fe.h` ein:

```
#include <libpq-fe.h>
```

  Wenn Sie das versäumt haben, erhalten Sie von Ihrem Compiler normalerweise Fehlermeldungen ähnlich dieser:

```
foo.c: In function `main':
foo.c:34: `PGconn' undeclared (first use in this function)
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
```

- Zeigen Sie Ihrem Compiler, in welchem Verzeichnis die PostgreSQL-Headerdateien installiert sind, indem Sie bei Ihrem Compiler die Option `-I`*`verzeichnis`* angeben. (In einigen Fällen wird der Compiler im fraglichen Verzeichnis automatisch suchen, sodass Sie diese Option dann auslassen können.) Die Befehlszeile zum Compilieren könnte zum Beispiel so aussehen:

```
cc -c -I/usr/local/pgsql/include testprog.c
```

  Wenn Sie `make`-Steuerdateien (*makefiles*) verwenden, fügen Sie die Option an die Variable `CPPFLAGS` an:

```
CPPFLAGS += -I/usr/local/pgsql/include
```

  Wenn die Möglichkeit besteht, dass Ihr Programm auch von anderen Benutzern compiliert werden könnte, sollten Sie die Verzeichnisnamen nicht auf diese Art festschreiben. Sie können stattdessen das Programm `pg_config` verwenden, um herauszufinden, wo die Headerdateien auf dem lokalen System sind:

```
$ pg_config --includedir
/usr/local/include
```

  Wenn Sie nicht die richtige Compileroption angegeben haben, erhalten Sie eine Fehlermeldung wie diese:

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- Wenn Sie das fertige Programm linken, geben Sie die Option `-lpq` an, um die `libpq`-Bibliothek einzubinden, sowie die Option `-L`*`verzeichnis`*, um dem Compiler das Verzeichnis zu zeigen, in dem die `libpq`-Bibliothek abgelegt ist. (Wiederum wird der Compiler einige Verzeichnisse automatisch durchsuchen.) Für maximale Portierbarkeit stellen Sie die `-L`-Option vor die Option `-lpq`. Zum Beispiel:

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
```

Das Bibliotheksverzeichnis können Sie auch mit `pg_config` herausfinden:

```
$ pg_config --libdir
/usr/local/pgsql/lib
```

Fehlermeldungen, die auf Probleme in diesem Bereich hinweisen, könnten folgendermaßen aussehen.

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

Dies bedeutet, dass Sie `-lpq` vergessen haben.

```
/usr/bin/ld: cannot find -lpq
```

Dies bedeutet, dass Sie die `-L`-Option vergessen oder nicht das richtige Verzeichnis angegeben haben.

Wenn Ihr Code die Headerdatei `libpq-int.h` verwendet und Sie sich weigern, Ihren Code zu ändern und sie nicht benutzen, dann befindet sich diese Datei seit PostgreSQL 7.2 in *`includedir`*`/postgresql/internal/libpq-int.h`. Sie müssen also die entsprechende `-I`-Option zur Compilerbefehlszeile hinzufügen.

27.13 Beispielprogramme

Beispiel 27.1: libpq-Beispielprogramm 1

```
/*
 * testlibpq.c
 *
 * Test the C version of libpq, the PostgreSQL frontend
 * library.
 */
#include <stdio.h>
#include <libpq-fe.h>

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
```

```
    char       *dbName;
    int         nFields;
    int         i,
                j;

    /* FILE *debug; */

    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */
    dbName = "template1";

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
```

```
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR SELECT * FROM pg_database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);
```

```
    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    /* fclose(debug); */
    return 0;

}
```

Beispiel 27.2: libpq-Beispielprogramm 2

```
/*
 * testlibpq2.c
 *  Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 values (new.i); NOTIFY TBL2);
 *
 * and do
 *
 *   INSERT INTO TBL1 values (10);
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */
    dbName = getenv("USER");    /* change this to the name of your test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
```

```
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d' received\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Beispiel 27.3: libpq-Beispielprogramm 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the PostgreSQL frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d real, p polygon);
```

```
 *
 * INSERT INTO test1 values (1, 3.567, polygon '(3.0, 4.0, 1.0, 2.0)');
 *
 * INSERT INTO test1 values (2, 89.05, polygon '(4.0, 3.0, 2.0, 1.0)');
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo_decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
```

```
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend server */

    dbName = getenv("USER");    /* change this to the name of your test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch rows from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR SELECT * FROM test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
```

```
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum), plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
```

```
            printf(" d = (%d bytes) %f,\n",
                   PQgetlength(res, i, d_fnum), *dval);
            printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo = %f,%f)\n",
                   PQgetlength(res, i, d_fnum),
                   pval->npts,
                   pval->boundbox.xh,
                   pval->boundbox.yh,
                   pval->boundbox.xl,
                   pval->boundbox.yl);
        }
        PQclear(res);

        /* close the cursor */
        res = PQexec(conn, "CLOSE mycursor");
        PQclear(res);

        /* commit the transaction */
        res = PQexec(conn, "COMMIT");
        PQclear(res);

        /* close the connection to the database and cleanup */
        PQfinish(conn);

        return 0;
    }
```

28

Large Objects

In PostgreSQL vor Version 7.1 konnte die Größe eine Zeile in der Datenbank die Größe einer Datenseite nicht überschreiten. Da die Größe einer Datenseite 8192 Bytes ist (der Vorgabewert, welcher bis auf 32768 erhöht werden kann), war die Obergrenze für die Größe eines Datenwerts ziemlich niedrig. Um die Speicherung von größeren atomaren Werten zu unterstützen, bot PostgreSQL, und bietet immer noch, eine Large-Object-Schnittstelle. Diese Schnittstelle bietet dateiähnlichen Zugriff auf Benutzerdaten, die in einer besonderen Large-Object-Struktur gespeichert sind.

Dieses Kapitel beschreibt die Implementierung und die Programmier- und SQL-Schnittstellen für Large-Object-Daten in PostgreSQL. Wir verwenden die C-Bibliothek `libpq` für die Beispiele in diesem Kapitel, aber die meisten PostgreSQL-spezifischen Programmierschnittstellen bieten die gleiche Funktionalität. Andere Schnittstellen verwenden möglicherweise die Large-Object-Schnittstelle intern, um allgemeine Unterstützung für große Werte anzubieten. Dies wird hier nicht beschrieben.

28.1 Geschichte

POSTGRES 4.2, der mittelbare Vorgänger von PostgreSQL, bot drei eingebaute Implementierungen von Large Objects: als Dateien außerhalb des POSTGRES-Servers, als externe Dateien, die vom POSTGRES-Server verwaltet wurden, und als Daten in der POSTGRES-Datenbank. Das verursachte beträchtliche Verwirrung unter den Benutzern. Als Ergebnis gibt es in PostgreSQL nur noch Unterstützung für Large Objects als Daten in der Datenbank. Obgleich der Zugriff darauf langsamer ist, bietet es striktere Datenintegrität. Aus historischen Gründen wird dieses Speichersystem als *Inversion large object* bezeichnet. (Gelegentlich werden Sie den Begriff *Inversion* als Synonym für Large Object verwendet sehen.) Seit PostgreSQL 7.1 werden alle Large-Object-Daten in einer Systemtabelle namens `pg_largeobject` abgespeichert.

PostgreSQL 7.1 führte einen Mechanismus (mit dem Spitznamen “TOAST”) ein, der es ermöglicht, dass Datenzeilen viel größer als einzelne Datenseiten sein können. Dadurch wird die Large-Object-Schnittstelle teilweise obsolet. Ein bleibender Vorteil der Large-Object-Schnittstelle ist, dass sie Werte bis 2 GB Größe erlaubt, wohingegen TOAST nur 1 GB verarbeiten kann.

28.2 Implementierungsmerkmale

Die Large-Object-Implementierung zerlegt Large-Object-Werte in “Stücke” und speichert die Stücke in Zeilen in der Datenbank. Ein B-Tree-Index ermöglicht schnellen Zugriff auf das korrekte Stück, wenn beliebige Datenteile gelesen und geschrieben werden.

28.3 Clientschnittstellen

Dieser Abschnitt beschreibt die Schnittstellen, die von PostgreSQL-Clientbibliotheken angeboten werden, um auf Large-Object-Daten zuzugreifen. Alle Manipulationen von Large Objects mit diesen Funktionen *müssen* in einem SQL-Transaktionsblock stattfinden. (Diese Anforderung wird seit

PostgreSQL 6.5 strikt durchgesetzt, obwohl sie schon in vorherigen Versionen eine unausgesprochene Anforderung war, die bei Vernachlässigung in Fehlverhalten resultierte.) Die Large-Object-Schnittstelle in PostgreSQL wurde der Dateisystemschnittstelle in UNIX nachempfunden, mit Analoga von `open` (öffnen), `read` (lesen), `write` (schreiben), `lseek` (suchen) usw.

Clientanwendungen, die die Large-Object-Schnittstelle in `libpq` verwenden, sollten die Headerdatei `libpq/libpq-fs.h` einbinden und die `libpq`-Bibliothek einlinken.

28.3.1 Ein Large Object erzeugen

Die Funktion

```
Oid lo_creat(PGconn *conn, int mode);
```

erzeugt eine neues Large Object. Der Parameter *mode* ist eine Bitmaske, die diverse Attribute des neuen Objekts beschreibt. Die hier beschriebenen symbolischen Konstanten sind in der Headerdatei `libpq/libpq-fs.h` definiert. Der Zugriffstyp (schreiben, lesen oder beides) wird durch die Bits *INV_READ* und *INV_WRITE*, mit “oder” verknüpft, kontrolliert. Die niedrigeren 16 Bits der Maske wurden früher in Berkeley verwendet, um den Speichermanager, auf dem das Large Object abgelegt werden soll, zu identifizieren. Diese Bits sollten jetzt null sein. Der Rückgabewert ist die dem neuen Large Object zugewiesene OID.

Ein Beispiel:

```
inv_oid = lo_creat(INV_READ|INV_WRITE);
```

28.3.2 Ein Large Object importieren

Um eine Datei aus dem Betriebssystem als Large Object zu importieren, rufen Sie

```
Oid lo_import(PGconn *conn, const char *filename);
```

auf.

filename gibt den Betriebssystemnamen der Datei an, die als Large Object importiert werden soll. Der Rückgabewert ist die dem neuen Large Object zugewiesene OID.

28.3.3 Ein Large Object exportieren

Um ein Large Object in eine Betriebssystemdatei zu exportieren, rufen Sie

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

auf.

Das Argument `lobjId` ist die OID des zu exportierenden Large Object und das Argument `filename` gibt den Betriebssystemnamen der Datei an.

28.3.4 Ein bestehendes Large Object öffnen

Um ein bestehendes Large Object zu öffnen, rufen Sie

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

auf .

Das Argument `lobjId` ist die OID des zu öffnenden Large Object. Die Bits in `mode` kontrollieren, ob das Objekt zum Lesen (*INV_READ*), Schreiben (*INV_WRITE*) oder zu beidem geöffnet wird. Ein Large Object kann nicht geöffnet werden, bevor es erzeugt worden ist. `lo_open` gibt einen Large-Object-Deskriptor zurück, der später mit `lo_read`, `lo_write`, `lo_lseek`, `lo_tell` und `lo_close` verwendet werden kann.

28.3.5 Daten in ein Large Object schreiben

Die Funktion

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

schreibt `len` Bytes aus `buf` in das Large Object `fd`. Das Argument `fd` muss vorher von `lo_open` erhalten worden sein. Die Anzahl der Bytes, die tatsächlich geschrieben wurden, wird zurückgegeben. Im Falle eines Fehlers ist der Rückgabewert negativ.

28.3.6 Daten aus einem Large Object lesen

Die Funktion

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

liest `len` Bytes aus dem Large Object `fd` in `buf`. Das Argument `fd` muss vorher von `lo_open` erhalten worden sein. Die Anzahl der Bytes, die tatsächlich gelesen wurden, wird zurückgegeben. Im Falle eines Fehlers ist der Rückgabewert negativ.

28.3.7 In einem Large Object suchen

Um die aktuelle Lese- oder Schreibposition in einem Large Object zu ändern, rufen Sie

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

auf.

Diese Funktion bewegt den aktuellen Positionszeiger des von `fd` identifizierten Large Object an die neue Position, die von `offset` angegeben wird. Die gültigen Werte für `whence` sind *SEEK_SET* (vom Objektanfang zählen), *SEEK_CUR* (von der aktuellen Position zählen) und *SEEK_END* (vom Objektende zählen). Der Rückgabewert ist die neue aktuelle Position.

28.3.8 Die aktuelle Schreibposition eines Large Object ermitteln

Um die aktuelle Lese- oder Schreibposition in einem Large Object zu erhalten, rufen Sie

```
int lo_tell(PGconn *conn, int fd);
```

auf.

Bei einem Fehler ist der Rückgabewert negativ.

28.3.9 Einen Large-Object-Deskriptor schließen

Ein Large Object kann mit

```
int lo_close(PGconn *conn, int fd);
```

geschlossen werden, wobei `fd` der von `lo_open` erhaltene Large-Object-Deskriptor ist. Bei Erfolg gibt `lo_close` null zurück. Bei einem Fehler ist der Rückgabewert negativ.

28.3.10 Ein Large Object entfernen

Um ein Large Object aus der Datenbank zu entfernen, rufen Sie

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

auf .

Das Argument `lobjId` gibt die OID des zu löschenden Large Object an. Im Falle eines Fehlers ist der Rückgabewert negativ.

28.4 Serverseitige Funktionen

Für den Zugriff auf Large Objects gibt es zwei eingebaute serverseitige Funktionen, `lo_import` und `lo_export`, welche zur Verwendung in SQL-Befehlen gedacht sind. Hier ist ein Beispiel, wie sie verwendet werden:

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));
```

```
SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

28.5 Beispielprogramm

Beispiel 28.1 ist ein Beispielprogramm, das zeigt, wie die Large-Object-Schnittstelle in `libpq` verwendet werden kann. Teile des Programms sind auskommentiert, aber im Quellcode belassen um dem Leser zu helfen. Dieses Programm kann auch in `src/test/examples/testlo.c` in der Quelltext-Distribution gefunden werden.

Beispiel 28.1: Beispielprogramm für Large Objects mit libpq

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
```

```
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "can't create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }
```

```
    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
```

```
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    export large object "lobjOid" to file "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "can't open large object %d\n",
                lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "can't open unix file %s\n",
                filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
```

```
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
```

```
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

29

pgtcl: Die Tcl-Bindungsbibliothek

`pgtcl` ist ein Tcl-Paket, mit dem Clientprogramme mit dem PostgreSQL-Server kommunizieren können. Es stellt die meisten Funktionen von `libpq` für Tcl-Skripts zur Verfügung.

29.1 Überblick

Tabelle 29.1 gibt einen Überblick über die in `pgtcl` verfügbaren Befehle. Diese Befehle sind im Einzelnen auf den folgenden Seiten beschrieben.

Befehl	Beschreibung
`pg_connect`	eine Verbindung zum Server öffnen
`pg_disconnect`	eine Verbindung zum Server schließen
`pg_conndefaults`	Verbindungsparameter und ihre Vorgabewerte ermitteln
`pg_exec`	einen Befehl an den Server senden
`pg_result`	Informationen über ein Befehlsergebnis ermitteln
`pg_select`	eine Schleife über ein Anfrageergebnis ausführen
`pg_execute`	eine Anfrage senden und wahlweise eine Schleife über das Ergebnis ausführen
`pg_listen`	einen Befehl für asynchrone Benachrichtigungsmitteilungen einsetzen oder ändern
`pg_on_connection_loss`	einen Befehl für unerwarteten Verbindungsverlust einsetzen oder ändern
`pg_lo_creat`	ein Large Object erzeugen
`pg_lo_open`	ein Large Object öffnen
`pg_lo_close`	ein Large Object schließen
`pg_lo_read`	aus einem Large Object lesen
`pg_lo_write`	in ein Large Object schreiben
`pg_lo_lseek`	den Positionszeiger eines Large Object setzen

Tabelle 29.1: `pgtcl`-Befehle

Befehl	Beschreibung
pg_lo_tell	den aktuellen Positionszeiger eines Large Object zurückgeben
pg_lo_unlink	ein Large Object löschen
pg_lo_import	ein Large Object aus einer Datei importieren
pg_lo_export	ein Large Object in eine Datei exportieren

Tabelle 29.1: pgtcl-Befehle (Forts.)

Die pg_lo_*-Befehle sind Schnittstellen zur Large-Object-Funktionalität PostgreSQL. Diese Funktionen wurden den analogen Funktionen in der Unix-Dateisystemschnittstelle nachempfunden. Die pg_lo_*-Befehle sollten innerhalb eines BEGIN/COMMIT-Transaktionsblocks verwendet werden, weil die von pg_lo_open zurückgegebenen Deskriptoren nur in der aktuellen Transaktion gültig sind. Die Befehle pg_lo_import und pg_lo_export *müssen* in einem BEGIN/COMMIT-Transaktionsblock verwendet werden.

29.2 pgtcl in eine Anwendung laden

Bevor Sie pgtcl-Befehle verwenden können, müssen Sie libpgtcl in Ihre Tcl-Anwendung laden. Das wird normalerweise mit dem Tcl-Befehl load gemacht. Hier ist ein Beispiel:

```
load libpgtcl[info sharedlibextension]
```

Es wird empfohlen, info sharedlibextension zu verwenden, anstatt .so oder .sl direkt in die Anwendung zu schreiben.

Der Befehl load wird fehlschlagen, wenn das dynamische Ladeprogramm des System nicht weiß, wo es nach der dynamischen Bibliothek libpgtcl suchen soll. Sie müssen womöglich mit ldconfig arbeiten, die Umgebungsvariable LD_LIBRARY_PATH setzen oder eine ähnliche Methode auf Ihrer Plattform finden. Sehen Sie sich die PostgreSQL-Installationsanweisungen an, um weitere Informationen zu erhalten.

libpgtcl hängt wiederum von libpq ab, also muss das Ladeprogramm auch die dynamische Bibliothek libpq finden können. In der Praxis ist das selten ein Problem, da beide diese Bibliotheken normalerweise im selben Verzeichnis abgelegt sind, aber in einigen Konfigurationen könnte dies eine Hürde sein.

Wenn Sie eine eigene Programmdatei für Ihre Anwendung bauen, könnten Sie libpgtcl auch statisch in die Programmdatei einbinden und so den load-Befehl und die ganzen möglichen Probleme beim dynamischen Linken vermeiden. Schauen Sie sich den Quellcode von pgtclsh als Beispiel an.

29.3 pgtcl-Befehlsreferenz

pg_connect

Name

pg_connect -- eine Verbindung zum Server öffnen

Synopsis

```
pg_connect -conninfo verbindungsparameter
pg_connect dbName ?-host hostname? ?-port portnummer? ?-tty tty? ?-options
serveroptionen?
```

Beschreibung

`pg_connect` öffnet eine Verbindung zum PostgreSQL-Server.

Zwei Syntaxen stehen zur Verfügung. In der älteren hat jeder mögliche Parameter einen getrennten Optionsschalter im `pg_connect`-Befehl. In der neueren Form wird eine einzige Zeichenkette angegeben, die mehrere Parameterwerte enthalten kann. `pg_conndefaults` kann verwendet werden, um Informationen über die in der neueren Syntax verfügbaren Optionen zu erhalten.

Argumente

Neuer Stil

`verbindungsparameter`

Eine Zeichenkette mit Verbindungsparametern, jeder in der Form `schlüssel = wert` geschrieben. Eine Liste der gültigen Parameter kann in der Beschreibung der `libpq`-Funktion `PQconnectdb` gefunden werden.

Alter Stil

`dbName`

Der Name der Datenbank, mit der verbunden werden soll.

`-host hostname`

Der Hostname des Datenbankservers, mit dem verbunden werden soll.

`-port portnummer`

Die TCP-Portnummer des Datenbankservers, mit dem verbunden werden soll.

`-tty tty`

Eine Datei oder ein TTY für zusätzliche Debugausgaben vom Server.

`-options serveroptionen`

Zusätzliche, an den Server zu übergebende Konfigurationsoptionen.

Rückgabewert

Wenn erfolgreich, wird eine Handle für die Datenbankverbindung zurückgegeben. Handles haben den Präfix `pgsql`.

pg_disconnect

Name

pg_disconnect -- eine Verbindung zum Server schließen

Synopsis

```
pg_disconnect verbindung
```

Beschreibung

`pg_disconnect` schließt eine Verbindung zum PostgreSQL-Server.

Argumente

`verbindung`

Die Handle der zu schließenden Verbindung.

Rückgabewert

Keiner

pg_conndefaults

Name

pg_conndefaults -- Verbindungsparameter und ihre Vorgabewerte ermitteln

Synopsis

```
pg_conndefaults
```

Beschreibung

`pg_conndefaults` gibt Informationen über die für `pg_connect -conninfo` zur Verfügung stehenden Verbindungsparameter und deren aktuelle Vorgabewerte zurück.

Argumente

Keine

Rückgabewert

Das Ergebnis ist eine Liste, die die möglichen Verbindungsparameter und deren aktuelle Vorgabewerte beschreibt. Jeder Eintrag ist wiederum eine Liste im folgenden Format:

```
{name beschriftung ausgzeichen ausggröße wert}
```

wobei *name* als Parameter in `pg_connect -conninfo` verwendet werden kann.

pg_exec

Name

pg_exec -- einen Befehl an den Server senden

Synopsis

```
pg_exec verbindung befehlstext
```

Beschreibung

`pg_exec` schickt einen Befehl an den PostgreSQL-Server und gibt ein Ergebnis zurück. Die Handles für Befehlsergebnisse bestehen aus der Verbindungshandle gefolgt von einem Punkt und der Ergebnisnummer.

Beachten Sie, dass das Wegbleiben eines Tcl-Fehlers nicht anzeigt, dass der SQL-Befehl erfolgreich war. Eine vom Server zurückgegebene Fehlermeldung ergibt ein Ergebnisobjekt mit einem Fehlerstatus, aber keinen Tcl-Fehler in `pg_exec`.

Argumente

`verbindung`

Die Handle der Verbindung, auf der der Befehl ausgeführt werden soll.

`befehlstext`

Der auszuführende SQL-Befehl.

Rückgabewert

Eine Ergebnishandle. Ein Tcl-Fehler wird erzeugt, wenn `pgtcl` keine Antwort vom Server erhalten konnte. Ansonsten wird ein Ergebnisobjekt erzeugt und eine Handle dafür zurückgegeben. Diese Handle kann an `pg_result` übergeben werden, um die Ergebnisse des Befehls zu ermitteln.

pg_result

Name

pg_result -- Informationen über ein Befehlsergebnis ermitteln

Synopsis

```
pg_result ergebnishandle ergebnisoption
```

Beschreibung

`pg_result` gibt Informationen aus einem Befehlsergebnisobjekt zurück, welches vorher durch `pg_exec` erzeugt wurde.

Sie können das Befehlsergebnis behalten, solange Sie wollen, aber wenn Sie damit fertig sind, sollten Sie es mit `pg_result -clear` freigeben. Ansonsten haben Sie ein Speicherleck und `pgtcl` wird sich irgendwann beschweren, dass Sie zu viele Ergebnisobjekte erzeugt haben.

Argumente

`ergebnishandle`

Die Handle des Befehlsergebnisses.

`ergebnisoption`

Eine der folgenden Optionen, die angibt, welcher Teil der Ergebnisinformationen zurückgegeben werden soll:

`-status`

Der Status des Ergebnisses.

`-error`

Die Fehlermeldung, wenn der Status einen Fehler anzeigt, ansonsten eine leere Zeichenkette.

`-conn`

Die Verbindung, auf der dieses Ergebnis erzeugt wurde.

`-oid`

Wenn der Befehl ein `INSERT` war, dann die OID der eingefügten Zeile, ansonsten 0.

`-numTuples`

Die Anzahl der von der Anfrage zurückgegebenen Zeilen (Tupel).

`-numAttrs`

Die Anzahl der Spalten (Attribute) in jeder Zeile.

`-assign` *`arrayname`*

Legt die Ergebnisse in einem Array ab, mit Indizes der Form (zeilennummer, spaltennummer).

-assignbyidx *arrayname [anhang]*

Legt die Ergebnisse in einem Array ab, mit den Werten der ersten Spalte und den Namen der übrigen Spalten als Indizes. Wenn anhang angegeben wird, dann wird es an jeden Indexwert angehängt. Kurz gesagt, werden alle Spalten außer der ersten in jeder Zeile im Array gespeichert, mit Indizes der Form (ersterSpaltenwert, spaltennameAnhang).

-getTuple *zeilennummer*

Gibt die Spalten der angegebenen Zeile als Liste zurück. Zeilennummern beginnen bei null.

-tupleArray *zeilennummer arrayname*

Speichert die Spalten der Zeile im Array arrayName, indiziert nach Spaltennamen. Zeilennummern beginnen bei null.

-attributes

Gibt eine Liste der Namen der Spalten im Ergebnis zurück.

-lAttributes

Gibt eine Liste von Listen der Form {name typoid typgröße} für jede Spalte zurück.

-clear

Gibt das Ergebnisobjekt wieder frei.

Rückgabewert

Das Ergebnis hängt von der gewählten Option ab, wie oben beschrieben.

pg_select

Name

pg_select – eine Schleife über ein Anfrageergebnis ausführen

Synopsis

```
pg_select verbindung befehlstext arrayvar prozedur
```

Beschreibung

lpg_select schickt eine Anfrage (SELECT-Befehl) an den PostgreSQL-Server und führt für jede Zeile im Ergebnis ein angegebenes Stück Code aus. Der befehlstext muss ein SELECT-Befehl sein, alles andere ist ein Fehler. Die Variable arrayvar ist ein Arrayname, der in der Schleife verwendet wird. Für jede Zeile wird arrayvar mit den Spaltenwerten gefüllt, mit den Spaltennamen als Arrayindizes. Dann wird die prozedur ausgeführt.

Zusätzlich zu den Spaltenwerten werden die folgenden besonderen Einträge in das Array eingefügt:

`.headers`

Eine Liste der von der Anfrage zurückgegebenen Spaltennamen.

`.numcols`

Die Anzahl der von der Anfrage zurückgegebenen Spalten.

`.tupno`

Die aktuelle Zeilennummer, beginnend bei null und bei jedem Schleifendurchlauf um eins erhöht.

Argumente

`verbindung`

Die Handle der Verbindung, auf der die Anfrage ausgeführt werden soll.

`befehlstext`

Die auszuführende SQL-Anfrage.

`arrayvar`

Eine Arrayvariable für die Ergebniszeilen.

`prozedur`

Eine Prozedur, die für jede Ergebniszeile ausgeführt wird.

Rückgabewert

Keiner

Beispiele

Dieses Beispiel geht davon aus, dass die Tabelle `tabelle1` Spalten namens `nummer` und `name` (und möglicherweise weitere) hat:

```
pg_select $pgconn "SELECT * FROM tabelle1;" array {
    puts [format "%5d %s" $array(nummer) $array(name)]
}
```

pg_execute

Name

pg_execute -- eine Anfrage senden und wahlweise eine Schleife über das Ergebnis ausführen

Synopsis

```
pg_execute ?-array arrayvar? ?-oid oidvar? verbindung befehlstext ?prozedur?
```

Beschreibung

`pg_execute` schickt einen Befehl an den PostgreSQL-Server.

Wenn der Befehl kein `SELECT` ist, wird die Anzahl der von dem Befehl betroffenen Zeilen zurückgegeben. Wenn der Befehl ein `INSERT` ist und eine einzige Zeile eingefügt wird, dann wird die OID der eingefügten Zeile in der Variable *`oidvar`* gespeichert, wenn die wahlfreie Option *`-oid`* angegeben wurde.

Wenn der Befehl ein `SELECT` ist, werden, für jede Zeile im Ergebnis, die Zeilenwerte in der Variable *`arrayvar`* gespeichert, wenn angegeben, mit den Spaltennamen als Arrayindizes, ansonsten in nach den Spalten benannten Variablen, und dann wird, wenn angegeben, die *`prozedur`* ausgeführt. (*`prozedur`* wegzulassen ist wahrscheinlich nur sinnvoll, wenn die Anfrage nur eine einzige Zeile ergibt.) Die Anzahl der Zeilen, die die Anfrage liefert, wird zurückgegeben.

Die *`prozedur`* kann die Tcl-Befehle `break`, `continue` und `return` mit dem zu erwartenden Verhalten verwenden. Beachten Sie, dass, wenn *`prozedur`* `return` ausführt, `pg_execute` nicht die Anzahl der betroffenen Zeilen zurückgibt.

`pg_execute` ist eine neuere Funktion, die mehr Fähigkeiten als `pg_select` bietet und in vielen Fällen `pg_exec` ersetzen kann, wenn kein Zugriff auf die Ergebnishandle benötigt wird.

Für Fehler vom Server wird `pg_execute` einen Tcl-Fehler erzeugen und eine Liste mit zwei Elementen zurückgeben. Das erste Element ist ein Fehlercode, wie zum Beispiel `PGRES_FATAL_ERROR`, und das zweite Element ist der Fehlertext vom Server. Bei ernsthafteren Fehlern, wie zum Beispiel einen Fehler bei der Kommunikation mit dem Server, wird `pg_execute` einen Tcl-Fehler erzeugen und nur den Fehlertext zurückgeben.

Argumente

`-array arrayvar`

Gibt den Namen einer Arrayvariablen an, wo die Ergebniszeilen abgelegt werden, indiziert nach Spaltennamen. Wenn *`befehlstext`* kein `SELECT` ist, dann wird diese Option ignoriert.

`-oid oidvar`

Gibt den Namen einer Variable an, in die die OID aus einem `INSERT`-Befehl gespeichert werden wird.

`verbindung`

Die Handle der Verbindung, auf der der Befehl ausgeführt werden soll.

`befehlstext`

Der auszuführende SQL-Befehl.

`prozedur`

Eine Prozedur, die für jede Ergebniszeile eines `SELECT`-Befehls ausgeführt wird.

Rückgabewert

Die Anzahl der Zeilen, die von dem Befehl betroffen oder zurückgegeben wurden.

Beispiele

In den folgenden Beispielen wurde die Fehlerprüfung mit `catch` weggelassen, um die Klarheit zu bewahren.

Füge eine Zeile ein und speichere die OID in `result_oid`:

```
pg_execute -oid result_oid $pgconn "INSERT INTO meinetabelle VALUES (1);"
```

Gebe die Spalten `nummer` und `wert` aus jeder Zeile aus:

```
pg_execute -array d $pgconn "SELECT nummer, wert FROM meinetabelle;" {
    puts "Nummer=$d(nummer) Wert=$d(wert)"
}
```

Finde die Maximal- und Minimalwerte und speichere sie in `$s(max)` und `$s(min)`:

```
pg_execute -array s $pgconn "SELECT max(value) AS max, min(value) AS min FROM
meinetabelle;"
```

Finde die Maximal- und Minimalwerte und speichere sie in `$max` und `$min`:

```
pg_execute $pgconn "SELECT max(value) AS max, min(value) AS min FROM
meinetabelle;"
```

pg_listen

Name

pg_listen -- einen Befehl für asynchrone Benachrichtigungsmitteilungen einsetzen oder ändern

Synopsis

```
pg_listen verbindung notifyName ?befehl?
```

Beschreibung

`pg_listen` setzt einen Befehl ein, der bei Ankunft einer asynchronen Benachrichtigungsmitteilung vom PostgreSQL-Server aufgerufen werden soll. Mit dem Parameter `befehl` wird ein Befehl eingesetzt oder der Text eines bestehenden Befehls ersetzt. Ohne den Parameter `befehl` wird ein schon registrierter Befehl widerrufen.

Nachdem ein Befehl mit `pg_listen` registriert worden ist, wird der angegebene Befehlstext immer ausgeführt, wenn eine Benachrichtigungsmitteilung mit dem angegebenen Namen vom Server eintrifft. Das geschieht, wenn eine beliebige PostgreSQL-Clientanwendung den Befehl `NOTIFY` mit Verweis auf den entsprechenden Namen ausführt. Der eingesetzte Befehl wird in der Idle-Schleife von Tcl ausgeführt. Bei einer mit Tk geschriebenen Anwendung wird diese Schleife im Leerlauf automatisch aufgerufen. In anderen Tcl-Shells können Sie `update` oder `vwait` aufrufen, um in die Idle-Schleife einzutreten.

Wenn Sie pg_listen verwenden, sollten Sie die SQL-Befehle LISTEN und UNLISTEN nicht direkt aufrufen. pgtclübernimmt das für Sie. Aber wenn Sie selbst eine Benachrichtigungsmitteilung senden wollen, dann führen Sie den SQL-Befehl NOTIFY mit pg_exec aus.

Argumente

verbindung

Die Handle, bei der auf Benachrichtigungen geachtet werden soll.

notifyName

Der Name der Benachrichtigung ,auf die geachtet werden soll.

befehl

Wenn angegeben, der Befehlstext, der ausgeführt werden soll, wenn eine passende Benachrichtigung eintrifft.

Rückgabewert

Keiner

pg_on_connection_loss

Name

pg_on_connection_loss -- einen Befehl für unerwarteten Verbindungsverlust einsetzen oder ändern

Synopsis

```
pg_on_connection_loss verbindung ?befehl?
```

Beschreibung

pg_on_connection_loss setzt einen Befehl ein, der aufgerufen werden soll, wenn ein unerwarteter Verlust der Verbindung geschieht. Mit dem Parameter befehl wird ein Befehl eingesetzt oder der Text eines bestehenden Befehls wird ersetzt. Ohne den Parameter befehl wird ein schon registrierter Befehl widerrufen.

Der eingesetzte Befehl wird in der Idle-Schleife von Tcl ausgeführt. Bei einer mit Tk geschriebenen Anwendung wird diese Schleife im Leerlauf automatisch aufgerufen. In anderen Tcl-Shells können Sie update oder vwait aufrufen, um in die Idle-Schleife einzutreten.

Argumente

verbindung

Die Handle, bei der auf Verbindungsverlust geachtet werden soll.

befehl

Wenn angegeben, der Befehlstext, der ausgeführt werden soll, wenn ein Verbindungsverlust entdeckt wird.

Rückgabewert

Keiner

pg_lo_creat

Name

pg_lo_creat -- ein Large Object erzeugen

Synopsis

```
pg_lo_creat verbindung modus
```

Beschreibung

`pg_lo_creat` erzeugt ein Large Object.

Argumente

verbindung

Die Handle einer Datenbankverbindung, in der das Large Object erzeugt werden soll.

modus

Der Zugriffsmodus für das Large Object. Er kann `INV_READ` (lesen) oder `INV_WRITE` (schreiben) oder beides mit "oder" verknüpfen. Der Verknüpfungsoperator ist `|`. Zum Beispiel:

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

Rückgabewert

Die OID des erzeugten Large Object.

pg_lo_open

Name

pg_lo_open -- ein Large Object öffnen

Synopsis

```
pg_lo_open verbindung loid modus
```

Beschreibung

`pg_lo_open` öffnet ein Large Object.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das zu öffnende Large Object liegt.

`loid`

Die OID des Large Object.

`modus`

Gibt den Zugriffsmodus für das Large Object an. Der Modus kann sein: `r` (lesen), `w` (schreiben) oder `rw` (beides).

Rückgabewert

Ein Deskriptor zur späteren Verwendung in Large-Object-Befehlen.

pg_lo_close

Name

pg_lo_close -- ein Large Object schließen

Synopsis

```
pg_lo_close verbindung deskriptor
```

Beschreibung

`pg_lo_close` schließt ein Large Object.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

deskriptor

Ein Deskriptor für das Large Object von `pg_lo_open`.

Rückgabewert

Keiner

pg_lo_read

Name

pg_lo_read – aus einem Large Object lesen

Synopsis

```
pg_lo_read verbindung deskriptor bufVar länge
```

Beschreibung

`pg_lo_read` liest höchstens `länge` Bytes aus einem Large Object in eine Variable namens `bufVar`.

Argumente

verbindung

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

deskriptor

Ein Deskriptor für das Large Object von `pg_lo_open`.

bufVar

Der Name einer Puffervariablen, die das Large-Object-Segment enthalten soll.

länge

Die Höchstzahl der zu lesenden Bytes.

Rückgabewert

Keiner

pg_lo_write

Name

pg_lo_write -- in ein Large Object schreiben

Synopsis

```
pg_lo_write verbindung deskriptor buf länge
```

Beschreibung

`_pg_lo_write` schreibt höchstens *`länge`* Bytes aus der Variablen *`buf`* in ein Large Object.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

`deskriptor`

Ein Deskriptor für das Large Object von `pg_lo_open`.

`buf`

Eine Zeichenkette (kein Variablenname), die die zu schreibenden Daten enthält.

`länge`

Die Höchstzahl der zu schreibenden Bytes.

Rückgabewert

Keiner

pg_lo_lseek

Name

pg_lo_lseek – den Positionszeiger eines Large Object setzen

Synopsis

```
pg_lo_lseek verbindung deskriptor offset whence
```

Beschreibung

`pg_lo_lseek` bewegt die aktuelle Lese- und Schreibposition *`offset`* Bytes von der durch *`whence`* angegebenen Position.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

`deskriptor`

Ein Deskriptor für das Large Object von `pg_lo_open`.

`offset`

Die neue Position in Bytes.

`whence`

Gibt an, von wo die neue Position berechnet werden soll: `SEEK_CUR` (von der aktuellen Position), `SEEK_END` (vom Ende) oder `SEEK_SET` (vom Start).

Rückgabewert

Keiner

pg_lo_tell

Name

pg_lo_tell -- den aktuellen Positionszeiger eines Large Object zurückgeben

Synopsis

```
pg_lo_tell verbindung deskriptor
```

Beschreibung

`pg_lo_tell` gibt die aktuelle Lese- und Schreibposition in Bytes vom Anfang des Large Objects zurück.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

`deskriptor`

Ein Deskriptor für das Large Object von `pg_lo_open`.

Rückgabewert

Der aktuelle Positionszeiger in Bytes, geeignet zur Übergabe an `pg_lo_lseek`.

pg_lo_unlink

Name

pg_lo_unlink -- ein Large Object löschen

Synopsis

```
pg_lo_unlink verbindung loid
```

Beschreibung

`pg_lo_unlink` löscht das angegebene Large Object.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

`loid`

Die OID des Large Object.

Rückgabewert

Keiner

pg_lo_import

Name

pg_lo_import -- ein Large Object aus einer Datei importieren

Synopsis

```
pg_lo_import verbindung dateiname
```

Beschreibung

`pg_lo_import` liest die angegebene Datei und legt den Inhalt in einem neuen Large Object ab.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object erzeugt werden soll.

`dateiname`

Gibt an, aus welcher Datei die Daten importiert werden sollen.

Rückgabewert

Die OID des erzeugten Large Objects.

Hinweise

`pg_lo_import` muss in einem `BEGIN`/`COMMIT`-Transaktionsblock aufgerufen werden.

pg_lo_export

Name

pg_lo_export -- ein Large Object in eine Datei exportieren

Synopsis

```
pg_lo_export verbindung loid dateiname
```

Beschreibung

`pg_lo_export` schreibt das angegebene Large Object in eine Datei.

Argumente

`verbindung`

Die Handle einer Datenbankverbindung, in der das Large Object liegt.

`loid`

Die OID des Large Object.

`dateiname`

Gibt an, in welche Datei die Daten exportiert werden sollen.

Rückgabewert

Keiner

Hinweise

`pg_lo_export` muss in einem `BEGIN`/`COMMIT`-Transaktionsblock aufgerufen werden.

29.4 Beispielprogramm

Beispiel 29.1 zeigt ein kleines Beispiel, wie die `pgtcl`-Befehle verwendet werden.

Beispiel 29.1: pgtcl-Beispielprogramm

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY datname;"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
        lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

30

ECPG: Eingebettetes SQL in C

Dieses Kapitel beschreibt das PostgreSQL-Paket für eingebettetes SQL. Es funktioniert mit C und C++.

Diese Dokumentation ist zugegebenermaßen ziemlich unvollständig. Aber da diese Schnittstelle standardisiert ist, können weitere Informationen in vielen Quellen über SQL gefunden werden.

30.1 Das Konzept

Ein Eingebettetes-SQL-Programm besteht aus Code in einer gewöhnlichen Programmiersprache, in diesem Fall C, vermischt mit SQL-Befehlen in besonders markierten Abschnitten. Um das Programm zu bauen, wird der Quellcode zuerst dem Präprozessor für eingebettetes SQL übergeben, welcher ihn in ein gewöhnliches C-Programm umwandelt, und nachher kann er mit den C-Compilerwerkzeugen verarbeitet werden.

Eingebettetes SQL hat Vorteile gegenüber anderen Methoden zum Umgang mit SQL-Befehlen in C-Programmen. Erstens wird das lästige Umherreichen von Informationen an und von Variablen aus dem C-Programm übernommen. Zweitens ist in C eingebettetes SQL im SQL-Standard definiert und wird von vielen anderen SQL-Datenbanken unterstützt. Es ist beabsichtigt, dass die Implementierung in PostgreSQL diesem Standard soweit wie möglich entspricht, und es ist normalerweise möglich, für andere Datenbanken geschriebene eingebettete SQL-Programme relativ einfach auf PostgreSQL zu portieren.

Wie angedeutet, sind Programme, die für die eingebettete SQL-Schnittstelle geschrieben sind, normale C-Programme mit besonderem Code, der Datenbankaktionen durchführt. Dieser besondere Code hat die Form

```
EXEC SQL ...;
```

Die Befehle nehmen syntaktisch den Platz einer C-Anweisung ein. Je nach dem Befehl im Einzelnen können sie im globalen Kontext oder innerhalb einer Funktion auftreten. Eingebettete SQL-Befehle folgen den Regeln zur Groß- und Kleinschreibung, die für normalen SQL-Code gelten, und nicht denen von C.

Die folgenden Abschnitte beschreiben alle eingebetteten SQL-Befehle.

30.2 Zum Datenbankserver verbinden

Man verbindet zu einer Datenbank mit dem folgenden Befehl:

```
EXEC SQL CONNECT TO ziel [AS verbindungsname] [USER benutzername];
```

Das *ziel* kann auf folgende Arten angegeben werden:

- *dbname*[*@hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*optionen*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*optionen*]
- *zeichenvariable*
- *zeichenkette*
- DEFAULT

Es gibt auch verschiedene Arten, den Benutzernamen anzugeben:

- *benutzername*
- *benutzername/passwort*
- *benutzername* IDENTIFIED BY *passwort*
- *benutzername* USING *passwort*

Der *benutzername* und das *passwort* können SQL-Namen, Zeichenkettenvariablen oder Zeichenkonstanten sein.

Der *verbindungsname* wird verwendet, um mehrere Verbindungen in einem Programm zu verwalten. Wenn ein Programm nur eine Verbindung verwendet, dann kann er ausgelassen werden.

30.3 Eine Verbindung schließen

Um eine Verbindung zu schließen, verwenden Sie den folgenden Befehl:

```
EXEC SQL DISCONNECT [verbindung];
```

Die *verbindung* kann auf folgende Arten angegeben werden:

- verbindungsname
- DEFAULT
- CURRENT
- ALL

30.4 SQL-Befehle ausführen

Jeder beliebige SQL-Befehl kann aus einer eingebetteten SQL-Anwendung ausgeführt werden. Unten sind einige Beispiele dafür.

Eine Tabelle erzeugen:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
```

```
EXEC SQL COMMIT;
```

Zeilen einfügen:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Zeilen löschen:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Eine Zeile auswählen:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Auswählen mit einem Cursor:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Aktualisieren:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Tokens der Form `:irgendwas` sind **Hostvariablen**, das heißt, sie verweisen auf Variablen im C-Programm. Sie werden im nächsten Abschnitt erklärt.

In der Standardeinstellung werden Transaktionen nur abgeschlossen, wenn `EXEC SQL COMMIT` ausgeführt wird. Die eingebettete SQL-Schnittstelle unterstützt auch das automatische Abschließen von Transaktionen (Autocommit), auch bekannt von anderen Schnittstellen, mit der Kommandozeilenoption `-t` von `ecpg` (siehe unten) oder dem Befehl `EXEC SQL SET AUTOCOMMIT TO ON`. Im Autocommit-Modus wird die automatische Transaktion um jeden Befehl automatisch abgeschlossen, außer wenn ein expliziter Transaktionsblock verwendet wird. Dieser Modus kann mit `EXEC SQL SET AUTOCOMMIT TO OFF` ausdrücklich ausgeschaltet werden.

30.5 Daten übergeben

Um Daten aus dem Programm an die Datenbank zu übergeben, zum Beispiel als Parameter in einer Anfrage, oder um Daten aus der Datenbank zurück an das Programm zu übergeben, müssen die C-Variablen, die diese Daten enthalten sollen, in speziell markierten Abschnitten deklariert werden, damit der SQL-Präprozessor sie erkennen kann.

Dieser Abschnitt fängt mit

```
EXEC SQL BEGIN DECLARE SECTION;
```

an und endet mit

```
EXEC SQL END DECLARE SECTION;
```

Zwischen diesen Zeilen müssen normale C-Variablendeklarationen stehen, wie zum Beispiel

```
int x;
char foo[16], bar[16];
```

Die Deklarationen werden in die Ausgabedatei als normale C-Variablen übertragen und müssen also nicht nochmal deklariert werden. Variablen, bei denen keine Verwendung mit SQL-Befehlen beabsichtigt ist, können noch einmal außerhalb dieser speziellen Abschnitte deklariert werden.

Die Definition einer Struktur oder Union muss ebenso innerhalb eines `DECLARE`-Abschnitts stehen. Ansonsten kann der Präprozessor diese Typen nicht verarbeiten, weil er die Definition nicht kennt.

Die besonderen Typen VARCHAR und VARCHAR2 werden für jede Variable in ein benanntes struct umgewandelt. Eine Deklaration wie

```
VARCHAR var[180];
```

wird umgewandelt in

```
struct varchar_var { int len; char arr[180]; } var;
```

Diese Struktur ist passend als Schnittstelle zu SQL-Werten vom Typ varchar.

Um eine ordnungsgemäß deklarierte C-Variable in einem SQL-Befehl zu verwenden, schreiben Sie `:varname`, wo ein Ausdruck erwartet wird. Siehe im vorigen Abschnitt für Beispiele.

30.6 Fehlerbehandlung

Die eingebettete SQL-Schnittstelle bietet einen einfachen und einen komplexen Weg, um Ausnahmebedingungen in einem Programm zu behandeln. Die erste Methode gibt eine Nachricht aus, wenn eine bestimmte Voraussetzung eintritt. Zum Beispiel:

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

oder

```
EXEC SQL WHENEVER not found sqlprint;
```

Diese Fehlerbehandlung bleibt im gesamten Programm eingeschaltet.

Anmerkung

Dies ist *kein* erschöpfendes Beispiel der Verwendung des Befehls `EXEC SQL WHENEVER`. Weitere Beispiele finden Sie in SQL-Büchern.

Für eine mächtigere Fehlerbehandlung bietet die eingebettete SQL-Schnittstelle ein struct und eine Variable namens `sqlca` wie folgt:

```
struct sqlca
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
    char sqlerrp[8];

    long sqlerrd[6];
    /* 0: leer                                          */
    /* 1: OID der aktuellen Zeile, wenn angebracht      */
    /* 2: Anzahl betroffener Zeilen bei INSERT, UPDATE  */
    /*    oder DELETE Befehl                            */
    /* 3: leer                                          */
    /* 4: leer                                          */
    /* 5: leer                                          */

    char sqlwarn[8];
    /* 0: 'W' wenn mindestens ein weiteres Feld 'W' ist */
    /* 1: wenn 'W' dann wurde mindestens eine Zeichen-  */
    /*    kette beim Speichern in eine Hostvariable     */
    /*    abgeschnitten                                 */
    /* 2: leer                                          */
    /* 3: leer                                          */
    /* 4: leer                                          */
    /* 5: leer                                          */
    /* 6: leer                                          */
    /* 7: leer                                          */

    char sqlext[8];
} sqlca;
```

(Viele der leeren Felder könnten in einer zukünftigen Version verwendet werden.)

Wenn beim letzten SQL-Befehl kein Fehler auftrat, wird `sqlca.sqlcode` 0 sein (*ECPG_NO_ERROR*). Wenn `sqlca.sqlcode` kleiner als null ist, ist das ein ernster Fehler, wie eine Datenbankdefinition, die nicht mit der Anfrage übereinstimmt. Wenn es größer als null ist, ist das ein normaler Fehler, als wenn eine Tabelle die gewünschte Zeile nicht enthält.

`sqlca.sqlerrm.sqlerrmc` wird den Text enthalten, der den Fehler beschreibt. Der Text endet mit der Zeilennummer in der Quelldatei.

Dies sind die Fehler, die auftreten können:

`-12: Out of memory in line %d.`

Sollte normalerweise nicht auftreten. Gibt an, dass der virtuelle Speicher aufgebraucht ist.

`-200 (ECPG_UNSUPPORTED): Unsupported type %s on line %d.`

Sollte normalerweise nicht auftreten. Zeigt an, dass der Präprozessor etwas erzeugt hat, dass die Bibliothek nicht kennt. Vielleicht haben Sie inkompatible Versionen von Präprozessor und Bibliothek verwendet.

`-201 (ECPG_TOO_MANY_ARGUMENTS): Too many arguments line %d.`

Dies bedeutet, dass der Server mehr Argumente zurückgegeben hat, als wir passende Variablen haben. Vielleicht haben Sie ein paar Hostvariablen in der Liste `INTO :var1, :var2` vergessen.

`-202 (ECPG_TOO_FEW_ARGUMENTS): Too few arguments line %d.`

Dies bedeutet, dass der Server weniger Argumente zurückgegeben hat, als wir Hostvariablen haben. Vielleicht haben Sie zu viele Hostvariablen in der Liste `INTO :var1, :var2`.

`-203 (ECPG_TOO_MANY_MATCHES): Too many matches line %d.`

Dies bedeutet, dass die Anfrage mehrere Zeilen ergab, aber die angegebenen Variablen keine Arrays sind. Der `SELECT`-Befehl sollte nur eine Zeile ergeben.

`-204 (ECPG_INT_FORMAT): Not correctly formatted int type: %s line %d.`

Dies bedeutet, dass die Hostvariable vom Typ int ist und das Feld in der PostgreSQL-Datenbank einen anderen Typ hat und einen Wert enthält, der nicht als int interpretiert werden kann. Die Bibliothek verwendet `strtol()` für diese Umwandlung.

`-205 (ECPG_UINT_FORMAT): Not correctly formatted unsigned type: %s line %d.`

Dies bedeutet, dass die Hostvariable vom Typ unsigned int ist und das Feld in der PostgreSQL-Datenbank einen anderen Typ hat und einen Wert enthält, der nicht als unsigned int interpretiert werden kann. Die Bibliothek verwendet `strtoul()` für diese Umwandlung.

`-206 (ECPG_FLOAT_FORMAT): Not correctly formatted floating-point type: %s line %d.`

Dies bedeutet, dass die Hostvariable vom Typ float ist und das Feld in der PostgreSQL-Datenbank einen anderen Typ hat und einen Wert enthält, der nicht als float interpretiert werden kann. Die Bibliothek verwendet `strtod()` für diese Umwandlung.

`-207 (ECPG_CONVERT_BOOL): Unable to convert %s to bool on line %d.`

Dies bedeutet, dass die Hostvariable vom Typ bool ist und das Feld in der PostgreSQL-Datenbank weder `'t'` noch `'f'` ist.

`-208 (ECPG_EMPTY): Empty query line %d.`

Die Anfrage war leer. (Das kann in einem eingebetteten SQL-Programm normalerweise nicht passieren und könnte daher einen internen Fehler aufzeigen.)

`-209 (ECPG_MISSING_INDICATOR): NULL value without indicator in line %d.`

Ein NULL-Wert wurde zurückgegeben und keine NULL-Indikatorvariable angegeben.

`-210 (ECPG_NO_ARRAY): Variable is not an array in line %d.`

Eine normale Variable wurde verwendet, wo ein Array erforderlich ist.

`-211 (ECPG_DATA_NOT_ARRAY): Data read from backend is not an array in line %d.`

Die Datenbank gab eine normale Variable zurück, wo ein Array erforderlich ist.

`-220 (ECPG_NO_CONN): No such connection %s in line %d.`

Das Programm versuchte auf eine Verbindung zuzugreifen, die nicht existiert.

`-221 (ECPG_NOT_CONN): Not connected in line %d.`

Das Programm versuchte auf eine Verbindung zuzugreifen, die existier,t aber nicht geöffnet ist.

`-230 (ECPG_INVALID_STMT): Invalid statement name %s in line %d.`

Der Befehl, den Sie versuchen zu verwenden, wurde nicht vorbereitet.

`-240 (ECPG_UNKNOWN_DESCRIPTOR): Descriptor %s not found in line %d.`

Der angegebene Deskriptor wurde nicht gefunden. Der Befehl, den Sie versuchen zu verwenden, wurde nicht vorbereitet.

`-241 (ECPG_INVALID_DESCRIPTOR_INDEX): Descriptor index out of range in line %d.`

Der angegebene Deskriptorindex ist außerhalb des gültigen Bereichs.

`-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM): Unknown descriptor item %s in line %d.`

Der angegebene Deskriptor wurde nicht gefunden. Der Befehl, den Sie versuchen zu verwenden, wurde nicht vorbereitet.

`-243 (ECPG_VAR_NOT_NUMERIC): Variable is not a numeric type in line %d.`

Die Datenbank gab einen numerischen Wert zurück und die Variable war nicht numerisch.

`-244 (ECPG_VAR_NOT_CHAR): Variable is not a character type in line %d.`

Die Datenbank gab einen nichtnumerischen Wert zurück und die Variable war numerisch.

`-400 (ECPG_PGSQL): '%s' in line %d.`

Irgendein PostgreSQL-Fehler. Der Text enthält die Fehlermeldung vom PostgreSQL-Server.

`-401 (ECPG_TRANS): Error in transaction processing line %d.`

Der PostgreSQL-Server hat signalisiert, dass wir die Transaktion nicht starten, abschließen oder zurückrollen können.

`-402 (ECPG_CONNECT): Could not connect to database %s in line %d.`

Der Verbindungsversuch zur Datenbank ist fehlgeschlagen.

`100 (ECPG_NOT_FOUND): Data not found line %d.`

Dies ist ein “normaler” Fehler, der aussagt, dass das, was sie abfragen wollen, nicht gefunden werden kann oder dass Sie am Ende des Cursors sind.

30.7 Dateien einbinden

Um eine externe Datei in Ihr eingebettetes SQL-Programm einzubinden, verwenden Sie:

```
EXEC SQL INCLUDE dateiname;
```

Der Präprozessor für das eingebettete SQL wird nach einer Datei namens `dateiname.h` suchen, sie verarbeiten und das Ergebnis in die C-Ausgabe einfügen. Eingebettete SQL-Befehle in der eingebundenen Datei werden also richtig verarbeitet.

Beachten Sie, dass dies *nicht* das Gleiche ist wie

```
#include <dateiname.h>
```

weil diese Datei nicht vom SQL-Präprozessor verarbeitet werden würde. Natürlich können Sie die C-Direktive `#include` weiterhin für andere Headerdateien verwenden.

Anmerkung

Beim Namen der eingebundenen Datei wird auf Groß- und Kleinschreibung geachtet, obwohl der Rest des Befehls `EXEC SQL INCLUDE` den normalen SQL-Regeln für Groß- und Kleinschreibung folgt.

30.8 Eingebettete SQL-Programme verarbeiten

Jetzt, da Sie eine Vorstellung haben, wie C-Programme mit eingebettetem SQL geschrieben werden, wollen Sie sicher wissen, wie Sie sie compilieren sollen. Vor dem Compilieren schicken Sie die Datei durch den Präprozessor für eingebettetes SQL in C, welcher die SQL-Befehle, die Sie verwendet haben, in besondere Funktionsaufrufe umwandelt. Nach dem Compilieren müssen Sie eine besondere Bibliothek einlinken, die die benötigten Funktionen enthält. Diese Funktionen holen sich Informationen aus den Argumenten, führen den SQL-Befehl mit der `libpq`-Schnittstelle aus und legen die Ergebnisse in den zur Ausgabe bestimmten Argumenten ab.

Das Präprozessorprogramm heißt `ecpg` und ist in einer normalen PostgreSQL-Installation enthalten. Eingebettete SQL-Programme haben typischerweise die Erweiterung `.pgc`. Wenn Sie eine Programmdatei namens `prog1.pgc` haben, dann können Sie sie einfach mit

```
ecpg prog1.pgc
```

verarbeiten. Das wird eine Datei mit Namen `prog1.c` erzeugen. Wenn Ihre Eingabedateien nicht dem empfohlenen Namensmuster folgen, können Sie die Ausgabedatei ausdrücklich mit der Option `-o` angeben.

Die verarbeitete Datei kann normal compiliert werden, zum Beispiel:

```
cc -c prog1.c
```

Die erzeugte C-Quelldatei verwendet Headerdateien aus der PostgreSQL-Installation, also wenn Sie PostgreSQL an einem Ort installiert haben, der normalerweise nicht durchsucht wird, müssen Sie eine Option wie `-I/usr/local/pgsql/include` zur Kommandozeile bei der Compilierung hinzufügen.

Um ein eingebettetes SQL-Programm zu linken, müssen Sie die Bibliothek `libecpg` einbinden, etwa so:

```
cc -o meinprog prog1.o prog2.o ... -lecpg
```

Wiederum müssen Sie vielleicht eine Option wie `-L/usr/local/pgsql/lib` zur Kommandozeile hinzufügen.

Wenn Sie den Bauvorgang eines größeren Projekts mit `make` verwalten, kann es praktisch sein, folgende implizite Regel in die `make`-Steuerdateien einzufügen:

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

Die komplette Syntax von `ecpg` steht auf Seite 801.

30.9 Bibliotheksfunktionen

Die Bibliothek `libecpg` enthält hauptsächlich "versteckte" Funktionen, die verwendet werden, um die Funktionalität der eingebetteten SQL-Befehle umzusetzen. Aber es gibt einige Funktionen, die auch direkt aufgerufen werden können. Beachten Sie, dass das Ihren Code unportierbar macht.

- `ECPGdebug(int an, FILE *strom)` schaltet das Debugloggen ein, wenn das erste Argument von null verschieden ist. Der Log wird in den angegebenen Strom geschrieben. Der Log enthält alle SQL-Befehle mit den eingesetzten Eingabevariablen und die Ergebnisse vom PostgreSQL-Server. Dies kann nützlich sein, um Fehler in Ihren SQL-Befehlen zu suchen.
- `ECPGstatus()` ergibt wahr, wenn man mit der Datenbank verbunden ist, und falsch, wenn nicht.

30.10 Interna

Dieser Abschnitt erklärt, wie `ECPG` intern funktioniert. Diese Informationen können gelegentlich helfen, damit Benutzer besser verstehen, wie man `ECPG` verwendet.

Die ersten vier Zeilen, die von `ecpg` in die Ausgabe geschrieben werden, stehen fest. Zwei sind Kommentare und zwei binden Headerdateien ein, um die Schnittstelle zur Bibliothek herzustellen. Danach liest der Präprozessor die Datei und schreibt die Ausgabe. Normalerweise wird einfach alles unverändert an die Ausgabe weitergeleitet.

Wenn er einen `EXEC SQL`-Befehl sieht, greift er ein und ändert ihn. Der Befehl beginnt mit `EXEC SQL` und endet mit `;`. Alles dazwischen wird als SQL-Befehl behandelt und nach Variablen, die ersetzt werden müssen, durchsucht.

Variablenersetzung tritt auf, wenn ein Symbol mit einem Doppelpunkt (`:`) anfängt. Die Variable mit diesem Namen wird unter den Variablen, die zuvor in `EXEC SQL DECLARE`-Abschnitten deklariert wurden, gesucht.

Die wichtigste Bibliotheksfunktion ist `ECPGdo`; sie übernimmt die Ausführung der meisten Befehle. Sie hat eine variable Zahl von Argumenten. Das kann leicht 50 oder mehr Argumente ergeben, und hoffentlich ist das kein Problem auf irgendeiner Plattform.

Die Argumente sind:

eine Zeilennummer

> Dies ist die Zeilennummer der ursprünglichen Zeile, nur für Fehlermeldungen.

eine Zeichenkette

> Dies ist der SQL-Befehl, der ausgeführt werden soll. Er wird durch die Eingabevariablen verändert, d.h. die Variablen, die bei der Compilierung noch nicht feststanden, aber in den Befehl eingesetzt werden sollen. Dort, wo die Variablen hin sollen, steht im Text ein `?`.

Eingabevariablen

> Für jede Eingabevariable werden zehn Argumente erzeugt. (Siehe unten.)

`ECPGt_EOIT`

> Ein enum, der das Ende der Eingabevariablen anzeigt.

Ausgabevariablen

> Für jede Ausgabevariable werden zehn Argumenet erzeugt. (Siehe unten.) Diese Variablen werden von der Funktion gefüllt.

`ECPGt_EORT`

> Ein `enum`, der das Ende der Variablen anzeigt.

Für jede Variable, die Teil eines SQL-Befehls ist, erhält die Funktion zehn Argumente:

1. Der Typ als spezielles Symbol.
2. Ein Zeiger auf den Wert oder ein Zeiger auf den Zeiger.
3. Die Größe der Variable, falls sie ein `char` oder `varchar` ist.
4. Die Anzahl der Elemente im Array (für Arraydaten).
5. Der Offset zum nächsten Element im Array (für Arraydaten).
6. Der Typ der Indikatorvariable als spezielles Symbol.
7. Ein Zeiger auf die Indikatorvariable.
8. 0
9. Die Anzahl der Elemente im Indikatorarray (für Arraydaten).
10. Der Offset zum nächsten Element im Indikatorarray (für Arraydaten).

Beachten Sie, dass nicht alle SQL-Befehle so behandelt werden. Zum Beispiel ein `OPEN`-Befehl wie

```
EXEC SQL OPEN cursor;
```

wird nicht in die Ausgabe kopiert. Anstelle dessen wird der `DECLARE`-Befehl des Cursors verwendet, weil er den Cursor auch mit öffnet.

Hier ist ein vollständiges Beispiel, das die Ausgabe des Präprozessors bei einer Datei `foo.pgc` beschreibt (Einzelheiten könnten je nach der genauen Version des Präprozessors variieren):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

wird umgewandelt in:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(Die Einrückung hier wurde der Lesbarkeit halber hinzugefügt und stammt nicht vom Präprozessor.)

31

Die JDBC-Schnittstelle

Die JDBC-API ist ein fester Bestandteil der Java-Plattform seit Version 1.1. Sie bietet eine standardisierte Schnittstelle für den Zugriff auf SQL-kompatible Datenbanken.

PostgreSQL stellt einen JDBC-Treiber vom **Typ 4** zu Verfügung. Typ 4 heißt, dass der Treiber in reinem Java geschrieben ist und über das datenbankeigene Netzwerkprotokoll mit dem Datenbanksystem kommuniziert. Daher ist der Treiber plattformunabhängig; wenn er einmal kompiliert wurde, kann er auf jedem beliebigen System verwendet werden.

Dieses Kapitel ist keine vollständige Anleitung zum Programmieren mit JDBC, aber es sollte Ihnen beim Einstieg helfen können. Weitere Informationen finden Sie in der offiziellen Dokumentation zur JDBC-API. Der Quellcode enthält auch einige Beispiele.

31.1 Einrichtung des JDBC-Treibers

Dieser Abschnitt beschreibt die Schritte, die Sie bewältigen müssen, bevor Sie Programme, die die JDBC-Schnittstelle verwenden, schreiben oder ausführen können.

31.1.1 Den Treiber besorgen

Vorkompilierte Versionen des Treibers können von der PostgreSQL JDBC Website heruntergeladen werden.

Andererseits können Sie den Treiber selbst aus den Quellen bauen, aber das müssen Sie eigentlich nur, wenn Sie am Quellcode Veränderungen vorgenommen haben. Einzelheiten dazu finden Sie in den PostgreSQL-Installationsanweisungen. Nach der Installation finden Sie den Treiber in *`PRÄFIX`*`/share/java/postgresql.jar`. Der Treiber wird für die Java-Version gebaut, die Sie gerade verwenden. Wenn Sie mit einer JDK der Version 1.1 bauen, erhalten Sie einen Treiber, der die JDBC-1-Spezifikation unterstützt, wenn Sie mit einer Java-2-JDK bauen (z.B. JDK 1.2 oder JDK 1.3), erhalten Sie einen Treiber, der die JDBC-2-Spezifikation unterstützt.

31.1.2 Den Klassenpfad einrichten

Um den Treiber verwenden zu können, muss das JAR-Archiv in den Klassenpfad (*classpath*) eingefügt werden, entweder, indem man die Umgebungsvariable `CLASSPATH` anpasst oder die entsprechende Kom-

mandozeilenoption beim Programm `java` verwendet. (Das JAR-Archiv heißt `postgresql.jar`, wenn Sie es selbst gebaut haben, ansonsten wahrscheinlich `jdbc-1.1.jar` oder `jdbc-1.2.jar` für die JDBC-1- bzw. die JDBC-2-Version.)

Nehmen wir an, wir haben eine Anwendung, die den JDBC-Treiber verwendet, um auf eine Datenbank zuzugreifen, und diese Anwendung ist als `/usr/local/lib/myapp.jar` installiert. Der PostgreSQL-JDBC-Treiber ist als `/usr/local/pgsql/share/java/postgresql.jar` installiert. Um die Anwendung auszuführen, würden wir die folgenden Befehle verwenden:

```
export CLASSPATH=/usr/local/lib/myapp.jar:/usr/local/pgsql/share/java/
postgresql.jar:.java MyApp
```

Wie Sie den Treiber innerhalb der Anwendung laden, steht in Abschnitt 31.2.

31.1.3 Den Datenbankserver auf JDBC vorbereiten

Weil Java nur TCP/IP-Verbindungen verwendet, muss der PostgreSQL-Server so konfiguriert werden, dass er TCP/IP-Verbindungen annimmt. Um das zu erreichen, können Sie in der Datei `postgresql.conf` `tcpip_socket = true` setzen oder die Option `-i` angeben, wenn Sie `postmaster` starten.

Die Clientauthentifizierungs-Einstellungen in der Datei `pg_hba.conf` müssen möglicherweise ebenfalls konfiguriert werden. Einzelheiten dazu finden Sie in Kapitel 19. Der JDBC-Treiber unterstützt die Authentifizierungsmethoden `trust`, `ident`, `password`, `md5` und `crypt`.

31.2 Initialisierung des Treibers

Dieser Abschnitt beschreibt, wie Sie den JDBC-Treiber in Ihren Programmen laden und initialisieren können.

31.2.1 JDBC importieren

Jede Quellcodedatei, die JDBC verwendet, muss das Paket `java.sql` importieren:

```
import java.sql.*;
```

Anmerkung

Importieren Sie nicht das Paket `org.postgresql`. Wenn Sie es doch tun, werden Sie Ihre Quellen nicht kompilieren können, weil der Compiler verwirrt sein wird.

31.2.2 Den Treiber laden

Bevor Sie mit einer Datenbank verbinden können, müssen Sie den Treiber laden. Es gibt zwei mögliche Methoden. Welche davon die bessere ist, hängt von Ihrem Code ab.

Bei der ersten Methode lädt Ihr Code den Treiber implizit mit der Methode `Class.forName()`. Für den PostgreSQL-Treiber würden Sie Folgendes verwenden:

```
Class.forName("org.postgresql.Driver");
```

Damit wird der Treiber geladen und beim Laden automatisch im JDBC-System registriert.

Anmerkung

Die Methode `forName()` kann die Exception `ClassNotFoundException` erzeugen, wenn der Treiber nicht verfügbar ist.

Diese Methode ist die am häufigsten verwendete, aber damit legen Sie Ihren Code auf PostgreSQL fest. Wenn Ihr Code in der Zukunft eventuell auch auf ein anderes Datenbanksystem zugreifen soll und Sie keine PostgreSQL-spezifische Funktionalität verwenden, dann ist die zweite Methode zu empfehlen.

Bei der zweiten Methode wird der Treiber als Parameter übergeben, wenn die JVM gestartet wird, und zwar mit der Option `-D`. Zum Beispiel:

```
java -Djdbc.drivers=org.postgresql.Driver example.ImageViewer
```

In diesem Beispiel wird die JVM versuchen, den Treiber bei ihrer Initialisierung zu laden. Danach wird `ImageViewer` gestartet.

Diese Methode ist besser, weil Ihr Code damit andere Datenbanksysteme verwenden kann, ohne neu kompiliert werden zu müssen. Das Einzige, was Sie noch ändern müssten, ist die Verbindungs-URL, welche als nächstes besprochen wird.

Eine Sache noch zum Schluss: Wenn Ihr Code eine neue Verbindung (Klasse `Connection`) öffnet und Sie eine `SQLException` mit der Fehlermeldung `No driver available` erhalten, dann liegt das wahrscheinlich daran, dass der Treiber nicht im Klassenpfad ist oder der übergebene Parameter nicht richtig war.

31.2.3 Mit der Datenbank verbinden

Innerhalb JDBC wird eine Datenbank durch eine URL (*Uniform Resource Locator*) dargestellt. In PostgreSQL nimmt diese eine der folgenden Formen an:

- jdbc:postgresql:*datenbank*
- jdbc:postgresql:*//host/datenbank*
- jdbc:postgresql:*//host:port/datenbank*

Die Parameter haben die folgenden Bedeutungen:

host

Der Hostname des Servers. Die Voreinstellung ist `localhost`.

port

Die Portnummer auf der der Server auf Verbindungen wartet. Die Voreinstellung ist die Standardportnummer von PostgreSQL (5432).

datenbank

Der Datenbankname.

Um zu verbinden, müssen die eine Instanz der Klasse `Connection` von JDBC erhalten. Dazu rufen Sie die Methode `DriverManager.getConnection()` auf:

```
Connection db = DriverManager.getConnection(url, benutzername, passwort);
```

31.2.4 Die Verbindung schließen

Um die Datenbankverbindung zu schließen, rufen Sie einfach die Methode `close()` des `Connection`-Objekts auf:

```
db.close();
```

31.3 Ausführung von Anfragen und Verarbeitung der Ergebnisse

Jedes Mal, wenn Sie einen SQL-Befehl an die Datenbank schicken wollen, benötigen Sie eine Instanz der Klasse `Statement` oder `PreparedStatement`. Wenn Sie eine Instanz von `Statement` oder `PreparedStatement` haben, können Sie eine Anfrage ausführen. Als Ergebnis erhalten Sie eine Instanz der Klasse `ResultSet`, welche das komplette Ergebnis enthält. Beispiel 31.1 illustriert diesen Vorgang.

Beispiel 31.1: Eine einfache Anfrage mit JDBC verarbeiten

Dieses Beispiel verwendet die Klasse `Statement`, um eine einfache Anfrage auszuführen, und gibt die erste Spalte jeder Zeile aus.

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM meinetabelle WHERE x = 500");
while (rs.next()) {
    System.out.print("Spalte 1 ergab ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

Dieses Beispiel führt die gleiche Anfrage wie oben aus, aber verwendet `PreparedStatement` und bindet einen Wert in die Anfrage ein.

```
int wert = 500;
PreparedStatement st = db.prepareStatement("SELECT * FROM meinetabelle WHERE x =
?");
st.setInt(1, wert);
ResultSet rs = st.executeQuery();
while (rs.next()) {
    System.out.print("Spalte 1 ergab ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

31.3.1 Verwendung der Interfaces Statement oder PreparedStatement

Folgendes muss bei der Verwendung der Interfaces `Statement` und `PreparedStatement` beachtet werden:

- Sie können eine Instanz von `Statement` so oft verwenden, wie Sie wollen. Sie könnten eine erzeugen, sobald Sie die Verbindung geöffnet haben, und sie für die Dauer der Verbindung verwenden. Aber Sie müssen bedenken, dass pro `Statement` oder `PreparedStatement` nur ein `ResultSet` auf einmal existieren kann.
- Wenn Sie eine Anfrage ausführen müssen, während Sie ein `ResultSet` verarbeiten, können Sie einfach ein neues `Statement`-Objekt erzeugen und verwenden.
- Wenn Sie Threads verwenden und mehrere Threads die Datenbank verwenden, muss jeder Thread ein eigenes `Statement`-Objekt verwenden. Lesen Sie auch Abschnitt 31.7, wenn Sie darüber nachdenken, Threads zu verwenden, weil dort einige wichtige Punkte besprochen werden.
- Wenn Sie ein `Statement` oder `PreparedStatement` nicht mehr benötigen, sollten Sie es schließen.

31.3.2 Verwenden des Interface ResultSet

Folgendes muss bei dem Verwenden des Interface `ResultSet` beachtet werden:

- Bevor Sie irgendwelche Werte lesen können, müssen Sie `next()` aufrufen. Das ergibt `true`, wenn ein Ergebnis vorhanden ist, und, was noch wichtiger ist, es bereitet die Zeile auf die Verarbeitung vor.
- Laut JDBC-Spezifikation sollten Sie auf jedes Feld nur einmal zugreifen. Am sichersten ist es, wenn Sie diese Regel befolgen, obwohl der PostgreSQL-Treiber Ihnen gegenwärtig erlaubt, auf jedes Feld so oft Sie wollen zuzugreifen.
- Wenn Sie ein `ResultSet` nicht mehr benötigen, müssen Sie es mit `close()` schließen.
- Wenn Sie mit dem `Statement`-Objekt, aus dem das `ResultSet` erzeugt wurde, eine neue Anfrage ausführen, wird die aktuelle Instanz von `ResultSet` automatisch geschlossen.
- `ResultSet` ist gegenwärtig eine Konstante. Sie können durch ein `ResultSet` keine Daten aktualisieren. Wenn Sie Daten aktualisieren wollen, müssen Sie es auf die normale Art und Weise mit dem SQL-Befehl `UPDATE` tun. Das ist in Übereinstimmung mit der JDBC-Spezifikation, welche nicht verlangt, dass Treiber aktualisierbare Ergebnismengen anbieten müssen.

31.4 Ausführung von Aktualisierungen

Um Daten zu verändern (ein `INSERT`, `UPDATE` oder `DELETE` auszuführen), verwenden Sie die Methode `executeUpdate()`. Diese Methode ist ähnlich der Methode `executeQuery()`, mit der wir den `SELECT`-Befehl ausgeführt haben, aber sie gibt kein `ResultSet` zurück; stattdessen gibt sie die Anzahl der Zeilen, die vom `INSERT`, `UPDATE` bzw. `DELETE` betroffen waren, zurück. Beispiel 31.2 illustriert die Verwendung.

Beispiel 31.2: Zeilen löschen mit JDBC

Dieses Beispiel führt einen einfachen `DELETE`-Befehl aus und gibt die Anzahl der gelöschten Zeilen aus.

```
int wert = 500;
PreparedStatement st = db.prepareStatement("DELETE FROM meinetabelle WHERE x =
?");
st.setInt(1, wert);
```

```
int rowsDeleted = st.executeUpdate();
System.out.println(rowsDeleted + " Zeilen gelöscht");
st.close();
```

31.5 Erzeugung und Modifizierung von Datenbankobjekten

Um Datenbankobjekte wie Tabelle oder Sichten zu erzeugen, modifizieren oder löschen, verwenden Sie die Methode `execute()`. Diese Methode ist ähnlich der Methode `executeQuery()`, aber sie gibt kein Ergebnis zurück. Beispiel 31.3 illustriert die Verwendung.

Beispiel 31.3: Eine Tabelle mit JDBC löschen

Dieses Beispiel wird eine Tabelle löschen.

```
Statement st = db.createStatement();
st.execute("DROP TABLE meinetabelle");
st.close();
```

31.6 Speicherung binärer Daten

PostgreSQL bietet zwei verschiedene Methoden, um binäre Daten zu speichern. Man kann binäre Daten in einer Tabelle mit dem Datentypen bytea speichern oder die Large-Objects-Funktionalität verwenden, wodurch binäre Daten in einer getrennten Tabelle gespeichert werden und wobei man auf diese Daten durch einen Wert des Typs oid zugreifen kann.

Um zu entscheiden, welche Methode für Sie passend ist, müssen Sie die Beschränkungen jeder Methode verstehen. Der Datentyp bytea ist nicht besonders gut geeignet, wenn man sehr große Mengen binärer Daten speichern will. Obwohl eine Spalte vom Typ bytea bis zu 1 GB Daten speichern kann, würde man eine riesige Menge Arbeitsspeicher benötigen, um einen so großen Wert zu verarbeiten. Die Large-Objects-Methode ist besser geeignet, um sehr große Werte zu speichern, hat aber ihre eigenen Beschränkungen. Insbesondere wird beim Löschen einer Zeile mit einem Verweis auf ein Large Object nicht das Large Object selbst gelöscht. Das Löschen des Large Objects muss als separate Operation durchgeführt werden. Large Objects haben auch einige Sicherheitsprobleme, weil jeder, der mit der Datenbank verbunden ist, jedes Large Object ansehen oder verändern kann, selbst wenn er keinen Zugriff auf die Zeile mit dem Verweis auf das Large Object hat.

Version 7.2 war die erste Version des JDBC-Treibers, die den Datentyp bytea unterstützt hat. Bei der Einführung dieser Funktionalität in 7.2 wurden einige Verhaltensweisen gegenüber früheren Versionen verändert. Seit 7.2 verarbeiten die Methoden `getBytes()`, `setBytes()`, `getBinaryStream()` und `setBinaryStream()` Werte des Datentyps bytea. In und vor 7.1 verarbeiteten diese Methoden Werte des Datentyps oid, die auf Large Objects verwiesen haben. Sie können den Treiber so einstellen, dass er sich wieder wie in Version 7.1 verhält, indem Sie die Eigenschaft (*property*) `compatible` im `Connection`-Objekt auf den Wert `7.1` setzen.

Um den Datentyp bytea zu verwenden, sollten Sie einfach die Methoden `getBytes()`, `setBytes()`, `getBinaryStream()` oder `setBinaryStream()` anwenden.

Um die Large-Objects-Funktionalität zu verwenden, können Sie entweder die Klasse `LargeObject` verwenden, die vom PostgreSQL-JDBC-Treiber zur Verfügung gestellt wird, oder die Methoden `getBLOB()` ud `setBLOB( )`.

Wichtig

Zugriffe auf Large Objects müssen innerhalb eines SQL-Transaktionsblocks geschehen. Einen Transaktionsblock können Sie mit `setAutoCommit(false)` starten.

Anmerkung

In einer zukünftigen Version des JDBC-Treibers könnte es sein, dass die Methoden `getBLOB()` und `setBLOB()` nicht mehr mit Large Objects funktionieren, sondern dann mit dem Datentyp bytea. Es ist also zu empfehlen, dass Sie die `LargeObject`-Schnittstelle verwenden, wenn Sie Large Objects verwenden wollen.

Beispiel 31.4 enthält einige Beispiele, wie Sie mit dem PostgreSQL-JDBC-Treiber binäre Daten verarbeiten können.

Beispiel 31.4: Binäre Daten mit JDBC verarbeiten

Nehmen wir zum Beispiel an, Sie haben eine Tabelle, die die Namen von Bilddateien enthält und Sie wollen das Bild in einer Spalte vom Typ bytea speichern:

```
CREATE TABLE bilder (name text, bild bytea);
```

Um ein Bild einzufügen, könnten Sie folgenden Code verwenden:

```
File file = new File("meinbild.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("INSERT INTO bilder VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setBinaryStream(2, fis, file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

Hier überträgt `setBinaryStream()` die angegebene Anzahl von Bytes aus dem Datenstrom in die Spalte vom Typ bytea. Sie könnten das auch mit der Methode `setBytes()` machen, wenn die Bilddaten bereits in einem `byte[]`-Array vorliegen.

Das Auslesen des Bilds ist noch einfacher. (Wir verwenden hier die Klasse `PreparedStatement`, aber `Statement` könnte genauso gut verwendet werden.)

```
PreparedStatement ps = con.prepareStatement("SELECT bild FROM bilder WHERE name =
?");
ps.setString(1, "meinbild.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while (rs.next()) {
        byte[] imgBytes = rs.getBytes(1);
        // verwenden Sie die Daten hier irgendwie
    }
    rs.close();
```

```
}
ps.close();
```

Hier haben wir die Daten als `byte[]` ausgelesen. Stattdessen könnten Sie auch ein `InputStream`-Objekt verwenden.

Wenn Sie eine sehr große Datei speichern müssen und die `LargeObject`-Schnittstelle verwenden wollen, könnten Sie Folgendes machen:

```
CREATE TABLE bilder (name text, bildoid oid);
```

Um das Bild einzufügen, verwenden Sie:

```
// Alle LargeObject-Aufrufe müssen in einem Transaktionsblock stehen
conn.setAutoCommit(false);

// Erzeuge einen Large-Object-Manager
LargeObjectManager lobj =
((org.postgresql.PGConnection)conn).getLargeObjectAPI();

// Erzeuge ein neues Large Object
int oid = lobj.create(LargeObjectManager.READ | LargeObjectManager.WRITE);

// Öffne das Large Object zum Schreiben
LargeObject obj = lobj.open(oid, LargeObjectManager.WRITE);

// Öffne die Datei
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);

// Kopiere die Daten aus der Datei in das Large Object
byte buf[] = new byte[2048];
int s, tl = 0;
while ((s = fis.read(buf, 0, 2048)) > 0) {
    obj.write(buf, 0, s);
    tl += s;
}

// Schließe das Large Object
obj.close();

// Füge die Zeile in die Tabelle ein
PreparedStatement ps = conn.prepareStatement("INSERT INTO bilder VALUES (?, ?)");
ps.setString(1, file.getName());
ps.setInt(2, oid);
ps.executeUpdate();
ps.close();
fis.close();
```

Zum Auslesen des Large Objects aus der Datenbank könnten Sie folgenden Code verwenden:

```
// Alle LargeObject-Aufrufe müssen in einem Transaktionsblock stehen
conn.setAutoCommit(false);

// Erzeuge einen Large-Object-Manager
LargeObjectManager lobj =
((org.postgresql.PGConnection)conn).getLargeObjectAPI();

PreparedStatement ps = con.prepareStatement("SELECT bildoid FROM bilder WHERE
name = ?");
ps.setString(1, "meinbild.gif");
ResultSet rs = ps.executeQuery();
if (rs != null) {
    while (rs.next()) {
        // Öffne das Large Object zum Schreiben
        int oid = rs.getInt(1);
        LargeObject obj = lobj.open(oid, LargeObjectManager.READ);

        // Lese die Daten
        byte buf[] = new byte[obj.size()];
        obj.read(buf, 0, obj.size());
        // Verwenden Sie die Daten hier irgendwie

        // Schließe das Objekt
        obj.close();
    }
    rs.close();
}
ps.close();
```

31.7 Verwendung des Treibers in einer Multithread- oder Servlet-Umgebung

Ein Problem bei vielen JDBC-Treibern ist, dass nur ein Thread ein `Connection`-Objekt auf einmal verwenden kann, ansonsten könnte ein Thread eine Anfrage senden, während ein anderer gerade Ergebnisse empfängt, und das könnte ziemlich große Verwirrung stiften.

Der PostgreSQL-JDBC-Treiber ist Thread-sicher. Sie müssen in Ihre Anwendungen also keine komplexen Algorithmen einbauen, um zu verhindern, dass mehr als ein Thread auf einmal auf die Datenbank zugreift.

Wenn ein Thread versucht, die Verbindung zu verwenden, während sie gerade von einem anderen verwendet wird, dann wartet er, bis der andere seine aktuelle Operation abgeschlossen hat. Wenn die Operation ein normaler SQL-Befehl ist, besteht die Operation aus dem Senden des Befehls und dem Empfangen des vollständigen `ResultSet`. Wenn sie ein Fastpath-Aufruf ist (zum Beispiel das Lesen eines Blocks aus einem Large Object), besteht sie aus dem Senden und Empfangen der entsprechenden Daten.

Dieses Verhalten ist vertretbar in normalen Anwendungen und Applets, kann aber bei Servlets Leistungsprobleme verursachen. Wenn Sie mehrere Threads haben, die Anfragen ausführen, dann werden alle außer einem warten müssen. Um dieses Problem zu lösen, ist zu empfehlen einen Verbindungspool zu erzeugen. Wenn ein Thread die Datenbank verwenden will, dann bittet er eine Verwaltungsklasse um ein `Connection`-Objekt. Der Verwalter übergibt dem Thread eine freie Verbindung und markiert sie als in Benutzung. Wenn keine freie Verbindung zur Verfügung steht, dann wird eine neue geöffnet. Wenn der Thread mit der Verbindung fertig ist, gibt er sie dem Verwalter zurück, der sie entweder schließt oder im Pool speichert. Der Verwalter würde auch prüfen, ob die Verbindung noch fehlerfrei ist, und würde Sie schließen, wenn nicht. Der Nachteil eines Verbindungspools ist, dass es die Belastung des Servers vergrößert, weil für jedes `Connection`-Objekt eine neue Sitzung gestartet wird. Wie Sie sich entscheiden, hängt von den Anforderungen Ihrer Anwendungen ab.

31.8 Verbindungspools und Data Sources

In JDBC 2 wurde ein standardisierter Verbindungspoolmechanismus in einer Zusatz-API namens JDBC 2.0 Optional Package (auch bekannt unter dem Namen JDBC 2.0 Standard Extension) eingeführt. Seit JDBC 3 ist diese Funktionalität fester Bestandteil der JDBC-Schnittstelle. Der PostgreSQL-JDBC-Treiber unterstützt diese Funktionalität, wenn er mit JDK 1.3.x in Kombination mit dem JDBC 2.0 Optional Package, oder mit JDK 1.4 oder höher (JDBC 3) kompiliert wurde. Die meisten Anwendungsserver enthalten das JDBC 2.0 Optional Package aber man kann es auch separat von Sun über die JDBC Download-Site beziehen.

31.8.1 Überblick

JDBC bietet eine Client- und eine Server-Schnittstelle für Verbindungspools. Die Clientschnittstelle ist `javax.sql.DataSource`; sie wird in der Regel von normalem Anwendungscode verwendet um eine gepoolte Datenbankverbindung zu erhalten. Die Serverschnittstelle ist `javax.sql.ConnectionPoolDataSource`; sie wird von den meisten Anwendungsservern verwendet, um auf den PostgreSQL-JDBC-Treiber zuzugreifen.

In einer Anwendungsserverumgebung wird die Konfiguration des Anwendungsservers normalerweise auf die `ConnectionPoolDataSource`-Implementierung von PostgreSQL verweisen, während der Anwendungscode die `DataSource`-Implementierung des Anwendungsservers (nicht die von PostgreSQL) verwendet.

Für eine Umgebung ohne Anwendungsserver bietet PostgreSQL zwei Implementierungen von `DataSource`, die von einer Anwendung direkt verwendet werden können. Eine Implementierung richtet einen Verbindungspool ein und die andere bietet einfach einen Zugang zu einer normalen Datenbankverbindung ohne Pool durch das `DataSource`-Interface. Nochmal sei gesagt, diese Implementierungen sollten nicht in einer Anwendungsserverumgebung verwendet werden, es sei denn der Anwendungsserver unterstützt das `ConnectionPoolDataSource`-Interface nicht.

31.8.2 Anwendungsserver: ConnectionPoolDataSource

PostgreSQL enthält eine Implementierung von `ConnectionPoolDataSource` für JDBC 2 und eine für JDBC 3, wie in Tabelle 31.1 gezeigt.

JDBC	Implementierungsklasse
2	`org.postgresql.jdbc2.optional.ConnectionPool`
3	`org.postgresql.jdbc3.Jdbc3ConnectionPool`

Tabelle 31.1: Implementierungen von `ConnectionPoolDataSource`

Beide Implementierungen verwenden das gleiche Konfigurationsschema. JDBC verlangt, dass `ConnectionPoolDataSource`-Objekte über JavaBean-Propertys konfiguriert werden, welche in Tabelle 31.2 gezeigt werden, also gibt es für jede dieser Propertys eine *Get*- und eine *Set*-Methode.

Property	Typ	Beschreibung
`serverName`	`String`	Hostname des PostgreSQL-Datenbankservers
`databaseName`	`String`	PostgreSQL-Datenbankname
`portNumber`	`int`	TCP-Port des PostgreSQL-Datenbankservers (oder 0, um den Standardport zu verwenden)
`user`	`String`	Benutzername für Datenbankverbindungen
`password`	`String`	Passwort für Datenbankverbindungen
`defaultAutoCommit`	`boolean`	Ob Autocommit auf Verbindungen an oder aus sein soll, wenn Sie an ein Programm übergeben werden. Die Voreinstellung ist `false`, also kein Autocommit.

Tabelle 31.2: Konfigurations-Propertys von `ConnectionPoolDataSource`

Viele Anwendungsserver verwenden eine Syntax im Property-Stil, um diese Einstellungen zu konfigurieren. Daher ist es nicht ungewöhnlich, diese Propertys als einen Textblock einzugeben. Wenn der Anwendungsserver alle Propertys in einem einzigen Bereich eingeben lässt, könnte das so aussehen:

```
serverName=localhost
databaseName=test
user=testbenutzer
password=testpasswort
```

Oder wenn Semikolons anstelle von Zeilenumbrüchen als Trennzeichen verwendet werden, könnte es so aussehen:

```
serverName=localhost;databaseName=test;user=testbenutzer;password=testpasswort
```

31.8.3 Anwendungen: DataSource

PostgreSQL enthält zwei Implementierungen von `DataSource` für JDBC 2 und zwei für JDBC 3, wie in Tabelle 31.3 gezeigt wird. Die Pool-Varianten schließen eine Verbindung nicht, wenn der Client die Methode `close` aufruft, sondern speichern die Verbindung im Pool, wo sie auf die Verwendung durch andere Clients wartet. Dadurch wird vermieden, dass ständig Verbindungen geöffnet und geschlossen werden müssen und es ermöglicht, dass eine großen Anzahl von Clients eine kleine Anzahl von Datenbankverbindungen unter sich aufteilen.

Die hier angebotene Verbindungspool-Implementierung hat nicht besonders viele Funktionen. Unter anderem werden Verbindungen erst geschlossen, wenn das Pool selbst geschlossen wird; man kann das Pool nicht schrumpfen lassen. Außerdem werden Verbindungen, die nicht den eingestellten Standardbenutzer verwenden, nicht durch das Pool bedient, sondern direkt aufgebaut. Viele Anwendungsserver bieten eine bessere Verbindungspool-Funktionalität und verwenden dafür stattdessen die Implementierung von `ConnectionPoolDataSource`.

JDBC	Pool	Implementierungsklasse
2	Nein	`org.postgresql.jdbc2.optional.SimpleDataSource`
2	Ja	`org.postgresql.jdbc2.optional.PoolingDataSource`
3	Nein	`org.postgresql.jdbc3.Jdbc3SimpleDataSource`
3	Ja	`org.postgresql.jdbc3.Jdbc3PoolingDataSource`

Tabelle 31.3: Implementierungen von `DataSource`

Alle Implementierungen verwenden das gleiche Konfigurationsschema. JDBC verlangt, dass `DataSource`-Objekte über JavaBean-Propertys konfiguriert werden, welche in Tabelle 31.4 gezeigt werden, also gibt es für jede dieser Propertys eine *Get*- und eine *Set*-Methode.

Property	Typ	Beschreibung
`serverName`	`String`	PostgreSQL-Datenbankname
`databaseName`	`String`	PostgreSQL-Datenbankname
`portNumber`	`int`	TCP-Port des PostgreSQL-Datenbankservers (oder 0 um den Standardport zu verwenden)
`user`	`String`	Benutzername für Datenbankverbindungen
`password`	`String`	Passwort für Datenbankverbindungen

Tabelle 31.4: Konfigurations-Propertys von `DataSource`

Die Pool-Implementierungen haben einige zusätzliche Konfigurations-Propertys, welche in Tabelle 31.5 gezeigt werden.

Property	Typ	Beschreibung
`dataSourceName`	`String`	Jede `DataSource` mit Poolfunktion muss einen eindeutigen Namen haben.
`initialConnections`	`int`	Die Anzahl der zu erzeugenden Datenbankverbindungen, wenn das Pool initialisiert wird.
`maxConnections`	`int`	Die erlaubte Höchstzahl offener Datenbankverbindungen. Wenn mehr Verbindungen verlangt werden, wird das Programm so lange blockiert, bis eine Verbindung an das Pool zurückgegeben wird.

Tabelle 31.5: Zusätzliche Pool-Konfigurations-Propertys von `DataSource`

Beispiel 31.5 zeigt ein Beispiel einer typischen Anwendung, die eine `DataSource`-Implementierung mit Poolfunktion verwendet.

Beispiel 31.5: DataSource-Codebeispiel

Der Code zum Initialisieren der `DataSource` mit Poolfunktion könnte so aussehen:

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("Test-DataSource");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testbenutzer");
source.setPassword("testpasswort");
source.setMaxConnections(10);
```

Der Code, der eine Verbindung aus dem Pool verwendet, könnte wie folgt aussehen. Beachten Sie, dass es äußert wichtig ist, dass alle Verbindungen irgendwann geschlossen werden. Ansonsten wird das Pool Verbindungen verlieren und früher oder später alle Clients blockieren.

```
Connection con = null;
try {
    con = source.getConnection();
    // Verbindung verwenden
} catch (SQLException e) {
    // Fehler loggen
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
```

31.8.4 Data Sources und JNDI

Alle Implementierungen von `ConnectionPoolDataSource` und `DataSource` können im JNDI-System gespeichert werden. Im Falle einer Implementierung ohne Poolfunktionen wird jedes Mal, wenn ein Objekt aus dem JNDI abgerufen wird, eine neue Instanz erzeugt, welche dieselben Einstellungen hat wie die gespeicherte Instanz. Bei Implementierungen mit Poolfunktion wird die selbe Instanz so lange verwendet, wie sie verfügbar ist (so lange zum Beispiel keine andere JVM das Pool aus dem JNDI abruft); ansonsten wird eine neue Instanz mit den gleichen Einstellungen erzeugt.

In einer Anwendungsserverumgebung wird normalerweise die `DataSource`-Instanz des Anwendungsservers im JNDI gespeichert, nicht die `ConnectionPoolDataSource`-Implementierung von PostgreSQL.

In einer Anwendungsumgebung kann die Anwendung die `DataSource`-Instanz im JNDI speichern, damit sie den Verweis auf die `DataSource` nicht für alle Anwendungskomponenten, die ihn benötigen, verfügbar machen muss. Ein Beispiel dafür wird in Beispiel 31.6 gezeigt.

Beispiel 31.6: DataSource-Codebeispiel mit JNDI

Anwendungscode, der eine `DataSource`-Instanz mit Poolfunktion initialisiert und sie im JNDI-System ablegt, könnte so aussehen:

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("Test-DataSource");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testbenutzer");
source.setPassword("testpasswort");
source.setMaxConnections(10);
new InitialContext().rebind("DataSource", source);
```

Der Code, der dann eine Verbindung aus dem Pool verwendet, könnte so aussehen:

```
Connection con = null;
try {
    DataSource source = (DataSource)new InitialContext().lookup("DataSource");
    con = source.getConnection();
```

```
    // Verbindung verwenden
} catch (SQLException e) {
    // Fehler loggen
} catch (NamingException e) {
    // DataSource wurde nicht im JNDI gefunden
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
}
```

31.9 Weitere Informationsquellen

Fall Sie sie noch nicht gelesen haben, sollten Sie unbedingt die JDBC-API-Dokumentation (in der JDK von Sun enthalten) und die JDBC-Spezifikation lesen. Beide sind von http://java.sun.com/products/jdbc/index.html erhältlich.

http://jdbc.postgresql.org enthält aktuelle Informationen, die noch nicht in diesem Kapitel enthalten sind, und bietet außerdem vorkompilierte Treiber an.

32

PyGreSQL: Die Python-Schnittstelle

PyGreSQL bietet eine alte, ausgereifte Schnittstelle im Modul pg und eine neuere Schnittstelle im Modul pgdb, welche dem von der Python DB-SIG-Gruppe entwickelten DB-API 2.0 Standard entspricht.

Hier wird nur die ältere pg-Schnittstelle beschrieben. Informationen über DB-API finden Sie auf der Seite http://www.python.org/topics/database/DatabaseAPI-2.0.html. Eine Tutorial-ähnliche Einführung in DB-API können Sie auf der Seite http://www2.linuxjournal.com/lj-issues/issue49/2605.html finden.

32.1 Das pg-Modul

Das Modul pg definiert drei Objekte:

- pgobject, welches Verbindungen mit der Datenbank verwaltet,
- pglargeobject, welches Zugriffe auf PostgreSQL-Large-Objects verwaltet, und
- pgqueryobject, welches Anfrageergebnisse verwaltet.

Wenn Sie einige einfache Beispiele für die Verwendung dieses Moduls sehen wollen, schauen Sie sich die Seite *http://www.druid.net/rides* an, wo Sie unten auf der Seite eine Verknüpfung zum eigentlichen Python-Code der Seite finden werden.

Konstanten

Im Dictionary des pg-Moduls sind einige Konstanten definiert. Diese sind dazu gedacht, als Parameter für Methodenaufrufe verwendet zu werden. Weitere Informationen über sie finden Sie in der Anleitung zu libpq (Kapitel 27). Diese Konstanten sind:

```
INV_READ,
INV_WRITE
```

Zugriffsmodi für Large Objects, verwendet von (pgobject.)locreate und (pglarge.)open

```
SEEK_SET,
SEEK_CUR,
```

```
SEEK_END
```

Positionszeiger für `(pglarge.)seek`

```
version,
__version__
```

Konstanten, die die aktuelle Version enthalten

32.2 Funktionen im pg-Modul

Das Modul `pg` definiert nur ein paar Methoden, um mit der Datenbank zu verbinden und um einige Vorgabewerte einzustellen, die die von PostgreSQL verwendeten Umgebungsvariablen übergehen.

Diese Vorgabewerte wurden entworfen, damit Sie in Ihrem Code die Verbindungsparameter einfacher handhaben können. Sie können den Benutzer nach einem Wert fragen, die Eingabe im Vorgabewert ablegen und müssen sich danach nicht mehr darum kümmern; ebenso müssen Sie nicht Ihre Umgebungsvariablen verändern. Die Unterstützung dieser Vorgabewerte kann beim Compilieren mit der Option `-DNO_DEF_VAR` in der Datei `Setup` abgeschaltet werden. Die Methoden, die von dieser Einstellung abhängig sind, sind mit [DV] markiert.

Alle Vorgabewerte werden bei der Initialisierung des Moduls auf `None` gesetzt, was heißt, dass die Umgebungsvariablen verwendet werden sollen.

connect

Name

connect – öffnet eine Verbindung zum Datenbankserver

Synopsis

```
connect([dbname], [host], [port], [opt], [tty], [user], [passwd])
```

Parameter

dbname

Name der Datenbank, mit der Verbunden werden soll (Zeichenkette/`None`)

host

Name des Serverhosts (Zeichenkette/`None`)

port

Port des Datenbankservers (ganze Zahl/-1)

opt

Optionen für den Server (Zeichenkette/`None`)

tty

Datei oder TTY für Debugausgaben vom Server (Zeichenkette/None)

user

PostgreSQL-Benutzer (Zeichenkette/None)

passwd

Passwort des Benutzers (Zeichenkette/None)

Rückgabetyp

pgobject

Wenn erfolgreich, dann wird ein Objekt zurückgegeben, das eine Handle für die Datenbankverbindung ist.

Exceptions

TypeError

Falscher Argumenttyp oder zu viele Argumente.

SyntaxError

Doppelt angegebenes Argument.

pg.error

Ein Fehler geschah während des Verbindungsaufbaus durch pg.

(zuzüglich aller Exceptions, die beim Alloziieren eines Objekts auftreten können)

Beschreibung

Diese Methode öffnet eine Verbindung zur angegebenen Datenbank auf dem angegebenen PostgreSQL-Server. Die Argumente können auch mit Schlüsselwörtern angegeben werden. Die Namen der Schlüsselwörter sind die, die in der Syntaxzeile stehen. Eine Beschreibung der genauen Bedeutung der Parameter finden Sie in Kapitel 27.

Beispiele

```
import pg
con1 = pg.connect('testdb', 'myhost', 5432, None, None, 'bob', None)
con2 = pg.connect(dbname='testdb', host='localhost', user='bob')
```

get_defhost

Name

get_defhost – ermittelt den Standardhostnamen [DV]

Synopsis

```
get_defhost()
```

Parameter

__¨`y_keine

Rückgabetyp

Zeichenkette oder `None`

Standardhostangabe

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

`get_defhost()` gibt den aktuell eingestellten Standardhost zurück, oder `None`, wenn Umgebungsvariablen verwendet werden sollen. Umgebungsvariablen werden bei der Ermittlung des Rückgabewerts nicht untersucht.

set_defhost

Name

set_defhost – setzt den Standardhostnamen [DV]

Synopsis

```
set_defhost(host)
```

Parameter

`host`

Neuer Standardhost (Zeichenkette/`None`).

Rückgabetyp

Zeichenkette oder `None`

Vorheriger Standardhost.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

Beschreibung

`set_defhost()` setzt den Standardhost für neue Verbindungen. Wenn `None` als Parameter angegeben wird, werden für zukünfige Verbindungen Umgebungsvariablen verwendet. Der vorherige Standardhost wird zurückgegeben.

get_defport

Name

get_defport – ermittelt den Standardport [DV]

Synopsis

```
get_defport()
```

Parameter

keine

Rückgabetyp

ganze Zahl oder `None`

Standardportangabe

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

`get_defport()` gibt den aktuell eingestellten Standardport zurück oder `None`, wenn Umgebungsvariablen verwendet werden sollen. Umgebungsvariablen werden bei der Ermittlung des Rückgabewerts nicht untersucht.

set_defport

Name

set_defport – setzt den Standardport [DV]

Synopsis

```
set_defport(port)
```

Parameter

port

Neuer Standardport (ganze Zahl/-1).

Rückgabetyp

Ganze Zahl oder `None`

Vorheriger Standardport.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

Beschreibung

`set_defport()` setzt den Standardport für neue Verbindungen. Wenn -1 als Parameter angegeben wird, dann werden für zukünfige Verbindungen Umgebungsvariablen verwendet. Der vorherige Standardport wird zurückgegeben.

get_defopt

Name

get_defopt – ermittelt die Standardoptionsangabe [DV]

Synopsis

```
get_defopt()
```

Parameter

Keine

Rückgabetyp

Zeichenkette oder `None`

Standardoptionsangabe

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

`get_defopt()` gibt die aktuell eingestellten Standardoptionen zurück oder `None` , wenn Umgebungsvariablen verwendet werden sollen. Umgebungsvariablen werden bei der Ermittlung des Rückgabewerts nicht untersucht.

set_defopt

Name

```
set_defopt – setzt die Standardoptionsangabe [DV]
```

Synopsis

Parameter

`options`

Die neue Standardoptionen-Zeichenkette (Zeichenkette/`None`).

Rückgabetyp

Zeichenkette oder `None`

Vorherige Standardoptionsangabe.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

Beschreibung

`set_defopt()` setzt die Standardoptionen für neue Verbindungen. Wenn `None` als Parameter angegeben wird, werden für zukünftige Verbindungen Umgebungsvariablen verwendet. Die vorherigen Standardoptionen werden zurückgegeben.

get_deftty

Name

get_deftty – ermittelt das Standardterminal für Debugausgaben [DV]

Synopsis

```
get_deftty()
```

Parameter

Keine

Rückgabetyp

Zeichenkette oder `None`

Standard-Debug-Terminal

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

`get_deftty()` gibt das aktuell eingestellte Standard-TTY für Debugausgaben zurück, oder `None` wenn Umgebungsvariablen verwendet werden sollen. Umgebungsvariablen werden bei der Ermittlung des Rückgabewerts nicht untersucht.

set_deftty

Name

set_deftty – setzt das Standardterminal für Debugausgaben [DV]

Synopsis

```
set_deftty(terminal)
```

Parameter

`terminal`

Neues Standard-Debug-Terminal (Zeichenkette/`None`)

Rückgabetyp

Zeichenkette oder `None`

Vorheriges Standard-Debug-Terminal.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

Beschreibung

`set_defhost()` setzt das Standard-Debug-TTY für neue Verbindungen. Wenn `None` als Parameter angegeben wird, dann werden für zukünftige Verbindungen Umgebungsvariablen verwendet. Das vorherige Standard-TTY wird zurückgegeben.

get_defbase

Name

get_defbase – ermittelt die Standarddatenbank [DV]

Synopsis

```
get_defbase(
```

Parameter

Keine

Rückgabetyp

Zeichenkette oder `None`

Standarddatenbankname

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

`get_defbase()` gibt die aktuell eingestellte Standarddatenbank zurück oder `None` ,wenn Umgebungsvariablen verwendet werden sollen. Umgebungsvariablen werden bei der Ermittlung des Rückgabewerts nicht untersucht.

set_defbase

Name

set_defbase – setzt die Standarddatenbank [DV]

Synopsis

```
set_defbase(database)
```

Parameter

database

Neuer Standarddatenbankname (Zeichenkette/None).

Rückgabetyp

Zeichenkette oder None

Vorheriger Standarddatenbankname

Exceptions

TypeError

Falscher Argumenttyp oder zu viele Argumente.

Beschreibung

set_defbase() setzt die Standarddatenbank für neue Verbindungen. Wenn None als Parameter angegeben wird, dann werden für zukünftige Verbindungen Umgebungsvariablen verwendet. Die vorherige Standarddatenbank wird zurückgegeben.

32.3 Verbindungsobjekt: pgobject

Das Objekt pgobject verwaltet eine Verbindung mit einer PostgreSQL-Datenbank. Es enthält und versteckt alle Parameter dieser Verbindung und erlaubt nur durch Funktionen Zugriff auf die wichtigen Parameter.

Einige Methoden geben Ihnen direkten Zugriff auf die Verbindungssocket. Diese sind mit [DA] markiert. *Verwenden Sie diese Methoden nur, wenn Sie ganz genau wissen, was Sie tun.* Wenn Sie diese Methoden lieber deaktivieren wollen, setzen Sie beim Compilieren die Option -DNO_DIRECT in der Datei Setup.

Einige andere Methoden bieten Zugriff auf Large Objects. Wenn Sie den Zugriff auf diese Methoden vom Modul aus verbieten wollen, dann setzen Sie die Option -DNO_LARGE in der Datei Setup. Diese Methoden sind mit [LO] markiert.

Jedes pgobject-Objekt definiert einige konstante Attribute, die den Status der Verbindung beschreiben. Diese Attribute sind:

host

Der Hostname des Servers (Zeichenkette).

port

Der Port des Servers (ganze Zahl).

db

Die gewählte Datenbank (Zeichenkette).

options

Die Verbindungsoptionen (Zeichenkette).

`tty`

Das Terminal für Debugausgaben (Zeichenkette).

`user`

Der Datenbankbenutzername (Zeichenkette).

`status`

Der Zustand der Verbindung (ganze Zahl: 1 = in Ordnung, 0 = fehlerhaft).

`error`

Die letzte Warn- oder Fehlermeldung vom Server (Zeichenkette).

query

Name

query – führt einen SQL-Befehl aus

Synopsis

```
query(command)
```

Parameter

command

Der SQL-Befehl (Zeichenkette).

Rückgabetyp

`pgqueryobject` oder `None`

Ergebnis des Befehls.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`ValueError`

Leerer SQL-Befehl.

`pg.error`

Ein Fehler bei der Verarbeitung des Befehls oder eine ungültige Verbindung.

Beschreibung

Die Methode `query()` sendet einen SQL-Befehl an die Datenbank. Wenn der Befehl ein `INSERT`-Befehl ist, ist der Rückgabewert die OID der neu eingefügten Zeile. Wenn der Befehl ein anderer ist, der kein Ergebnis liefert (d.h. kein `SELECT`), dann wird `None` zurückgegeben. Ansonsten wird ein Objekt vom Typ `pgqueryobject` zurückgegeben, aus dem man die Anfrageergebnisse mit dem Methoden `getresult()` oder `dictresult()` abrufen kann oder das man einfach ausdrucken kann.

reset

Name

`reset – setzt die Verbindung zurück`

Synopsis

```
  reset(
```

Parameter

Keine

Rückgabetyp

Keiner

Exceptions

`TypeError`

Zu viele Argumente.

Beschreibung

Die Methode `reset()` setzt die aktuelle Verbindung zur Datenbank zurück.

close

Name

close – schließt die Datenbankverbindung

Synopsis

```
close()
```

Parameter

keine

Rückgabetyp

Keiner

Exceptions

`TypeError`

Zu viele Argumente.

Beschreibung

Die Methode `close()` schließt die Datenbankverbindung. Die Verbindung wird sowieso geschlossen, wenn das Verbindungsobjekt gelöscht wird, aber mit dieser Methode können Sie sie ausdrücklich schließen. Hauptsächlich ist diese Funktion vorhanden, damit die DB-SIG-Schnittstelle ihre entsprechende `close`-Funktion implementieren kann.

fileno

Name

fileno – gibt die Socketnummer der Datenbankverbindung zurück

Synopsis

```
fileno()
```

Parameter

Keine

Rückgabetyp

Socketnummer

Die Socketnummer, über die mit der Datenbank verbunden wurde.

Exceptions

`TypeError`

Zu viele Argumente.

Beschreibung

Die Methode `fileno()` gibt die Socketnummer zurück, über die mit der Datenbank verbunden wurde. Diese Information kann zum Beispiel für `select` verwendet werden.

getnotify

Name

`getnotify` – gibt die letzte Benachrichtigung vom Server zurück

Synopsis

```
getnotify(
```

Parameter

Keine

Rückgabetyp

Tupel, `None`

Die letzte Benachrichtigung vom Server.

Exceptions

`TypeError`

Zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Die Methode `getnotify()` versucht vom Server eine Benachrichtigung abzuholen. (Benachrichtigungen werden mit dem SQL-Befehl `NOTIFY` erzeugt.) Wenn der Server keine Benachrichtigung liefert, gibt die Methode `None` zurück. Ansonsten ergibt Sie ein Tupel (ein Paar) `(name, pid)`, wobei *name* der Name der Benachrichtigung ist und *pid* die Prozess-ID der Sitzung, die die Benachrichtigung ausgelöst hat. Denken Sie daran, dass zuerst `LISTEN` ausführen müssen, weil ansonsten `getnotify` immer `None` zurückgibt.

inserttable

Name

inserttable – fügt eine Liste in eine Tabelle ein

Synopsis

```
inserttable(table, values)
```

Parameter

table

Der Tabellenname (Zeichenkette).

values

Die Liste der einzufügenden Zeilenwerte (Liste).

Rückgabetyp

Keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Mit der Methode `inserttable()` können Sie schnell große Mengen Daten in eine Tabelle einfügen: Sie fügt die gesamte Werteliste in die Tabelle ein. Die Liste ist eine Liste von Tupeln oder Listen, die die Werte jeder einzufügenden Zeile bestimmen. Die Zeilenwerte können vom Typ string, integer, long oder double sein. *Achtung:* Diese Methode prüft nicht, ob die Felder mit der Tabellendefinition übereinstimmen; sie prüft nur, ob sie selbst die Typen verarbeiten kann.

putline

Name

putline – schreibt eine Zeile für die Serversocket [DA]

Synopsis

```
putline(line)
```

Parameter

line

Zu schreibende Zeile (Zeichenkette).

Rückgabetyp

keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Mit der Methode `putline()` können Sie eine Zeichenkette direkt zur Serversocket schreiben.

getline

Name

getline – liest eine Zeile von der Serversocket [DA]

Synopsis

```
getline()
```

Parameter

Keine

Rückgabetyp

Zeichenkette

Die gelesene Zeile.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Mit der Methode `getline()` können Sie eine Zeichenkette direkt von der Serversocket lesen.

endcopy

Name

endcopy – synchronisiert Client und Server [DA]

Synopsis

```
endcopy()
```

Parameter

Keine

Rückgabetyp

Keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Durch die Verwendung der Methoden, die direkt auf die Socket zugreifen, könnten Client und Server desynchronisiert werden. Mit dieser Methode wird die Synchronisierung zwischen Client und Server wiederhergestellt.

locreate

Name

locreate – erzeugt ein Large Object in der Datenbank [LO]

Synopsis

```
locreate(mode)
```

Parameter

mode

Erzeugungsmodus

Rückgabetyp

`pglarge`

Ein Objekt, das eine Handle für ein PostgreSQL-Large-Object darstellt.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung oder Fehler bei der Erzeugung des Objekts.

Beschreibung

Die Methode `locreate()` erzeugt ein Large Object in der Datenbank. Der Modus kann durch eine "Oder"-Verknüpfung der im Modul `pg` definierten Konstanten `INV_READ` (lesen) und `INV_WRITE` (schreiben) angegeben werden.

getlo

Name

getlo – öffnet ein Large Object [LO]

Synopsis

```
getlo(oid)
```

Parameter

`oid`

OID eines bestehenden Large Object (ganze Zahl)

Rückgabetyp

`pglarge`

Eine Handle für das Large Object

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung.

Beschreibung

Mit der Methode `getlo()` können Sie auf ein früher erzeugtes Large Object durch die Klasse `pglarge` zugreifen, wenn Sie die OID kennen.

loimport

Name

loimport – importiert ein Large Object aus einer Datei [LO]

Synopsis

```
loimport(filename)
```

Parameter

`filename`

Der Name der zu importierenden Datei (Zeichenkette).

Rückgabetyp

`pglarge`

Eine Handle für das Large Object.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`pg.error`

Ungültige Verbindung oder Fehler beim Importieren.

Beschreibung

Mit der Methode `loimport()` können Sie ein Large Object ganz einfach erzeugen. Geben Sie einfach den Namen der Datei an, die die Daten enthält, die Sie verwenden wollen.

32.4 Datenbank-Wrapper-Klasse: DB

Das Modul `pg` enthält eine Klasse namens `DB`. Alle Methoden der Klasse `pgobject` sind auch in diesem Objekt enthalten. Eine Reihe zusätzlicher Methoden wird unten beschrieben. Die beste Art und Weise, diese Klasse zu verwenden, ist wie folgt:

```
import pg
```

```
db = pg.DB(...)

for r in db.query("SELECT foo, bar FROM foo_bar_table WHERE foo !~
bar;").dictresult():
    print '%(foo)s %(bar)s' % r
```

Im Folgenden werden die Methoden und Variablen dieser Klasse beschrieben.

Die Initialisierung der Klasse `DB` erfolgt mit den gleichen Argumenten wie die Methode `pg.connect`. Außerdem werden dabei einige interne Variablen initialisiert. Die Anweisung `db = DB()` öffnet eine Verbindung zu einem lokalen Datenbankserver mit allen Vorgabeeinstellungen, genau wie es `pg.connect()` tut.

pkey

Name

pkey – gibt den Primärschlüssel einer Tabelle zurück

Synopsis

```
pkey(table)
```

Parameter

`table`

Name der Tabelle.

Rückgabetyp

Zeichenkette

Name der Spalte, die der Primärschlüssel der Tabelle ist.

Beschreibung

Die Methode `pkey()` gibt den Primärschlüssel einer Tabelle zurück. Beachten Sie, dass diese Methode eine Exception auslöst, wenn die Tabelle keinen Primärschlüssel hat.

get_databases

Name

get_databases – ermittelt eine Liste der Datenbanken im System

Synopsis

```
get_databases()
```

Parameter

Keine

Rückgabetyp

Liste

Liste der Datenbanken im System .

Beschreibung

Obwohl Sie diese Informationen auch mit einer direkten Anfrage ermitteln können, ist die Verwendung dieser Methode bequemer.

get_tables

Name

get_tables – ermittelt eine Liste aller Tabellen in der aktuellen Datenbank

Synopsis

```
get_tables()
```

Parameter

Keine

Rückgabetyp

Liste

Liste aller Tabellen in der aktuellen Datenbank.

Beschreibung

Obwohl Sie diese Informationen auch mit einer direkten Anfrage ermitteln können, ist die Verwendung dieser Methode bequemer.

get_attnames

Name

get_attnames – gibt die Spaltennamen einer Tabelle zurück

Synopsis

```
get_attnames(table)
```

Parameter

tabelle

Name der Tabelle.

Rückgabetyp

Dictionary

Die Schlüssel des Dictionary sind die Spaltennamen, die Werte sind die Typennamen der Spalten.

Beschreibung

Diese Methode ermittelt für die angegebene Tabelle die Namen der Spalten und deren Typen.

get

Name

get – liest eine Zeile aus einer Datenbanktabelle

Synopsis

```
get(table, arg, [keyname])
```

Parameter

table

Name der Tabelle.

arg

Ein Dictionary oder ein Wert, der gesucht werden soll.

keyname

Name einer Spalte, die als Schlüssel verwendet werden soll (optional).

Rückgabetyp

Dictionary

Ein Dictionary mit Spaltennamen als Schlüssel und Zeilenwerten als Werte.

Beschreibung

Diese Methode ist ein einfacher Mechanismus, um eine einzelne Zeile zu lesen. Der Suchschlüssel muss eine eindeutige Zeile auswählen. Wenn `keyname` nicht angegeben ist, wird der Primärschlüssel der Tabelle verwendet. Wenn `arg` ein Dictionary ist, wird daraus der Wert des Schlüssels entnommen und es wird um die gelesenen Werte ergänzt, wenn nötig durch Ersetzen der alten Werte. Die OID wird ebenso in das Dictionary aufgenommen, aber damit man das Dictionary mit mehreren Tabellen verwenden kann, wird der Spaltenname verändert, um ihn eindeutig zu machen: Er besteht aus der Zeichenkette `oid_` gefolgt vom Namen der Tabelle.

insert

Name

insert – fügt eine Zeile in eine Datenbanktabelle ein

Synopsis

```
insert(table, a)
```

Parameter

`table`

Name der Tabelle.

`a`

Ein Dictionary mit Werten.

Rückgabetyp

ganze Zahl

Die OID der neu eingefügten Zeile.

Beschreibung

Diese Methode fügt Werte aus dem Dictionary in die angegebene Tabelle ein. Danach lädt es die Werte aus der Datenbank zurück in das Dictionary. Dadurch wird das Dictionary aktuell gehalten, wenn die Werte durch Regeln, Trigger usw. verändert worden sind.

Wegen der Art, wie diese Funktion implementiert ist, werden Sie feststellen, dass die Einfügeoperationen länger und länger dauern, je größer die Tabelle wird. Um dieses Problem zu überwinden, erzeugen Sie einfach einen Index für die OID-Spalte jeder Tabelle, von der Sie denken, dass sie sehr groß werden könnte.

update

Name

update – aktualisiert eine Datenbanktabelle

Synopsis

```
update(table, a)
```

Parameter

table

Name der Tabelle.

a

Ein Dictionary mit Werten.

Rückgabetyp

ganze Zahl

Die OID der aktualisierten Zeile.

Beschreibung

Ähnlich wie `insert`, aber aktualisiert eine vorhandene Zeile. Die Aktualisierung wird auf Basis der OID vorgenommen, wobei der Name der OID-Spalte wie bei `get` verändert wird. Das zurückgegebene Dictionary ist das gleiche wie das übergebene, mit eventuellen Veränderungen durch Trigger, Regeln, Vorgabewerte usw.

clear

Name

clear – setzt die Werte einer Datenbanktabelle zurück

Synopsis

```
clear(table, [a])
```

Parameter

table

Name der Tabelle.

a

Ein Dictionary mit Werten.

Rückgabetyp

Dictionary

Ein Dictonary mit einer leeren Zeile.

Beschreibung

Diese Methode setzt die Spalten einer Tabelle auf typabhängige Werte zurück. Numerische Typen werden auf 0 gesetzt, Datumstypen auf `'today'`, alles andere auf eine leere Zeichenkette. Wenn das Dictionary-Argument vorhanden ist, werden nur die Spalten zurückgesetzt, die im Dictionary vorkommen, und alle anderen werden nicht verändert.

delete

Name

delete – löscht eine Zeile aus einer Datenbanktabelle

Synopsis

```
delete(table, [a])
```

Parameter

table

Name der Tabelle.

a

Ein Dictionary mit Werten.

Rückgabetyp

Keiner

Beschreibung

Diese Methode löscht eine Zeile aus einer Tabelle. Die Löschoperation verwendet die OID aus dem Dictionary, wie oben beschrieben.

32.5 Anfrageergebnisobjekt: pgqueryobject

Das Objekt `pgqueryobject` enthält ein Anfrageergebnis und bietet Methoden um dieses Anfragergebnis auszuwerten. Objekte dieses Typs werden von der Methode `pgobject.query()` erzeugt.

getresult

Name

getresult – gibt die von einer Anfrage ermittelten Werte zurück

Synopsis

```
getresult()
```

Parameter

Keine

Rückgabetyp

Liste

Eine Liste von Tupeln.

Exceptions

`SyntaxError`

Zu viele Argumente.

`pg.error`

Das Anfrageergebnis ist ungültig.

Beschreibung

Die Methode `getresult()` gibt eine Liste von Werten zurück, die durch die Anfrage ausgewählt wurden. Weitere Informationen über das Anfrageergebnis können mit den Methoden `listfields`, `fieldname` und `fieldnum` ermittelt werden.

dictresult

Name

dictresult – gibt die von einer Anfrage ermittelten Werte als Dictionary zurück

Synopsis

```
dictresult()
```

Parameter

Keine

Rückgabetyp

Liste

Eine Liste von Dictionaries.

Exceptions

`SyntaxError`

Zu viele Argumente.

`pg.error`

Das Anfrageergebnis ist ungültig.

Beschreibung

Die Methode `dictresult()` gibt eine Liste der von der Anfrage ermittelten Werte zurück, wobei jede Zeile als Dictionary mit den Spaltennamen als Schlüssel dargestellt wird.

listfields

Name

listfields – gibt die Spaltennamen des Anfrageergebnisses zurück

Synopsis

```
listfields()
```

Parameter

Keine

Rückgabetyp

Liste

Liste der Spaltennamen

Exceptions

`SyntaxError`

Zu viele Argumente.

`pg.error`

Ungültiges Anfrageergebnis oder ungültige Verbindung.

Beschreibung

Die Methode `listfields()` ergibt eine Liste der Spaltennamen im Anfrageergebnis. Die Spaltennamen sind in derselben Reihenfolge wie die Ergebniswerte.

fieldname

Name

fieldname – ermittelt den Spaltennamen für eine Spaltennummer

Synopsis

```
fieldname(i)
```

Parameter

i

Spaltennummer (ganze Zahl)

Rückgabetyp

Zeichenkette

Spaltenname

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`ValueError`

Ungültige Spaltennummer.

`pg.error`

Ungültiges Anfrageergebnis oder ungültige Verbindung.

Beschreibung

Mit der Methode `fieldname()` können Sie den Spaltennamen aus einer Spaltennummer ermitteln. Das kann zum Beispiel nützlich sein, wenn Sie die Ergebnisse tabellarisch darstellen wollen. Die Spalten sind in derselben Reihenfolge wie die Ergebniswerte.

fieldnum

Name

fieldnum – ermittelt die Spaltennummer für einen Spaltennamen

Synopsis

```
fieldnum(name)
```

Parameter

name

Spaltenname (Zeichenkette)

Rückgabetyp

ganze Zahl

Spaltennummer

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`ValueError`

Unbekannter Spaltenname.

`pg.error`

Ungültiges Anfrageergebnis oder ungültige Verbindung.

Beschreibung

Die Methode `fieldnum()` ermittelt die Spaltennummer für einen Spaltennamen. Damit kann man zum Beispiel eine Funktion schreiben, die die Werte eines Anfrageergebnisses anhand einer dem Programm bekannten Tabellendefinition in ihre korrekten Typen umwandelt. Die zurückgegebene Nummer zählt in der Reihenfolge der Ergebniswerte.

ntuples

Name

ntuples – gibt die Anzahl der Ergebniszeilen in einem Ergebnisobjekt zurück

Synopsis

```
ntuples()
```

Parameter

Keine

Rückgabetyp

ganze Zahl

Anzahl der Zeilen im Ergebnisobjekt.

Exceptions

`SyntaxError`

Zu viele Argumente.

Beschreibung

Die Methode `ntuples()` gibt die Anzahl der von der Anfrage ermittelten Zeilen zurück.

32.6 Large Object: pglarge

Das Objekt `pglarge` verwaltet alle Operationen mit PostgreSQL-Large-Objects. Es speichert und versteckt alle "wiederkehrenden" Variablen (Objekt-ID und Verbindung), genauso wie `pgobject`-Objekte es tun, wodurch nur die wichtigen Parameter in Funktionsaufrufen angegeben werden müssen. Es speichert einen Verweis auf das `pgobject`-Objekt, aus dem es erzeugt wurde, und leitet die eigenen Aufrufe an dieses weiter. Jede Veränderung am `pgobject`-Objekt wirkt sich daher auf das `pglarge`-Objekt aus. `Python` gibt den Speicher des `pgobject`-Objekts erst frei, wenn alle daraus erzeugten `pglarge`-Objekte auch freigegeben wurden. Alle Funktionen geben bei einem Fehler eine allgemeine Fehlermeldung zurück, egal, was der genaue Fehler war. Das Attribut `error` des Objekts enthält die genaue Fehlermeldung.

`pglarge`-Objekte definieren einige konstante Attribute, die Informationen über das Objekt enthalten. Diese Attribute sind:

`oid`

Die OID des Objekts.

`pgcnx`

Das zugehörige `pgobject`-Objekt.

`error`

Die letzte Warn- oder Fehlermeldung auf der Verbindung.

Das OID-Attribut ist sehr interessant, weil Sie damit später ein `pglarge`-Objekt mit der Methode `getlo()` der Klasse `pgobject` erzeugen können.

Weitere Informationen über die Large-Object-Schnittstelle in PostgreSQL finden Sie in Kapitel 28.

Wichtig

In Multithread-Umgebungen kann `error` durch einen anderen Thread, der dasselbe `pgobject`-Objekt verwendet, verändert werden. Beachten Sie, dass diese Objekte geteilt sind und nicht dupliziert werden; wenn Sie in solchen Situationen die Fehlermeldung prüfen wollen, müssen Sie selbst Sperren einrichten.

open

Name

open – öffnet das Large Object

Synopsis

```
open(mode)
```

Parameter

mode

Zugriffsmodus (ganze Zahl).

Rückgabetyp

Keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`IOError`

Schon geöffnet oder Fehler beim Öffnen.

`pg.error`

Ungültige Verbindung.

Beschreibung

Die Methode `open()` öffnet ein Large Object zum Lesen oder Schreiben, ähnlich wie die Unix-Funktion `open()`. Der Modus kann durch eine "Oder"-Verknüpfung der im Modul `pg` definierten Konstanten `INV_READ` (lesen) und `INV_WRITE` (schreiben) angegeben werden.

close

Name

close – schließt das Large Object

Synopsis

```
close()
```

Parameter

Keine

Rückgabetyp

Keiner

Exceptions

`SyntaxError`

Zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Schließen.

`pg.error`

Ungültige Verbindung.

Beschreibung

Die Methode `close()` schließt ein vorher geöffnetes Large Object, ähnlich wie die Unix-Funktion `close()`.

read

Name

read – liest aus dem Large Object

Synopsis

```
read(size)
```

Parameter

size

Maximal zu lesende Daten (ganze Zahl).

Rückgabetyp

Zeichenkette

Die gelesenen Daten.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Schreiben.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Die Methode `read()` liest Daten aus dem Large Object, ausgehend von der aktuellen Position.

write

Name

write – schreibt in das Large Object

Synopsis

```
write(string)
```

Parameter

string

Zu schreibende Daten (Zeichenkette).

Rückgabetyp

Keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Schreiben.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Die Methode `write()` schreibt Daten in das Large Object, ausgehend von der aktuellen Position.

seek

Name

seek – ändert die aktuelle Position im Large Object

Synopsis

```
seek(offset, whence)
```

Parameter

`offset`

Neue Position (ganze Zahl).

`whence`

Ausgangspunkt für die neue Position (ganze Zahl).

Rückgabetyp

ganze Zahl

Die neue aktuelle Position im Objekt.

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Positionieren.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Die Methode `seek()` bewegt die aktuelle Lese- und Schreibposition im Large Object. Der Parameter `whence` kann durch eine der im Modul `pg` definierten Konstanten `SEEK_SET` (zählt vom Anfang des Objekts), `SEEK_CUR` (zählt von der aktuellen Position aus) und `SEEK_END` (zählt vom Ende des Objekts) angegeben werden.

tell

Name

tell – gibt die aktuelle Position im Large Object zurück

Synopsis

```
tell()
```

Parameter

Keine

Rückgabetyp

ganze Zahl

Aktuelle Position im Objekt.

Exceptions

`SyntaxError`

Zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Positionieren.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Die Methode `tell()` gibt die aktuelle Lese- und Schreibposition im Large Object zurück.

unlink

Name

unlink – löscht das Large Object

Synopsis

```
unlink()
```

Parameter

Keine

Rückgabetyp

Keiner

Exceptions

`SyntaxError`

Zu viele Argumente.

`IOError`

Objekt ist nicht geschlossen oder Fehler beim Löschen.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Die Methode `unlink()` löscht ein Large Object.

size

Name

size – gibt die Größe des Large Object zurück

Synopsis

```
size()
```

Parameter

Keine

Rückgabetyp

Ganze Zahl

Die Größe des Large Object.

Exceptions

`SyntaxError`

Zu viele Argumente.

`IOError`

Objekt ist nicht geöffnet oder Fehler beim Positionieren.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Mit der Methode `size()` können Sie die Größe des Large Objects ermitteln. Sie wurde implementiert, weil Sie für Web-Anwendungen sehr nützlich ist. Gegenwärtig müssen Sie das Large Object erst öffnen.

export

Name

export – speichert das Large Object in einer Datei

Synopsis

```
export(filename)
```

Parameter

`filename`

Name der zu erzeugenden Datei.

Rückgabetyp

Keiner

Exceptions

`TypeError`

Falscher Argumenttyp oder zu viele Argumente.

`IOError`

Objekt ist nicht geschlossen oder Fehler beim Exportieren.

`pg.error`

Ungültige Verbindung oder ungültiges Objekt.

Beschreibung

Mit der Methode `export()` können Sie den Inhalt eines Large Objects auf eine ganze einfache Art und Weise in einer Datei abspeichern. Die Exportdatei wird auf dem Rechner des Programms, nicht auf dem Serverrechner, erzeugt.

Teil V

Server-Programmierung

In diesem Teil geht es um die Erweiterung der Serverfunktionalität durch benutzerdefinierte Funktionen, Datentypen, Trigger usw. Das sind fortgeschrittene Themen, denen Sie sich wahrscheinlich erst widmen sollten, wenn Sie die gesamte restliche Benutzerdokumentation zu PostgreSQL verstanden haben. Dieser Teil beschreibt auch die in der PostgreSQL-Distribution verfügbaren serverseitigen Programmiersprachen und allgemeine einige allgemeine Themen über Serverprogrammiersprachen. Diese Informationen sind wahrscheinlich nur für die Leser nützlich, die zumindest die ersten paar Kapitel dieses Teils gelesen haben.

33

SQL erweitern

In den folgenden Abschnitten besprechen wir, wie man die SQL-Sprache in PostgreSQL erweitern kann, und zwar mit benutzerdefinierten

- Funktionen (ab Abschnitt 33.3)
- Datentypen (ab Abschnitt 33.10)
- Operatoren (ab Abschnitt 33.11)
- Aggregatfunktionen (ab Abschnitt 33.13)

33.1 Wie die Erweiterbarkeit funktioniert

PostgreSQL ist erweiterbar, weil seine Operation von Systemkatalogen gesteuert wird. Wenn Sie sich mit normalen relationalen Datenbanksystemen auskennen, wissen Sie, dass diese die Informationen über Datenbanken, Tabellen, Spalten in so genannten Systemkatalogen speichern. (Manche Systeme nennen dies auch Data Dictionary.) Die Kataloge erscheinen für den Benutzer als normale Tabellen, aber das Datenbanksystem speichert darin seine internen Informationen. Ein wichtiger Unterschied zwischen PostgreSQL und normalen relationalen Datenbanksystemen ist, dass PostgreSQL viel mehr Informationen in seinen Katalogen speichert: nicht nur Informationen über Tabellen und Spalten, sondern auch Informationen über Datentypen, Funktionen, Zugriffsmethoden und so weiter. Diese Tabellen können vom Benutzer verändert werden, und da die Operation von PostgreSQL von diesen Tabellen gesteuert wird, bedeutet das, dass PostgreSQL von Benutzern erweitert werden kann. Im Gegensatz dazu können herkömmliche Datenbanksysteme nur erweitert werden, indem der Quellcode selbst verändert wird oder indem spezielle Module vom DBMS-Hersteller geladen werden.

Der PostgreSQL-Server kann außerdem Anwendercode durch dynamisches Laden aufnehmen. Das heißt, der Benutzer kann eine Objektdatei (zum Beispiel eine dynamische Bibliothek) angeben, die einen neuen Typ oder eine neue Funktion implementiert, und PostgreSQL wird diese bei Bedarf laden. Code, der in SQL geschrieben ist, kann sogar noch einfacher in den Server integriert werden. Diese Fähigkeit, die Operation im laufenden Betrieb verändern zu können, macht PostgreSQL auf einzigartige Weise geeignet für schnelles Prototyping von neuen Anwendungen und Speicherstrukturen.

33.2 Das PostgreSQL-Typensystem

Ein Datentyp ist entweder ein Basis- oder ein zusammengesetzter Typ. Basistypen sind die, die in einer Sprache wie C implementiert sind, wie zum Beispiel int4. Diese entsprechen im Allgemeinen so genannten abstrakten Datentypen. PostgreSQL kann mit diesen Typen nur durch vom Benutzer zur Verfügung gestellte Methoden umgehen und versteht das Verhalten dieser Typen nur so weit, wie der Benutzer es beschreibt. Zusammengesetzte Typen entstehen immer, wenn der Benutzer eine Tabelle erzeugt. Der Benutzer kann auf die Attribute dieser Typen durch die Anfragesprache zugreifen.

33.3 Benutzerdefinierte Funktionen

PostgreSQL stellt vier Arten von Funktionen zur Verfügung:

- in SQL geschriebene Funktionen (Abschnitt 33.4)
- in einer prozeduralen Sprache geschriebene Funktionen (zum Beispiel `PL/Tcl` oder `PL/pgSQL`) (Abschnitt 33.5)
- interne Funktionen (Abschnitt 33.6)
- in C geschriebene Funktionen (Abschnitt 33.7)

Jede Art von Funktion kann Basistypen, zusammengesetzte Typen oder beide als Argumente haben. Außerdem kann jede Art von Funktion einen Basistyp oder einen zusammengesetzten Typ zurückgeben.

Am einfachsten sind in SQL geschriebene Funktionen, daher fangen wir mit denen an. Die Beispiele in diesem Abschnitt finden Sie auch in `funcs.sql` und `funcs.c` im Tutorialverzeichnis.

Während Sie dies lesen, kann es nützlich sein, die Referenzseite des Befehls `CREATE FUNCTION` parat zu haben, um die Beispiele besser verstehen zu können.

33.4 SQL-Funktionen

SQL-Funktionen führen eine beliebige Liste von SQL-Befehlen aus und geben das Ergebnis der letzten Anfrage in der Liste zurück. (Die Anfrage muss ein `SELECT`-Befehl sein.) Im einfachen Fall (ohne Ergebnismenge) wird die erste Zeile aus dem Ergebnis der letzten Anfrage zurückgegeben. (Bedenken Sie, dass "die erste Zeile" eines mehrzeiligen Ergebnisses nur genau bestimmbar ist, wenn Sie `ORDER BY` verwenden.) Wenn die letzte Anfrage zufällig keine Zeilen zurückgibt, ist das Ergebnis der Funktion der NULL-Wert.

Andererseits kann eine SQL-Funktion auch eine Ergebnismenge zurückgeben, wenn der Rückgabetyp der Funktion als `SETOF typ` angegeben wurde. In dem Fall werden alle Ergebniszeilen der letzten Anfrage zurückgegeben. Einzelheiten dazu folgen weiter unten.

Der Körper einer SQL-Funktion sollte eine Liste aus einem oder mehreren SQL-Befehlen, durch Semikolons getrennt, sein. Beachten Sie, dass die Syntax von `CREATE FUNCTION` erfordert, dass der Körper der Funktion von Apostrophen eingeschlossen wird und dass daher Apostrophe im Funktionskörper durch Fluchtfolgen ersetzt werden müssen, entweder durch zwei Apostrophe (`''`) oder einen Backslash vor dem Apostroph (`\'`).

Auf die Argumente einer SQL-Funktion kann im Funktionskörper mit der Syntax `$n` verwiesen werden: `$1` ist das erste Argument, `$2` ist das zweite und so weiter. Wenn ein Argument einen zusammengesetzten Typ hat, kann die Punktschreibweise, z.B. `$1.name`, verwendet werden, um auf die Attribute des Arguments zuzugreifen.

33.4.1 SQL-Funktionen mit Basistypen

Die einfachste SQL-Funktion hat keine Argumente und hat einen Basistyp als Rückgabewert, zum Beispiel integer:

```
CREATE FUNCTION eins() RETURNS integer AS '
    SELECT 1 AS ergebnis;
' LANGUAGE SQL;

SELECT eins();

 eins
------
    1
```

Wie Sie sehen, haben wir im Funktionskörper einen Aliasnamen für das Ergebnis der Funktion angegeben (mit dem Namen `ergebnis`), aber dieser Aliasname ist außerhalb der Funktion nicht sichtbar. Daher heißt die Ergebnisspalte `eins` und nicht `ergebnis`.

Fast genauso einfach kann man SQL-Funktionen schreiben, die Basistypen als Argumente verwenden. Beachten Sie, wie auf die Argumente mit `$1` und `$2` verwiesen wird.

```
CREATE FUNCTION addition(integer, integer) RETURNS integer AS '
    SELECT $1 + $2;
' LANGUAGE SQL;

SELECT addition(1, 2) AS antwort;

 antwort
---------
       3
```

Hier ist eine etwas nützlichere Funktion, die zum Belasten eines Kontos verwendet werden könnte:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS '
    UPDATE bank
        SET kontostand = kontostand - $2
        WHERE konto_nr = $1;
    SELECT 1;
' LANGUAGE SQL;
```

Um 100 Euro vom Konto Nummer 17 abzuziehen, könnte der Benutzer Folgendes ausführen:

```
SELECT tf1(17, 100.0);
```

In der Praxis möchte man wahrscheinlich ein nützlicheres Ergebnis aus der Funktion zurückgeben, anstatt einfach immer 1. Eine bessere Definition wäre daher:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS '
    UPDATE bank
        SET kontostand = kontostand - $2
```

```
        WHERE konto_nr = $1;
    SELECT kontostand FROM bank WHERE konto_nr = $1;
' LANGUAGE SQL;
```

Diese Funktion belastet das Konto und gibt den neuen Kontostand zurück.

Jede Sammlung von SQL-Befehlen kann in einer Funktion zusammengefasst werden. Neben Anfragen mit `SELECT` können die Befehle auch Datenmodifizierungsbefehle (d.h. `INSERT`, `UPDATE` und `DELETE`) sein. Der letzte Befehl muss jedoch ein `SELECT` sein, das ein Ergebnis zurückgibt, das mit dem angegebenen Rückgabetyp der Funktion übereinstimmt. Wenn sie allerdings eine SQL-Funktion definieren wollen, die bestimmte Aktionen durchführt, aber keinen sinnvollen Wert zurückgibt, können Sie als Rückgabetyp void angeben. In diesem Fall darf der Funktionskörper nicht mit einem `SELECT` enden. Zum Beispiel:

```
CREATE FUNCTION mitarbeiter_löschen() RETURNS void AS '
    DELETE FROM mitarbeiter
        WHERE gehalt <= 0;
' LANGUAGE SQL;

SELECT mitarbeiter_löschen();

 mitarbeiter_löschen
---------------------

(1 row)
```

33.4.2 SQL-Funktionen mit zusammengesetzten Typen

Wenn man Funktionen mit Argumenten aus zusammengesetzten Typen schreibt, muss man nicht nur angeben, welches Argument man verwenden möchte (wie oben mit `$1` und `$2`), sondern auch welches Attribut des Arguments. Nehmen wir an, dass die Tabelle `mitarbeiter` Daten über die Mitarbeiter einer Organisation enthält. Gleichzeitig ist das auch der Name des zusammengesetzten Typs jeder Zeile der Tabelle. Hier ist eine Funktion `doppeltes_gehalt`, die berechnet, wie hoch das Gehalt wäre, wenn es verdoppelt würde:

```
CREATE TABLE mitarbeiter (
    name    text,
    gehalt  integer,
    alter   integer,
    büro    point
);

CREATE FUNCTION doppeltes_gehalt(mitarbeiter) RETURNS integer AS '
    SELECT $1.gehalt * 2 AS gehalt;
' LANGUAGE SQL;

SELECT name, doppeltes_gehalt(mitarbeiter) AS traum
    FROM mitarbeiter
    WHERE mitarbeiter.büro ~= point '(2,1)';
```

```
 name | traum
------+-------
 Sam  |  2400
```

Beachten Sie, wie mit der Syntax `$1.gehalt` ein Feld aus dem Zeilenwert des Arguments ausgewählt wird. Sie sehen auch, wie der `SELECT`-Befehl den Namen der Tabelle als Verweis auf die gesamte aktuelle Zeile dieser Tabelle verwendet.

Sie können aus einer Funktion auch einen zusammengesetzten Typ zurückgeben. Hier ist ein Beispiel für eine Funktion, die eine einzelne Zeile vom Typ mitarbeiter zurückgibt:

```
CREATE FUNCTION neuer_mitarbeiter() RETURNS mitarbeiter AS '
    SELECT text ''leer'' AS name,
        1000 AS gehalt,
        25 AS alter,
        point ''(2,2)'' AS büro;
' LANGUAGE SQL;
```

In diesem Fall haben wir für jedes Attribut einen konstanten Wert angegeben, aber jede Berechnung hätte anstelle dieser Konstanten stehen können.

Beachten Sie zwei wichtige Dinge bei der Definition der Funktion:

- Die Reihenfolge in der Select-Liste in der Anfrage muss genau die gleiche sein wie die Reihenfolge, in der die Spalten in der zu dem zusammengesetzten Typ gehörenden Tabelle angeordnet sind. (Die Namen der Spalten, die wir hier vergeben haben, sind für das System irrelevant.)
- Die Ausdrücke müssen in die Datentypen, die zur Definition des zusammengesetzten Typs passen, umgewandelt werden oder Sie werden Fehler wie diesen sehen:

  ```
  ERROR:  function declared to return mitarbeiter returns varchar instead of text
  at column 1
  ```

Eine Funktion, die eine Zeile (einen zusammengesetzten Typ) zurückgibt, kann als Tabellenfunktion verwendet werden, wie weiter unten beschrieben. Sie kann auch in einem SQL-Ausdruck aufgerufen werden, aber nur, wenn Sie ein einzelnes Attribut aus der Ergebniszeile herausziehen oder wenn Sie die gesamte Zeile einer anderen Funktion übergeben, die denselben zusammengesetzten Typ akzeptiert.

So kann man zum Beispiel ein Attribut aus der Zeile auswählen:

```
SELECT (neuer_mitarbeiter()).name;

 name
------
 leer
```

Die zusätzlichen Klammern werden benötigt, um Zweideutigkeiten bei der Syntaxanalyse auszuschließen:

```
SELECT neuer_mitarbeiter().name;
ERROR:  parser: parse error at or near "."
```

Eine andere Möglichkeit ist, die Funktionsschreibweise zum Herausziehen des Attributs zu verwenden. Das lässt sich so erklären, dass die Schreibweisen `attribut(tabelle)` und `tabelle.attribut` gleichbedeutend sind.

```
SELECT name(neuer_mitarbeiter());
```

```
 name
------
 leer
-- Dies ist das gleiche wie:
-- SELECT mitarbeiter.name AS jüngere FROM mitarbeiter WHERE mitarbeiter.alter <
30

SELECT name(mitarbeiter) AS jüngere
    FROM mitarbeiter
    WHERE alter(mitarbeiter) < 30;

 jüngere
---------
 Sam
```

Die andere Möglichkeit, eine Funktion mit einem Zeilentypergebnis zu verwenden, ist, eine zweite Funktion zu deklarieren, die den Zeilentyp als Argument nimmt, um dieser das Ergebnis der ersten Funktion zu übergeben:

```
CREATE FUNCTION get_name(mitarbeiter) RETURNS text AS
'SELECT $1.name;'
LANGUAGE SQL;

SELECT get_name(neuer_mitarbeiter());
 get_name
----------
 leer
(1 row)
```

33.4.3 SQL-Funktionen als Tabellenquelle

Alle SQL-Funktionen können in der `FROM`-Klausel einer Anfrage verwendet werden, aber besonders nützlich ist das für Funktionen, die zusammengesetzte Typen zurückgeben. Wenn die Funktion einen Basistyp zurückgibt, ergibt die Tabellenfunktion eine Tabelle mit eine Spalte. Wenn die Funktion einen zusammengesetzten Typ zurückgibt, ergibt die Tabellenfunktion eine Spalte für jedes Attribut des zusammengesetzten Typs.

```
Hier ist ein Beispiel:
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;
```

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(2 rows)
```

Wie dieses Beispiel zeigt, kann man mit den Spalten des Funktionsergebnisses genauso umgehen, als ob sie Spalten einer normalen Tabelle wären.

Beachten Sie, dass wir aus der Funktion nur eine Zeile erhalten haben, weil wir `SETOF` nicht verwendet haben. Das wird im nächsten Abschnitt beschrieben.

33.4.4 SQL-Funktionen mit Mengenergebnissen

Wenn der Rückgabetyp einer Funktion als `SETOF typ` angegeben ist, wird das letzte `SELECT` der Funktion bis zum Ende ausgeführt und jede Zeile, die die Anfrage liefert, wird als Element der Ergebnismenge zurückgegeben.

Diese Funktionalität wird normalerweise verwendet, wenn die Funktion in der `FROM`-Klausel aufgerufen wird. In dem Fall wird jede von der Funktion zurückgegebene Zeile eine Zeile der von der Anfrage gesehenen Tabelle. Nehmen wir zum Beispiel an, dass die Tabelle `foo` den gleichen Inhalt wie oben hat und wir Folgendes ausführen:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS '
    SELECT * FROM foo WHERE fooid = $1;
' LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Dann würden wird Folgendes erhalten:

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

Gegenwärtig können Funktionen, die Ergebnismengen zurückgeben, auch in der Select-Liste einer Anfrage aufgerufen werden. Für jede Zeile, die die Anfrage selbst erzeugt, wird die Funktion aufgerufen und für jedes Element in der Ergebnismenge der Funktion eine Ergebniszeile für die Anfrage erzeugt. Diese Funktionalität ist jedoch veraltet und könnte in einer zukünftigen Version entfernt werden. Hier ist ein Beispiel für eine Funktion, die eine Ergebnismenge aus der Select-Liste zurückgibt:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS
'SELECT name FROM nodes WHERE parent = $1'
LANGUAGE SQL;

SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
```

```
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

Beachten Sie, dass im letzten `SELECT`-Befehl keine Ausgabezeilen für `Child2`, `Child3` usw. auftauchen. Der Grund ist, dass `listchildren` für diese Argumente eine leere Menge zurückgibt und damit keine Ergebniszeilen erzeugt werden.

33.5 Funktionen in prozeduralen Sprachen

Prozedurale Sprachen sind nicht in den PostgreSQL-Server eingebaut; sie werden als ladbare Module angeboten. Bitte schauen Sie in der Beschreibung der jeweiligen prozeduralen Sprache nach, um Einzelheiten über die Syntax und die Interpretation des Funktionskörpers in der Sprache zu erhalten.

Es gibt gegenwärtig vier prozedurale Sprachen in der PostgreSQL-Standarddistribution: `PL/pgSQL`, `PL/Tcl`, `PL/Perl` und `PL/Python`. Weitere Sprachen können von Benutzern definiert werden. Weitere Informationen dazu finden Sie in Kapitel 37. Die Grundlagen der Entwicklung einer neuen prozeduralen Sprache werden in Abschnitt 33.9 besprochen.

33.6 Interne Funktionen

Interne Funktionen sind in C geschrieben und wurden statisch in den PostgreSQL-Server eingebunden. Der “Körper” der Funktionsdefinition gibt den C-Namen der Funktion an, welcher nicht der gleiche

Namen wie der Name der SQL-Funktion sein muss. (Der Rückwartskompatibilität halber bedeutet ein leerer Funktionskörper, dass der Name der C-Funktion der gleiche wie der der SQL-Funktion ist.)

Normalerweise werden alle im Server vorhandenen internen Funktionen bei der Initialisierung des Datenbankclusters (mit `initdb`) deklariert. Ein Benutzer könnte jedoch mit `CREATE FUNCTION` Aliasnamen für interne Funktionen erzeugen. Interne Funktionen werden in `CREATE FUNCTION` mit dem Sprachnamen `internal` deklariert. Zum Beispiel könnte man so einen Aliasnamen für die Funktion `sqrt` erzeugen:

```
CREATE FUNCTION quadratwurzel(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Die meisten internen Funktionen gehen davon aus, dass sie als “strikt” deklariert worden sind.)

Anmerkung

Nicht alle vordefinierten Funktionen sind in diesem Sinne “intern”. Einige vordefinierte Funktionen sind in SQL geschrieben.

33.7 C-Funktionen

Benutzerdefinierte Funktionen können in C geschrieben werden (oder einer Sprache, die mit C kompatibel gemacht werden kann, wie zum Beispiel C++). Solche Funktionen werden in dynamisch ladbare Objekte (dynamische Bibliotheken/*shared libraries*) kompiliert und vom Server bei Bedarf geladen. Das dynamische Laden unterscheidet die C-Funktionen von den “internen” Funktionen, die Quellcode-Schnittstelle ist im Prinzip dieselbe für beide. (Daher sind die internen Funktionen eine reichhaltige Quelle von Beispielen für benutzerdefinierte C-Funktionen.)

Gegenwärtig gibt es zwei unterschiedliche Aufrufskonventionen für C-Funktionen. Um die neuere Konvention, “Version 1”, auszuwählen, muss für die Funktion ein Aufruf des Makros `PG_FUNCTION_INFO_V1()` getätigt werden, wie unten gezeigt. Wenn dieser Makroaufruf fehlt, wird der alte Stil (“Version 0”) verwendet. Der in `CREATE FUNCTION` angegebene Sprachname ist in beiden Fällen `C`. Der alte Stil wird nicht mehr empfohlen, weil er Portabilitätsprobleme hat und ihm diverse Funktionalität fehlt, aber er wird der Kompatibilität halber weiter unterstützt.

33.7.1 Dynamisches Laden

Wenn eine benutzerdefinierte Funktion in einer bestimmten ladbaren Objektdatei zum ersten Mal in einer Sitzung aufgerufen wird, dann wird die Objektdatei vom dynamischen Lader in den Speicher geladen, damit die Funktion aufgerufen werden kann. Der Befehl `CREATE FUNCTION` muss für eine benutzerdefinierte C-Funktion also zwei Informationen angeben: der Name der ladbaren Objektdatei und der C-Name (das Link-Symbol) der in dieser Objektdatei aufzurufenden Funktion. Wenn der C-Name nicht ausdrücklich angegeben wird, dann wird angenommen, dass er der gleiche ist, wie der Name der Funktion auf SQL-Ebene.

Der folgende Algorithmus wird verwendet, um die dynamische Objektdatei auf Grundlage des im Befehl `CREATE FUNCTION` angegebenen Namen zu finden:

1. Wenn der Name ein absoluter Pfad ist, wird die angegebene Datei geladen.
2. Wenn der Name mit der Zeichenkette `$libdir` anfängt, wird dieser Teil durch den Namen des PostgreSQL-Paketverzeichnisses, welches beim Kompilieren festgelegt wird, ersetzt.

3. Wenn der Name keinen Verzeichnisteil enthält, wird die Datei im vom Konfigurationsparameter `dynamic_library_path` festgelegten Pfad gesucht.
4. Ansonsten (die Datei wurde nicht im Pfad gefunden oder sie enthält einen nicht absoluten Verzeichnisteil) wird versucht, die Datei mit dem angegebenen Namen zu laden, was wahrscheinlich scheitern wird. (Man kann sich nicht auf ein bestimmtes aktuelles Arbeitsverzeichnis verlassen.)

Wenn diese Schritte nicht funktionieren, wird die plattformspezifische Dateierweiterung für dynamische Bibliotheken (oft `.so`) an den angegebenen Namen angehängt und die Schritte erneut versucht. Wenn das auch nicht zum Erfolg führt, ist der Ladeversuch fehlgeschlagen.

Anmerkung

Die Benutzerkennung, unter der der PostgreSQL-Server läuft, muss auf alle Teile des Pfades zu der Datei, die Sie laden möchten, Zugriff haben. Ein häufiger Fehler ist es, dem `postgres`-Benutzer für ein höheres Verzeichnis keine Lese- und/oder Ausführrechte zu geben.

Auf jeden Fall wird der Dateiname, der im Befehl `CREATE FUNCTION` angegeben wird, unverändert abgespeichert. Wenn also die Datei erneut geladen werden muss, wird der selbe Vorgang durchgeführt.

Anmerkung

PostgreSQL kompiliert die C-Funktion nicht automatisch. Die Objektdatei muss kompiliert werden, bevor Sie im Befehl `CREATE FUNCTION` angegeben werden kann. Weitere Informationen dazu finden Sie in Abschnitt 33.7.6.

Anmerkung

Eine dynamisch geladene Objektdatei wird nach der ersten Verwendung im Hauptspeicher behalten. Wenn die Funktion(en) in derselben Sitzung noch einmal aufgerufen werden, entsteht dadurch nur ein geringer Aufwand bei der Suche in der Symboltabelle. Wenn Sie eine Objektdatei zwingend neu laden wollen, zum Beispiel nachdem Sie sie neu kompiliert haben, können Sie den Befehl `LOAD` verwenden oder eine neue Sitzung beginnen.

Es wird empfohlen, dynamische Bibliotheken entweder relativ zu `$libdir` abzulegen oder durch den dynamischen Bibliothekspfad finden zu lassen. Dadurch vereinfacht sich der Umstieg auf eine neue Version, wenn diese an einer anderen Stelle abgelegt ist. Für welches Verzeichnis `$libdir` tatsächlich steht, können Sie mit dem Befehl `pg_config --pkglibdir` herausfinden.

Anmerkung

Vor PostgreSQL Version 7.2 konnten in CREATE FUNCTION nur genaue, absolute Pfade zu Objektdateien angegeben werden. Diese Methode wird jetzt nicht mehr empfohlen, da sie die Funktionsdefinition unnötigerweise unportierbar macht. Am besten gibt man nur den Namen der dynamischen Bibliothek ohne Pfad und ohne Erweiterung an und lässt den Suchmechanismus den Rest erledigen.

33.7.2 Basistypen in C-Funktionen

Um zu wissen, wie man C-Funktionen schreibt, müssen Sie wissen, wie PostgreSQL Basisdatentypen intern darstellt und wie sie in und aus Funktionen übergeben werden können. Intern betrachtet PostgreSQL einen Basistyp einfach als ein Stück Speicher. Die benutzerdefinierten Funktionen, die Sie für einen Typ definieren, bestimmen dann, wie PostgreSQL mit diesem Typ umgehen kann. Das heißt, dass PostgreSQL die Daten nur speichert und wieder von der Festplatte liest und dann Ihre benutzerdefinierten Funktionen verwendet, um die Daten einzugeben, zu verarbeiten und auszugeben.

Basistypen können drei verschiedene interne Formate haben:

- Wertübergabe, feste Länge
- Referenzübergabe, feste Länge
- Referenzübergabe, variable Länge

Typen mit Wertübergabe können nur 1, 2 oder 4 Bytes lang sein (oder auch 8 Bytes, wenn auf Ihrer Maschine `sizeof(Datum)` 8 ist). Sie sollten darauf achten, dass Ihre Typen auf allen Architekturen die gleiche Größe (in Bytes) haben. Der Typ long wäre zum Beispiel gefährlich, weil er auf manchen Maschinen 4 Bytes und auf anderen 8 Bytes groß ist. Der Typ int ist jedoch auf den meisten Unix-Maschinen 4 Bytes groß. Eine sinnvolle Implementierung des Typs int4 auf Unix-Maschinen wäre zum Beispiel:

```
/* 4-Byte ganze Zahl, Wertübergabe */
typedef int int4;
```

Typen mit Referenzübergabe können andererseits beliebige feste Längen haben. Hier ist zum Beispiel eine Implementierung eines PostgreSQL-Typs:

```
/* 16-Byte Struktur, Referenzübergabe */
typedef struct
{
    double  x, y;
} Point;
```

Solche Typen können nur über Zeiger in und aus PostgreSQL-Funktionen übergeben werden. Um einen Wert eines solchen Typs zurückzugeben, teilen Sie mit `palloc` entsprechend viel Speicher zu, füllen Sie den Speicher und geben Sie einen Zeiger darauf zurück. (Sie können auch einen Eingabewert, der den gleichen Typ wie der Rückgabewert hat, direkt zurückgeben, indem Sie den Zeiger auf den Eingabewert zurückgeben. Verändern Sie jedoch *niemals* den Inhalt eines Eingabewerts mit Referenzübergabe.)

Schließlich müssen auch alle Typen mit variabler Länge als Referenz übergeben werden. Alle Typen mit variabler Länge müssen mit einem Längenfeld beginnen, das 4 Bytes groß ist, und alle zu dem Wert gehörenden Daten müssen im Speicher unmittelbar nach diesem Längenfeld abgelegt werden. Das Längenfeld enthält die Gesamtlänge der Struktur, das heißt, es schließt die Länge des Längenfelds selbst mit ein.

Als Beispiel können wir den Typ text folgendermaßen definieren:

```
typedef struct {
       int4 length;
       char data[1];
} text;
```

Natürlich ist das hier deklarierte Datenfeld nicht lang genug, um alle möglichen Zeichenketten aufzunehmen. Da es aber in C nicht möglich ist, eine Struktur mit variabler Länge zu deklarieren, verlassen wir uns darauf, dass der C-Compiler die Arraygrenzen nicht überprüft. Wir teilen einfach den benötigten Speicherplatz zu und greifen dann auf das Array zu, als ob es mit der richtigen Länge deklariert worden wäre. (Das ist ein bekannter Trick, über den Sie in vielen Lehrbüchern über C nachlesen können.)

Wenn man Typen mit variabler Länge manipuliert, dann muss man darauf achten, dass man die richtige Menge Speicher zuteilt und das Längenfeld korrekt setzt. Wenn wir zum Beispiel 40 Bytes in einer `text`-Struktur speichern wollen, können wir ein Codestück wie das Folgende verwenden:

```
#include "postgres.h"
...
char buffer[40]; /* unsere Quelldaten */
```

```
...
text *ziel = (text *) palloc(VARHDRSZ + 40);
ziel->length = VARHDRSZ + 40;
memcpy(ziel->data, buffer, 40);
...
```

VARHDRSZ ist das Gleiche wie `sizeof(int4)`, aber es ist ein besserer Stil das Makro VARHDRSZ zu verwenden, um die Größe des Längenfeldes eines Typs mit variabler Länge anzugeben.

Tabelle 33.1 gibt an, welcher C-Typ welchem SQL-Typ entspricht, wenn man eine C-Funktion schreiben möchte, die einen in PostgreSQL eingebauten Datentyp verwendet. Die Spalte “Definiert in” gibt an, welche Headerdatei eingebunden werden muss, um an die Definition des Typs zu kommen. (Die eigentliche Definition könnte auch in einer anderen Datei sein, die von der gelisteten eingebunden wird. Es wird aber empfohlen, sich an die definierte Schnittstelle zu halten.) Beachten Sie, dass Sie in jeder Quelldatei immer zuerst `postgres.h` einbinden sollten, da dort eine Reihe von Sachen deklariert werden, die Sie sowieso benötigen.

SQL-Typ	C-Typ	Definiert In
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h **(eventuell im Compiler eingebaut)**
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	**(im Compiler eingebaut)**
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 **or** int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 **or** int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	utils/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h

Tabelle 33.1: Äquivalente C-Typen für eingebaute SQL-Typen

SQL-Typ	C-Typ	Definiert In
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	utils/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Tabelle 33.1: Äquivalente C-Typen für eingebaute SQL-Typen (Forts.)

Jetzt, da wir alle möglichen Strukturen für Basistypen kennen, können wir einige Beispiele für richtige Funktionen zeigen.

33.7.3 Aufrufskonvention Version 0 für C-Funktionen

Als Erstes zeigen wir den “alten Stil” der C-Funktionen. Obwohl dieser Stil nicht mehr empfohlen wird, ist er am Anfang einfacher zu verstehen. In der Version-0-Methode werden die Argumente und der Rückgabetyp im normalen C-Stil deklariert, wobei man darauf achten muss, den richtigen C-Typ für den entsprechenden SQL-Typ zu verwenden, wie oben gezeigt.

Hier sind einige Beispiele:

```
#include "postgres.h"
#include <string.h>

/* Wertübergabe */

int
eins_addieren(int arg)
{
    return arg + 1;
}

/* Referenzübergabe, feste Länge */

float8 *
eins_addieren_float8(float8 *arg)
{
    float8    *ergebnis = (float8 *) palloc(sizeof(float8));

    *ergebnis = *arg + 1.0;

    return ergebnis;
}

Point *
makepoint(Point *pointx, Point *pointy)
```

```
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* Referenzübergabe, variable Länge */

text *
text_kopieren(text *t)
{
    /*
     * VARSIZE ist die Gesamtgröße der Struktur in Bytes.
     */
    text *neu_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(neu_t) = VARSIZE(t);
    /*
     * VARDATA ist ein Zeiger auf den Datenbereich der Struktur.
     */
    memcpy((void *) VARDATA(neu_t), /* Ziel */
           (void *) VARDATA(t),     /* Quelle */
           VARSIZE(t)-VARHDRSZ);    /* wieviele Bytes */
    return neu_t;
}

text *
text_verknuepfen(text *arg1, text *arg2)
{
    int32 neue_text_groesse = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *neu_text = (text *) palloc(neue_text_groesse);

    VARATT_SIZEP(neu_text) = neue_text_groesse;
    memcpy(VARDATA(neu_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(neu_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    return neu_text;
}
```

Nehmen wir an, dass der obige Code in einer Datei namens `funktionen.c` steht und in eine dynamische Bibliothek kompiliert wurde. Dann könnten wir diese Funktionen in PostgreSQL mit folgenden Befehlen definieren:

```
CREATE FUNCTION eins_addieren(integer) RETURNS integer
     AS 'VERZEICHNIS/funktionen', 'eins_addieren'
```

```
     LANGUAGE C STRICT;

-- Beachten Sie, dass der SQL-Funktionsname "eins_addieren" hier überladen wird.
CREATE FUNCTION eins_addieren(double precision) RETURNS double precision
     AS 'VERZEICHNIS/funktionen', 'eins_addieren_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'VERZEICHNIS/funktionen', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION text_kopieren(text) RETURNS text
     AS 'VERZEICHNIS/funktionen', 'text_kopieren'
     LANGUAGE C STRICT;

CREATE FUNCTION text_verknüpfen(text, text) RETURNS text
     AS 'VERZEICHNIS/funktionen', 'text_verknuepfen',
     LANGUAGE C STRICT;
```

Hier steht *VERZEICHNIS* für das Verzeichnis mit der dynamischen Bibliotheksdatei. (Besser wäre es, in der AS-Klausel einfach `'funktionen'` zu schreiben, nachdem man *VERZEICHNIS* zum Suchpfad hinzugefügt hat. Die Dateierweiterung für dynamische Bibliotheken, meistens `.so` oder `.sl`, kann in jedem Fall weggelassen werden.)

Beachten Sie, dass wir die Funktionen mit dem Attribut `STRICT` definiert haben, was bedeutet, dass das System automatisch den NULL-Wert als Rückgabewert annimmt, wenn irgendein Argument den NULL-Wert hat. Dadurch müssen wir im Funktionscode die Argumente nicht auf NULL-Werte überprüfen. Ohne dieses Attribut müsste man die Argumente auf NULL-Werte prüfen, indem man bei jedem Argument mit Referenzübergabe prüft, ob ein NULL-Zeiger übergeben wurde. (Bei Argumenten mit Wertübergabe gibt es gar keine Möglichkeit, um NULL-Werte zu entdecken.)

Obwohl diese Aufrufskonvention einfach zu verwenden ist, ist sie nicht sehr portabel; auf einigen Architekturen gibt es Probleme bei der Übergabe von Typen, die kleiner als int sind. Außerdem gibt es keinen einfachen Weg, einen NULL-Wert zurückzugeben und auch keinen Weg, mit NULL-Argumenten umzugehen, außer die Funktion als `STRICT` zu deklarieren. Die Aufrufskonvention Version 1, die als Nächstes beschrieben wird, beseitigt diese Mängel.

33.7.4 Aufrufskonvention Version 1 für C-Funktionen

Die Aufrufskonvention Version 1 verwendet Makros, um die Komplexität der Argument- und Ergebnisübergabe zu verstecken. Die C-Deklaration einer Version-1-Funktion ist immer

```
Datum funktionsname(PG_FUNCTION_ARGS)
```

Zusätzlich muss der Makroaufruf

```
PG_FUNCTION_INFO_V1(funktionsname);
```

in derselben Quellcodedatei getätigt werden. (Üblicherweise wird er direkt vor die Funktion selbst geschrieben.) Für Funktionen in der Sprache `internal` wird dieser Makroaufruf nicht benötigt, da PostgreSQL annimmt, dass alle internen Funktionen die Version-1-Konvention verwenden. Für dynamisch geladene Funktionen ist er jedoch notwendig.

In einer Funktion, die die Version-1-Konvention verwendet, wird jedes Argument mit einem dem Datentyp des Arguments entsprechenden Makro `PG_GETARG_`*xxx*`()` ermittelt und das Ergebnis wird mit einem dem Rückgabetyp entsprechenden Makro `PG_RETURN_`*xxx*`()` zurückgegeben. `PG_GETARG_`*xxx*`()` hat als sein Argument die Nummer des zu ermittelnden Funktionsarguments, wobei die Zählung bei 0 anfängt. `PG_RETURN_`*xxx*`()` hat als sein Argument den zurückzugebenden Wert.

Hier zeigen wir dieselben Funktionen wie oben, aber unter Verwendung des Version-1-Stils:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* Wertübergabe */

PG_FUNCTION_INFO_V1(eins_addieren);

Datum
eins_addieren(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* Referenzübergabe, feste Länge */

PG_FUNCTION_INFO_V1(eins_addieren_float8);

Datum
eins_addieren_float8(PG_FUNCTION_ARGS)
{
    /* Die Makros für FLOAT8 verstecken die Referenzübergabe. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Die Referenzübergabe von Point wird hier nicht versteckt. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
```

```
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* Referenzübergabe, variable Länge */

PG_FUNCTION_INFO_V1(text_kopieren);

Datum
text_kopieren(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE ist die Gesamtgröße der Struktur in Bytes.
     */
    text     *neu_t = (text *) palloc(VARSIZE(t));
    VARATT_SIZEP(neu_t) = VARSIZE(t);
    /*
     * VARDATA ist ein Zeiger auf die Datenregion der Struktur.
     */
    memcpy((void *) VARDATA(neu_t), /* Ziel */
           (void *) VARDATA(t),     /* Quelle */
           VARSIZE(t)-VARHDRSZ);    /* wieviele Bytes */
    PG_RETURN_TEXT_P(neu_t);
}

PG_FUNCTION_INFO_V1(text_verknuepfen);

Datum
text_verknüpfen(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 neue_text_groesse = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *neu_text = (text *) palloc(neue_text_groesse);

    VARATT_SIZEP(neu_text) = neue_text_groesse;
    memcpy(VARDATA(neu_text), VARDATA(arg1), VARSIZE(arg1)-VARHDRSZ);
    memcpy(VARDATA(neu_text) + (VARSIZE(arg1)-VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
    PG_RETURN_TEXT_P(neu_text);
}
```

Die `CREATE FUNCTION`-Befehle sind dieselben wie bei der Version 0.

Auf den ersten Blick mag die Version-1-Aufrufskonvention wie sinnloser Obskurantismus aussehen. Sie bietet jedoch eine Reihe von Verbesserungen, weil die Makros unnütze Details verstecken. Ein Beispiel ist, dass wir in der Funktion `eins_addieren_float8` nicht mehr wissen müssen, dass der Typ float8 Referenzübergabe verwendet. Ein weiteres Beispiel ist, dass die `GETARG`-Makros für Typen mit variabler Länge ein effizienteres Auslesen von "getoasteten" (komprimierten oder außerhalb des Heaps gespeicherten) Werten ermöglichen.

Ein große Verbesserung der Version-1-Konvention ist der bessere Umgang mit Argumenten und Ergebnissen mit dem NULL-Wert. Mit dem Makro `PG_ARGISNULL(n)` kann getestet werden, ob ein Argument der NULL-Wert ist. (Das ist natürlich nur notwendig, wenn die Funktion nicht als "strikt" deklariert wurde.) Wie bei den `PG_GETARG_`*`xxx`*`()`-Makros fängt die Zählung bei 0 an. Beachten Sie, dass man `PG_GETARG_`*`xxx`*`()` erst aufrufen sollte, nachdem man sich versichert hat, dass das Argument nicht der NULL-Wert ist. Um einen NULL-Wert als Ergebnis zurückzugeben, führen Sie `PG_RETURN_NULL()` aus; das funktioniert bei strikten und bei nicht strikten Funktionen.

Weitere Möglichkeiten, die die neue Schnittstelle bietet, sind zwei Varianten der `PG_GETARG_`*`xxx`*`()`-Makros. Die erste Variante, `PG_GETARG_`*`xxx`*`_COPY()`, garantiert, dass Sie eine Kopie des angegebenen Arguments erhalten, in die Sie problemlos schreiben können. (Die normalen Makros geben manchmal einen Zeiger auf den direkt in der Tabelle gespeicherten Wert zurück, der nicht überschrieben werden darf. Mit `PG_GETARG_`*`xxx`*`_COPY()` erhalten Sie garantiert ein beschreibbares Ergebnis.) Die zweite Variante sind die `PG_GETARG_`*`xxx`*`_SLICE()`-Makros, welche drei Argumente haben. Das erste ist die Nummer des Funktionsarguments (wie oben). Das zweite und dritte sind der Anfang und die Länge des zu ermittelnden Segments. Der Anfangswert zählt von 0 an und eine negative Länge gibt an, dass der Rest des Werts zurückgegeben werden soll. Diese Makros ermöglichen effizienteren Zugriff auf Teile von großen Werten, die den Speichertyp external haben. (Der Speichertyp einer Spalte kann mit dem Befehl `ALTER TABLE` *`tabellenname`* `ALTER COLUMN` *`spaltenname`* `SET STORAGE` *`speichertyp`* angegeben werden. *`speichertyp`* muss einer der Werte `plain`, `external`, `extended` oder `main` sein.)

Schließlich können Funktionen, die die Aufrufskonvention Version 1 verwenden, Ergebnismengen zurückgeben (Abschnitt 33.7.9) und es ist mit der Version-1-Konvention möglich, Triggerfunktionen (Kapitel 35) und Handler für prozedurale Sprachen (Abschnitt 33.9) zu schreiben. Version-1-Code ist außerdem portierbarer als Version-0-Code, weil er nicht das Funktionsaufrufsprotokoll aus dem C-Standard verletzt. Einzelheiten dazu finden Sie in `src/backend/utils/fmgr/README` in der Quellendistribution.

33.7.5 Code schreiben

Bevor wir uns fortgeschritteneren Themen zuwenden, sollten wir einige Regeln für den Code von C-Funktionen für PostgreSQL besprechen. Obwohl es möglich sein könnte, in anderen Sprachen geschriebene Funktionen in PostgreSQL zu laden, ist das meistens schwierig (wenn überhaupt möglich), weil andere Sprachen, wie C++, FORTRAN oder Pascal, oft eine andere Aufrufskonvention als C verwenden. Das heißt, dass andere Sprachen Argumente und Rückgabewerte zwischen Funktionen nicht auf dieselbe Weise wie C übergeben. Aus diesem Grund gehen wir davon aus, dass Ihre C-Funktionen auch tatsächlich in C geschrieben werden.

Die grundlegenden Regeln, um C-Funktionen zu schreiben und zu bauen, sind folgende:

- Verwenden Sie `pg_config --includedir-server`, um herauszufinden, wo die PostgreSQL-Server-Headerdateien auf Ihrem System (oder auf dem System Ihrer Anwender) installiert sind. Diese Option existiert seit PostgreSQL 7.2. Für PostgreSQL 7.1 sollten Sie die Option `--includedir` verwenden. (`pg_config` gibt einen von 0 verschiedenen Status zurück, wenn es eine unbekannte Option entdeckt.) Für Versionen vor 7.1 müssen Sie raten, aber da das vor der Einführung der aktuellen Aufrufskonvention war, sind Sie wahrscheinlich an solchen Versionen sowieso nicht mehr interessiert.
- Wenn Sie Speicher zuteilen, verwenden Sie die PostgreSQL-Funktionen `palloc` und `pfree` statt der entsprechenden C-Bibliotheksfunktionen `malloc` und `free`. Der von `palloc` zugeteilte Speicher wird automatisch am Ende der Transaktion freigegeben, womit Speicherlecks verhindert werden.

- Setzen Sie die Bytes aller Strukturen immer zuerst auf null zurück, etwa mit `memset` oder `bzero`. Mehrere Routinen (wie die Hash-Zugriffsmethode, Hash-Verbunde und der Sortieralgorithmus) berechnen Funktionen direkt aus den Bits in Ihren Strukturen. Selbst wenn Sie alle Felder in Ihren Strukturen initialisieren, kann es trotzdem mehrere Bytes zur Ausrichtungsbegleichung (Löcher in der Struktur) geben, die ansonsten Müll enthalten.
- Die meisten internen PostgreSQL-Typen sind in der Datei `postgres.h` deklariert, während die Funktionsmanager-Schnittstellen (*PG_FUNCTION_ARGS* usw.) in `fmgr.h` deklariert sind. Also müssen Sie mindestens diese beiden Dateien einbinden. Der Portierbarkeit wegen binden Sie am besten `postgres.h` *zuerst*, vor allen System- oder Benutzer-Headerdateien, ein. Durch `postgres.h` wird auch automatisch `elog.h` und `palloc.h` eingebunden.
- Die in den Objektdateien definierten Symbolnamen dürfen keine Konflikte mit anderen Objektdateien oder den im PostgreSQL-Server selbst definierten Symbolen erzeugen. Wenn Sie Fehlermeldungen erhalten, die darauf hindeuten, dann müssen Sie Ihre Funktionen oder Variablen umbenennen.
- Das Kompilieren und Linken Ihres Codes, damit er dynamisch in den PostgreSQL-Server geladen werden kann, erfordert immer besondere Optionen. In Abschnitt 33.7.6 steht geschrieben, wie genau es auf Ihrem Betriebssystem funktioniert.

33.7.6 Kompilieren und Linken von dynamisch ladbaren Funktionen

Bevor Sie Ihre in C geschriebenen PostgreSQL-Erweiterungsfunktionen verwenden können, müssen diese auf eine besondere Art und Weise kompiliert und gelinkt werden, um eine Datei zu erzeugen, die vom Server dynamisch geladen werden kann. Diese Art von Datei heißt dynamische Bibliothek (oder *shared library*).

Um Informationen zu erhalten, die über den Inhalt dieses Abschnitts hinausgehen, sollten Sie die Dokumentation Ihres Betriebssystem lesen, insbesondere die Anleitungen des C-Compilers, `cc`, und des Link-Editors, `ld`. Außerdem enthält der PostgreSQL-Quellcode mehrere funktionierende Beispiele im `contrib`-Verzeichnis. Wenn Sie sich auf diese Beispiele verlassen, werden Ihre Module jedoch von der Verfügbarkeit des PostgreSQL-Quellcodes abhängen.

Das Erzeugen von dynamischen Bibliotheken ist im Allgemeinen analog dem Erzeugen von ausführbaren Programmen: Zuerst werden die Quelldateien in Objektdateien kompiliert, dann werden die Objektdateien zusammengelinkt. Die Objektdateien müssen als *positionsunabhängiger* Code (*position-independent code*, PIC) erzeugt werden, was im Prinzip bedeutet, dass sie an einer beliebigen Stelle im Speicher abgelegt werden können, wenn Sie von einem Programm geladen werden. (Objektdateien, die für normale Programme gedacht sind, werden nicht auf diese Art kompiliert.) Der Befehl, der die dynamische Bibliothek linkt, enthält besondere Optionen, um ihn vom Linken eines normalen Programms zu unterscheiden. – Das ist zumindest die Theorie. Auf einigen Systemen ist die Praxis viel grausamer.

Im den folgenden Beispielen gehen wir davon aus, dass Ihr Quellcode in einer Datei `foo.c` ist und dass wir eine dynamische Bibliothek `foo.so` erzeugen werden. Die Objektdatei zwischendurch wird `foo.o` heißen, wenn nichts anderes angegeben ist. Eine dynamische Bibliotheksdatei kann mehrere Objektdateien enthalten, aber wir verwenden hier nur eine.

BSD/OS

Die Compileroption, um PIC zu erzeugen, ist `-fpic`. Die Linkeroption, um eine dynamische Bibliothek zu erzeugen, ist `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

Dies gilt ab Version 4.0 von BSD/OS.

FreeBSD

Die Compileroption, um PIC zu erzeugen, ist `-fpic`. Um eine dynamische Bibliothek zu erzeugen, ist die Compileroption `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Dies gilt ab Version 3.0 von FreeBSD.

HP-UX

Die Compileroption des Systemcompilers, um PIC zu erzeugen, ist `+z`. Wenn Sie GCC verwenden, ist sie `-fpic`. Die Linkeroption für dynamische Bibliotheken ist `-b`. Also

```
cc +z -c foo.c
```

oder

```
gcc -fpic -c foo.c
```

und dann

```
ld -b -o foo.sl foo.o
```

HP-UX verwendet im Gegensatz zu den meisten anderen Systemen die Dateierweiterung `.sl` für dynamische Bibliotheken.

IRIX

PIC ist die Standardeinstellung, keine besonderen Compileroptionen sind notwendig. Die Linkeroption, um dynamische Bibliotheken zu erzeugen, ist `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

Die Compileroption, um PIC zu erzeugen, ist `-fpic`. Auf einigen Plattformen muss in einigen Situationen `-fPIC` verwendet werden, wenn `-fpic` nicht funktioniert. Weitere Informationen dazu finden Sie in der GCC-Anleitung. Die Compileroption, um eine dynamische Bibliothek zu erzeugen, ist `-shared`. Ein vollständiges Beispiel sieht so aus:

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

MacOS X

Hier ist Beispiel. Es setzt voraus, dass alle Entwicklerwerkzeuge installiert sind.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

Die Compileroption, um PIC zu erzeugen, ist `-fpic`. Auf ELF-Systemen wird der Compiler mit der Option `-shared` verwendet, um dynamische Bibliotheken zu linken. Auf den älteren Nicht-ELF-Systemen wird `ld -Bshareable` verwendet.

```
gcc -fpic -c foo.c
```

```
gcc -shared -o foo.so foo.o
```

OpenBSD

Die Compileroption um PIC zu erzeugen ist `-fpic. ld -Bshareable` wird zum Linken von dynamischen Bibliotheken verwendet.

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

Die Compileroption, um PIC zu erzeugen, ist `-KPIC` mit dem Sun-Compiler und `-fpic` mit GCC. Um dynamische Bibliotheken zu linken ist die Compileroption entweder `-G` für beide Compiler oder `-shared` mit GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

oder

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC ist die Voreinstellung, also ist der Compiler-Befehl der übliche. ld mit besonderen Optionen wird zum Linken verwendet:

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

Die gleichen Befehle werden verwendet, wenn GCC anstatt des Systemcompilers verwendet wird; keine besonderen Optionen sind nötig.

UnixWare

Die Compileroption, um PIC zu erzeugen, ist `-K PIC` mit dem SCO-Compiler und `-fpic` mit GCC. Um dynamische Bibliotheken zu linken, ist die Compileroption entweder `-G` für den SCO-Compiler und `-shared` mit GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

oder

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Tipp

Wenn Ihnen das zu kompliziert ist, sollten Sie die Verwendung von *GNU Libtool* in Erwägung ziehen, das die Plattformunterschiede hinter einer einheitlichen Schnittstelle verbirgt.

Die dynamische Bibliothek, die sich aus diesem Vorgang ergibt, kann in den PostgreSQL-Server geladen werden. Wenn der Dateiname im Befehl `CREATE FUNCTION` angegeben wird, muss das der Name der dynamische Bibliotheksdatei sein, nicht der der zwischendurch entstandenen Objektdatei. Beach-

ten Sie, dass die für das jeweilige System normale Dateierweiterung für dynamischen Bibliotheken (üblicherweise `.so` oder `.sl`) im Befehl `CREATE FUNCTION` weggelassen werden kann und normalerweise der besseren Portierbarkeit wegen auch weggelassen werden sollte.

Oben in Abschnitt 33.7.1 finden Sie Informationen darüber, wo der Server die dynamischen Bibliotheken zu finden versucht.

33.7.7 Argumente aus zusammengesetzten Typen in C-Funktionen

Zusammengesetzte Typen haben kein festes Layout wie eine C-Struktur. Zusammengesetzte Typen können NULL-Werte in Feldern enthalten. Außerdem können zusammengesetzte Typen, die Teil einer Vererbungshierarchie sind, andere Felder haben als andere Mitglieder derselben Vererbungshierarchie. Daher bietet PostgreSQL eine Schnittstelle mit Funktionen für den Zugriff auf die Felder zusammengesetzter Typen in C.

Angenommen, wir wollen eine Funktion für folgende Anfrage schreiben:

```
SELECT name, ueberbezahlt(mitarbeiter, 1500)
    FROM mitarbeiter
    WHERE name = 'Bill' OR name = 'Sam';
```

Mit der Aufrufskonvention Version 0 kann die Funktion `ueberbezahlt` so definiert werden:

```
#include "postgres.h"
#include "executor/executor.h"  /* für GetAttributeByName() */

bool
ueberbezahlt(TupleTableSlot *t, /* aktuelle Zeile von "mitarbeiter" */
           int32 limit)
{
    bool ist_null;
    int32 gehalt;

    salary = DatumGetInt32(GetAttributeByName(t, "gehalt", &ist_null));
    if (ist_null)
        return false;
    return gehalt > limit;
}
```

In Version 1 würde diese Funktion so aussehen:

```
#include "postgres.h"
#include "executor/executor.h"  /* für GetAttributeByName() */

PG_FUNCTION_INFO_V1(ueberbezahlt);

Datum
ueberbezahlt(PG_FUNCTION_ARGS)
{
```

```
    TupleTableSlot  *t = (TupleTableSlot *) PG_GETARG_POINTER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool ist_null;
    int32 gehalt;

    gehalt = DatumGetInt32(GetAttributeByName(t, "gehalt", &ist_null));
    if (ist_null)
        PG_RETURN_BOOL(false);
    /* Man könnte in diesem Fall vielleicht auch PG_RETURN_NULL() verwenden. */

    PG_RETURN_BOOL(gehalt > limit);
}
```

`GetAttributeByName` ist die PostgreSQL-Systemfunktion, die Attribute (Spaltenwerte) aus der angegebenen Zeile ermittelt. Sie hat drei Argumente: das der Funktion übergebene Argument vom Typ TupleTableSlot*, der Name des gewünschten Attributs und ein Parameter, in dem zurückgegeben wird, ob das Attribut den NULL-Wert hat. `GetAttributeByName` gibt einen Wert vom Typ Datum zurück, den Sie mit den Makros `DatumGet`*`XXX`*`()` in den richtigen Datentyp umwandeln können.

Der folgende Befehl deklariert die Funktion `ueberbezahlt` in SQL:

```
CREATE FUNCTION ueberbezahlt(mitarbeiter, integer)
    RETURNS boolean
    AS 'DIRECTORY/funcs', 'ueberbezahlt'
     LANGUAGE C;
```

33.7.8 Rückgabe von Zeilen (zusammengesetzten Typen) aus C-Funktionen

Um eine Zeile oder einen zusammengesetzten Typ aus einer C-Funktion zurückzugeben, können Sie eine spezielle API verwenden, die Makros und Funktionen anbietet, die einen Großteil der Komplexität beim Zusammenbauen von zusammengesetzten Datentypen verstecken. Um diese API verwenden zu können, muss die Quellcodedatei Folgendes enthalten:

```
#include "funcapi.h"
```

Die Unterstützung für das Zurückgeben von zusammengesetzten Datentypen (oder Zeilen) fängt mit der Struktur `AttInMetadata` an. Diese Struktur enthält Arrays mit Informationen über die Attribute, die benötigt werden, um eine Zeile aus einfachen C-Zeichenketten aufzubauen. Die Informationen in dieser Struktur werden aus einer `TupleDesc`-Struktur abgeleitet, werden aber hier gespeichert, um im Falle einer Funktion mit Ergebnismenge (siehe nächster Abschnitt), wiederholte Berechnungen derselben Informationen bei jedem Aufruf zu vermeiden. Wenn eine Funktion eine Ergebnismenge zurückgibt, sollte die Struktur `AttInMetadata` einmal berechnet und dann für die Wiederverwendung in den folgenden Aufrufen gespeichert werden. `AttInMetadata` speichert auch einen Zeiger auf die ursprüngliche `TupleDesc`-Struktur.

```
typedef struct AttInMetadata
{
    /* voller TupleDesc */
    TupleDesc       tupdesc;
```

```
    /* Array mit "finfo" für Typeingabefunktion */
    FmgrInfo       *attinfuncs;

    /* Array mit "typelem" für Attributtyp */
    Oid            *attelems;

    /* Array mit "typmod" für Attributtyp */
    int32          *atttypmods;
}       AttInMetadata;
```

Um diese Struktur zu füllen, können Ihnen mehrere Funktionen und Makros helfen. Verwenden Sie

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

um ein `TupleDesc` für die benannte Relation zu erhalten, oder

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

um ein `TupleDesc` anhand einer Typ-OID zu erhalten. Damit können Sie einen `TupleDesc` für Basistypen und für zusammengesetzte (Tabellen-) Typen erhalten. Danach ergibt

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

einen Zeiger auf ein `AttInMetadata`, das anhand des angegebenen `TupleDesc` initialisiert wurde. `AttInMetadata` kann zusammen mit C-Zeichenketten verwendet werden, um einen richtigen Zeilenwert (intern Tupel genannt) zu konstruieren.

Um ein Tupel zurückzugeben, müssen Sie einen Tuple-Slot auf Grundlage des `TupleDesc` erzeugen. Sie können einen Tuple-Slot mit

```
TupleTableSlot *TupleDescGetSlot(TupleDesc tupdesc)
```

initialisieren oder einen auf einem anderen (benutzerdefinierten) Weg erhalten. Der Tuple-Slot wird benötigt, um ein Datum zu erzeugen, das von der Funktion zurückgegeben werden kann. Derselbe Slot kann (und sollte) bei jedem Aufruf wiederverwendet werden.

Nachdem Sie eine `AttInMetadata`-Struktur erzeugt haben, können Sie mit

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

ein `HeapTuple` anhand von Benutzerdaten in C-Zeichenkettenform erzeugen. `values` ist ein Array aus C-Zeichenketten, eine pro Attribut in der zurückzugebenden Zeile. Jede C-Zeichenkette sollte in der Form sein, die die Eingabefunktion des Attributdatentyps erwartet. Wenn eines der Attribute den NULL-Wert haben soll, dann sollte der entsprechende Zeiger im Array `values` auf *NULL* gesetzt werden. Diese Funktion müssen Sie für jede Zeile, die Sie zurückgeben wollen, neu aufrufen.

Ein Tupel mit `TupleDescGetAttInMetadata` und `BuildTupleFromCStrings` aufzubauen, ist nur bequem, wenn Ihre Funktion die zurückzugebenden Werte sowieso als Zeichenketten berechnet. Wenn Ihr Code die Werte aber als Datum-Werte berechnet, sollten Sie die Funktion `heap_formtuple` verwenden (die auch `BuildTupleFromCStrings` zugrunde liegt), um die Datum-Werte direkt in ein Tupel umzuwandeln. Sie benötigen trotzdem einen `TupleDesc` und einen `TupleTableSlot`, aber nicht `AttInMetadata`.

Wenn Sie ein Tupel erzeugt haben, das Sie aus Ihrer Funktion zurückgeben wollen, muss dieses in ein Datum umgewandelt werden. Dazu verwenden Sie

```
TupleGetDatum(TupleTableSlot *slot, HeapTuple tuple)
```

unter Angabe eines Tupels und eines Slots. Das Datum, das das Ergebnis dieser Funktion ist, kann direkt zurückgegeben werden, wenn Sie nur eine einzelne Zeile zurückgeben wollen, oder es kann als aktueller Rückgabewert in einer Funktion mit Ergebnismenge verwendet werden.

Ein Beispiel finden Sie im nächsten Abschnitt.

33.7.9 Rückgabe von Ergebnismengen aus C-Funktionen

Es gibt auch eine spezielle API, die die Rückgabe von Ergebnismengen (mehrere Zeilen) aus C-Funktionen unterstützt. Eine solche Funktion muss die Aufrufskonvention Version 1 verwenden. Außerdem müssen die Quellcodedateien wie oben die Headerdatei `funcapi.h` einbinden.

Eine Funktion mit Ergebnismenge (*set-returning function*, SRF) wird einmal für jedes zurückzugebende Element aufgerufen. Die SRF muss also genug Informationen zwischenspeichern, um zu wissen, was sie zuletzt getan hat, und muss dann bei jedem Aufruf das nächste Element zurückgeben. Um dabei zu helfen, gibt es die Struktur `FuncCallContext`. Innerhalb einer Funktion wird `fcinfo->flinfo->fn_extra` verwendet, um einen Zeiger auf `FuncCallContext` zwischen Aufrufen zu speichern.

```
typedef struct
{
    /*
     * Wie oft wir bisher aufgerufen wurden.
     *
     * call_cntr wird von SRF_FIRSTCALL_INIT() auf 0 gesetzt und von
     * jedem SRF_RETURN_NEXT() erhöht.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximale Anzahl von Aufrufen
     *
     * max_calls ist nur der Bequemlichkeit halber da und ist OPTIONAL.
     * Wenn es nicht gesetzt ist, dann müssen Sie auf andere Weise wissen,
     * wann die Funktion fertig ist.
     */
    uint32 max_calls;

    /*
     * OPTIONAL Zeiger auf Ergebnis-Slot
     *
     * slot kann verwendet werden, wenn Sie Tupel (d.h. zusammengesetzte
     * Datentypen) zurückgeben und wird für Basistypen nicht benötigt.
     */
    TupleTableSlot *slot;
```

```
    /*
     * OPTIONAL Zeiger auf diverse Kontextinformationen vom Benutzer
     *
     * user_fctx ist ein Zeiger auf Ihre eigenen Daten, die Sie zwischen
     * Aufrufen Ihrer Funktion speichern wollen.
     */
    void *user_fctx;

    /*
     * OPTIONAL Zeiger auf Struktur mit Atttributtyp-Metainformationen
     *
     * attinmeta kann verwendet werden, wenn Sie Tupel (d.h. zusammengesetzte
     * Datentypen) zurückgeben und wird für Basistypen nicht benötigt.  Es wird
     * nur benötigt, wenn Sie BuildTupleFromCStrings() verwenden wollen um das
     * Ergebnistupel zu bauen.
     */
    AttInMetadata *attinmeta;

    /*
     * Speicherkontext für Strukturen, die über mehrere Aufrufe verwendet werden
sollen
     *
     * multi_call_memory_ctx wird von SRF_FIRSTCALL_INIT() für Sie eingerichtet
     * und wird von SRF_RETURN_DONE() zum Aufräumen verwendet.  Es ist der
passendste
     * Speicherkontext für Speicher, der von mehreren Aufrufen einer SRF wieder-
     * verwendet werden soll.
     */
    MemoryContext multi_call_memory_ctx;
} FuncCallContext;
```

Eine SRF verwendet mehrere Funktionen und Makros, die den `FuncCallContext` automatisch bearbeiten (und ihn in `fn_extra` erwarten). Verwenden Sie

```
SRF_IS_FIRSTCALL()
```

um zu ermitteln, ob Ihre Funktion zum ersten Mal aufgerufen wird. Im ersten Aufruf (und nur dann), verwenden Sie

```
SRF_FIRSTCALL_INIT()
```

um den `FuncCallContext` zu initialisieren. Bei jedem Aufruf, einschließlich des ersten, verwenden Sie

```
SRF_PERCALL_SETUP()
```

um den `FuncCallContext` einzurichten und schon zurückgegebene Daten zu löschen, die vom vorigen Durchgang übrig geblieben sind.

Wenn Ihre Funktion Daten zurückgeben soll, dann verwenden Sie

```
SRF_RETURN_NEXT(funcctx, ergebnis)
```

(`ergebnis` muss den Typ Datum haben, entweder ein einzelner Wert oder ein Tupel, das wie oben beschrieben vorbereitet wurde.) Wenn Ihre Funktion schließlich keine Daten mehr zurückzugeben hat, dann rufen Sie

```
SRF_RETURN_DONE(funcctx)
```

auf, um aufzuräumen und die SRF zu beenden.

Der Speicherkontext, der in einer SRF aktiv ist, ist ein vorübergehender Kontext, der zwischen den Aufrufen gelöscht wird. Das bedeutet, dass Sie für die Sachen, die Sie mit `palloc` angelegt haben, nicht `pfree` aufrufen müssen; sie werden sowieso gelöscht. Wenn Sie jedoch irgendwelche Datenstrukturen ablegen wollen, die über mehrere Aufrufe verfügbar sein sollen, müssen Sie diese irgendwo anders ablegen. Der Speicherkontext, auf den `multi_call_memory_ctx` zeigt, ist ein passender Speicherkontext, der erst gelöscht wird, wenn die SRF abgeschlossen hat. In dem meisten Fällen heißt das, dass Sie in den Kontext `multi_call_memory_ctx` wechseln müssen, während Sie die Initialisierungen beim ersten Aufruf vornehmen.

Ein vollständiges Beispiel mit Pseudocode sieht so aus:

```
Datum
meine_ergebnismengen_funktion(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             ergebnis;
    MemoryContext     alter_kontext;
    weitere Deklarationen nach Bedarf

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        alter_kontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* Initialisierung beim ersten Aufruf kommt hierhin: */
        Benutzercode
        Wenn zusammengesetzter Typ zurückgegeben wird
            TupleDesc und eventuell AttInMetadata bauen
            Slot erhalten
            funcctx->slot = slot;
        Ende von Wenn zusammengesetzter Typ zurückgegeben wird
        Benutzercode
        MemoryContextSwitchTo(alter_kontext);
    }

    /* Initialisierung bei jedem Aufruf kommt hierhin: */
    Benutzercode
    funcctx = SRF_PERCALL_SETUP();
    Benutzercode

    /* Dies ist nur eine Möglichkeit, wie man das Ende ermitteln kann: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
```

```
        /* Hier geben wir ein weiteres Element zurück: */
        Benutzercode
        Datum-Wert in "ergebnis" speichern
        SRF_RETURN_NEXT(funcctx, ergebnis);
    }
    else
    {
        /* Hier sind wir fertig und räumen auf: */
        Benutzercode
        SRF_RETURN_DONE(funcctx);
    }
}
```

Ein vollständiges Beispiel für eine SRF, die einen zusammengesetzten Typ zurückgibt, sieht so aus:

```
PG_FUNCTION_INFO_V1(testpassbyval);

Datum
testpassbyval(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    TupleTableSlot      *slot;
    AttInMetadata       *attinmeta;

     /* nur beim ersten Aufruf der Funktion */
     if (SRF_IS_FIRSTCALL())
     {
        MemoryContext

        /* erzeuge Funktionskontext für allgemeine Informationen */
        funcctx = SRF_FIRSTCALL_INIT();

        /* wechsele Speicherkontext um Informationen zwischen Aufrufen zu
speichern */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* Gesamtzahl zurückzugebender Tupel */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* baue TupleDesc für ein __testpassbyval Tupel */
        tupdesc = RelationNameGetTupleDesc("__testpassbyval");

        /* erzeuge Slot für ein Tupel mit diesem TupleDesc */
        slot = TupleDescGetSlot(tupdesc);
```

```
        /* weise den Slot dem Funktionskontext zu */
        funcctx->slot = slot;

        /*
         * erzeuge Attributmetadaten um später Tupel aus C-Zeichenketten erzeugen
         * zu können
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* bei jedem Aufruf der Funktion */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    slot = funcctx->slot;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* wenn wir noch nicht fertig sind */
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * Bereite Wertearray für die Speicherung im Slot vor.
         * Dies sollte ein Array mit C-Zeichenketten sein, die
         * später von den Typeingabefunktionen verarbeitet werden.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* baue ein Tupel */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* erzeuge Datum aus Tupel */
```

```
        result = TupleGetDatum(slot, tuple);

        /* räume auf (nicht unbedingt nötig) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* wenn wir fertig sind */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

Der SQL-Code, der diese Funktion deklariert, sieht so aus:

```
CREATE TYPE __testpassbyval AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION testpassbyval(integer, integer) RETURNS SETOF
__testpassbyval
    AS 'VERZEICHNIS', 'testpassbyval'
    LANGUAGE C IMMUTABLE STRICT;
```

Weitere Beispiele für Funktionen mit Ergebnismengen finden Sie im Verzeichnis `contrib/tablefunc` in der Quellendistribution.

33.8 Funktionen überladen

Sie können mehrere Funktionen mit dem gleichen SQL-Namen definieren, solange die Argumente verschieden sind. Anders ausgedrückt, können Funktionen **überladen** werden. Wenn eine Anfrage ausgeführt wird, bestimmt der Server die aufzurufende Funktion anhand der Datentypen und der Anzahl der angegebenen Argumente. Das Überladen kann auch verwendet werden, um Funktionen mit einer variablen Anzahl von Argumenten zu simulieren, bis zu einer endlichen Höchstzahl.

Eine Funktion kann auch den gleichen Namen haben wie ein Attribut. (Erinnern Sie sich, dass `attribut(tabelle)` gleichbedeutend ist mit `tabelle.attribut`.) Wenn es dabei Zweideutigkeiten gibt zwischen einer Funktion, die mit einem zusammengesetzten Typ verwendet wird, und einem Attribut eines komplexen Typs, wird immer das Attribut verwendet.

Wenn Sie eine Familie aus überladenen Funktionen erzeugen, sollten Sie darauf achten, keine Unklarheiten zu erzeugen. Zum Beispiel ist es bei den Funktionen

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

nicht sofort klar, welche Funktion mit einer einfachen Eingabe wie `test(1, 1.5)` aufgerufen würde. Die gegenwärtig implementierten Auflösungsregeln sind in Kapitel 10 beschrieben, aber es wäre nicht besonders schlau, ein System zu entwerfen, das sich indirekt darauf verlässt.

Wenn eine C-Funktion überladen wird, gibt es eine zusätzliche Einschränkung: Der C-Name jeder Funktion in der Familie der überladenen Funktionen muss sich von den C-Namen aller anderen Funktionen, egal ob intern oder dynamisch geladen, unterscheiden. Wenn diese Regel nicht befolgt wird, ist das Verhalten plattformabhängig. Sie könnten zur Laufzeit einen Linkerfehler erhalten oder eine der Funktionen wird aufgerufen (meistens die interne). Mit der alternativen Form der AS-Klausel des SQL-Befehls CREATE FUNCTION kann ein C-Funktionsname angegeben werden, der sich vom Namen der SQL-Funktion unterscheidet. Zum Beispiel:

```
CREATE FUNCTION test(int) RETURNS int
    AS 'dateiname', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'dateiname', 'test_2arg'
    LANGUAGE C;
```

Die hier gewählten Dateinamen der C-Funktionen spiegeln nur eine der möglichen Namenskonventionen wieder.

33.9 Handler für prozedurale Sprachen

Alle Funktionsaufrufe, die in einer anderen Sprache als der aktuellen Version-1-Schnittstelle für kompilierte Sprachen geschrieben worden sind (das schließt Funktionen in benutzerdefinierten prozeduralen Sprachen, Funktionen in SQL und Funktionen, die die Version-0-Schnittstelle für kompilierte Sprachen verwenden, ein), werden von einer Handlerfunktion für die bestimmte Sprache verarbeitet. Es liegt in der Verantwortung des Handlers, die Funktion auf eine sinnvolle Art und Weise auszuführen, zum Beispiel indem der angegebene Quelltext interpretiert wird. Dieser Abschnitt beschreibt, wie ein Handler für eine Sprache geschrieben werden kann. Diese Aufgabe kommt nicht sehr häufig vor; in der Tat wurde es in der Geschichte von PostgreSQL bisher nur ein paar Mal gemacht. Aber dieses Thema passt am besten in dieses Kapitel, und das Material kann ein wenig Einblick in die erweiterbare Natur des PostgreSQL-Systems geben.

Der Handler einer prozedurale Sprache ist eine “normale” Funktion, die in einer kompilierten Sprache wie C geschrieben wird, unter Verwendung der Version-1-Schnittstelle, und in PostgreSQL als Funktion ohne Argument und mit dem Rückgabetyp language_handler registriert wird. Dieser besondere Pseudotyp identifiziert die Funktion als Sprachhandler und verhindert, dass sie direkt in SQL-Befehlen aufgerufen wird.

Der Handler wird genauso wie jede andere Funktion aufgerufen: Er erhält einen Zeiger auf ein FunctionCallInfoData struct, das die Argumentwerte und Informationen über die aufgerufene Funktion enthält, und er muss einen Wert vom Typ Datum zurückgeben (und eventuell das Feld isnull in der Struktur FunctionCallInfoData setzen, wenn er einen NULL-Wert zurückgeben will). Der Unterschied zwischen einem Handler und einer normalen Funktion ist, dass das Feld flinfo->fn_oid der Struktur FunctionCallInfoData die OID der eigentlich aufgerufenen Funktion enthält und nicht die des Handlers selbst. Der Handler muss dieses Feld verwenden, um zu ermitteln, welche Funktion ausgeführt werden soll. Außerdem wurde die übergebene Argumentliste entsprechend der Deklaration der Zielfunktion eingerichtet und nicht nach der Deklaration des Handlers.

Es liegt an dem Handler, den Eintrag für die Funktion aus der Systemtabelle pg_proc zu lesen und die Argumenttypen und den Rückgabetyp der aufgerufenen Funktion zu analysieren. Die AS-Klausel aus dem CREATE FUNCTION-Befehl der Funktion steht in der Spalte prosrc von pg_proc. Das könnte der Quelltext in der prozeduralen Sprache selbst sein (wie bei PL/Tcl) oder ein Dateiname oder irgendetwas anderes, das dem Handler mitteilt, was genau er zu tun hat.

Oft wird dieselbe Funktion mehrere Male pro SQL-Befehl aufgerufen. Ein Handler kann es vermeiden, wiederholt Informationen über die aufgerufene Funktion ermitteln zu müssen, indem er das Feld `flinfo->fn_extra` verwendet. Dieses Feld ist am Anfang *NULL*, kann aber vom Handler auf Informationen über die aufgerufene Funktion gesetzt werden. Bei den folgenden Aufrufen, wenn `flinfo->fn_extra` nicht mehr *NULL* ist, können die Informationen wiederverwendet werden. Der Handler muss dafür sorgen, dass `flinfo->fn_extra` auf einen Speicherbereich zeigt, der mindestens bis zum Ende der aktuellen Anfrage erhalten bleibt, da die `FmgrInfo`-Datenstruktur so lange erhalten bleiben könnte. Eine Möglichkeit, das zu erreichen, ist, die Extradaten im von `flinfo->fn_mcxt` angegebenen Speicherkontext abzulegen; solche Daten haben normalerweise die gleiche Lebensdauer wie die `FmgrInfo`-Struktur selbst. Aber der Handler könnte auch einen längerlebigen Speicherkontext verwenden, damit er die Funktionsinformationen über mehrere Anfragen hinweg speichern kann.

Wenn eine in einer prozeduralen Sprache geschriebene Funktion als Trigger aufgerufen wird, werden keine Argumente auf die übliche Art übergeben, aber das Feld `context` der `FunctionCallInfoData`-Struktur zeigt auf eine `TriggerData`-Struktur, anstatt *NULL* zu sein. Ein Sprachhandler sollte einen Mechanismus vorsehen, durch den die Funktionen an die Triggerinformationen gelangen können.

Hier ist eine Vorlage für einen in C geschriebenen Sprachhandler:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plbeispiel_call_handler);

Datum
plbeispiel_call_handler(PG_FUNCTION_ARGS)
{
    Datum          ergebnis;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Als Triggerprozedur aufgerufen
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        ergebnis = ...
    }
    else
    {
        /*
         * Als Funktion aufgerufen
         */
```

```
        ergebnis = ...
    }

    return ergebnis;
}
```

Um den Handler zu vervollständigen, müssen nur ein paar tausend Zeilen Code anstelle der Punkte eingefügt werden.

Nachdem die Handlerfunktion in ein ladbares Modul kompiliert wurde (siehe Abschnitt 33.7.6), kann die Beispielsprache mit folgenden Befehlen registriert werden:

```
CREATE FUNCTION plbeispiel_call_handler() RETURNS language_handler
    AS 'dateiname'
    LANGUAGE C;
CREATE LANGUAGE plbeispiel
    HANDLER plbeispiel_call_handler;
```

33.10 Benutzerdefinierte Datentypen

Wie oben beschrieben, gibt es in PostgreSQL zwei Arten von Datentypen: Basistypen und zusammengesetzte Typen. Dieser Abschnitt beschreibt, wie man neue Basistypen definiert.

Die Beispiele in diesem Abschnitt können Sie auch in complex.sql und complex.c im Tutorial-Verzeichnis finden.

Ein benutzerdefinierter Typ muss eine Eingabe- und eine Ausgabefunktion haben. Diese Funktionen bestimmen, wie der Typ in Zeichenketten erscheint (bei Eingabe vom Benutzer und bei Ausgabe zum Benutzer) und wieder Typ im Speicher organisiert wird. Die Eingabefunktion nimmt eine C-Zeichenkette (mit Null-Byte am Ende) als Argument und gibt die interne (im Speicher) Darstellung des Typs zurück. Die Ausgabefunktion nimmt die interne Darstellung des Typs als Argument und gibt eine C-Zeichenkette zurück.

Nehmen wir an, wir wollen einen Typ namens complex definieren, der komplexe Zahlen darstellt. Eine nahe liegende Möglichkeit, eine komplexe Zahl im Speicher zu organisieren, ist die folgende C-Struktur:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

Als externe Zeichenkettendarstellung des Typs wählen wir die Form (x,y).

Das Schreiben der Eingabe- und Ausgabefunktionen ist in der Regel nicht schwer. Aber wenn Sie die externe Zeichenkettendarstellung des Typs festlegen, dann müssen Sie daran denken, dass die Eingabefunktion einen vollständigen und robusten Parser für diese Darstellung enthalten muss. Zum Beispiel:

```
Complex *
complex_in(char *str)
{
```

```
    double x, y;
    Complex *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
    {
        elog(ERROR, "complex_in: error in parsing %s", str);
        return NULL;
    }
    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    return result;
}
```

Die Ausgabefunktion kann einfach so aussehen:

```
char *
complex_out(Complex *complex)
{
    char *result;

    if (complex == NULL)
        return(NULL);
    result = (char *) palloc(60);
    sprintf(result, "(%g,%g)", complex->x, complex->y);
    return result;
}
```

Sie sollten versuchen, die Eingabe- und Ausgabefunktionen so zu schreiben, dass sie jeweils die Umkehrung der anderen sind. Wenn Sie das nicht tun, werden Sie große Probleme haben, wenn Sie Ihre Daten in Textdateien speichern und wieder einlesen. Das ist besonders bei Fließkommazahlen ein häufiges Problem.

Um den Typ complex zu definieren, müssen wir zuerst die zwei benutzerdefinierten Funktionen complex_in und complex_out definieren:

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'dateiname'
    LANGUAGE C;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'dateiname'
    LANGUAGE C;
```

Beachten Sie, dass die Deklarationen der Eingabe- und Ausgabefunktionen auf den noch nicht definierten Typ verweisen müssen. Das ist erlaubt, erzeugt aber eine Warnmeldung, die Sie ignorieren können.

Schließlich können wir den Datentyp deklarieren:

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

Wenn Sie einen neuen Basistyp definieren, erstellt PostgreSQL automatisch die Unterstützung für Arrays dieses Typs. Aus historischen Gründen hat der Arraytyp den gleichen Namen wie der Basistyp mit einem Unterstrich (_) davor.

Wenn die Werte Ihres Datentyps die Größe von ein paar hundert Bytes überschreitet (in der internen Form), sollten Sie den Typ TOAST-fähig machen. Dazu muss die interne Organisation dem Standardlayout für Typen mit variabler Länge folgen: Die ersten vier Bytes sind ein int32 mit der Gesamtlänge des Werts (einschließlich der Längenangabe selbst). Wenn Sie den Befehl CREATE TYPE ausführen, wählen Sie als interne Länge variable und wählen Sie die passende Speicheroption.

Weitere Einzelheiten finden Sie in der Beschreibung des Befehls CREATE TYPE in Teil VI.

33.11 Benutzerdefinierte Operatoren

Jeder Operator ist eine syntaktische Verzierung für einen Aufruf einer Funktion, die die eigentliche Arbeit erledigt. Sie müssen also zuerst eine Funktion definieren, die dem zu definierenden Operator zugrunde liegt. Ein Operator ist jedoch *nicht nur* eine syntaktische Verzierung, weil er zusätzliche Informationen enthält, die dem Anfrageplaner helfen, Anfragen, die den Operator verwenden, zu optimieren. Der nächste Abschnitt widmet sich diesen zusätzlichen Informationen.

PostgreSQL unterstützt linke unäre (ein Argument), rechte unäre (ein Argument) und binäre (zwei Argumente) Operatoren. Operatoren können überladen werden; das heißt, derselbe Operatorname kann für verschiedene Operatoren verwendet werden, die verschiedene Operandentypen oder eine andere Anzahl von Operanden haben. Wenn eine Anfrage ausgeführt wird, bestimmt das System den aufzurufenden Operator anhand der Datentypen und der Anzahl der angegebenen Operanden.

Hier ist ein Beispiel für einen Operator, der zwei komplexe Zahlen addiert. Wir nehmen an, dass wir bereits den Typ complex erzeugt haben (siehe Abschnitt 33.10). Als erstes benötigen wir eine Funktion, die die Arbeit erledigt, dann können wir den Operator definieren:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'dateiname', 'complex_add'
    LANGUAGE C;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Jetzt könnten wir eine solche Anfrage ausführen:

```
SELECT (a + b) AS c FROM test_complex;

        c
----------------
 (5.2,6.05)
 (133.42,144.95)
```

Hier haben wir gezeigt, wie man einen binären Operator erzeugt. Um einen unären Operator zu erzeugen, lassen Sie einfach `leftarg` oder `rightarg` weg (für einen linken unären bzw. rechten unären). Die Klausel `procedure` und die Argumentklauseln sind die einzigen erforderlichen Elemente in `CREATE OPERATOR`. Die im Beispiel gezeigte `commutator`-Klausel ist ein wahlfreier Hinweis für den Anfrageoptimierer. Weitere Einzelheiten über `commutator` und andere Optimiererhinweise folgen im nächsten Abschnitt.

33.12 Operator-Optimierungsinformationen

Die Definition eines PostgreSQL-Operators kann mehrere optionale Klauseln enthalten, die dem System nützliche Dinge darüber mitteilen, wie der Operator sich verhält. Diese Klauseln sollten angegeben werden, wenn es angebracht ist, weil sie in Anfragen, die den Operator verwenden, für eine erhebliche Beschleunigung sorgen können. Aber wenn Sie sie angeben, müssen sie auch richtig sein! Falsche Verwendung der Optimierungsklauseln kann zu Abstürzen des Serverprozesses, schwer zu entdeckenden falschen Ergebnissen und anderen Problemen führen. Sie können die Optimierungsklauseln immer auch weglassen, wenn Sie sich Ihrer Sache nicht sicher sind; die einzige Folge wäre dann, dass Anfragen langsamer als nötig laufen könnten.

In zukünftigen Versionen von PostgreSQL könnten weitere Optimierungsklauseln hinzugefügt werden. Die hier beschriebenen sind alle die, die Version versteht.

33.12.1 COMMUTATOR

Die `COMMUTATOR`-Klausel nennt, wenn angegeben, einen Operator, der der Kommutator des zu definierenden Operators ist. Ein Operator A ist der Kommutator eines Operators B, wenn (x A y) gleich (y B x) für alle möglichen Eingabewerte x, y ist. Wie Sie bemerken werden, ist B dann auch der Kommutator von A. Zum Beispiel sind die Operatoren < und > eines bestimmten Datentyps normalerweise Kommutatoren voneinander und der Operator + ist normalerweise kommutativ mit sich selbst. Aber der Operator – ist normalerweise mit keinem anderen Operator kommutativ.

Der Typ des linken Operanden eines kommutativen Operators ist derselbe wie der Typ des rechten Operanden seines Kommutators und umgekehrt. Der Name des Kommutatoroperators reicht aus, damit PostgreSQL den Kommutator finden kann, und ist daher alles, was in der `COMMUTATOR`-Klausel angegeben werden muss.

Wenn Sie einen selbstkommutativen Operator definieren, können Sie das einfach so machen. Wenn Sie ein Paar kommutativer Operatoren definieren, wird es etwas schwieriger: Wie kann der zuerst definierte auf den anderen verweisen, der noch nicht definiert wurde? Für dieses Problem gibt es zwei Lösungen:

- Eine Möglichkeit ist, die `COMMUTATOR`-Klausel beim zuerst definierten Operator wegzulassen und nur beim zweiten eine anzugeben. Weil PostgreSQL weiß, dass kommutative Operatoren in Paaren auftreten, wird es, wenn es die zweite Definition sieht, automatisch die fehlende `COMMUTATOR`-Klausel in der ersten Definition ergänzen.

- Die andere, einfachere Möglichkeit ist, einfach in beiden Definitionen eine `COMMUTATOR`-Klausel anzugeben. Wenn PostgreSQL die erste Definition verarbeitet und erkennt, dass die `COMMUTATOR`-Klausel auf einen nicht existierenden Operator verweist, erzeugt das System im Systemkatalog einen provisorischen Eintrag für diesen Operator. Dieser provisorische Eintrag hat nur gültige Daten für den Operatornamen, den Typ des linken und rechten Operanden und den Ergebnistyp, da das alles ist, was PostgreSQL zu diesem Zeitpunkt ableiten kann. Der Katalogeintrag des ersten Operators verweist auf diesen provisorischen Eintrag. Wenn Sie später den zweiten Operator definieren, ergänzt das System den provisorischen Eintrag mit den in der zweiten Definition angegebenen zusätzlichen Informationen. Wenn Sie versuchen, den provisorischen Operator zu verwenden, bevor er vollständig definiert wurde, dann erhalten Sie eine Fehlermeldung.

33.12.2 NEGATOR

Die `NEGATOR`-Klausel nennt, wenn angegeben, einen Operator, der die Umkehrung des zu definierenden Operators ist. Ein Operator A ist die Umkehrung eines Operators B, wenn beide ein Ergebnis vom Typ boolean zurückgeben und (x A y) gleich NOT (x B y) für alle möglichen Eingabewerte x, y ist. Wie Sie bemerken werden, ist B dann auch die Umkehrung von A. Zum Beispiel sind die Operatoren `<` und `>=` Umkehrungspaare für die meisten Datentypen. Ein Operator kann niemals seine eigene Umkehrung sein.

Im Gegensatz zu Kommutatoren könnte auch ein Paar unärer Operatoren als Umkehrungspaar markiert werden; das würde heißen, dass (A x) gleich NOT (B x) für alle x ist, oder entsprechend für rechte unäre Operatoren.

Der Umkehrungsoperator muss denselben linken und/oder rechten Operandentyp wie der zu definierende Operator haben, also muss, wie bei `COMMUTATOR`, bei `NEGATOR` nur der Operatorname angegeben werden.

Die Angabe des Umkehrungsoperators ist sehr nützlich, damit der Anfrageoptimierer Ausdrücke wie `NOT (x = y)` in die einfachere Form `x <> y` umwandeln kann. Das kommt öfter vor, als Sie vielleicht denken, weil `NOT`-Operationen als Folge anderer Umstellungen entstehen können.

Paare von Umkehrungsoperatoren können mit den gleichen Methoden definiert werden, wie oben für Kommutatorpaare erklärt wurde.

33.12.3 RESTRICT

Die `RESTRICT`-Klausel nennt, wenn angegeben, eine Auswahlselektivitätsschätzfunktion für den Operator. (Beachten Sie, dass das ein Funktionsname ist, kein Operatorname.) `RESTRICT`-Klauseln sind nur für binäre Operatoren, die boolean zurückgeben, sinnvoll. Der Zweck der Auswahlselektivitätsschätzfunktion ist, dass sie schätzen soll, welcher Anteil der Zeilen in einer Tabelle eine `WHERE`-Klausel-Bedingung der Form

```
spalte OP konstante
```

mit dem aktuellen Operator und einem bestimmten konstanten Wert erfüllen wird. Das hilft dem Optimierer, indem es ihm eine Ahnung gibt, wie viele Zeilen von `WHERE`-Klauseln dieser Form eliminiert werden. (Was passiert, wenn die Konstante auf der linken Seite steht, fragen Sie sich vielleicht? Dafür ist ja der Kommutator da ...)

Das Schreiben einer neuen Selektivitätsschätzfunktion geht weit über den Rahmen dieses Kapitels hinaus, aber glücklicherweise können Sie für viele Ihrer eigenen Operatoren normalerweise einfach eine der Standardschätzfunktionen im System verwenden. Diese sind:

`eqsel` für `=`

`neqsel` für `<>`

`scalarltsel` für < oder <=

`scalargtsel` für > oder >=

Es kommt Ihnen vielleicht etwas merkwürdig vor, dass das die Kategorien sind, aber wenn Sie darüber nachdenken, hat das Sinn. = akzeptiert typischerweise nur einen kleinen Anteil der Zeilen in einer Tabelle; <> verwirft typischerweise nur einen kleinen Anteil. < akzeptiert einen Anteil, der davon abhängt, wo die angegebene Konstante in den Bereich der Werte dieser Spalte fällt. (Diese Informationen werden von `ANALYZE` eingesammelt und der Schätzfunktion zur Verfügung gestellt.) <= wird einen etwas höheren Anteil als < für dieselbe Konstante akzeptieren, aber sie sind nahe genug beieinander, dass es sich nicht lohnt, sie zu unterscheiden, da es sich sowieso nur um eine Schätzung handelt. Ähnliche Bemerkungen gelten auch für > und >=.

Oft werden Sie auch davon kommen, wenn Sie entweder `eqsel` oder `neqsel` für Operatoren mit sehr hoher bzw. sehr niedriger Selektivität verwenden, selbst wenn diese nicht wirklich Gleichheit oder Ungleichheit testen. Zum Beispiel verwenden die geometrischen Operatoren, die auf ungefähre Gleichheit testen, auch `eqsel`, unter der Annahme, dass sie normalerweise nur einen kleinen Teil der Zeilen einer Tabelle auswählen.

`scalarltsel` und `scalargtsel` können Sie für Vergleiche von Datentypen verwenden, die auf eine sinnvolle Weise in numerische Skalarwerte umgewandelt werden können, die für Bereichsvergleiche verwendet werden können. Wenn möglich, fügen Sie den Datentyp zu der Liste hinzu, die von der Funktion `convert_to_scalar()` in `src/backend/utils/adt/selfuncs.c` verstanden wird. (Irgendwann sollte diese Funktion durch datentypspezifische Funktionen ersetzt werden, die in einer Spalte des Systemkatalogs `pg_type` gespeichert wird. Aber zurzeit ist das noch nicht der Fall.) Wenn Sie das nicht tun, funktioniert alles trotzdem, aber die Schätzungen des Optimierers werden nicht so gut sein, wie sie sein könnten.

Zusätzliche Selektivitätsschätzfunktionen gibt es für geometrische Operatoren in `src/bakkend/utils/adt/geo_selfuncs.c`: `areasel`, `positionsel` und `contsel`. Zurzeit geben diese Funktionen nur konstante Werte zurück, aber vielleicht wollen Sie sie trotzdem verwenden (oder sie vielleicht sogar verbessern.)

33.12.4 JOIN

Die `JOIN`-Klausel nennt, wenn angegeben, eine Verbundselektivitätsschätzfunktion für den Operator. (Beachten Sie, dass das ein Funktionsname ist, kein Operatorname.) `JOIN`-Klauseln sind nur für binäre Operatoren, die boolean zurückgeben, sinnvoll. Der Zweck der Verbundselektivitätsschätzfunktion ist, dass sie schätzen soll, welcher Anteil der Zeilen in einem Tabellenpaar eine `WHERE`-Klausel-Bedingung der Form

tabelle1.spalte1 OP tabelle2.spalte2

mit dem aktuellen Operator erfüllen wird. Wie bei `RESTRICT` hilft das dem Optimierer erheblich, weil er dadurch ermitteln kann, welche von mehreren möglichen Verbundreihenfolgen am wahrscheinlichsten den geringsten Aufwand erfordert.

Wie zuvor wird dieses Kapitel nicht versuchen zu erklären, wie man eine solche Verbundselektivitätsschätzfunktion schreibt, sondern empfiehlt, dass Sie, wenn es angebracht ist, eine der Standardfunktionen verwenden:

`eqjoinsel` für =

`neqjoinsel` für <>

`scalarltjoinsel` für < oder <=

`scalargtjoinsel` für > oder >=

`areajoinsel` für 2D-Vergleiche der Fläche

`positionjoinsel` für 2D-Vergleiche der Position

`contjoinsel` für 2D-Vergleiche, die prüfen, ob ein Objekt in einem anderen enthalten ist

33.12.5 HASHES

Die `HASHES`-Klausel, wenn angegeben, sagt dem System, dass es möglich ist, für Verbunde mit diesem Operator die Hash-Verbund-Methode zu verwenden. `HASHES` ist nur für einen binären Operator, der boolean zurückgibt, sinnvoll, und in der Praxis sollte dieser Operator auch der Ist-Gleich-Operator eines Datentyps sein.

Die Annahme, die dem Hash-Verbund zugrunde liegt, ist, dass ein Verbundoperator nur wahr ergeben kann, wenn der linke und der rechte Wert den gleichen Hash-Code ergeben. Wenn zwei Werte verschiedene Hash-Codes ergeben, wird der Verbund sie gar nicht erst vergleichen, weil er implizit annimmt, dass das Ergebnis des Verbundoperators falsch sein muss. Es ist also niemals sinnvoll, `HASHES` bei Operatoren anzugeben, die keine Gleichheitsprüfung darstellen.

Tatsächlich reicht die logische Gleichheit noch nicht aus; der Operator muss vielmehr die bitweise Gleichheit darstellen, weil die Hashfunktion aus der Speicherdarstellung des Werts berechnet wird, unabhängig von der Bedeutung der Bits. Zum Beispiel stellt der Polygonoperator `~=`, der prüft, ob zwei Polygone deckungsgleich sind, keine bitweise Gleichheit dar, weil zwei Polygone auch deckungsgleich sein können, wenn Ihre Eckpunkte in unterschiedlichen Reihenfolgen angegeben sind. Das würde bedeuten, dass ein Verbund mit `~=` zwischen Polygonspalten andere Ergebnisse liefern würde, wenn er als Hash-Verbund durchgeführt würde, als wenn er mit einer anderen Methode durchgeführt würde, weil ein großer Teil der Werte, die übereinstimmen sollten, verschiedene Hash-Werte ergeben und vom Hash-Verbund gar nicht erst verglichen werden. Wenn der Optimierer aber eine andere Verbundmethode auswählt, werden alle Paare, die vom Operator `~=` als gleich erachtet werden, auch gefunden. Widersprüche dieser Art gilt es zu vermeiden, und daher wird der Polygonoperator `~=` nicht als Hash-fähig markiert.

Es gibt auch maschinenabhängige Gründe, warum ein Hash-Verbund nicht richtig funktionieren könnte. Wenn Ihr Datentyp zum Beispiel eine Struktur ist, die uninteressante Füllbits enthält, ist es nicht sicher, den Ist-gleich-Operator als Hash-fähig zu markieren. (Es sei denn, Sie schreiben Ihre anderen Operatoren und Funktionen so, dass die unbenutzten Bits immer null sind, was die empfohlene Verfahrensweise ist.) Ein weiteres Beispiel ist, dass die Fließkommadatentypen nicht für Hash-Verbunde sicher sind. Auf Maschinen, die den IEEE-Fließkommastandard befolgen, sind minus null und plus null verschiedene Werte (verschiedene Bitmuster), werden aber bei Vergleichen als gleich erachtet. Wenn also der Ist-gleich-Operator von Fließkommadatentypen als Hash-fähig markiert würde, dann würden minus null und plus null von einem Hash-Verbund wahrscheinlich nicht als gleich befunden, aber von allen anderen Verbundmethoden.

Zusammengefasst kann man sagen, dass Sie `HASHES` wahrscheinlich nur bei Ist-gleich-Operatoren verwenden sollten, die durch `memcmp()` implementiert sind (oder so implementiert werden könnten).

33.12.6 MERGES (SORT1, SORT2, LTCMP, GTCMP)

Die `MERGES`-Klausel, wenn angegeben, sagt dem System, dass es möglich ist, für Verbunde mit diesem Operator die Merge-Verbund-Methode zu verwenden. `MERGES` ist nur für einen binären Operator, der boolean zurückgibt, sinnvoll, und in der Praxis muss dieser Operator auch der Ist-Gleich-Operator eines Datentyps oder zwischen zwei Datentypen sein.

Der Merge-Verbund basiert auf der Idee, dass die linken und rechten Tabellen in einem Verbund sortiert und dann parallel durchsucht werden. Also muss es bei beiden Datentypen möglich sein, sie vollständig zu sortieren, und der Verbundoperator sollte so funktionieren, dass er nur für Wertepaare, die in der Sortierreihenfolge an "derselben Stelle” stehen, erfolgreich ist. In der Praxis bedeutet das, dass sich der Verbundoperator wie ein Ist-gleich-Operator verhalten muss. Aber im Gegensatz zum Hash-Verbund, wo beide Datentypen identisch (oder zumindest bitweise äquivalent) sein müssen, kann ein Merge-Verbund

mit zwei unterschiedlichen Operatoren ausgeführt werden, solange diese logisch kompatibel sind. Zum Beispiel ist der Ist-gleich-Operator zwischen smallint und integer Merge-Verbund-fähig. Wir benötigen nur Sortieroperatoren, die die beiden Datentypen in eine logische kompatible Reihenfolge bringen.

Für die Ausführung eines Merge-Verbunds benötigt das System vier Operatoren neben dem Merge-Verbund-fähigen Ist-gleich-Operator: den Kleiner-als-Operator für den linken Operandentyp, den Kleiner-als-Operator für den rechten Operandentyp, den Kleiner-als-Operator zwischen den beiden Datentypen und den Größer-als-Operator zwischen den beiden Datentypen. (Das sind vier verschiedene Operatoren, wenn der Merge-Verbund-fähige Operator zwei unterschiedliche Operandentypen hat; aber wenn die Operandentypen dieselben sind, dann sind die drei Kleiner-als-Operatoren alle dieselben.) Die Namen dieser Operatoren können einzeln mit den Optionen `SORT1`, `SORT2`, `LTCMP` bzw. `GTCMP` angegeben werden. Wenn `MERGES` angegeben und einer oder mehrere dieser Operatoren weggelassen wurden, setzt das System automatisch die Standardnamen <, <, < bzw. > ein. Ferner wird `MERGES` automatisch angenommen, wenn eine dieser vier Operatoroptionen entdeckt wird, sodass man einfach einige von ihnen angeben kann und den Rest vom System auffüllen lässt.

Die Operandendatentypen der vier Vergleichsoperatoren können von den Operandentypen des Merge-Verbund-fähigen Operators abgeleitet werden, also müssen Sie wie bei `COMMUTATOR` nur die Operatornamen selbst in diesen Klauseln angeben. Wenn Sie keine eigenartigen Namen für Ihre Operatoren ausgewählt haben, reicht es aus, wenn Sie `MERGES` schreiben und die Einzelheiten dem System überlassen. (Wie bei `COMMUTATOR` und `NEGATOR` kann das System provisorische Operatoreinträge anlegen, wenn Sie die Ist-gleich-Operatoren vor den anderen erzeugen.)

Es gibt einige zusätzliche Einschränkungen für Operatoren, die als Merge-Verbund-fähig eingestuft werden. Diese Einschränkungen werden gegenwärtig nicht von `CREATE OPERATOR` überprüft, aber wenn sie nicht befolgt werden, könnten Sie damit Fehler verursachen:

- Ein Merge-Verbund-fähiger Operator muss einen Merge-Verbund-fähigen Kommutator haben (er selbst, wenn die beiden Operandendatentypen gleich sind, oder ein verwandter Ist-gleich-Operator, wenn sie unterschiedlich sind).
- Wenn es einen Merge-Verbund-fähigen Operator zwischen zwei Datentypen A und B gibt und einen weiteren Merge-Verbund-fähigen Operator zwischen B und einem dritten Datentyp C, muss es zwischen A und C auch einen Merge-Verbund-fähigen Operator geben. Anders ausgedrück,t muss das Vorhandensein eines Merge-Verbund-fähigen Operators zwischen zwei Datentypen transitiv sein.
- Wenn die vier Vergleichsoperatoren die Daten nicht auf kompatible Art und Weise sortieren,+ werden Sie bizarre Ergebnisse erhalten.

Anmerkung

In PostgreSQL-Versionen vor 7.3 gab es die Kurzform mit `MERGES` nicht: Um einen Merge-Verbund-fähigen Operator zu erzeugen, musste man sowohl `SORT1` als auch `SORT2` ausdrücklich angeben. Die Optionen `LTCMP` und `GTCMP` gab es auch nicht; die Namen dieser Operatoren waren auf < bzw. > festgelegt.

33.13 Benutzerdefinierte Aggregatfunktionen

Aggregatfunktionen werden in PostgreSQL mit *Zustandswerten* und *Übergangsfunktionen* ausgedrückt. Das heißt, eine Aggregatfunktion kann als Zustand gesehen werden, der bei jedem Eingabewert verändert wird. Um eine neue Aggregatfunktion zu definieren, wählt man einen Datentyp für den Zustandswert, einen Anfangswert für den Zustandswert und eine Übergangsfunktion. Die Übergangsfunktion ist einfach eine normale Funktion, die auch außerhalb der Aggregatfunktion verwendet werden könnte. Außerdem kann eine *Abschlussfunktion* angegeben werden, wenn sich das gewünschte Ergebnis der Aggregatfunktion von den Daten, die im laufenden Zustandswert gespeichert werden, unterscheidet.

Neben dem Argument- und dem Ergebnistyp, die vom Benutzer der Aggregatfunktion gesehen werden, gibt es also den Datentyp des internen Zustandswerts, der sich von dem Argument- und dem Ergebnistyp unterscheiden kann.

Wenn wir eine Aggregatfunktion ohne Abschlussfunktion definieren, haben wir eine Aggregatfunktion, die aus den laufenden Spaltenwerten jeder Zeile einen Wert errechnet. Ein Beispiel dafür ist die Aggregatfunktion `sum`. `sum` fängt bei null an und addiert immer den Wert der aktuellen Zeile zum laufenden Gesamtwert. Wenn wir zum Beispiel eine Aggregatfunktion `sum` für einen Datentyp, der komplexe Zahlen darstellt, erzeugen wollen, benötigen wir nur die Additionsfunktion dieses Datentyps. Die Definition der Aggregatfunktion wäre:

```
CREATE AGGREGATE complex_sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;

 complex_sum
-------------
 (34,53.9)
```

(In der Praxis würden wir die Aggregatfunktion einfach `sum` nennen und es PostgreSQL überlassen, die richtige Funktion für eine Spalte des Typs complex zu finden.)

Die obige Definition von `sum` gibt null (den Anfangswert) zurück, wenn es keine vom NULL-Wert verschiedene Eingabewerte gibt. Vielleicht wollen wir in dem Fall den NULL-Wert zurückgeben. Der SQL-Standard verlangt, dass sich `sum` so verhält. Dazu können wir einfach die `initcond`-Klausel weglassen, damit der Anfangswert der NULL-Wert ist. Normalerweise würde das bedeuten, dass die `sfunc` (Übergangsfunktion) überprüfen muss, ob der Zustandswert der NULL-Wert ist, aber für `sum` und einige andere einfache Aggregatfunktionen wie `max` und `min` würde es ausreichen, den ersten von NULL verschiedenen Eingabewert im Zustandswert abzulegen und dann die Übergangsfunktion erst beim zweiten von NULL verschiedenen Eingabewert anzuwenden. PostgreSQL macht das automatisch, wenn der Anfangswert der NULL-Wert ist und die Übergangsfunktion als "strikt" markiert ist (d.h., sie wird nicht aufgerufen, wenn ein Argument der NULL-Wert ist).

Außerdem wird, wenn die Übergangsfunktion "strikt" ist, der vorherige Zustandswert unverändert belassen, wenn der Eingabewert der NULL-Wert ist. NULL-Werte werden also ignoriert. Wenn Sie ein anderes Verhalten für NULL-Wert in der Eingabe benötigen, definieren Sie Ihre Übergangsfunktion nicht als strikt und schreiben Sie den Code so, dass er NULL-Werte erkennt und sie wie gewünscht verarbeitet.

Die Aggregatfunktion `avg` (Durchschnitt) ist ein komplexeres Beispiel. Sie benötigt zwei Zustandswerte: die Summe und die Anzahl der Eingabewerte. Das Endergebnis wird ermittelt, indem die Summe durch die Anzahl dividiert wird. `avg` wird in der Regel mit einem Array aus zwei Elementen als Zustandswert definiert. Die eingebaute Implementierung von `avg(float8)` sieht zum Beispiel so aus:

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
```

```
);
```

Weitere Einzelheiten finden Sie in der Beschreibung des Befehls `CREATE AGGREGATE` in Teil VI.

33.14 Erweiterungen für Indexe vorbereiten

Bisher haben wir beschrieben, wie man neue Typen, neue Funktionen und neue Operatoren definiert. Aber wir können noch keinen Index für eine Spalte eines neuen Datentyps erzeugen. Dazu müssen wir zuerst eine *Operatorklasse* für den neuen Datentyp erzeugen. Später in diesem Abschnitt werden wir dieses Konzept anhand eines Beispiels veranschaulichen: eine neue Operatorklasse für die B-Tree-Indexmethode, die komplexe Zahlen in der aufsteigenden Größe ihres Betrags sortiert.

Anmerkung

Vor PostgreSQL Version 7.3 musste man von Hand Änderungen in den Systemkatalogen `pg_amop`, `pg_amproc` und `pg_opclass` vornehmen, um benutzerdefinierte Operatorklassen zu erzeugen. Diese Methode wird jetzt nicht mehr empfohlen, weil `CREATE OPERATOR CLASS` eine viel einfachere und weniger fehleranfällige Methode ist, um die notwendigen Katalogeinträge zu erzeugen.

33.14.1 Indexmethoden und Operatorklassen

Die Tabelle `pg_am` enthält eine Zeile für jede Indexmethode. Die Unterstützung für den normalen Zugriff auf Tabellen ist in PostgreSQL eingebaut, aber alle Indexmethoden werden durch `pg_am` beschrieben. Es ist auch möglich eine neue Indexmethode zu definieren, indem man die erforderlichen Schnittstellenroutinen schreibt und dann eine Zeile in `pg_am` einfügt. Aber das würde den Rahmen dieses Kapitels sprengen.

Die Routinen für eine Indexmethode wissen selbst nichts über die Datentypen, mit denen die Indexmethoden arbeiten sollen. Stattdessen bestimmt eine *Operatorklasse* einen Satz Operationen, die die Indexmethode verwenden muss, um mit einem Datentyp zu arbeiten. Operatorklassen heißen so, weil sie eine Gruppe Operatoren angeben, die mit dem Index verwendet werden können, wenn Sie in einer `WHERE`-Klausel auftreten (das heißt, die in eine Indexscan-Bedingung umgewandelt werden können). Eine Operatorklasse kann außerdem einige *Unterstützungsprozeduren* bestimmen, die intern von der Indexmethode verwendet werden, aber nichts direkt mit einem `WHERE`-Klausel-Operator, der mit dem Index verwendet werden kann, zu tun haben.

Man kann mehrere Operatorklassen für denselben Datentyp und dieselbe Indexmethode definieren. Damit kann man erreichen, dass ein Datentyp auf verschiedene Arten indiziert werden kann. Zum Beispiel erfordert ein B-Tree-Index, dass für jeden Datentyp, mit dem er arbeiten soll, eine Sortierreihenfolge definiert wird. Für einen Datentyp, der komplexe Zahlen darstellt, könnte es nützlich sein, eine B-Tree-Operatorklasse zu haben, die die Daten nach dem Betrag der komplexen Zahl sortiert, und eine andere, die nach Realteil sortiert, und so weiter. Normalerweise wird eine der Operatorklassen, die man für die am häufigsten verwendete hält, als Standardoperatorklasse für den entsprechenden Datentyp und die Indexmethode markiert.

Derselbe Operatorklassenname kann für mehrere verschiedene Indexmethoden verwendet werden (zum Beispiel haben sowohl die B-Tree- als auch die Hash-Indexmethode eine Operatorklasse mit dem Namen `oid_ops`), aber eine jede solche Klasse ist ein unabhängiges Objekt und muss gesondert definiert werden.

33.14.2 Indexmethodenstrategien

Die zu einer Operatorklasse gehörenden Operatoren werden durch “Strategienummern” identifiziert, die anzeigen, welche Bedeutung jeder Operator im Zusammenhang mit der Operatorklasse hat. Da zum Beispiel B-Trees die Schlüssel strikt sortieren, kleiner nach größer, sind für B-Trees Operatoren wie “kleiner als” und “größer als oder gleich” von Interesse. Weil Benutzer in PostgreSQL eigene Operatoren definieren können, kann PostgreSQL nicht einfach vom Namen des Operators (z.B. < oder >=) ableiten, was für einen Vergleich er durchführt. Stattdessen definiert die Indexmethode einen Satz “Strategien”, welche man sich auch als verallgemeinerte Operatoren vorstellen kann. Jede Operatorklasse gibt an, welche echten Operatoren welchen Strategien für einen bestimmten Datentyp und eine bestimmte Indizierungsvariante entsprechen.

Die B-Tree-Indexmethode definiert fünf Strategien, welche in Tabelle 33.2 gezeigt werden.

Operation	Strategienummer
kleiner als	1
kleiner als oder gleich	2
gleich	3
größer als oder gleich	4
größer als	5

Tabelle 33.2: B-Tree-Strategien

Hash-Indexe drücken nur die bitweise Übereinstimmung aus und verwenden daher nur eine Strategie, welche in Tabelle 33.3 gezeigt wird.

Operation	Strategienummer
gleich	1

Tabelle 33.3: Hash-Strategien

R-Tree-Indexe speichern, welche Objekte in Rechtecken enthalten sind. Sie verwenden acht Strategien, welche in Tabelle 33.4 gezeigt werden.

Operation	Strategienummer
links von	1
links von oder überschneidend	2
überschneidend	3
rechts von oder überschneidend	4
rechts von	5
identisch	6
schließt ein	7
ist eingeschlossen von	8

Tabelle 33.4: R-Tree-Strategien

GiST-Indexe sind noch flexibler: Sie verwenden überhaupt keinen festen Satz Strategien. Stattdessen interpretiert die Unterstützungsroutine “Konsistenz” einer jeden GiST-Operatorklasse die Strategienummern, wie sie möchte.

Beachten Sie, dass alle Strategieoperatoren Boole'sche Ergebnisse liefern. In der Praxis müssen alle als Indexmethodenstrategie definierten Operatoren den Rückgabetyp boolean haben, weil sie in der obersten Ebene der `WHERE`-Klausel erscheinen müssen, um mit einem Index verwendet werden zu können.

Übrigens gibt die Spalte `amorderstrategy` in `pg_am` an, ob die Indexmethode geordnete Scans unterstützt. Null bedeutet, dass sie es nicht tut; wenn sie es doch tut, enthält `amorderstrategy` die Nummer der Strategie, die dem Sortieroperator entspricht. Beim B-Tree ist zum Beispiel `amorderstrategy` = 1, was die Strategienummer für “kleiner als” ist.

33.14.3 Indexmethoden-Unterstützungsroutinen

Strategien reichen in der Regel nicht aus, damit das System weiß, wie es einen Index zu verwenden hat. Meistens benötigen die Indexmethoden zusätzliche Unterstützungsroutinen, um ihre Arbeit zu erledigen. Zum Beispiel muss die B-Tree-Indexmethode zwei Schlüssel vergleichen können und ermitteln, ob einer kleiner als, gleich groß wie oder größer als der andere ist. Die R-Tree-Indexmethode muss Schnittflächen, Vereinigungsflächen und Größen von Rechtecken berechnen können. Diese Operationen entsprechen keinen Operatoren, die in den Bedingungen in SQL-Befehlen verwendet werden; sie sind interne Verwaltungsroutinen der Indexmethoden.

Wie bei den Strategien gibt die Operatorklasse an, welche genaue Funktion jede dieser Rollen für einen bestimmten Datentyp und eine Indizierungsvariante ausfüllen soll. Die Indexmethode definiert, welche Funktionen sie benötigt, und die Operatorklasse identifiziert die richtigen Funktionen, indem sie ihnen die Nummern der Unterstützungsfunktionen zuordnet.

B-Trees benötigen eine einzige Unterstützungsfunktion, welche in Tabelle 33.5 gezeigt wird.

Funktion	Nummer
Vergleiche zwei Schlüssel und gebe eine ganze Zahl kleiner als null, gleich null oder größer als null zurück, die angibt, ob der erste Schlüssel kleiner als, gleich groß wie oder größer als der zweite ist.	1

Tabelle 33.5: B-Tree-Unterstützungsfunktionen

Hash-Indexe benötigen ebenfalls eine Unterstützungsfunktion, welche in Tabelle 33.6 gezeigt wird.

Funktion	Nummer
Berechne den Hash-Wert für einen Schlüssel	1

Tabelle 33.6: Hash-Unterstützungsfunktionen

R-Tree-Indexe benötigen drei Unterstützungsfunktionen, welche in Tabelle 33.7 gezeigt werden.

Funktion	Nummer
Vereinigungsfläche	1
Schnittfläche	2
Größe	3

Tabelle 33.7: R-Tree-Unterstützungsfunktionen

GiST-Indexe benötigen sieben Unterstützungsfunktionen, welche in Tabelle 33.8 gezeigt werden.

Funktion	Nummer
consistent	1
union	2
compress	3
decompress	4
penalty	5
picksplit	6
equal	7

Tabelle 33.8: GiST-Unterstützungsfunktionen

Im Gegensatz zu den Strategieoperatoren geben die Unterstützungsfunktionen den Datentyp zurück, den die bestimmte Indexmethode eben verlangt, zum Beispiel bei der Vergleichsfunktion für B-Trees den Typ integer.

33.14.4 Ein Beispiel

Jetzt, da wir uns mit der Theorie vertraut gemacht haben, kommt hier das versprochene Beispiel für die Erzeugung einer neuen Operatorklasse. Als Namen wählen wir `complex_abs_ops`, weil die Operatorklasse für den Typ complex ist und die Werte nach ihrem Betrag (*absolute value*) sortiert. Als erstes benötigen wir eine Gruppe von Operatoren. Wie man neue Operatoren definiert, wurde in Abschnitt 33.11 beschrieben. Für eine Operatorklasse für B-Trees benötigen wir folgende Operatoren:

- Ist Betrag kleiner als? (Strategie 1)
- Ist Betrag kleiner als oder gleich? (Strategie 2)
- Ist Betrag gleich? (Strategie 3)
- Ist Betrag größer als oder gleich? (Strategie 4)
- Ist Betrag größer als? (Strategie 5)

Der C-Code für den Ist-gleich-Operator sieht so aus:

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

bool
complex_abs_eq(Complex *a, Complex *b)
{
    double amag = Mag(a), bmag = Mag(b);
    return (amag == bmag);
}
```

Die anderen vier Operatoren sind sehr ähnlich. Den Code dafür können Sie in `src/tutorial/complex.c` und `src/tutorial/complex.sql` in der Quellcode-Distribution finden.

Jetzt deklarieren Sie die Funktionen und die Operatoren, die auf diesen Funktionen aufbauen:

```
CREATE FUNCTION complex_abs_eq(complex, complex) RETURNS boolean
    AS 'dateiname', 'complex_abs_eq'
    LANGUAGE C;
```

```
CREATE OPERATOR = (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_abs_eq,
    restrict = eqsel,
    join = eqjoinsel
);
```

Es ist wichtig, dass die Auswahl- und Verbundselektivitätsschätzfunktionen angegeben werden, ansonsten kann der Optimierer den Index später nicht effektiv verwenden. Beachten Sie, dass die Operatoren für kleiner als, gleich und größer als verschiedene Schätzfunktionen verwenden sollten.

Weitere erwähnenswerte Dinge passieren hier:

- Es kann nur einen Operator namens = geben, bei dem beide Operanden den Typ complex haben. In diesem Fall gibt es keinen anderen Operator namens = für den Typ complex, aber wenn wir einen praktischen Datentyp bauen würden, dann würden wir den Operatornamen = wahrscheinlich für den normalen Vergleich von komplexen Zahlen nehmen (und nicht für den Vergleich des Betrags). In dem Fall müssten wir einen anderen Operatornamen für `complex_abs_eq` verwenden.
- PostgreSQL kann mit Funktionen umgehen, die den gleichen Namen haben, solange sie unterschiedliche Argumentdatentypen haben. Aber in C kann es nur eine globale Funktion mit einem bestimmten Namen geben. Also sollten wir der C-Funktion keinen einfachen Namen wie `abs_eq gegeben`. Es ist in der Regel zu empfehlen, den Namen des Datentyps in den Namen der C-Funktion einzubauen, damit keine Konflikte mit Funktionen für andere Datentypen entstehen.
- Wir hätten der Funktion innerhalb von PostgreSQL den Namen `abs_eq` geben und uns darauf verlassen können, dass PostgreSQL sie von anderen Funktionen mit dem gleichen Namen durch die Argumentdatentypen unterscheiden kann. Aber um das Beispiel einfach zu halten, haben wir der Funktion in PostgreSQL den gleichen Namen gegeben wie in C.

Der nächste Schritt ist die Registrierung der von B-Trees benötigten Unterstützungsroutine. Den Beispielcode, der diese Routine implementiert, finden Sie in der selben Datei, die die Operatorfunktionen enthält. So wird die Funktion deklariert:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'dateiname'
     LANGUAGE C;
```

Jetzt haben wir die benötigten Operatoren und die Unterstützungsroutine und können endlich die Operatorklasse erzeugen:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Das war's! Jetzt sollte es möglich sein, B-Tree-Indexe für Spalten vom Typ complex zu erzeugen und zu verwenden.

Wir hätten die Operatoreinträge auch ausführlicher schreiben können, wie

```
        OPERATOR        1       < (complex, complex) ,
```

mpl aber dazu besteht keine Notwendigkeit, wenn die Operandendatentypen des Operators mit dem Datentyp, für den die Operatorklasse definiert wird, identisch sind.

Das obige Beispiel geht davon aus, dass Sie die neue Operatorklasse zur Standardoperatorklasse für B-Trees und den Datentyp complex machen wollen. Wenn nicht, lassen Sie einfach das Wort `DEFAULT` weg.

33.14.5 Optionale Merkmale von Operatorklassen

Es gibt zwei optionale Merkmale von Operatorklassen, die wir noch nicht besprochen haben, hauptsächlich weil sie für die Standardmethode B-Tree nicht besonders nützlich sind.

Normalerweise bedeutet es, wenn ein Operator ein Mitglied einer Operatorklasse ist, dass dann die Indexmethode genau die gleichen Zeilen auswählt wie eine `WHERE`-Bedingung, die den Operator verwendet. Zum Beispiel kann

```
SELECT * FROM tabelle WHERE integer_spalte < 4;
```

genau von einem B-Tree-Index für die verwendete Spalte bedient werden. Aber es gibt Fälle, in denen ein Index als ungenauer Hinweis auf die übereinstimmenden Zeilen nützlich ist. Wenn zum Beispiel ein R-Tree-Index nur Rechtecke speichert, die die Objekte umschließen, kann es keiner `WHERE`-Bedingung genau entsprechen, die das Überschneiden zweier nicht rechteckiger Objekte, wie Polygone, prüft. Aber wir könnten den Index verwenden, um Objekte zu finden, deren umschließendes Rechteck sich mit dem umschließenden Rechteck des Zielobjekts überschneidet, und dann die richtige Überschneidungsprüfung nur mit den vom Index gefundenen Objekten durchführen. Wenn dieses Szenario zutrifft, nennt man den Index für den Operator verlustbehaftet und wir fügen der `OPERATOR`-Klausel im Befehl `CREATE OPERATOR CLASS` das Schlüsselwort `RECHECK` hinzu. `RECHECK` gibt an, dass der Index garantiert alle erforderlichen Zeilen zurückgibt, plus möglicherweise zusätzliche Zeilen, welche dann ausgeschlossen werden können, indem der ursprüngliche Operatoraufruf durchgeführt wird.

Betrachten wir noch einmal die Situation, in der wir nur das umschließende Rechteck eines komplexen Objekts, wie eines Polygons, im Index speichern. In dem Fall hat es nicht viel Sinn, das ganze Polygon im Indexeintrag zu speichern; wir könnten auch ein einfacheres Objekt vom Typ box speichern. Diese Situation wird durch die Option `STORAGE` in `CREATE OPERATOR CLASS` ausgedrückt: Wir könnten etwa Folgendes schreiben:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

Gegenwärtig unterstützt nur die Indexmethode GiST einen `STORAGE`-Typ, der sich vom Datentyp der Spalte unterscheidet. Die GiST-Routinen `compress` und `decompress` müssen sich um die Datentypumwandlung kümmern, wenn `STORAGE` verwendet wird.

34

Server Programming Interface

Mit dem *Server Programming Interface* (SPI) können Benutzer in benutzerdefinierten C-Funktionen SQL-Befehle ausführen. SPI besteht aus einer Sammlung von Befehlen, die eine vereinfachte Schnittstelle zum Parser, Planer, Optimierer und Executor von PostgreSQL darstellen. Außerdem übernimmt SPI einige Aspekte der Speicherverwaltung.

Um Missverständnissen vorzubeugen, verwenden wir im Folgenden den Begriff “Funktion”, wenn wir von einer Funktion der SPI-Schnittstelle sprechen, und den Begriff “Prozedur”, wenn wir von einer benutzerdefinierten C-Funktion sprechen, die möglicherweise SPI verwendet.

Beachten Sie, dass, wenn in einer Prozedur wegen eines Fehlers in einem Befehl die Transaktion abgebrochen wird, die Kontrolle dann nicht wieder an ihre Prozedur zurückkehrt. Stattdessen werden alle Änderungen zurückgerollt und der Server wartet auf den nächsten Befehl vom Client. Außerdem kann man in Prozeduren die Transaktionskontrollbefehle `BEGIN`, `COMMIT` und `ROLLBACK` nicht ausführen. Diese beiden Einschränkungen werden wahrscheinlich in der Zukunft einmal geändert werden.

Bei Erfolg geben die SPI-Funktionen ein nicht negatives Ergebnis zurück (entweder direkt im Rückgabewert oder in der globalen Variable `SPI_result`, wie unten beschrieben). Bei einem Fehler wird ein negatives Ergebnis oder `NULL` zurückgegeben.

Quellcodedateien, die SPI verwenden, müssen die Headerdatei `executor/spi.h` einbinden.

Anmerkung

Die vorhandenen prozeduralen Sprachen bieten andere Wege, in Prozeduren SQL-Befehle auszuführen. Einige davon sind der SPI-Schnittstelle nachempfunden, daher kann diese Dokumentation für Benutzer dieser Sprachen trotzdem nützlich sein.

34.1 Schnittstellenfunktionen

SPI_connect

Name

SPI_connect – verbindet eine Prozedur mit dem SPI-Manager

Synopsis

```
int SPI_connect(void)
```

Beschreibung

`SPI_connect` öffnet eine Verbindung von dem Prozedureaufruf zum SPI-Manager. Sie müssen diese Funktion aufrufen, wenn Sie Befehle über SPI ausführen wollen. Einige SPI-Unterstützungsfunktionen können auch aus nicht verbundenen Prozeduren aufgerufen werden.

Wenn Ihre Prozedur schon verbunden ist, gibt `SPI_connect` den Fehlercode *SPI_ERROR_CONNECT* zurück. Das kann auch passieren, wenn eine Prozedur, die `SPI_connect` aufgerufen hat, direkt eine andere Prozedur aufruft, die selbst ebenfalls `SPI_connect` aufruft. Rekursive Aufrufe des SPI-Managers sind erlaubt, wenn ein durch SPI ausgeführter SQL-Befehl eine andere Prozedur aufruft, die selbst SPI verwendet, aber direkte, verschachtelte Aufrufe von `SPI_connect` und `SPI_finish` sind verboten.

Rückgabewert

`SPI_OK_CONNECT`

Bei Erfolg.

`SPI_ERROR_CONNECT`

Bei einem Fehler.

SPI_finish

Name

SPI_finish – beendet die Verbindung einer Prozedur mit dem SPI-Manager

Synopsis

```
int SPI_finish(void)
```

Beschreibung

`SPI_finish` schließt eine bestehende Verbindung mit dem SPI-Manager. Sie müssen diese Funktion aufrufen, nachdem Sie alle SPI-Operationen, die im aktuellen Aufruf Ihrer Prozedur vorgesehen sind, abgeschlossen haben. Wenn Sie die Transaktion jedoch mit `elog(ERROR)` abbrechen, müssen Sie sich darum nicht kümmern, denn dann räumt SPI automatisch selbst auf.

`SPI_finish` gibt den Fehlercode *SPI_ERROR_UNCONNECTED* zurück, wenn es aufgerufen wird und keine gültige Verbindung besteht. Das ist kein grundlegendes Problem; es bedeutet, dass der SPI-Manager nichts tun muss.

Rückgabewert

`SPI_OK_FINISH`

Wenn die Verbindung ordnungsgemäß beendet wurde.

`SPI_ERROR_UNCONNECTED`

Wenn die Prozedur nicht verbunden ist.

SPI_exec

Name

SPI_exec – führt einen Befehl aus

Synopsis

```
int SPI_exec(char * befehl, int anzahl)
```

Beschreibung

`SPI_exec` führt den angegebenen SQL-Befehl für die angegebene Anzahl von Zeilen aus.

Diese Funktion sollte nur aus einer verbundenen Prozedur aufgerufen werden. Wenn *`anzahl`* null ist, wird der Befehl für alle Zeilen ausgeführt, auf die er zutrifft. Wenn *`anzahl`* größer als null ist, wird die Anzahl der Zeilen, für die der Befehl ausgeführt wird, beschränkt (wie mit der `LIMIT`-Klausel). Dieser Befehl wird zum Beispiel höchstens fünf Zeilen in die Tabelle einfügen:

```
SPI_exec("INSERT INTO tab SELECT * FROM tab", 5);
```

Sie können in der Befehlszeichenkette mehrere Befehle angeben und die Befehl könnten auch durch Regeln umgeschrieben werden. `SPI_exec` gibt das Ergebnis des zuletzt ausgeführten Befehls zurück.

Die Anzahl der Zeilen, für die der (letzte) Befehl tatsächlich ausgeführt wurde, wird in der globalen Variablen `SPI_processed` zurückgegeben (es sei denn, der Rückgabewert der Funktion ist *SPI_OK_UTILITY*). Wenn *SPI_OK_SELECT* zurückgegeben wird, können Sie über den globalen Zeiger `SPITupleTable *SPI_tuptable` auf die Ergebniszeilen zugreifen.

Die Struktur `SPITupleTable` ist folgendermaßen definiert:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* Speicherkontext der Ergebnistabelle */
    uint32      alloced;        /* Anzahl der Werte */
    uint32      free;           /* Anzahl leerer Werte */
    TupleDesc   tupdesc;        /* Zeilenbeschreibung */
    HeapTuple  *vals;           /* Zeilen */
} SPITupleTable;
```

`vals` ist ein Array aus Zeigern auf Zeilen. (Die Anzahl der gültigen Einträge wird durch `SPI_processed` angegeben.) `tupdesc` ist eine Zeilenbeschreibung, die Sie den SPI-Funktionen, die mit Zeilen umgehen, übergeben können. Die Felder `tuptabcxt`, `alloced` und `free` sind für interne Zwecke und nicht für die Verwendung durch SPI-Benutzer gedacht.

`SPI_finish` gibt alle in der aktuellen Prozedur erzeugten `SPITupleTable`-Strukturen frei. Wenn Sie eine Ergebnistabelle eher freigeben wollen, verwenden Sie `SPI_freetuptable`.

Argumente

`char * befehl`

Zeichenkette mit dem auszuführenden SQL-Befehl.

`int anzahl`

Maximale Anzahl Zeilen, die verarbeitet oder zurückgegeben werden sollen.

Rückgabewert

Wenn die Ausführung des Befehls erfolgreich war, wird einer der folgenden (nicht negativen) Werte zurückgegeben:

`SPI_OK_SELECT`

Wenn ein `SELECT` (aber nicht `SELECT ... INTO`) ausgeführt wurde.

`SPI_OK_SELINTO`

Wenn ein `SELECT ... INTO` ausgeführt wurde.

`SPI_OK_DELETE`

Wenn ein `DELETE` ausgeführt wurde.

`SPI_OK_INSERT`

Wenn ein `INSERT` ausgeführt wurde.

`SPI_OK_UPDATE`

Wenn ein `UPDATE` ausgeführt wurde.

`SPI_OK_UTILITY`

Wenn ein Unterstützungsbefehl (z.B. `CREATE TABLE`) ausgeführt wurde.

Bei einem Fehler wird einer der folgenden negativen Werte zurückgegeben:

`SPI_ERROR_ARGUMENT`

Wenn `befehl NULL` ist oder *`anzahl`* kleiner als 0 ist.

`SPI_ERROR_COPY`

Wenn ein `COPY TO stdout` oder `COPY FROM stdin` versucht wurde.

`SPI_ERROR_CURSOR`

Wenn ein `DECLARE`, `CLOSE` oder `FETCH` versucht wurde.

`SPI_ERROR_TRANSACTION`

Wenn ein `BEGIN`, `COMMIT` oder `ROLLBACK` versucht wurde.

`SPI_ERROR_OPUNKNOWN`

Wenn der Typ des Befehls unbekannt ist (sollte nicht passieren können).

`SPI_ERROR_UNCONNECTED`

Wenn die Prozedur nicht verbunden ist.

Hinweise

Die Funktionen `SPI_exec`, `SPI_execp` und `SPI_prepare` ändern `SPI_processed` und `SPI_tuptable` (nur den Zeiger, nicht den Inhalt der Struktur). Sichern Sie diese globalen Variablen in lokalen Prozedurvariablen, wenn Sie die Ergebnisse von `SPI_exec` oder `SPI_execp` nach weiteren Aufrufen noch verwenden wollen.

SPI_prepare

Name

SPI_prepare – bereitet einen Plan für einen Befehl vor, ohne ihn schon auszuführen

Synopsis

```
void * SPI_prepare(char * befehl, int narg, Oid * argtypen)
```

Beschreibung

`SPI_prepare` erzeugt einen Ausführungsplan für einen Befehl und gibt ihn zurück, führt den Befehl aber nicht aus. Diese Funktion sollte nur aus einer verbundenen Prozedur aufgerufen werden.

Wenn derselbe oder ähnliche Befehle wiederholt ausgeführt werden müssen, kann es von Vorteil sein, das Planen nur einmal durchzuführen. `SPI_prepare` wandelt eine Befehlszeichenkette in einen Ausführungsplan um, der wiederholt mit `SPI_execp` ausgeführt werden kann.

Ein vorbereiteter Befehl kann allgemein gehalten werden, indem an den Stellen, wo im normalen Befehl Konstanten stehen würden, Parameter (`$1`, `$2` usw.) geschrieben werden. Die tatsächlichen Werte für die Parameter werden dann angegeben, wenn `SPI_execp` aufgerufen wird. Damit kann ein vorbereiteter Befehl in vielfältigeren Situationen verwendet werden, als ohne Parameter möglich wäre. Wenn ein Befehl Parameter verwendet, müssen ihre Zahl und ihre Datentypen im Aufruf von `SPI_prepare` angegeben werden.

Der von `SPI_prepare` zurückgegebene Plan kann nur im aktuellen Aufruf der Prozedur verwendet werden, weil `SPI_finish` den von dem Plan belegten Speicher freigibt. Mit der Funktion `SPI_saveplan` können Sie einen Plan länger speichern.

Argumente

`char * befehl`

Befehlszeichenkette

`int narg`

Anzahl der Eingabeparameter (`$1`, `$2` usw.).

`Oid * argtypen`

Zeiger auf ein Array mit den OIDs der Datentypen der Eingabeparameter.

Rückgabewert

`SPI_prepare` gibt einen gültigen Zeiger (nicht `NULL`) auf einen Ausführungsplan zurück. Bei einem Fehler wird `NULL` zurückgegeben. `SPI_result` wird auf Werte analog zu den möglichen Rückgabewerten von `SPI_exec` gesetzt, außer dass es auf *SPI_ERROR_ARGUMENT* gesetzt wird, wenn `befehl NULL` ist, oder wenn `narg` kleiner als 0 ist, oder wenn `narg` größer als 0 und `argtypen NULL` ist.

Hinweise

Es gibt bei der Verwendung von Parametern einen Nachteil: Da der Planer die Werte, die für die Parameter eingesetzt werden, nicht kennt, könnte er schlechtere Pläne wählen, als wenn er in einem normalen Befehl alle Konstanten kennt.

SPI_execp

Name

SPI_execp – führt einen von SPI_prepare vorbereiteten Plan aus

Synopsis

```
int SPI_execp(void * plan, Datum * werte, char * nulls, int anzahl)
```

Beschreibung

`SPI_execp` führt einen von `SPI_prepare` vorbereiteten Plan aus. `anzahl` hat dieselbe Bedeutung wie bei `SPI_exec`.

Argumente

`void * plan`

Ausführungsplan (von `SPI_prepare` zurückgegeben).

`Datum * werte`

Tatsächliche Parameterwerte.

`char * nulls`

Ein Array, das beschreibt, welche Parameter den `NULL`-Wert haben. Ein `NULL`-Wert wird durch `n` angezeigt (Eintrag in `werte` wird ignoriert), ein Leerzeichen zeigt einen normalen Wert an (Eintrag in `werte` ist gültig).

Wenn `nulls` `NULL` ist, dann wird davon ausgegangen, dass keine Parameter den `NULL`-Wert haben.

`int anzahl`

Anzahl Zeilen, für die der Plan ausgeführt werden soll.

Rückgabewert

Der Rückgabewert ist der gleiche wie bei `SPI_exec` oder einer der folgenden:

`SPI_ERROR_ARGUMENT`

Wenn `plan` `NULL` ist oder wenn `anzahl` kleiner als 0 ist.

`SPI_ERROR_PARAM`

Wenn `werte` `NULL` ist und `plan` mit Parametern vorbereitet wurde.

Bei Erfolg werden `SPI_processed` und `SPI_tuptable` wie bei `SPI_exec` gesetzt.

Hinweise

Wenn eines der Objekte (eine Tabelle, Funktion usw.), die von dem vorbereiteten Plan verwendet werden, während der Sitzung gelöscht wird, sind die Ergebnisse von `SPI_execp` undefiniert.

SPI_cursor_open

Name

SPI_cursor_open – richtet einen Cursor für einen Plan von SPI_prepare ein

Synopsis

```
Portal SPI_cursor_open(char * name, void * plan, Datum * werte, char * nulls)
```

Beschreibung

`SPI_cursor_open` richtet eine Cursor (intern Portal genannt) ein, der einen von `SPI_prepare` vorbereiteten Plan ausführen wird.

Die Verwendung eines Cursors hat gegenüber der direkten Ausführung des Plans zwei Vorteile. Erstens kann man die Ergebniszeilen in kleinen Gruppen abrufen und so verhindern, dass bei vielen Zeilen der Speicher überlauft. Zweitens kann ein Portal über die aktuelle Prozedur hinaus erhalten bleiben (es kann genau gesagt bis zum Ende der Transaktion bestehen bleiben). Wenn man den Portalnamen aus der Prozedur zurückgibt, kann man somit eine Ergebnismenge zurückgeben.

Argumente

`char * name`

Name für das Portal oder `NULL`, um das System einen Namen wählen zu lassen.

`void * plan`

Ausführungsplan (von `SPI_prepare` zurückgegeben).

`Datum * values`

Tatsächliche Parameterwerte.

`char * nulls`

Ein Array, das beschreibt, welche Parameter den `NULL`-Wert haben. Ein `NULL`-Wert wird durch `n` angezeigt (Eintrag in `werte` wird ignoriert; ein Leerzeichen zeigt einen normalen Wert an (Eintrag in `werte` ist gültig).

Wenn `nulls` `NULL` ist, wird davon ausgegangen, dass keine Parameter den `NULL`-Wert haben.

Rückgabewert

Zeiger auf ein Portal mit dem Cursor oder `NULL` bei einem Fehler.

SPI_cursor_find

Name

SPI_cursor_find – findet einen bestehenden Cursor anhand des Namens

Synopsis

```
Portal SPI_cursor_find(char * name)
```

Beschreibung

`SPI_cursor_find` findet ein bestehendes Portal anhand des Namens. Das ist hauptsächlich nützlich, um Cursornamen aufzulösen, die aus anderen Funktionen als Text zurückgegeben wurden.

Argumente

`char * name`

Name des Portals.

Rückgabewert

Zeiger auf das Portal mit dem angegebenen Namen oder `NULL`, wenn keines gefunden wurde.

SPI_cursor_fetch

Name

SPI_cursor_fetch – liest Zeilen aus einem Cursor.

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool vorwaerts, int anzahl)
```

Beschreibung

`SPI_cursor_fetch` liest einige Zeilen aus einem Cursor. Das entspricht dem SQL-Befehl `FETCH`.

Argumente

`Portal` *`portal`*

Portal des Cursors.

`bool` *`vorwaerts`*

Wahr, um vorwärts zu lesen, falsch, um rückwärts zu lesen.

`int` *`anzahl`*

Maximale Anzahl zu lesender Zeilen.

Rückgabewert

Bei Erfolg werden `SPI_processed` und `SPI_tuptable` wie bei `SPI_exec` gesetzt.

SPI_cursor_move

Name

SPI_cursor_move – ändert die aktuelle Position eines Cursors

Synopsis

```
void SPI_cursor_move(Portal portal, bool vorwaerts, int anzahl)
```

Beschreibung

`SPI_cursor_move` überspringt eine Anzahl von Zeilen in einem Cursor. Das entspricht dem SQL-Befehl `MOVE`.

Argumente

`Portal` *`portal`*

Portal des Cursors

`bool` *`vorwaerts`*

Wahr, um vorwärts zu gehen, falsch, um rückwärts zu gehen.

`int` *`anzahl`*

Maximale Anzahl zu überspringender Zeilen,

SPI_cursor_close

Name

SPI_cursor_close – schließt einen Cursor

Synopsis

```
void SPI_cursor_close(Portal portal)
```

Beschreibung

`SPI_cursor_close` schließt einen zuvor erzeugten Cursor und gibt die Portalstrukturen frei.

Alle offenen Cursor werden automatisch am Ende der Transaktion geschlossen. `SPI_cursor_close` muss nur aufgerufen werden, wenn man die Ressourcen früher freigeben möchte.

Argumente

`Portal portal`

Portal des Cursors

SPI_saveplan

Name

SPI_saveplan – speichert einen Plan

Synopsis

```
void * SPI_saveplan(void * plan)
```

Beschreibung

l`SPI_saveplan` speichert den übergebenen Plan (der mit `SPI_prepare` vorbereitet wurde) in einem Speicherbereich, der nicht von `SPI_finish` oder dem Transaktionsmanager gelöscht wird, und gibt einen Zeiger auf den gespeicherten Plan zurück. Damit kann man vorbereitete Pläne in den folgenden Aufrufen der Prozedur in der aktuellen Sitzung wiederverwenden. Den zurückgegebenen Zeiger kann man in einer lokalen Variablen speichern. Prüfen Sie immer, ob dieser Zeiger `NULL` ist, entweder wenn der Plan vorbereitet wird oder wenn Sie einen vorbereiteten Plan an `SPI_execp` übergeben.

Argumente

`void * plan`

Zu speichernder Plan.

Rückgabewert

Zeiger auf den gespeicherten Plan; `NULL`, wenn nicht erfolgreich. Bei einem Fehler wird `SPI_result` wie folgt gesetzt:

`SPI_ERROR_ARGUMENT`

Wenn `plan NULL` ist.

`SPI_ERROR_UNCONNECTED`

Wenn die Prozedur nicht verbunden ist.

Hinweise

Wenn eines der Objekte (eine Tabelle, Funktion usw.), die von dem vorbereiteten Plan verwendet werden, während der Sitzung gelöscht wird, dann sind die Ergebnisse von `SPI_execp` undefiniert.

34.2 Schnittstellenunterstützungsfunktionen

Mit den hier beschriebenen Funktionen kann man bequem auf die Ergebnismengen, die von `SPI_exec` und anderen SPI-Funktionen zurückgegeben werden, zugreifen.

Alle in diesem Abschnitt beschriebenen Funktionen können in verbundenen und nicht verbundenen Prozeduren verwendet werden.

SPI_fname

Name

SPI_fname – ermittelt den Spaltennamen für die angegebene Spaltennummer

Synopsis

```
char * SPI_fname(TupleDesc zeilenbeschreibung, int spaltennummer)
```

Beschreibung

`SPI_fname` gibt den Spaltennamen der angegebenen Spalte zurück. (Sie können die Kopie des Namens mit `pfree` freigeben, wenn Sie ihn nicht mehr benötigen.)

Argumente

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile

`int` *`spaltennummer`*

Spaltennummer (Zählung fängt bei 1 an)

Rückgabewert

Der Spaltenname; `NULL`, wenn `spaltennummer` außerhalb des gültigen Bereichs ist. Bei einem Fehler wird `SPI_result` auf `SPI_ERROR_NOATTRIBUTE` gesetzt.

SPI_fnumber

Name

SPI_fnumber – ermittelt die Spaltennummer für den angegebenen Spaltennamen

Synopsis

```
int SPI_fnumber(TupleDesc zeilenbeschreibung, char * spaltenname)
```

Beschreibung

`SPI_fnumber` gibt die Spaltennummer für die Spalte mit dem angegebenen Namen zurück.

Wenn `spaltenname` auf eine Systemspalte verweist (z.B. `oid`), wird die entsprechende negative Spaltennummer zurückgegeben. Die Prozedur sollte den Rückgabewert genau mit `SPI_ERROR_NOATTRIBUTE` vergleichen, um Fehler zu entdecken; ein Vergleich auf kleiner oder gleich 0 ist nicht korrekt, es sei denn, man möchte keine Systemspalten verarbeiten.

Argumente

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile.

`char *` *`spaltenname`*

Spaltenname

Rückgabewert

Nummer der Spalte (Zählung fängt bei 1 an) oder `SPI_ERROR_NOATTRIBUTE`, wenn die genannte Spalte nicht gefunden wurde.

SPI_getvalue

Name

SPI_getvalue – gibt die Zeichenkettendarstellung der angegebenen Spalte zurück

Synopsis

```
char * SPI_getvalue(HeapTuple zeile, TupleDesc zeilenbeschreibung, int
spaltennummer)
```

Beschreibung

`SPI_getvalue` gibt die externe Zeichenkettendarstellung des Werts der angegebenen Spalte zurück.

Das Ergebnis wird in mit `palloc` zugeteiltem Speicher abgelegt. (Sie können den Speicher mit `pfree` freigeben, wenn Sie ihn nicht mehr benötigen.)

Argumente

`HeapTuple` *`zeile`*

Zu untersuchende Eingabezeile.

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile.

`int` *`spaltennummer`*

Spaltennummer (Zählung fängt bei 1 an).

Rückgabewert

Spaltenwert oder `NULL`, wenn die Spalte den `NULL`-Wert hat, `spaltennummer` außerhalb des gültigen Bereichs ist (`SPI_result` wird auf `SPI_ERROR_NOATTRIBUTE` gesetzt) oder keine Ausgabefunktion verfügbar ist (`SPI_result` wird auf `SPI_ERROR_NOOUTFUNC` gesetzt).

SPI_getbinval

Name

SPI_getbinval – gibt den binären Wert der angegebenen Spalte zurück

Synopsis

```
Datum SPI_getbinval(HeapTuple zeile, TupleDesc zeilenbeschreibung, int
spaltennummer, bool * ist_null)
```

Beschreibung

`SPI_getbinval` gibt den Wert der angegebenen Spalte in der internen Form (als Typ Datum) zurück.

Diese Funktion legt keinen neuen Speicher für den Wert an. Bei einem Datentyp mit Referenzübergabe ist der Rückgabewert ein Zeiger in die übergebene Zeile.

Argumente

`HeapTuple` *`zeile`*

Zu untersuchende Eingabezeile.

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile.

`int` *`spaltennummer`*

Spaltennummer (Zählung fängt bei 1 an).

`bool *` *`ist_null`*

Signalisiert einen `NULL`-Wert in der Spalte.

Rückgabewert

Der binäre Wert der Spalte wird zurückgegeben. Die Variable, auf die `ist_null` zeigt, wird auf wahr gesetzt, wenn die Spalte den `NULL`-Wert hat, ansonsten auf falsch.

Bei einem Fehler wird `SPI_result` auf `SPI_ERROR_NOATTRIBUTE` gesetzt.

SPI_gettype

Name

SPI_gettype – gibt den Datentypnamen der angegebenen Spalte zurück

Synopsis

```
char * SPI_gettype(TupleDesc zeilenbeschreibung, int spaltennummer)
```

Beschreibung

`SPI_gettype` gibt den Namen des Datentyps der angegebenen Spalte zurück. (Sie können die Kopie des Namens mit `pfree` freigeben, wenn Sie ihn nicht mehr benötigen.)

Argumente

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile.

`int` *`spaltennummer`*

Spaltennummer (Zählung fängt bei 1 an).

Rückgabewert

Der Datentypname der angegebenen Spalte oder `NULL` bei einem Fehler. Bei einem Fehler wird `SPI_result` auf `SPI_ERROR_NOATTRIBUTE` gesetzt.

SPI_gettypeid

Name

SPI_gettypeid – gibt die Datentyp-OID der angegebenen Spalte zurück

Synopsis

```
Oid SPI_gettypeid(TupleDesc zeilenbeschreibung, int spaltennummer)
```

Beschreibung

`SPI_gettypeid` gibt die OID des Datentyps der angegebenen Spalte zurück.

Argumente

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der Eingabezeile.

`int` *`spaltennummer`*

Spaltennummer (Zählung fängt bei 1 an).

Rückgabewert

Die OID des Datentyps der angegebenen Spalte oder *InvalidOid* bei einem Fehler. Bei einem Fehler wird SPI_result auf SPI_ERROR_NOATTRIBUTE gesetzt.

SPI_getrelname

Name

SPI_getrelname – gibt den Namen der angegebenen Relation zurück

Synopsis

```
char * SPI_getrelname(Relation rel)
```

Beschreibung

SPI_getrelname gibt den Namen der angegebenen Relation zurück. (Sie können die Kopie des Namens mit pfree freigeben, wenn Sie ihn nicht mehr benötigen.)

Argumente

Relation *rel*

Eingaberelation

Rückgabewert

Der Name der angegebenen Relation.

34.3 Speicherverwaltung

PostgreSQL verwaltet alle dynamisch zugeteilte Speicher in **Speicherkontexten**. Damit lassen sich Speichersegmente, die an vielen verschiedenen Stellen zugeteilt wurden und unterschiedlich lange leben müssen, einfach verwalten. Wenn ein Kontext gelöscht wird, dann wird aller darin enthaltener Speicher freigegeben. Daher muss man sich nicht die einzelnen Objekte merken, um Speicherlecks zu vermeiden, sondern es müssen nur einige wenige Speicherkontexte verwaltet werden. palloc und verwandte Funktionen teilen Speicher im “aktuellen” Kontext zu.

SPI_connect erzeugt einen neuen Speicherkontext und macht ihn zum aktuellen. SPI_finish löscht den von SPI_connect erzeugten Speicherkontext und stellt den vorher aktiven wieder her. Damit wird

sichergestellt, dass aller Speicher, der in ihren Prozeduren zugeteilt wird, am Ende der Prozedur freigegeben wird und somit Speicherlecks vermieden werden.

Wenn ihre Prozedur jedoch ein Objekt im zugeteilten Speicher aus der Prozedur zurückgeben möchte (zum Beispiel einen Wert eines Datentyps mit Referenzübergabe), können Sie diesen Speicher nicht mit `palloc` zuteilen, zumindest nicht, solange Sie mit SPI verbunden sind. Wenn Sie es versuchen, wird ihr Objekt von `SPI_finish` wieder freigegeben und ihre Prozedur wird nicht verlässlich funktionieren. Um dieses Problem zu lösen, verwenden Sie `SPI_palloc` um Speicher für das zurückzugebende Objekt zu erhalten. `SPI_palloc` teilt Speicher aus dem "übergeordneten Executor-Kontext" zu, das heißt, aus dem Speicherkontext, der vor dem Aufruf von `SPI_connect` aktuell war, was genau der richtige Kontext ist um aus Ihrer Prozedur einen Wert zurückzugeben.

Wenn `SPI_palloc` aufgerufen wird, während die Prozedur nicht mit SPI verbunden ist, verhält es sich wie ein normales `palloc`. Bevor eine Prozedur mit dem SPI-Manager verbindet, ist also der übergeordnete Executor-Kontext der aktuelle Speicherkontex,t und der von `palloc` oder SPI-Unterstützungsfunktionen zugeteilte Speicher wird in diesem Kontext angelegt.

Wenn `SPI_connect` aufgerufen wird, wird der private Kontext der Prozedur, der von `SPI_connect` erzeugt wurde, zum aktuellen Kontext. Aller Speicher, der von `palloc`, `repalloc` oder SPI-Unterstützungsfunktionen (außer `SPI_copytuple`, `SPI_copytupledesc`, `SPI_copytupleintoslot`, `SPI_modifytuple` und `SPI_palloc`) zugeteilt wird, wird in diesem Kontext angelegt. Wenn eine Prozedur die Verbindung mit dem SPI-Manager beendet (mit `SPI_finish`), dann wird der übergeordnete Executor-Kontext wieder zum aktuellen Kontext und der im Speicherkontext der Prozedur angelegte Speicher wird freigegeben und kann nicht mehr verwendet werden.

Alle in diesem Abschnitt beschriebenen Funktionen können in verbundenen und in nicht verbundenen Prozeduren verwendet werden. In einer nicht verbundenen Prozedur verhalten sie sich genauso wie die ihnen zugrundeliegenden normalen Serverfunktionen (`palloc` usw.).

SPI_palloc

Name

SPI_palloc – teilt Speicher im übergeordneten Executor-Kontext zu

Synopsis

```
void * SPI_palloc(Size groesse)
```

Beschreibung

`SPI_palloc` teilt Speicher im übergeordneten Executor-Kontext zu.

Argumente

`Size` *`groesse`*

Größe des zuzuteilenden Speichers in Bytes.

Rückgabewert

Zeiger auf einen neuen Speicherbereich der angegebenen Größe.

SPI_repalloc

Name

SPI_repalloc – teilt Speicher im übergeordneten Executor-Kontext neu zu

Synopsis

```
void * SPI_repalloc(void * zeiger, Size groesse)
```

Beschreibung

`SPI_repalloc` ändert die Größe eines Speichersegments, das zuvor mit `SPI_palloc` zugeteilt wurde.

Diese Funktion unterscheidet sich nicht mehr vom normalen `repalloc`. Sie wird nur für die Kompatibilität mit altem Code beibehalten.

Argumente

`void *` *`zeiger`*

Zeiger auf den zu ändernden Speicherbereich

`Size` *`groesse`*

Größe des zuzuteilenden Speichers in Bytes

Rückgabewert

Zeiger auf einen neuen Speicherbereich der angegebenen Größe mit dem Inhalt des alten Bereichs kopiert.

SPI_pfree

Name

SPI_pfree – gibt Speicher im übergeordneten Executor-Kontext frei

Synopsis

```
void SPI_pfree(void * zeiger)
```

Beschreibung

`SPI_pfree` gibt Speicher frei, der von `SPI_palloc` oder `SPI_repalloc` zugeteilt wurde.

Diese Funktion unterscheidet sich nicht mehr vom normalen `pfree`. Sie wird nur für die Kompatibilität mit altem Code beibehalten.

Argumente

`void * zeiger`

Zeiger auf den freizugebenden Speicher.

SPI_copytuple

Name

SPI_copytuple – macht eine Kopie einer Zeile im übergeordneten Executor-Kontext

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple zeile)
```

Beschreibung

`SPI_copytuple` macht eine Kopie einer Zeile im übergeordneten Executor-Kontext.

Argumente

`HeapTuple zeile`

Zu kopierende Zeile.

Rückgabewert

Kopierte Zeile, nicht `NULL`, wenn `zeile` nicht `NULL` ist und die Kopieroperation erfolgreich war, `NULL` nur, wenn `zeile` `NULL` ist.

SPI_copytupledesc

Name

SPI_copytupledesc – macht eine Kopie einer Zeilenbeschreibung im übergeordneten Executor-Kontext

Synopsis

```
TupleDesc SPI_copytupledesc(TupleDesc zeilenbeschreibung)
```

Beschreibung

`SPI_copytupledesc` macht eine Kopie einer Zeilenbeschreibung im übergeordneten Executor-Kontext.

Argumente

`TupleDesc` *`zeilenbeschreibung`*

Zu kopierende Zeilenbeschreibung.

Rückgabewert

Kopierte Zeilenbeschreibung, nicht `NULL`, wenn `zeilenbeschreibung` nicht `NULL` ist und die Kopieroperation erfolgreich war, `NULL` nur, wenn `zeilenbeschreibung` `NULL` ist.

SPI_copytupleintoslot

Name

SPI_copytupleintoslot – macht eine Kopie einer Zeile und einer Beschreibung im übergeordneten Executor-Kontext

Synopsis

```
TupleTableSlot * SPI_copytupleintoslot(HeapTuple zeile, TupleDesc
zeilenbeschreibung)
```

Beschreibung

`SPI_copytupleintoslot` macht eine Kopie einer Zeile im übergeordneten Executor-Kontext und gibt sie in der Form einer gefüllten `TupleTableSlot`-Struktur zurück.

Argumente

`HeapTuple` *`zeile`*

Zu kopierende Zeile.

`TupleDesc` *`zeilenbeschreibung`*

Beschreibung der zu kopierenden Zeile.

Rückgabewert

`TupleTableSlot` mit der kopierten Zeile und Beschreibung, nicht `NULL`, wenn `zeile` und `zeilenbeschreibung` nicht `NULL` sind und die Kopieroperation erfolgreich war, `NULL` nur, wenn `zeile` oder `zeilenbeschreibung` `NULL` ist.

SPI_modifytuple

Name

SPI_modifytuple – erzeugt eine Zeilenstruktur, indem ausgewählte Spalten in der angegebenen Zeilenstruktur ersetzt werden

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple zeile, int nspalten, int *
spalten, Datum * werte, char * nulls)
```

Beschreibung

`SPI_modifytuple` erzeugt eine neue Zeilenstruktur, indem in ausgewählten Spalten neue Werte eingesetzt und in den übrigen Zeilen die Werte aus der alten Zeile kopiert werden. Die Eingabezeile wird nicht verändert.

Argumente

`Relation` *`rel`*

Wird nur als Quelle für die Zeilenbeschreibung der Eingabezeile verwendet. (Sinnvoller wäre es, die Zeilenbeschreibung direkt zu übergeben, aber das lässt sich jetzt nicht mehr ändern.)

`HeapTuple` *`zeile`*

Die zu verändernde Eingabezeile.

`int` *`nspalten`*

Anzahl der Spaltennummern im Array `spalten`.

`int *` *`spalten`*

Array mit den Nummern der Spalten, die geändert werden sollen (Spaltennummern zählen von 1 an).

`Datum * werte`

Neue Werte für die angegebenen Spalten.

`char * nulls`

Welche Werte den `NULL`-Wert haben (siehe `SPI_execp` wegen des Formats).

Rückgabewert

Neue Zeile mit den Änderungen, im übergeordneten Executor-Kontext angelegt; nicht `NULL`, wenn `zeile` nicht `NULL` ist und die Änderungen erfolgreich waren, `NULL`, wenn `zeile` `NULL` ist oder ein Fehler aufgetreten ist.

Bei einem Fehler wird `SPI_result` wie folgt gesetzt:

`SPI_ERROR_ARGUMENT`

Wenn `rel` `NULL` ist, oder wenn `zeile` `NULL` ist, oder wenn `nspalten` kleiner als oder gleich 0 ist, oder wenn `spalten` `NULL` ist, oder wenn `werte` `NULL` ist.

`SPI_ERROR_NOATTRIBUTE`

Wenn `spalten` eine ungültige Spaltennummer enthält (kleiner als oder gleich 0 oder größer als die Anzahl der Spalten in `zeile`).

SPI_freetuple

Name

SPI_freetuple – – gibt eine im übergeordneten Executor-Kontext angelegte Zeile frei

Synopsis

```
void SPI_freetuple(HeapTuple zeiger)
```

Beschreibung

`SPI_freetuple` gibt eine zuvor im übergeordneten Executor-Kontext angelegte Zeile frei.

Diese Funktion unterscheidet sich nicht mehr vom normalen `heap_freetuple`. Sie wird nur für die Kompatibilität mit altem Code beibehalten.

Argumente

`HeapTuple zeiger`

Zeiger auf die freizugebende Zeile.

SPI_freetuptable

Name

SPI_freetuptable- gibt eine Zeilenmenge frei, die von SPI_exec oder einer ähnlichen Funktion erzeugt wurde

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Beschreibung

`SPI_freetuptable` gibt eine Zeilenmenge frei, die zuvor von einer SPI-Funktion wie `SPI_exec` bei der Ausführung eines SQL-Befehls erzeugt worden ist. Daher wird diese Funktion normalerweise mit der globalen Variable `SPI_tuptable` als Argument aufgerufen.

Diese Funktion ist nützlich, wenn eine SPI-Prozedur mehrere Befehle ausführen möchte, aber nicht die Ergebnisse aller Befehle bis zum Ende speichern möchte. Alle Zeilenmengen, die nicht ausdrücklich freigegeben wurden, werden automatisch von `SPI_finish` freigegeben.

Argumente

`SPITupleTable * tuptable`

Zeiger auf die freizugebende Zeilenmenge.

SPI_freeplan

Name

SPI_freeplan – gibt einen zuvor gespeicherten Plan frei

Synopsis

```
int SPI_freeplan(void * plan)
```

Beschreibung

`SPI_freeplan` gibt einen Befehlsausführungsplan frei, der von `SPI_prepare` zurückgegeben oder mit `SPI_saveplan` gespeichert wurde.

Argumente

`void * plan`

der freizugebende Plan

Rückgabewert

`SPI_ERROR_ARGUMENT`, wenn `plan NULL` ist.

34.4 Sichtbarkeit von Datenveränderungen

Die folgenden beiden Regeln bestimmen die Sichtbarkeit von Datenveränderungen in Funktionen, die SPI verwenden (oder auch jeder anderen C-Funktion):

- Während der Ausführung eines SQL-Befehls sind alle Datenveränderungen, die von dem Befehl vorgenommen werden (oder von Funktionen, die durch den Befehl aufgerufen werden, einschließlich Triggerfunktionen), für den Befehl unsichtbar. Zum Beispiel sind im Befehl

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  die eingefügten Zeilen für den `SELECT`-Teil unsichtbar.
- Änderungen, die von einem Befehl B getätigt wurden, sind für alle Befehle, die nach B gestartet wurden, sichtbar, egal,w ob er innerhalb von B (während der Ausführung von B) gestartet wurde, oder nachdem B fertig ist.

Der nächste Abschnitt enthält ein Beispiel, das die Anwendung dieser Regeln veranschaulicht.

Beispiel

Dieser Abschnitt enthält ein einfaches Beispiel für die Verwendung von SPI. Die Prozedur `execq` nimmt einen SQL-Befehl als erstes Argument und eine Zeilenzahl als zweites Argument, führt den Befehl mit `SPI_exec` aus und gibt die Anzahl der von dem Befehl verarbeiteten Zeilen zurück. Umfangreichere Beispiele für SPI finden Sie im Quellcodebaum in `src/test/regress/regress.c` und in `contrib/spi`.

```
#include "executor/spi.h"

int execq(text *sql, int zahl);

int
execq(text *sql, int zahl)
{
    char *befehl;
    int status;
    int proc;

    /* Wandle übergebenes text-Objekt in C-Zeichenkette um. */
```

```
    befehl = DatumGetCString(DirectFunctionCall1(textout,
                                                 PointerGetDatum(sql)));

    SPI_connect();

    status = SPI_exec(befehl, zahl);
    proc = SPI_processed;

    /*
     * Wenn es ein SELECT ist und Zeilen zurückgegeben wurden,
     * dann werden die Zeilen mit elog(INFO) ausgegeben.
     */
    if (status == SPI_OK_SELECT && SPI_processed > 0)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                         SPI_getvalue(tuple, tupdesc, i),
                         (i == tupdesc->natts) ? " " : " |");
            elog (INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(befehl);

    return proc;
}
```

(Diese Funktion verwendet die Aufrufskonvention Version 0, damit das Beispiel einfacher zu verstehen ist. In echten Anwendungen sollten Sie aber die neuere Version-1-Schnittstelle verwenden.)

So deklarieren Sie diese Funktion, nachdem Sie sie in eine dymische Bibliothek kompiliert haben:

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Hier ist eine Beispielsitzung:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 167631 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- von execq eingefügt
INFO:  EXECQ:  1    -- von execq zurückgegeben und vom äußeren INSERT eingefügt

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, nur eine Zeile eingefügt - wie angegeben

 execq
-------
     3              -- 10 ist nur der Höchstwert, 3 ist die wirkliche Zeilenzahl
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 167712 1
=> SELECT * FROM a;
 x
---
 1                  -- keine Zeilen in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  0
```

```
INSERT 167713 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- eine Zeile war in a + 1
(2 rows)

-- Dies demonstriert die Sichtbarkeitsregeln bei Datenveränderungen:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 Zeilen * 1 (x in erster Zeile)
 6                  -- 3 Zeilen (2 + 1 gerade eingefügt) * 2 (x in zweiter Zeile)
(4 zeilen)             ^^^^^^^^
                           für execq() bei den Aufrufen sichtbare Zeilen
```

35 Trigger

Dieses Kapitel beschreibt, wie man Triggerfunktionen schreibt. Insbesondere beschreibt es die C-Schnittstelle für Triggerfunktionen. Die Trigger-Schnittstellen in den meisten prozeduralen Sprachen funktionieren analog. (Triggerfunktionen können nicht in SQL geschrieben werden.)

Triggerfunktionen können vor oder nach einem Befehl ausgeführt werden, und die Befehle `INSERT`, `UPDATE` und `DELTE` können Trigger auslösen. Ein Trigger wird für jede modifizierte Zeile ausgeführt. Trigger auf Befehlsebene werden in der aktuellen Version noch nicht unterstützt.

35.1 Triggerdefinition

Wenn ein Triggerereignis auftritt, wird der Triggermanager vom Executor aufgerufen. Dieser richtet eine Informationsstruktur vom Typ `TriggerData` ein (unten beschrieben) und ruft die Triggerfunktion auf, um das Ereignis zu bearbeiten.

Die Triggerfunktion muss erzeugt werden, bevor der Trigger selbst erzeugt werden kann. Die Triggerfunktion muss als Funktion deklariert werden, die keine Argumente und den Ergebnistyp `trigger` hat. (Die Triggerfunktion erhält ihre Argumente durch die `TriggerData`-Struktur, nicht als normale Funktionsargumente.) Wenn die Funktion in C geschrieben ist, muss sie die Aufrufskonvention “Version 1” verwenden.

Die Syntax zum Erzeugen von Triggern wird in Teil VI beschrieben.

Triggerfunktionen geben an den Executor einen Wert vom Typ `HeapTuple` zurück, der eine Tabellenzeile verkörpert. Bei einem Trigger, der nach der eigentlichen Operation ausgeführt wird, wird der Rückgabewert ignoriert. Wenn der Trigger vor der Operation ausgeführt wird, hat er folgende Möglichkeiten:

- Er kann einen *NULL*-Zeiger zurückgeben, um die Operation für die aktuelle Zeile zu überspringen (sodass die Zeile nicht eingefügt/aktualisiert/gelöscht wird).
- Bei `INSERT`- und `UPDATE`-Triggern ist die zurückgegebene Zeile diejenige Zeile, die eingefügt wird bzw. die die zu aktualisierende Zeile ersetzt. Somit kann eine Triggerfunktion die einzufügende oder zu aktualisierende Zeile verändern.

Ein Trigger, der vor der Operation ausgeführt wird und keines dieser beiden Verhalten verursachen möchte, muss darauf achten, dass er dieselbe Zeile zurückgibt, die ihm als neue Zeile übergeben wurde (siehe unten).

Wenn mehrere Trigger für das gleiche Ereignis und dieselbe Relation definiert sind, werden die Trigger ihrem Namen nach in alphabetischer Reihenfolge ausgelöst. Bei "Vorher"-Triggern wird die möglicherweise veränderte Zeile dem nächsten Trigger als Eingabe übergeben. Wenn irgendein Vorher-Trigger einen *NULL*-Zeiger zurückgibt, wird die Operation abgebrochen und die folgenden Trigger nicht mehr ausgeführt.

Wenn eine Triggerfunktion SQL-Befehle ausführt (durch SPI), könnten diese Befehle wiederum Trigger auslösen. Es gibt keine direkte Beschränkung der Anzahl dieser geschachtelten Triggeraufrufe. Es ist in einer solchen Situation auch möglich, dass derselbe Trigger rekursiv aufgerufen wird; zum Beispiel könnte ein `INSERT`-Trigger einen Befehl ausführen, der in dieselbe Tabelle eine weitere Zeile einfügt, wodurch der `INSERT`-Trigger erneut aufgerufen werden würde. Es liegt in der Verantwortung des Triggerprogrammierers, in solchen Szenarien endlose Rekursion zu vermeiden.

Wenn ein Trigger definiert wird, können für ihn Argumente angegeben werden. Der Zweck davon ist, dass auf diese Weise verschiedene Trigger mit ähnlichen Anforderungen dieselbe Funktion aufrufen können. Zum Beispiel könnte man eine flexible Triggerfunktion schreiben, die als Argumente zwei Spaltennamen erhält und den aktuellen Benutzer in einer und die aktuelle Zeit in der anderen ablegt. Wenn die Funktion richtig geschrieben wird, ist sie unabhängig von der Tabelle, für die sie später verwendet wird. Diese Funktion könnte also für `INSERT`-Ereignisse in Tabellen mit passenden Spalten verwendet werden, um die Erstellung von Datensätzen in dieser Tabelle automatisch zu verfolgen. Sie könnte auch für `UPDATE`-Ereignisse definiert werden, um die jeweils letzte Aktualisierung aufzuzeichnen.

35.2 Umgang mit dem Triggermanager

Dieser Abschnitt beschreibt die Einzelheiten der Schnittstelle für Triggerfunktionen. Diese Informationen benötigen Sie nur, wenn Sie eine Triggerfunktion in C schreiben. Wenn Sie eine höhere Sprache verwenden, kümmert sich die Sprache um diese Dinge.

Wenn eine Funktion vom Triggermanager aufgerufen wird, dann werden ihr keine normalen Argumente übergeben, sondern ein "Kontext"-Zeiger, der auf eine `TriggerData`-Struktur zeigt. C-Funktionen können überprüfen, ob sie vom Triggermanager aufgerufen worden sind, indem sie das Makro

```
CALLED_AS_TRIGGER(fcinfo)
```

aufrufen, welches in den folgenden Ausdruck aufgelöst wird:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Wenn das wahr ergibt, kann `fcinfo->context` in den Typ `TriggerData *` umgewandelt werden und die `TriggerData`-Struktur, auf die gezeigt wird, kann verwendet werden.

`struct TriggerData` ist in `commands/trigger.h` definiert:

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
} TriggerData;
```

Die Felder sind folgendermaßen definiert:

`type`

Immer `T_TriggerData`.

`tg_event`

Beschreibt das Ereignis, für das die Funktion aufgerufen wurde. Um `tg_event` zu untersuchen, können Sie folgende Makros verwenden:

`TRIGGER_FIRED_BEFORE(tg_event)`

Ergibt wahr, wenn der Trigger vor der Operation ausgelöst wurde.

`TRIGGER_FIRED_AFTER(tg_event)`

Ergibt wahr, wenn der Trigger nach der Operation ausgelöst wurde.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Ergibt wahr, wenn der Trigger für die Zeile ausgelöst wurde.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Ergibt wahr, wenn der Trigger für den Befehl ausgelöst wurde.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Ergibt wahr, wenn der Trigger durch einen `INSERT`-Befehl ausgelöst wurde.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Ergibt wahr, wenn der Trigger durch einen `UPDATE`-Befehl ausgelöst wurde.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Ergibt wahr, wenn der Trigger durch einen `DELETE`-Befehl ausgelöst wurde.

`tg_relation`

Ein Zeiger auf eine Struktur, die die Relation beschreibt, für die der Trigger ausgelöst wurde. Einzelheiten über diese Struktur finden Sie in `utils/rel.h`. Die interessantesten Felder sind `tg_relation->rd_att` (ein Deskriptor der Tupel in der Relation) und `tg_relation->rd_rel->relname` (der Name der Relation; der Typ ist nicht char* sondern NameData; verwenden Sie `SPI_getrelname(tg_relation),` um ein char*, zu erhalten, wenn Sie ein Kopie des Namens benötigen).

`tg_trigtuple`

Ein Zeiger auf die Zeile, für die der Trigger ausgelöst wurde. Das ist die Zeile, die eingefügt, aktualisiert bzw. gelöscht werden soll. Wenn der Trigger von einem `INSERT` oder `DELETE` ausgelöst worden ist, dann ist das der Wert, den Sie aus der Funktion zurückgeben sollten, wenn Sie die Zeile nicht durch eine andere ersetzen (bei `INSERT`) oder die Operation überspringen wollen.

`tg_newtuple`

Eine Zeiger auf die neue Version der Zeile, wenn der Trigger von einem `UPDATE` ausgelöst wurde, oder `NULL`, wenn es ein `INSERT` oder `DELETE` ist. Das ist der Wert, den Sie aus der Funktion zurückgeben müssen, wenn das Ereignis ein `UPDATE` ist und Sie diese Zeile nicht durch eine andere ersetzen oder die Operation abbrechen wollen.

```
tg_trigger
```

Ein Zeiger auf eine Struktur vom Typ `Trigger`, welcher in `utils/rel.h` definiert ist:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    Oid         tgconstrrelid;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

`tgname` ist der Name des Triggers, `tgnargs` ist die Anzahl der Argumente in `tgargs`, und `tgargs` ist ein Array mit Zeigern auf die Argumente, die im Befehl `CREATE TRIGGER` angegeben wurden. Die anderen Felder sind nur für interne Zwecke.

35.3 Sichtbarkeit von Datenveränderungen

Wenn Sie die SPI-Schnittstelle verwenden, um in Ihren in C geschriebenen Triggerfunktionen SQL-Befehle auszuführen (oder wenn Sie eine andere Sprache verwenden und dort irgendwie SQL-Befehle ausführen, die dann intern auch durch SPI verarbeitet werden), sollten Sie Abschnitt 34.4 lesen, damit Sie wissen, welche Daten zu welchem Zeitpunkt während der Ausführung eines Triggers sichtbar sind. Für Trigger sind die wichtigsten Folgen der Datensichtbarkeitsregeln:

- Die einzufügende Zeile (`tg_trigtuple`) ist für SQL-Befehle, die in einem Vorher-Trigger ausgeführt werden, *nicht sichtbar.*
- Die einzufügende Zeile (`tg_trigtuple`) ist für SQL-Befehle, die in einem Nachher-Trigger ausgeführt werden, *sichtbar* (weil sie gerade eingefügt wurde).
- Eine soeben eingefügte Zeile ist für alle SQL-Befehle sichtbar, die in irgendeinem Trigger ausgeführt werden, der im späteren Verlauf des äußeren Befehls ausgelöst wurde (z.B. für die nächste Zeile).

Der nächste Abschnitt enthält eine Veranschaulichung dieser Regeln.

35.4 Ein vollständiges Beispiel

Hier ist ein sehr einfaches Beispiel für eine in C geschriebene Triggerfunktion. Die Funktion `trigf` gibt die Anzahl der Zeilen in der Tabelle `ttest` aus und lässt die eigentliche Operation weg, wenn der Befehl versucht, in die Spalte `x` einen NULL-Wert einzufügen. (Der Trigger verhält sich also wie ein NOT-NULL-Constraint, bricht aber nicht die Transaktion ab.)

Zuerst die Definition der Tabelle:

```
CREATE TABLE ttest (
    x integer
);
```

Dies ist der Quellcode der Triggerfunktion:

```
#include "postgres.h"
#include "executor/spi.h"       /* das brauchen Sie für SPI */
#include "commands/trigger.h"   /* ... und für Trigger */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* prüfe, ob sie überhaupt als Trigger aufgerufen wurde */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: nicht als Trigger aufgerufen");

    /* Rückgabewert */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* prüfe auf NULL-Werte */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;
```

```
    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "vorher ";
    else
        when = "nachher";

    tupdesc = trigdata->tg_relation->rd_att;

    /* verbinde mit SPI-Manager */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (%s aufgerufen): SPI_connect ergab %d", when, ret);

    /* ermittle Anzahl der Zeilen in der Tabelle */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(NOTICE, "trigf (%s aufgerufen): SPI_exec ergab %d", when, ret);

    /* count(*) gibt int8 zurück, also ordnungsgemäß umwandeln */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (%s aufgerufen): Es sind %d Zeilen in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

Nachdem Sie den Quellcode kompiliert haben, deklarieren Sie die Funktion und die Trigger:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'dateiname'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

```
CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Jetzt können Sie das Verhalten des Triggers ausprobieren:

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (vorher  aufgerufen): Es sind 0 Zeilen in ttest
INSERT 0 0

-- Einfügen wurde ausgelassen und AFTER-Trigger nicht ausgelöst

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (vorher  aufgerufen): Es sind 0 Zeilen in ttest
INFO:  trigf (nachher aufgerufen): Es sind 1 Zeilen in ttest
                                           ^^^^^^^^
                              denken Sie an die Datensichtbarkeitsregeln
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (vorher  aufgerufen): Es sind 1 Zeilen in ttest
INFO:  trigf (nachher aufgerufen): Es sind 2 Zeilen in ttest
                                           ^^^^^^^^
                              denken Sie an die Datensichtbarkeitsregeln
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (vorher  aufgerufen): Es sind 2 Zeilen in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (vorher  aufgerufen): Es sind 2 Zeilen in ttest
INFO:  trigf (nachher aufgerufen): Es sind 2 Zeilen in ttest
```

```
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (vorher  aufgerufen): Es sind 2 Zeilen in ttest
INFO:  trigf (nachher aufgerufen): Es sind 1 Zeilen in ttest
INFO:  trigf (vorher  aufgerufen): Es sind 1 Zeilen in ttest
INFO:  trigf (nachher aufgerufen): Es sind 0 Zeilen in ttest
                                           ^^^^^^^^
                             denken Sie an die Datensichtbarkeitsregeln
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Umfangreichere Beispiele finden Sie in `src/test/regress/regress.c` und in `contrib/spi`.

36

Das Regelsystem

Dieses Kapitel bespricht das Regelsystem von PostgreSQL. Produktionsregelsysteme sind im Prinzip ganz einfach, aber es gibt einige kleine Details, die man bei der Verwendung beachten muss.

Einige andere Datenbanksysteme definieren aktive Datenbankregeln, die meistens gespeicherte Prozeduren und Trigger sind. Diese kann man in PostgreSQL auch mit Funktionen und Triggern implementieren.

Das Regelsystem (genauer gesagt das Anfrageumschreiberegelsystem) unterscheidet sich vollkommen von gespeicherten Prozeduren und Triggern. Es verändert Anfragen anhand der Regeln und übergibt die veränderte Anfrage an den Planer und Executor zur Ausführung. Es ist sehr leistungsfähig und kann für viele Anwendungen, wie SQL-Prozeduren, Sichten und Versionen, verwendet werden. Die theoretischen Grundlagen und die Möglichkeiten dieses Regelsystems werden auch in *Stonebraker, Jhingran, Goh 1990* und in *Ong & Goh 1990* (siehe Literaturverzeichnis) besprochen.

36.1 Der Anfragebaum

Um zu verstehen, wie das Regelsystem funktioniert, muss man wissen, wann es aufgerufen wird und was seine Eingabe- und Ausgabewerte sind.

Das Regelsystem befindet sich zwischen Parser und Planer. Es nimmt die Ausgabe des Parsers, einen Anfragebaum, und die benutzerdefinierten Umschreiberegeln, die ebenfalls Anfragebäume mit einigen zusätzlichen Informationen sind, und erzeugt daraus null oder mehr Anfragebäume als Ergebnis. Die Eingaben und Ausgaben des Regelsystems sind also immer Strukturen, die der Parser selbst hätte erzeugen können, und sind daher im Prinzip alle als SQL-Befehl darstellbar.

Was ist also ein solcher Anfragebaum? Er ist eine interne Darstellung eines SQL-Befehls, in der die einzelnen Teil getrennt gespeichert sind. Diese Anfragebäume können Sie im Serverlog anzeigen lassen, indem Sie die Konfigurationsparameter `debug_print_parse`, `debug_print_rewritten` oder `debug_print_plan` setzen. Die Regelaktionen werden auch als Anfragebaum gespeichert, und zwar im Systemkatalog `pg_rewrite`. Sie sind nicht wie die Logausgabe formatiert, aber sie enthalten genau die gleichen Informationen.

Das Lesen eines Anfragebaums erfordert einige Erfahrung. Aber da die SQL-Darstellung eines Anfragebaums zum Verstehen des Regelsystems ausreicht, wird dieses Kapitel Ihnen nicht zeigen, wie sie zu lesen sind.

Wenn Sie die SQL-Darstellung eines Anfragebaums in diesem Kapitel lesen, ist es nötig, die Teile zu identifizieren, in die ein Befehl in der Anfragebaumstruktur aufgespalten wird. Die Teile des Anfragebaums sind:

- der Befehlstyp

  Dies ist ein einfacher Wert, der anzeigt, welcher Befehl (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) den Anfragebaum erzeugt hat.

- die Range-Tabelle

  Die Range-Tabelle ist eine Liste der Relationen, die in der Anfrage verwendet werden. Bei einem `SELECT`-Befehl sind das die nach dem Schlüsselwort `FROM` angegebenen Relationen.

  Jeder Eintrag in der Range-Tabelle identifiziert eine Tabelle oder eine Sicht und zeigt an, unter welchem Namen sie in anderen Teilen der Anfrage verwendet wird. Im Anfragebaum wird auf die Einträge der Range-Tabelle mit Nummern verwiesen, sodass es hier egal ist, ob es Namen doppelt gibt, im Gegensatz zu SQL-Befehlen. Das kann vorkommen, wenn die Range-Tabelle mit den Range-Tabellen der Regeln verschmolzen wird. In den Beispielen in diesem Kapitel wird das aber nicht vorkommen.

- die Ergebnisrelation

  Das ist ein Index in die Range-Tabelle, der anzeigt, in welcher Relation das Ergebnis der Anfrage gespeichert werden soll.

  `SELECT`-Anfragen haben normalerweise keine Ergebnisrelation. Der Sonderfall `SELECT INTO` ist im Großen und Ganzen identisch mit einem `CREATE TABLE` gefolgt von einem `INSERT ... SELECT` und wird hier nicht gesondert besprochen.

  Bei den Befehlen `INSERT`, `UPDATE` und `DELETE` ist die Ergebnisrelation die Tabelle (oder die Sicht!), auf die der Befehl sich auswirkt.

- die Target-Liste

  Die Target-Liste ist eine Liste von Ausdrücken, die das Ergebnis der Anfrage definieren. Bei einem `SELECT`-Befehl sind dies die Ausdrücke, die das Endergebnis der Anfrage erzeugen. Sie entsprechen den Ausdrücken zwischen den Schlüsselwörtern `SELECT` und `FROM`. (`*` ist nur eine Abkürzung für alle Spalten einer Relation. Sie wird vom Parser in die einzelnen Spalten aufgelöst, sodass das Regelnsystem nichts davon mitbekommt.)

  `DELETE`-Befehle benötigen keine Target-Liste, weil sie kein Ergebnis erzeugen. Der Planer fügt in die leere Target-Liste einen speziellen CTID-Eintrag ein, aber das passiert nach dem Regelnsystem und wird später besprochen; für das Regelsystem ist die Target-Liste leer.

  Bei `INSERT`-Befehlen beschreibt die Target-Liste die neuen Zeilen, die in die Ergebnisrelation eingefügt werden sollen. Sie besteht aus den Ausdrücken in der `VALUES`-Klausel oder aus denen in der `SELECT`-Klausel, wenn es sich um einen `INSERT ... SELECT`-Befehl handelt. Der erste Schritt des Umschreibevorgangs fügt in die Target-Liste Einträge für alle Spalten ein, die im ursprünglichen Befehl nicht angegeben wurden, aber Vorgabewerte haben. Alle übrigen Spalten (ohne angegebenen Wert und ohne Vorgabewert) werden vom Planer mit NULL-Werten aufgefüllt.

  Bei `UPDATE`-Befehlen beschreibt die Target-Liste die neuen Zeilen, die die alten ersetzen sollen. Im Regelsystem enthält es einfach die Ausdrücke aus den `SET spalte = ausdruck` Teilen des Befehls. Der Planer fügt für die nicht angegebenen Spalten Ausdrücke ein, die die Werte aus der alten Zeile in die neue kopieren. Und wie bei `DELETE` wird hier auch ein spezieller CTID-Eintrag eingefügt.

  Jeder Eintrag in der Target-Liste enthält einen Ausdruck, der ein konstanter Wert, eine Variable, die auf eine Relation in der Range-Tabelle verweist, ein Parameter oder ein Ausdrucksbaum aus Funktionsaufrufen, Konstanten, Variablen, Operatoren usw. sein kann.

- die Bedingung

  Die Bedingung einer Anfrage ist ein Ausdruck ähnlich derer in den Einträgen der Target-Liste. Das Ergebnis dieses Ausdrucks ist ein Boole'scher Wert, der anzeigt, ob die Operation (`INSERT`, `UPDATE`, `DELETE` oder `SELECT`) für die endgültige Ergebniszeile ausgeführt werden soll oder nicht. Sie entspricht der `WHERE`-Klausel in einem SQL-Befehl.

- der Verbundbaum

  Der Verbundbaum einer Anfrage zeigt die Struktur der FROM-Klausel. Bei einer einfachen Anfrage wie SELECT ... FROM a, b, c ist der Verbundbaum einfach die Liste der Elemente in der FROM-Klausel, weil diese in jeder beliebigen Reihenfolge verbunden werden können. Aber wenn JOIN-Ausdrücke, insbesondere äußere Verbunde, verwendet werden, müssen die Relationen in der angegebenen Reihenfolge verbunden werden. In diesem Fall zeigt der Verbundbaum die Struktur der JOIN-Ausdrücke. Die Verbundbedingungen der jeweiligen JOIN-Ausdrücke (aus den Klauseln ON oder USING) werden als Bedingungsausdrücke in den Verbundbaum-Knoten gespeichert. Es hat sich als praktisch ergeben, den obersten Ausdruck aus der WHERE-Klausel auch als Bedingung im obersten Verbundbaumelement zu speichern. Also enthält der Verbundbaum in Wirklichkeit sowohl die FROM- als auch die WHERE-Klausel eines SELECT-Befehls.

- die anderen

  Die anderen Teile eines Anfragebaums, wie die ORDER BY-Klausel, sind hier nicht von Interesse. Das Regelsystem wird dort bei der Anwendung der Regeln einige Einträge ersetzen, aber das hat mit den Grundlagen des Regelsystems nicht viel zu tun.

36.2 Sichten und das Regelsystem

Sichten sind in PostgreSQL mit dem Regelsystem implementiert. Es gibt in der Tat zwischen dem Befehl

```
CREATE VIEW meine_sicht AS SELECT * FROM meine_tabelle;
```

und den Befehlen

```
CREATE TABLE meine_sicht (selbe spalten wie meine_tabelle);
CREATE RULE "_RETURN" AS ON SELECT TO meine_sicht DO INSTEAD
    SELECT * FROM meine_tabelle;
```

kaum einen Unterschied, weil es der Befehl CREATE VIEW intern genauso macht. Das hat einige Auswirkungen. Eine davon ist, dass die Informationen über eine Sicht in den PostgreSQL-Systemkatalogen die gleichen wie bei einer Tabelle sind. Für den Parser besteht also absolut kein Unterschied zwischen einer Tabelle und einer Sicht. Sie sind dieselbe Art von Objekt: Relationen.

36.2.1 Wie SELECT-Regeln funktionieren

ON SELECT-Regeln werden auf alle Anfragen als letzten Schritt angewendet, selbst wenn der eigentliche Befehl ein INSERT, UPDATE oder DELETE war. Und sie haben eine andere Semantik als Regeln für andere, Befehlsarten, weil sie den Anfragebaum direkt verändern anstatt einen neuen zu erzeugen. Daher werden die SELECT-Regeln zuerst beschrieben.

Gegenwärtig kann eine ON SELECT-Regel nur eine Aktion haben, und diese muss eine SELECT-Aktion ohne Bedingung und mit INSTEAD sein. Diese Einschränkung ist erforderlich, um Regeln sicher genug zu machen, damit normale Benutzer sie verwenden können, und sie beschränkt ON SELECT-Regeln auf wirkliche Sicht-Regeln.

Die Beispiele in diesem Kapitel sind zwei Verbundsichten, die einige Berechnungen durchführen, und einige weitere Sichten, die darauf aufbauen. Eine der ersten beiden Sichten wird später verfeinert, indem Regeln für INSERT, UPDATE und DELETE hinzugefügt werden, sodass das Endergebnis eine Sicht ist, die sich mit ein bisschen Magie wie eine richtige Tabelle verhält. Dieses Beispiel ist nicht besonders einfach und erschwert vielleicht den Einstieg. Aber es ist besser ein einziges Beispiel zu verwenden, das alle wich-

tigen Themen Schritt für Schritt erklären kann, als viele verschiedene zu verwenden, die man dann leicht durcheinander bringen kann.

Für das Beispiel benötigen wir eine kleine Funktion `min`, die die kleinere aus zwei ganzen Zahlen zurückgibt. Diese können wir so erzeugen:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS '
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
' LANGUAGE SQL STRICT;
```

Die Tabellen, die wir für die ersten zwei Beschreibungen des Regelsystems benötigen, sind diese:

```
CREATE TABLE shoe_data (      -- Schuhdaten
    shoename   text,          -- Primärschlüssel
    sh_avail   integer,       -- vorhandene Paare
    slcolor    text,          -- bevorzugte Schnürsenkelfarbe
    slminlen   real,          -- minimale Schnürsenkellänge
    slmaxlen   real,          -- maximale Schnürsenkellänge
    slunit     text           -- Längeneinheit
);

CREATE TABLE shoelace_data (  -- Schnürsenkeldaten
    sl_name    text,          -- Primärschlüssel
    sl_avail   integer,       -- vorhandene Paare
    sl_color   text,          -- Schnürsenkelfarbe
    sl_len     real,          -- Schnürsenkellänge
    sl_unit    text           -- Längeneinheit
);

CREATE TABLE unit (           -- Einheiten
    un_name    text,          -- Primärschlüssel
    un_fact    real           -- Umrechnungsfaktor in cm
);
```

Wie sie sehen, enthalten sie Daten aus einem Schuhladen.

Die Sichten werden folgendermaßen erzeugt:

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;
```

```
CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

Der Befehl `CREATE VIEW` für die Sicht `shoelace` (welche hier die einfachste ist) erzeugt eine Relation namens `shoelace` und einen Eintrag in `pg_rewrite`, der besagt, dass es eine Umschreiberegel gibt, die immer dann angewendet werden muss, wenn die Relation `shoelace` in der Range-Tabelle einer Anfrage verwendet wird. Die Regel hat keine Regelbedingung (welche später besprochen werden, da `SELECT`-Regeln gegenwärtig keine haben können) und die Regelaktion ist `INSTEAD` (also "anstatt" der ursprünglichen Aktion). Beachten Sie, dass Regelbedingungen nicht dasselbe sind wie Anfragebedingungen. Die Aktion der Regel hat eine Anfragebedingung. Die Aktion unserer Regel ist eine Kopie des Anfragebaums vom `SELECT`-Befehl, der bei der Erzeugung der Sicht angegeben wurde.

Anmerkung

Die zwei zusätzlichen Range-Tabellen-Einträge für *`NEW`* und *`OLD`* (aus historischen Gründen in der Ausgabeform des Anfragebaums als *`*NEW*`* und *`*OLD*`* erscheinend), die Sie im *`pg_rewrite`*-Eintrag sehen können, sind für *`SELECT`*-Regeln ohne Belang.

Jetzt füllen wir `unit`, `shoe_data` und `shoelace_data` mit Daten und führen eine einfache Anfrage mit der Sicht aus:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
```

```
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Das ist das einfachste `SELECT`, das man mit unseren Sichten ausführen kann, und daher verwenden wir es um die Grundlagen der Sichtregeln zu erklären. Der Befehl `SELECT * FROM shoelace` wurde vom Parser interpretiert und ergab folgenden Anfragebaum:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

Dieser wurde dem Regelsystem übergeben. Das Regelsystem durchsucht die Range-Tabelle und prüft, ob es für irgendeine Relation Regeln gibt. Dabei findet es für die Relation `shoelace` (bis jetzt der einzige Eintrag in der Range-Tabelle) die Regel `_RETURN`, die folgenden Anfragebaum hat:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

Um die Sicht aufzulösen, erzeugt der Umschreiber einfach einen Range-Tabellen-Eintrag für eine Unteranfrage, die den Anfragebaum der Regelaktion enthält, und setzt diesen Range-Tabellen-Eintrag für den ein, der ursprünglich auf die Sicht verwiesen hat. Der daraus resultierende Anfragebaum sieht aus, als ob Sie Folgendes direkt eingegeben hätten:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
```

```
        shoelace.sl_unit, shoelace.sl_len_cm
   FROM (SELECT s.sl_name,
                s.sl_avail,
                s.sl_color,
                s.sl_len,
                s.sl_unit,
                s.sl_len * u.un_fact AS sl_len_cm
           FROM shoelace_data s, unit u
          WHERE s.sl_unit = u.un_name) shoelace;
```

Es gibt jedoch einen Unterschied: Die Range-Tabelle der Unteranfrage hat zwei zusätzliche Einträge `shoelace *OLD*` und `shoelace *NEW*`. Diese Einträge nehmen nicht direkt an der Anfrage teil, da sie nicht im Verbundbaum der Unteranfrage oder in der Target-Liste verwendet werden. Der Umschreiber verwendet Sie, um Informationen zur Prüfung der Zugriffsprivilegien zu speichern, die ursprünglich im Range-Tabellen-Eintrag der Sicht enthalten waren. Dadurch kann der Executor trotzdem prüfen, ob der Benutzer die richtigen Privilegien hat, um auf die Sicht zuzugreifen, obwohl die Sicht in der umgeschriebenen Anfrage nicht mehr direkt vorkommt.

Das war die Anwendung der ersten Regel. Das Regelsystem prüft auch noch die verbleibenden Range-Tabellen-Einträge der obersten Anfrage (in diesem Beispiel gibt es keine weiteren) und prüft dann rekursiv die Range-Tabellen-Einträge der hinzugefügten Unteranfrage, um zu sehen, ob irgendeiner davon auf eine Sicht verweist. (Aber `*OLD*` und `*NEW*` werden nicht weiter geprüft, ansonsten wären wir in einer Endlosschleife!) In diesem Beispiel gibt es keine Umschreiberegeln für `shoelace_data` oder `unit` und somit ist die Umschreibephase abgeschlossen und das oben gesehene Endergebnis wird an den Planer weitergereicht.

Jetzt wollen wir eine Anfrage schreiben, die herausfindet, für welche Schuhe im Laden wir passende Schnürsenkel (Farbe und Länge) haben und wo die Gesamtzahl der passenden Paare größer oder gleich zwei ist.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

Die Ausgabe des Parsers ist folgender Anfragebaum:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Die erste angewendete Regel wird die für die Sicht `shoe_ready` sein und ergibt folgenden Anfragebaum:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
```

```
                rsl.sl_name,
                rsl.sl_avail,
                min(rsh.sh_avail, rsl.sl_avail) AS total_avail
           FROM shoe rsh, shoelace rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
  WHERE shoe_ready.total_avail >= 2;
```

Auf dieselbe Art werden die Regeln für `shoe` und `shoelace` in die Range-Tabelle der Unteranfrage eingesetzt, was am Ende folgenden Anfragebaum mit drei Ebenen ergibt:

```
  SELECT shoe_ready.shoename, shoe_ready.sh_avail,
         shoe_ready.sl_name, shoe_ready.sl_avail,
         shoe_ready.total_avail
    FROM (SELECT rsh.shoename,
                 rsh.sh_avail,
                 rsl.sl_name,
                 rsl.sl_avail,
                 min(rsh.sh_avail, rsl.sl_avail) AS total_avail
            FROM (SELECT sh.shoename,
                         sh.sh_avail,
                         sh.slcolor,
                         sh.slminlen,
                         sh.slminlen * un.un_fact AS slminlen_cm,
                         sh.slmaxlen,
                         sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                         sh.slunit
                    FROM shoe_data sh, unit un
                   WHERE sh.slunit = un.un_name) rsh,
                 (SELECT s.sl_name,
                         s.sl_avail,
                         s.sl_color,
                         s.sl_len,
                         s.sl_unit,
                         s.sl_len * u.un_fact AS sl_len_cm
                    FROM shoelace_data s, unit u
                   WHERE s.sl_unit = u.un_name) rsl
           WHERE rsl.sl_color = rsh.slcolor
             AND rsl.sl_len_cm >= rsh.slminlen_cm
             AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
  WHERE shoe_ready.total_avail > 2;
```

Es stellt sich heraus, dass der Planer diesen Baum in einen Anfragebaum mit zwei Ebenen zusammenfasst: Die untersten `SELECT`-Befehle werden in das mittlere `SELECT` "heraufgezogen", da es keinen Grund gibt sie getrennt zu verarbeiten. Aber das mittlere `SELECT` bleibt vom obersten getrennt, weil es Aggregatfunktionen enthält. Wenn wir es heraufziehen würden, dann würde sich das Verhalten des obersten `SELECT`

verändern, was es zu verhindern gilt. Das Zusammenfassen des Anfragebaums ist allerdings eine Optimierung, um die sich das Umschreibesystem nicht kümmern muss.

Anmerkung

Es gibt im Regelsystem gegenwärtig keinen Stoppmechanismus für Rekursion in Sichtregeln (im Gegensatz zu den anderen Regelarten). Das ist nicht weiter schlimm, denn die einzige Möglichkeit, damit eine Endlosschleife zu erzeugen (die den Serverprozess aufblähen würde, bis der Speicher aufgebraucht ist), ist zwei Tabellen zu erzeugen und dann Sichtregeln von Hand mit `CREATE RULE` so einzurichten, dass die eine aus der anderen liest und die andere aus der einen. Das kann niemals passieren, wenn `CREATE VIEW` verwendet wird, weil beim ersten `CREATE VIEW` die zweite Relation noch nicht existiert und daher die erste Sicht nicht aus der zweiten lesen kann.

36.2.2 Sichtregeln in Nicht-SELECT-Befehlen

Zwei Einzelheiten des Anfragebaums wurden in der obigen Beschreibung der Sichtregeln noch nicht angesprochen. Das wären der Befehlstyp und die Ergebnisrelation. Fakt ist, Sichtregeln brauchen diese Informationen nicht.

Es gibt nur wenige Unterschiede zwischen dem Anfragebaum für ein `SELECT` und einem für einen der anderen Befehle. Natürlich haben sie einen anderen Befehlstyp und bei anderen Befehlen außer `SELECT` zeigt die Ergebnisrelation auf den Range-Tabellen-Eintrag, wo das Ergebnis hin soll. Alles andere ist genau gleich. Wenn wir also zwei Tabellen `t1` und `t2` mit den Spalten `a` und `b` hätten, würden die Anfragebäume für die beiden Befehle

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

fast identisch aussehen. Im Einzelnen:

- Die Range-Tabellen enthalten Einträge für die Tabellen `t1` und `t2`.
- Die Target-Listen enthalten eine Variable, die auf die Spalte `b` des Range-Tabellen-Eintrags für die Tabelle `t2` zeigt.
- Die Bedingungsausdrücke vergleichen die Spalten `a` beider Range-Tabellen-Einträge auf Gleichheit.
- Die Verbundbäume zeigen einen einfachen Verbund zwischen `t1` und `t2`.

Die Folge daraus ist, dass beide Anfragebäume ähnliche Ausführungspläne ergeben: Sie sind beide Verbunde aus zwei Tabellen. Bei dem `UPDATE`-Befehl werden die fehlenden Spalten von `t1` vom Planer zur Target-Liste hinzugefügt und der endgültige Anfragebaum sieht dann so aus:

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

Daher ergibt die Ausführung des Verbunds durch den Executor genau die gleiche Ergebnismenge wie bei

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Aber beim `UPDATE` gibt es ein kleines Problem: Den Executor interessiert es nicht, wofür die Ergebnisse des Verbunds, den er gerade ausführt, gedacht sind. Er erzeugt nur eine Menge aus Zeilen. Der Unterschied, dass ein Befehl ein `SELECT` ist und der andere ein `UPDATE`, wird woanders verwaltet. Man weiß (indem man sich den Anfragebaum ansieht), dass der Befehl ein `UPDATE` ist, und man weiß, dass die Ergebnisse in die Tabelle `t1` gelangen sollen. Aber welche der Zeilen dort soll durch die neue Zeile ersetzt werden?

Um dieses Problem zu lösen, wird bei `UPDATE`-Befehlen (und auch bei `DELETE`-Befehlen) in die Target-Liste ein zusätzlicher Eintrag eingefügt: die aktuelle Tupel-ID (*current tuple ID*, CTID). Das ist eine System-

spalte, die von der jeweiligen Zeile die Dateiblocknummer und die Position innerhalb des Blocks enthält. Wenn man die Tabelle kennt, kann man anhand der CTID die ursprüngliche Version der in `t1` zu aktualisierenden Zeile ermitteln. Nachdem die CTID zur Target-Liste hinzugefügt wurde, sieht die Anfrage dann im Prinzip so aus:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Jetzt kommt ein weiteres Detail von PostgreSQL ins Spiel. Alte Tabellenzeilen werden nicht überschrieben, was der Grund ist, warum `ROLLBACK` schnell geht. Bei einem `UPDATE` wird die neue Ergebniszeile in die Tabelle eingefügt (nachdem die CTID entfernt wurde), und im Tupelkopf der alten Zeile, auf die die CTID gezeigt hat, werden die Felder `cmax` und `xmax` auf die aktuelle Befehlsnummer bzw. die aktuelle Transaktionsnummer gesetzt. Dadurch wird die alte Zeile versteckt, und nachdem die Transaktion abgeschlossen wurde, kann sie mit `VACUUM` wirklich entfernt werden.

Nachdem wir das alles wissen, können wir Sichtregeln einfach auf dieselbe Art und Weise bei allen Befehlen anwenden. Es gibt da keinen Unterschied.

36.2.3 Die Leistungsfähigkeit von Sichten in PostgreSQL

Oben wurde gezeigt, wie das Regelsystem die Definition der Sicht in den ursprünglichen Anfragebaum integriert hat. Im zweiten Beispiel erzeugte ein einfacher `SELECT`-Befehl mit einer Sicht letztendlich einen Anfragebaum, der ein Verbund von vier Tabellen darstellte (`unit` wurde zweimal mit verschiedenen Namen verwendet).

Der Vorteil, wenn Sichten mit dem Regelsystem implementiert werden, ist, dass der Planer alle Informationen darüber, welche Tabellen durchsucht werden müssen, welche Verhältnisse zwischen den Tabellen bestehen, was die Bedingung aus der Sicht ist und welche Bedingungen in der eigentlichen Anfrage angegeben wurden, in einem einzigen Anfragebaum hat. Und das gilt auch noch, wenn die ursprüngliche Anfrage schon ein Verbund aus Sichten ist. Der Planer muss entscheiden, wie die Anfrage am besten ausgeführt werden kann, und je mehr Informationen der Planer hat, desto besser kann diese Entscheidung sein. Und so wie das Regelsystem in PostgreSQL implementiert ist, wird sichergestellt, dass alle bis zu diesem Zeitpunkt verfügbaren Informationen im Planer verwendet werden können.

36.2.4 Sichten aktualisieren

Was passiert, wenn eine Sicht als Zielrelation in einem `INSERT`-, `UPDATE`- oder `DELETE`-Befehl verwendet wird? Nachdem die oben beschriebenen Umwandlungen durchgeführt worden sind, haben wir einen Anfragebaum, bei dem die Ergebnisrelation auf einen Range-Tabellen-Eintrag für eine Unteranfrage zeigt. Das wird nicht funktionieren, weswegen der Umschreiber einen Fehler erzeugt, wenn er so eine Konstruktion sieht.

Um das zu ändern, können wir Regeln definieren, die das Verhalten von diesen Befehlstypen verändern. Das ist das Thema des nächsten Abschnitts.

36.3 Regeln für INSERT, UPDATE und DELETE

Regeln, die für `INSERT`, `UPDATE` und `DELETE` definiert sind, unterscheiden sich erheblich von den Sichtregeln, die im vorangegangenen Abschnitt beschrieben wurden. Erstens gibt der Befehl `CREATE RULE` bei ihnen mehr Möglichkeiten:

- Sie können auch überhaupt keine Aktion haben.
- Sie können mehrere Aktionen haben.

- Sie können INSTEAD sein oder nicht.
- Die Pseudorelationen NEW und OLD werden nützlich.
- Sie können Regelbedingungen haben.

Zweitens verändern sie den Anfragebaum nicht direkt, sondern erzeugen an seiner Stelle null oder mehr neue Anfragebäume und können den ursprünglichen verwerfen.

36.3.1 Wie Update-Regeln funktionieren

Merken Sie sich die Syntax

```
CREATE RULE regelname AS ON ereignis
    TO objekt [WHERE regelbedingungen]
    DO [INSTEAD] [aktion | (aktionen) | NOTHING];
```

Im Folgenden nennen wir Regeln, die für die Befehle INSERT, UPDATE oder DELETE definiert sind, zusammengefasst **Update-Regeln**.

Update-Regeln werden vom Regelsystem angewendet, wenn die Ergebnisrelation und der Befehlstyp des Anfragebaums mit dem Objekt und dem Ereignis, das im Befehl CREATE RULE angegeben wurde, übereinstimmt. Bei Update-Regeln erzeugt das Regelsystem eine Liste von Anfragebäumen. Am Anfang ist die Liste leer. Es kann null (bei NOTHING), eine oder mehrere Aktionen geben. Um es einfach zu halten, werden wir uns eine Regel mit einer Aktion ansehen. Diese Regel kann eine Bedingung haben und sie kann INSTEAD sein oder nicht.

Was ist eine Regelbedingung? Das ist eine Einschränkung, die bestimmt, wann die Aktionen einer Regel ausgeführt werden sollen und wann nicht. Diese Bedingung kann nur auf die Pseudorelationen NEW und/oder OLD verweisen, welche im Prinzip die als Objekt angegebene Relation darstellen (aber mit einer besonderen Bedeutung).

Wir haben also vier Fälle, die bei einer Regel mit einer Aktion folgende Anfragebäume erzeugen.

- Keine Bedingung und nicht INSTEAD

  Der Anfragebaum der Regelaktion, zusätzlich mit der Bedingung aus dem ursprünglichen Anfragebaum.

- Keine Bedingung, aber INSTEAD

  Der Anfragebaum der Regelaktion, zusätzlich mit der Bedingung aus dem ursprünglichen Anfragebaum.

- Mit Bedingung und nicht INSTEAD

  Der Anfragebaum der Regelaktion, zusätzlich mit der Regelbedingung und der Bedingung aus dem ursprünglichen Anfragebaum.

- Mit Bedingung und INSTEAD

  Der Anfragebaum der Regelaktion, zusätzlich mit der Regelbedingung und der Bedingung aus dem ursprünglichen Anfragebaum; und der ursprüngliche Anfragebaum mit der negierten Regelbedingung.

Schließlich wird, wenn die Regel nicht INSTEAD ist, der unveränderte Originalanfragebaum zur Liste hinzugefügt. Da nur INSTEAD-Regeln mit Regelbedingung den ursprünglichen Anfragebaum schon hinzugefügt haben, erhalten wir entweder einen oder zwei Anfragebäume als Ergebnis bei einer Regel mit einer Aktion.

Bei ON INSERT-Regeln wird die ursprüngliche Anfrage vor den Regelaktionen ausgeführt (wenn sie nicht durch INSTEAD unterdrückt wurde). Dadurch können die Aktionen die eingefügten Zeilen sehen. Aber bei ON UPDATE- und ON DELETE-Regeln wird die ursprüngliche Anfrage nach den Aktionen der Regeln ausgeführt. Dadurch können die Aktionen die zu aktualisierenden oder zu löschenden Zeilen sehen;

ansonsten machen die Aktionen vielleicht gar nichts, weil sie keine Zeilen finden, die mit ihren Bedingungen übereinstimmen.

Die Anfragebäume, die sich aus den Regelaktionen ergeben, werden wiederum durch das Umschreibesystem geschickt, und es werden möglicherweise weitere Regeln angewendet, wodurch sich mehr oder weniger Anfragebäume ergeben. Die Anfragebäume der Regelaktionen müssen daher entweder einen anderen Befehlstyp oder eine andere Ergebnisrelation haben, ansonsten wird dieser rekursive Vorgang in einer Schleife enden. Es gibt eine feste Rekursionsgrenze von gegenwärtig 100 Iterationen. Wenn es nach 100 Iterationen immer noch Update-Regeln gibt, die angewendet werden können, geht das Regelsystem von einer Schleife über mehrere Regeldefinitionen aus und erzeugt einen Fehler.

In Anfragebäumen, die im Systemkatalog `pg_rewrite` gespeichert werden, sind nur Vorlagen. Da sie auf Range-Tabellen-Einträge für `NEW` und `OLD` verweisen können, müssen einige Ersetzungen getätigt werden, bevor sie verwendet werden können. Für jeden Verweis auf `NEW` wird die Target-Liste der ursprünglichen Anfrage nach einem entsprechenden Eintrag durchsucht. Wenn einer gefunden wird, ersetzt der Ausdruck dieses Eintrags den Verweis. Ansonsten bedeutet `NEW` dasselbe wie `OLD` (bei `UPDATE`) oder wird durch den NULL-Wert ersetzt (bei `INSERT`). Verweise auf `OLD` werden durch einen Verweis auf den Range-Tabellen-Eintrag ersetzt, der die Ergebnisrelation darstellt.

Nachdem das System die Update-Regeln angewendet hat, wendet es auf die erzeugten Anfragebäume eventuelle Sicht-Regeln an. Sichten können keine weiteren Update-Aktionen einfügen, also müssen nach der Sichtumschreibung keine Update-Regeln mehr angewendet werden.

Eine erste Regel Schritt für Schritt

Nehmen wir an, wir wollen Änderungen in der Spalte `sl_avail` der Relation `shoelace_data` verfolgen. Dazu richten wir eine Logtabelle ein und erzeugen eine Regel, die bei jedem `UPDATE` von `shoelace_data` einen Logeintrag schreibt.

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- geänderter Schnürsenkel
    sl_avail   integer,       -- neuer Lagerbestand
    log_who    text,          -- wer hat die Änderungen vorgenommen
    log_when   timestamp      -- wann
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Jetzt führt jemand Folgendes aus:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

und wir sehen uns die Logtabelle an:

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
```

```
---------+----------+---------+---------------------------------
 sl7     |        6 |      Al | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Das haben wir so erwartet. Im Hintergrund ist Folgendes passiert. Der Parser hat diesen Anfragebaum erzeugt:

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

Es gibt eine Regel `log_shoelace`, die `ON UPDATE` ist und den Regelbedingungsausdruck

```
NEW.sl_avail <> OLD.sl_avail
```

und die Aktion

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

hat. (Das sieht etwas seltsam aus, weil man normalerweise nicht `INSERT ... VALUES ... FROM` schreiben kann. Die `FROM`-Klausel gibt hier nur an, dass es in der Range-Tabelle Einträge für `*NEW*` und `*OLD*` gibt. Diese werden benötigt, damit Variablen im Anfragebaum des `INSERT`-Befehls darauf verweisen können.)

Die Regel ist eine Regel mit Bedingung und ohne `INSTEAD`, sodass das Regelsystem zwei Anfragebäume zurückgeben muss: die modifizierte Regelaktion und der ursprüngliche Anfragebaum. In Schritt 1 wird die Range-Tabelle der ursprünglichen Anfrage in den Anfragebaum der Regelaktion aufgenommen. Das ist das Ergebnis:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data;
```

In Schritt 2 wird die Regelbedingung hinzugefügt, damit die Ergebnismenge auf Zeilen beschränkt wird, wo sich `sl_avail` ändert:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail;
```

(Das sieht noch seltsamer aus, weil `INSERT ... VALUES` auch keine `WHERE`-Klausel hat, aber der Planer und Executor haben damit keine Probleme. Sie müssen diese Funktionalität für `INSERT ... SELECT` sowieso unterstützen.)

In Schritt 3 wird die Bedingung aus dem ursprünglichen Anfragebaum hinzugefügt, wodurch die Ergebnismenge weiter beschränkt wird, und zwar auf die Zeilen, die auch von der ursprünglichen Anfrage gefunden worden wären:

```
INSERT INTO shoelace_log VALUES (
       *NEW*.sl_name, *NEW*.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE *NEW*.sl_avail <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Schritt 4 ersetzt die Verweise auf `NEW` durch Target-Listen-Einträge aus dem ursprünglichen Anfragebaum oder durch passende Variablenverweise aus der Ergebnisrelation:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> *OLD*.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Schritt 5 ändert Verweise auf `OLD` in Verweise auf die Ergebnisrelation:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Das war's. Da die Regel nicht `INSTEAD` ist, geben wir dazu den ursprünglichen Anfragebaum aus. Zusammengefasst ist die Ausgabe des Regelsystems eine Liste von zwei Anfragebäumen, die diesen Befehlen entsprechen:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

Diese werden in dieser Reihenfolge ausgeführt und es ist genau das, was die Regel machen sollte.

Die Ersetzungen und die hinzugefügten Bedingungen sorgen dafür, dass, wenn die ursprüngliche Anfrage zum Beispiel

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

wäre, dass dann kein Logeintrag geschrieben würde. In diesem Fall enthält der ursprüngliche Anfragebaum keinen Target-Listen-Eintrag für `sl_avail` und daher wird `NEW.sl_avail` durch `shoelace_data.sl_avail` ersetzt. Dadurch wird der von der Regel erzeugte zusätzliche Befehl so aussehen:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Und diese Bedingung kann niemals wahr sein.

Es funktioniert auch, wenn die ursprüngliche Anfrage mehrere Zeilen verändert. Wenn zum Beispiel jemand den Befehl

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

ausführt, werden im konkreten Fall vier Zeilen aktualisiert (`sl1`, `sl2`, `sl3` und `sl4`). Aber `sl3` hat schon `sl_avail = 0`. In diesem Fall ist die Bedingung im ursprünglichen Anfragebaum eine andere und das ergibt als von der Regel erzeugten zusätzlichen Anfragebaum

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

Dieser Anfragebaum wird ganz sicher drei Logeinträge erzeugen. Und das ist vollkommen richtig.

Hier können wir sehen, warum es wichtig ist, dass der ursprüngliche Anfragebaum zuletzt ausgeführt wird. Wenn das `UPDATE` zuerst ausgeführt worden wäre, wären alle Zeilen schon auf null gesetzt worden und das `INSERT` für den Logeintrag würde keine Zeilen finden, bei denen `0 <> shoelace_data.sl_avail`.

36.3.2 Zusammenarbeit mit Sichten

Eine einfache Möglichkeit, Sichtrelationen vor der erwähnten Möglichkeit zu schützen, dass jemand `INSERT`, `UPDATE` oder `DELETE` mit ihnen ausführen könnte, ist, dafür zu sorgen, dass diese Anfragebäume weggeworfen werden. Dazu erzeugen wir folgende Regeln:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

Wenn jetzt jemand eine dieser Operationen mit der Sichtrelation `shoe` versucht, wird das Regelsystem diese Regeln anwenden. Da diese Regeln keine Aktionen haben und `INSTEAD` sind, wird die Ergebnisliste der Anfragebäume leer sein und die ganze Anfrage verschwindet, weil es nichts mehr zu optimieren oder auszuführen gibt, nachdem das Regelsystem fertig ist.

Eine etwas bessere Verwendung das Regelsystems wäre es, Regeln zu erzeugen, die den Anfragebaum so umschreiben, dass er die richtigen Operationen an den eigentlichen Tabellen vornimmt. Um das mit der Sicht `shoelace` zu tun, erzeugen wir folgende Regeln:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Jetzt nehmen wir an, dass ab und zu eine große Packung Schnürsenkel im Laden ankommt, mit einer langen Lieferliste. Aber Sie wollen die Sicht `shoelace` nicht jedes Mal von Hand aktualisieren. Stattdessen richten wir zwei kleine Tabellen ein: eine, wo man die Posten von der Lieferliste einfügen kann, und eine mit einem besonderen Trick. Die Befehle zur Erzeugung sind:

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);
```

```
CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Jetzt können Sie die Tabelle `shoelace_arrive` mit den Daten aus der Lieferliste füllen:

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Schauen Sie noch einmal auf die aktuellen Daten:

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Jetzt fügen Sie die neu angekommenen Schnürsenkel ein:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

Und so sieht das Ergebnis aus:

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
```

```
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+---------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

Von dem einen `INSERT ... SELECT` bis zu diesen Ergebnissen ist es ein langer Weg. Und die Beschreibung der Anfragebaumtransformation wird die letzte in diesem Kapitel sein. Zuerst kommt die Ausgabe des Parsers:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Jetzt wird die erste Regel `shoelace_ok_ins` angewendet und macht daraus

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

und wirft den ursprünglichen `INSERT`-Befehl für `shoelace_ok` weg. Diese umgeschriebene Anfrage wird wiederum dem Regelsystem zugeführt und die zweite angewendete Regel `shoelace_upd` erzeugt

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Wieder ist es eine INSTEAD-Regel und der vorherige Anfragebaum wird verworfen. Beachten Sie, dass diese Anfrage immer noch die Sicht shoelace verwendet. Aber das Regelsystem ist bei diesem Schritt noch nicht fertig, also macht es weiter und wendet die Regel _RETURN an, wodurch wir Folgendes erhalten:

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Schließlich wird die Regel log_shoelace angewendet und erzeugt den zusätzlichen Anfragebaum

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
       shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data shoelace_data,
       shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u,
       shoelace_data *OLD*, shoelace_data *NEW*
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

Danach hat das Regelsystem keine Regeln mehr und gibt die erzeugten Anfragebäume zurück.

Am Ende haben wir also zwei Anfragebäume, die folgenden SQL-Befehlen entsprechen:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
```

```
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

Als Ergebnis wird das Einfügen von Daten aus einer Relation in eine andere, was in eine Aktualisierung einer dritten umgewandelt wird, was wiederum in eine Aktualisierung einer vierten umgewandelt wird, plus das Loggen dieser letzten Aktualisierung in eine fünfte auf zwei Anfragen reduziert.

Es gibt hier ein etwas unglückliches Detail. Wenn Sie sich die zwei Anfragen ansehen, werden Sie bemerken, dass die Relation `shoelace_data` zweimal in der Range-Tabelle auftaucht, obwohl einmal auf jeden Fall reichen würde. Der Planer kann das nicht erkennen und daher wird der Ausführungsplan des `INSERT` so aussehen:

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

Ohne diesen zusätzlichen Range-Tabellen-Eintrag könnte er so aussehen:

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

Beides erzeugt genau die gleichen Einträge in der Logtabelle. Das Regelsystem verursacht also eine zusätzliche Suche durch die Tabelle `shoelace_data`, die vollkommen überflüssig ist. Und im `UPDATE` wird diese überflüssige Suche noch einmal durchgeführt. Aber es war schon schwer genug, das überhaupt möglich zu machen.

Jetzt folgt eine letzte Demonstration der Leistungsfähigkeit des PostgreSQL-Regelnsystems. Nehmen wir an, dass Sie einige Schnürsenkel mit außergewöhnlichen Farben in die Datenbank eingefügt haben:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Wir möchten eine Sicht erzeugen, die prüft, welche `shoelace`-Einträge farblich zu keinem Schuh passen. Die Sicht dafür ist:

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Die Ausgabe davon ist:

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Jetzt wollen wir es so einrichten, dass unpassende Schnürsenkel, die nicht auf Lager sind, aus der Datenbank gelöscht werden. Um es für PostgreSQL etwas schwerer zu machen, wollen wir sie nicht direkt löschen, sondern erzeugen noch eine Sicht

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

und machen es so:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

Ein `DELETE` mit einer Sicht, mit einer Unteranfrage in der Bedingung, die insgesamt vier geschachtelte bzw. verbundene Sichten verwendet, von denen eine selbst eine Unteranfrage mit Sicht in der Bedingung hat, und wo berechnete Sichtspalten verwendet werden, wird in einen einzelnen Anfragebaum umgeschrieben, der die gewünschten Daten aus einer echten Tabelle löscht.

In der richtigen Welt gibt es wahrscheinlich nur wenige Situationen, in denen eine solche Konstruktion wirklich notwendig ist. Aber es ist beruhigend zu wissen, dass es funktioniert.

36.4 Regeln und Privilegien

Wegen des Umschreibens durch Regeln können von einer Anfrage auch Tabellen oder Sichten verwendet werden, die nicht in der ursprünglichen Anfrage angegeben wurden. Wenn Update-Regeln verwendet werden, kann das auch Schreibzugriffe einschließen.

Umschreiberegeln haben keinen Eigentümer. Der Eigentümer der Relation (Tabelle oder Sicht) ist automatisch der Eigentümer der dafür definierten Umschreiberegeln. Das Regelsystem verändert das normale Verhalten der Privilegienprüfung. Relationen, die aufgrund von Regeln verwendet werden, werden anhand der Privilegien des Eigentümers der Regeln überprüft, nicht mit den Privilegien des Benutzers, der die Regel ausgelöst hat. Das bedeutet, dass ein Benutzer die erforderlichen Privilegien nur für Tabellen/Sichten benötigt, die er ausdrücklich in seinen Anfragen angibt.

Ein Beispiel: Ein Benutzer hat eine Liste mit Telefonnummern, von denen einige privat sind und einige auch für den Assistenten im Büro zugänglich sein sollen. Er kann Folgendes zusammenstellen:

```
CREATE TABLE telefondaten (person text, telefon text, privat boolean);
CREATE VIEW telefonnummer AS
    SELECT person, telefon FROM telefondaten WHERE NOT privat;
GRANT SELECT ON telefonnummer TO assistent;
```

Niemand außer ihm (und den Datenbank-Superusern) kann auf die Tabelle `telefondaten` zugreifen. Aber wegen des `GRANT`-Befehls kann der Assistent einen `SELECT`-Befehl in der Sicht `telefonnummer` ausführen. Das Regelsystem wird den `SELECT`-Befehl in `telefonnummer` in einen `SELECT`-Befehl in `telefondaten` umschreiben und fügt die Bedingung hinzu, dass nur Einträge, bei denen `privat` falsch ist, ausgewählt werden sollen. Da der Benutzer der Eigentümer von `telefondaten` und daher der Eigentümer der Regel ist, werden die Leseprivilegien mit ihm geprüft und der Zugriff wird gewährt. Die Zugriffsprüfung für `telefonnummer` wird auch durchgeführt, aber das wird mit den Privilegien des aufrufenden Benutzers gemacht, und somit kann nur der Benutzer und der Assistent darauf zugreifen.

Die Privilegien werden Regel für Regel geprüft. Bis jetzt ist der Assistent der Einzige, der die öffentlichen Telefonnummern sehen kann. Aber der Assistent kann eine weitere Sicht einrichten und allen darauf Zugriff gewähren. Dann kann jeder Daten aus `telefonnummer` durch die Sicht des Assistenten sehen. Was der Assistent aber nicht machen kann, ist, eine Sicht zu erzeugen, die direkt auf `telefondaten` zugreift. (Eigentlich doch, aber sie wird nicht funktionieren, weil bei jeder Rechteprüfung der Zugriff verweigert würde.) Und sobald der Benutzer mitbekommt, dass der Assistent seine Sicht `telefonnummer` geöffnet hat, kann er seinen Zugriff entziehen. Jeder Zugriff auf die Sicht des Assistenten würde dann sofort scheitern.

Man mag vielleicht denken, dass diese Regel-für-Regel-Prüfung ein Sicherheitsloch darstellt. Aber wenn es nicht so funktionieren würde, könnte der Assistent eine Tabelle mit den gleichen Spalten wir `telefonnummer` einrichten und die Daten einmal am Tag dorthin kopieren. Dann sind es seine eigenen Daten und er kann jedem Benutzer nach Belieben darauf Zugriff gewähren. Ein `GRANT`-Befehl bedeutet: “Ich vertraue dir”. Wenn jemand, dem Sie vertrauen, Dinge wie oben macht, ist es an der Zeit, nochmal darüber nachzudenken und `REVOKE` anzuwenden.

Dieser Mechanismus funktioniert auch bei Update-Regeln. In den Beispielen im vorigen Abschnitt könnte der Eigentümer der Beispieltabellen jemandem für die Sicht `shoelace` die Privilegien `SELECT`, `INSERT`, `UPDATE` und `DELETE` gewähren, aber nur `SELECT` für `shoelace_log`. Die Regelaktion, die die Logeinträge schreibt, würde trotzdem erfolgreich ausgeführt und dieser andere Benutzer kann die Logeinträge auch sehen. Aber er kann keine gefälschten Einträge einfügen oder bestehende verändern oder löschen.

36.5 Regeln und der Befehlsstatus

Der PostgreSQL-Server gibt für jeden Befehl eine Statuszeichenkette wie `INSERT 149592 1` zurück. Das ist ganz einfach, wenn keine Umschreiberegeln im Spiel sind, aber was passiert, wenn die Anfrage von Regeln umgeschrieben wurde?

Regeln beeinflussen den Befehlsstatus wie folgt:

- Wenn es für die Anfrage keine Regel ohne Bedingung, aber mit `INSTEAD` gibt, wird die ursprüngliche Anfrage ausgeführt und ihr Befehlsstatus ganz normal zurückgegeben. (Beachten Sie aber, dass, wenn es `INSTEAD`-Regeln mit Bedingung gab, dass dann die Negierung ihrer Bedingungen zur ursprünglichen Anfrage hinzugefügt worden sind. Das kann die Anzahl der verarbeiteten Zeilen verringern und so den zurückgegebenen Status beeinflussen.)
- Wenn es für die Anfrage eine Regel ohne Bedingung, aber mit `INSTEAD` gibt, wird die ursprüngliche Anfrage gar nicht ausgeführt. In diesem Fall gibt der Server den Befehlsstatus der letzten Anfrage zurück, die von einer `INSTEAD`-Regel (mit oder ohne Bedingung) eingefügt wurde und denselben Befehlstyp (`INSERT`, `UPDATE` oder `DELETE`) wie die ursprüngliche Anfrage hat. Wenn keine Anfrage diese Voraussetzungen erfüllt, zeigt der zurückgegebene Befehlsstatus den Typ der ursprünglichen Anfrage und null für die Zeilenzahl und die OID.

(Dieses System wurde in PostgreSQL 7.3 geschaffen. In den Versionen davor könnte der Befehlsstatus andere Ergebnisse liefern, wenn Regeln vorhanden sind.)

Der Programmierer kann dafür sorgen, dass eine bestimmte `INSTEAD`-Regel diejenige ist, die im zweiten Fall den Befehlsstatus setzt, indem ihr der alphabetisch letzte Name unter den aktiven Regeln gegeben wird, damit sie zuletzt angewendet wird.

36.6 Regeln und Trigger

Viele Sachen, die man mit Triggern machen kann, können auch mit dem PostgreSQL-Regelsystem implementiert werden. Eine Sache, die nicht durch Regeln implementiert werden kann, sind bestimmte Arten von Constraints, insbesondere Fremdschlüssel. Es ist möglich, eine Regel mit Bedingung zu erzeugen, die einen Befehl in `NOTHING`, also nichts, umwandelt, wenn ein Wert aus einer Spalte in einer anderen Tabelle nicht vorhanden ist. Aber dann werden die Daten heimlich weggeworfen und das wäre keine besonders gute Idee. Wenn die Gültigkeit von Werten geprüft werden soll und im Fall eines ungültigen Wertes eine Fehlermeldung erzeugt werden soll, dann muss das mit einem Trigger gemacht werden.

Andererseits kann ein Trigger, der von `INSERT` in eine Sicht ausgelöst wird, dasselbe wie eine Regel erledigen: die Daten woanders ablegen und das Einfügen in die Sicht unterbinden. Aber er kann das Gleiche nicht bei `UPDATE` oder `DELETE` machen, weil es in der Sichtrelation keine echten Daten gibt, die durchsucht werden könnten, und somit würde der Trigger niemals aufgerufen werden. Nur eine Regel kann da helfen.

Für Sachen, die durch beide implementiert werden können, hängt die bessere Wahl von der Verwendung der Datenbank ab. Ein Trigger wird einmal für jede betroffene Zeile ausgelöst. Eine Regel verändert den Anfragebaum oder erzeugt einen zusätzlichen. Wenn ein Befehl also auf viele Zeilen Auswirkung hat, dann ist eine Regel, die einen zusätzlichen Befehl ausführt, meistens besser als ein Trigger, der für jede einzelne Zeile aufgerufen wird und seine Operationen viele Male ausführen muss.

Hier zeigen wie ein Beispiel dafür, wie sich die Wahl zwischen Regeln und Triggern in einer bestimmten Situation auswirkt. Es gibt zwei Tabellen:

```
CREATE TABLE computer (
    hostname        text,    -- mit Index
```

```
    hersteller      text     -- mit Index
);

CREATE TABLE software (
    software        text,    -- mit Index
    hostname        text     -- mit Index
);
```

Beide Tabellen haben viele tausend Zeilen und der Indexe für `hostname` sind Unique Indexe. Die Regel oder der Trigger sollen einen Constraint implementieren, der Zeilen aus `software` löscht, die auf einen gelöschten Computer verweisen. Der Trigger würde folgenden Befehl verwenden:

```
DELETE FROM software WHERE hostname = $1;
```

Da der Trigger für jede Zeile einzeln aufgerufen wird, kann er den Plan für den Befehl vorbereiten und speichern und den Wert für `hostname` in dem Parameter übergeben. Die Regel würde so aussehen:

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Jetzt schauen wir uns verschiedene Arten von Löschvorgängen an. Im Fall eines

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

wird die Tabelle `computer` über einen Index durchsucht (schnell) und der vom Trigger ausgeführte Befehl würde auch einen Indexscan verwenden (auch schnell). Der zusätzliche Befehl von der Regel wäre

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Da entsprechende Indexe eingerichtet wurden, wird der Planer folgenden Plan erzeugen:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Also würde es hier keinen großen Geschwindigkeitsunterschied zwischen den Implementierungen mit Triggern und mit Regeln geben.

Mit dem nächsten Löschvorgang wollen wir alle 2000 Computer löschen, bei denen `hostname` mit `alt` anfängt. Dafür gibt es zwei mögliche Befehle. Einer ist

```
DELETE FROM computer WHERE hostname >= 'alt'
                       AND hostname < 'alu'
```

Der Befehl, den die Regel hinzufügt, ist:

```
DELETE FROM software WHERE computer.hostname >= 'alt'
AND computer.hostname < 'alu' AND software.hostname = computer.hostname;
```

Der Plan für diesen Befehl wird folgender sein:

```
Hash Join
```

```
    ->  Seq Scan on software
    ->  Hash
      ->  Index Scan using comp_hostidx on computer
```

Der andere mögliche Befehl ist

```
DELETE FROM computer WHERE hostname ~ '^old';
```

mit diesem Ausführungsplan für den von der Regel hinzugefügten Befehl:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Das zeigt, dass der Planer nicht erkennt, dass die Bedingung für `hostname` in `computer` auch als Indexscan für `software` verwendet werden könnte, wenn mehrere Bedingungsausdrücke mit `AND` verknüpft sind, was er bei der Version mit dem regulären Ausdruck tut. Der Trigger wird für jeden der 2000 zu löschenden alten Computer einmal aufgerufen werden und das ergibt einen Indexscan durch `computer` und 2000 Indexscans durch `software`. Die Regel-Implementierung schafft es mit zwei Befehlen, die Indexe zu verwenden. Und ob die Regel mit der sequenziellen Suche immer noch schneller sein wird, hängt von der Gesamtgröße der Tabelle `software` ab. 2000 Befehle über den SPI-Manager auszuführen, dauern auch eine Weile, selbst wenn alle Indexblöcke sehr bald im Cache sein werden.

Den letzten Befehl, den wir uns anschauen, ist

```
DELETE FROM computer WHERE hersteller = 'bim';
```

Damit könnten wieder viele Zeilen aus `computer` gelöscht werden. Also wird der Trigger wieder viele Befehle über den Executor ausführen. Der von der Regel erzeugte Befehl ist

```
DELETE FROM software WHERE computer.hersteller = 'bim'
                       AND software.hostname = computer.hostname;
```

Der Plan für diesen Befehl wird wieder ein Nested-Loop-Verbund mit zwei Indexscans sein, nur mit einem anderen Index für `computer`:

```
Nestloop
  ->  Index Scan using comp_herstidx on computer
  ->  Index Scan using soft_hostidx on software
```

In allen diesen Fällen sind die zusätzlichen Befehle aus dem Regelsystem mehr oder weniger unabhängig von der Anzahl der von dem Befehl betroffenen Zeilen.

Die Zusammenfassung ist: Regeln sind nur dann erheblich langsamer als Trigger, wenn sich aus ihren Aktionen schlecht qualifizierte Verbunde ergeben, weil der Planer das nicht gut verarbeiten kann.

37

Prozedurale Sprachen

In PostgreSQL können Benutzer dem System neue Programmiersprachen zum Schreiben von Funktionen und Prozeduren hinzufügen. Diese Sprachen werden *prozedurale Sprachen* genannt. Wenn eine Funktion oder Triggerprozedur in einer prozeduralen Sprache geschrieben worden ist, weiß der Datenbankserver selbst nicht, wie der Quellcode der Funktion zu interpretieren ist. Stattdessen wird diese Aufgabe von einer besonderen Handler-Funktion übernommen, die die Details der Sprache kennt. Der Handler könnte entweder die ganze Arbeit (Parsen, Syntaxanalyse, Ausführung usw.) selbst erledigen oder er könnte als Zwischenstück zwischen PostgreSQL und einer bestehenden Implementierung einer Programmiersprache fungieren. Der Handler selbst ist eine besondere Funktion, die in C geschrieben, in eine dynamische Bibliothek kompiliert und bei Bedarf geladen wird.

Wie man einen Handler für eine neue prozedurale Sprache schreibt, wird in Abschnitt 33.9 beschrieben. In der Standarddistribution von PostgreSQL sind mehrere prozedurale Sprachen enthalten, welche als Beispiele dienen können.

37.1 Installation von prozeduralen Sprachen

Eine prozedurale Sprache muss in jeder Datenbank, in der sie verwendet werden soll, "installiert" werden. Prozedurale Sprachen, die in der Datenbank `template1` installiert werden, stehen jedoch in jeder danach erzeugten Datenbank automatisch zur Verfügung. Der Datenbankadministrator kann also entscheiden, welche Sprachen in welchen Datenbanken zur Verfügung stehen sollen, und kann einige Sprachen zur Standardaustattung jeder Datenbank hinzufügen.

Für die Sprachen, die in der Standarddistribution enthalten sind, kann das Programm `createlang` verwendet werden um die Sprache zu installieren anstatt die Einzelheiten von Hand durchzuführen. Um zum Beispiel die Sprache `PL/pgSQL` in der Datenbank `template1` zu installieren, verwenden Sie

```
createlang plpgsql template1
```

Das unten beschriebene manuelle Verfahren ist nur für Sprachen zu empfehlen, die `createlang` nicht kennt.

Manuelle Installation einer prozeduralen Sprache

Eine prozedurale Sprache wird mit drei Schritten in einer Datenbank installiert. Diese Schritte müssen von einem Datenbank-Superuser durchgeführt werden. Das Programm `createlang` automatisiert Schritt 2. und Schritt 3..

1. Das dynamische Objekt mit dem Handler der Sprache muss kompiliert und installiert werden. Das funktioniert genauso wie das Kompilieren und Installieren von Modulen mit normalen benutzerdefinierten C-Funktionen; siehe Kapitel 33.7.6.
2. Der Handler muss mit folgendem Befehl deklariert werden:

   ```
   CREATE FUNCTION handler_funktionsname()
       RETURNS language_handler
       AS 'pfad-zum-dynamischen-objekt'
       LANGUAGE C;
   ```

   Der besondere Rückgabetyp language_handler sagt dem Datenbanksystem, dass diese Funktionen keinen der definierten SQL-Datentypen zurückgibt und nicht direkt in SQL-Befehlen verwendet werden kann.
3. Die prozedurale Sprache muss mit folgendem Befehl deklariert werden:

   ```
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE sprachname
       HANDLER handler_funktionsname;
   ```

   Das optionale Schlüsselwort `TRUSTED` (etwa: vertrauenswürdig) gibt an, dass normale Datenbankbenutzer ohne Superuser-Privilegien diese Sprache zur Erzeugung von Funktionen und Triggerprozeduren verwenden dürfen. Da Funktionen innerhalb des Datenbankservers ausgeführt werden, sollte `TRUSTED` nur bei Sprachen verwendet werden, die keinen Zugriff auf die internen Strukturen des Datenbankservers oder auf das Dateisystem erlauben. Die Sprachen `PL/pgSQL`, `PL/Tcl`, `PL/Perl` und `PL/Python` können als vertrauenswürdig eingestuft werden; die Sprachen `PL/TclU` und `PL/PerlU` sind dafür gedacht, dass sie unbegrenzte Funktionalität anbieten, und sollten *nicht* als vertrauenswürdig eingestuft werden.

Beispiel 37.1 zeigt, wie der manuelle Installationsvorgang bei der Sprache `PL/pgSQL` funktioniert.

Beispiel 37.1: Manuelle Installation von PL/pgSQL

Der folgende Befehl sagt dem Datenbankserver, wo das dynamische Objekt für die Handler-Funktion der Sprache `PL/pgSQL` zu finden ist.

```
CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
    '$libdir/plpgsql' LANGUAGE C;
```

Der Befehl

```
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler;
```

legt dann fest, dass die vorher deklarierte Handler-Funktion aufgerufen werden soll, wenn die Sprache `plpgsql` angegeben wurde.

In der Standardinstallation von PostgreSQL wird der Handler für die Sprache `PL/pgSQL` automatisch im Bibliotheksverzeichnis installiert. Wenn Tcl/Tk-Unterstützung eingeschaltet wurde, werden die Handler für `PL/Tcl` und `PL/TclU` im selben Verzeichnis installiert. Ebenso werden `PL/Perl` und `PL/PerlU` installiert, wenn Perl-Unterstützung eingeschaltet wurde, und `PL/Python` wird installiert, wenn Python-Unterstützung ausgewählt wurde.

38

PL/pgSQL: SQL prozedurale Sprache

PL/pgSQL ist eine ladbare prozedurale Sprache für das PostgreSQL-Datenbanksystem. Das Ziel bei der Entwicklung von PL/pgSQL war es, eine ladbare prozedurale Sprache zu erstellen, die

- verwendet werden kann, um Funktionen und Triggerprozeduren zu schreiben,
- Kontrollstrukturen zur Sprache SQL hinzufügt,
- komplexe Berechnungen ausführen kann,
- alle benutzerdefinierten Typen, Funktionen und Operatoren zur Verfügung hat,
- vom Server als TRUSTED eingestuft werden kann,
- einfach anzuwenden ist.

38.1 Überblick

Das PL/pgSQL-Modul liest den Quelltext der Funktion und erstellt einen internen binären Anweisungsbaum, wenn die Funktion zum ersten Mal in einer Sitzung aufgerufen wird. Der Anweisungsbaum spiegelt die Struktur der PL/pgSQL-Anweisungen wider, aber einzelne SQL-Ausdrucke und -Befehle werden nicht sofort übersetzt.

Wenn ein SQL-Ausdruck oder eine Anfrage zum ersten Mal in einer Funktion verwendet wird, erzeugt der PL/pgSQL-Interpreter einen vorbereiteten Ausführungsplan (mit den SPI-Funktionen SPI_prepare und SPI_saveplan). Wenn der Ausdruck oder der Befehl danach noch einmal ausgeführt werden muss, wird der vorbereitete Plan wiederverwendet. In einer Funktion mit Auswahlstrukturen, in denen mehrere Anweisungen enthalten sind, für die Ausführungspläne benötigt werden könnten, werden also nur die Pläne vorbereitet und gespeichert, die während der Lebensdauer der Datenbankverbindung tatsächlich verwendet werden. Dadurch kann bei einer PL/pgSQL-Funktion die zum Parsen und Erzeugen der Ausführungspläne benötigte Zeit erheblich reduziert werden. Ein Nachteil ist, dass Fehler in einem bestimmten Ausdruck oder einem Befehl erst entdeckt werden können, wenn der entsprechende Teil der Funktion ausgeführt wird.

Wenn PL/pgSQL für einen bestimmten Befehl in einer Funktion einen Ausführungsplan erstellt hat, wird dieser für die Dauer der Datenbankverbindung wiederverwendet. Das ist normalerweise ein Leistungsgewinn, kann aber Probleme verursachen, wenn Sie Ihr Datenbankschema dynamisch verändern. Ein Beispiel:

```
CREATE FUNCTION test() RETURNS integer AS '
DECLARE
```

```
    -- Deklarationen
BEGIN
    PERFORM meine_funktion();
END;
' LANGUAGE plpgsql;
```

Wenn Sie diese Funktion ausführen, wird die OID der Funktion `meine_funktion()` im Ausführungsplan des Befehls `PERFORM` gespeichert. Wenn Sie die Funktion `meine_funktion()` später löschen und neu erzeugen, wird `test()` die Funktion `meine_funktion()` nicht mehr finden können. Sie müssten die Funktion `test()` dann neu erzeugen oder zumindest eine neue Datenbanksitzung starten, damit sie neu kompiliert würde.

Weil `PL/pgSQL` die Ausführungspläne auf diese Art und Weise speichert, müssen SQL-Befehle, die direkt in einer `PL/pgSQL`-Funktion verwendet werden, bei jeder Ausführung auf dieselben Tabellen und Spalten verweisen; das heißt, man kann keine Parameter als Name einer Tabelle oder einer Spalte in einem SQL-Befehl verwenden. Um diese Beschränkung zu umgehen, können Sie mit dem `PL/pgSQL`-Befehl `EXECUTE` dynamische Befehle bauen, wobei dann allerdings ein neuer Ausführungsplan bei jeder Ausführung erstellt werden muss.

Anmerkung

Der `PL/pgSQL`-Befehl `EXECUTE` hat nichts mit dem Befehl `EXECUTE` im PostgreSQL-Server zu tun. Der `EXECUTE`-Befehl des Servers kann in `PL/pgSQL` nicht verwendet werden (wird aber auch nicht benötigt).

Außer Eingabe- und Ausgabefunktionen für benutzerdefinierte Typen und Funktionen, die mit solchen Typen Berechnungen anstellen, können alle Funktion, die in C geschrieben werden können, auch in `PL/pgSQL` geschrieben werden. Zum Beispiel können Sie Funktionen erzeugen, die komplexe Berechnungen durchführen und diese dann verwenden, um Operatoren zu definieren oder einen Funktionsindex zu erzeugen.

38.1.1 Vorteile von PL/pgSQL

SQL ist die Sprache, die PostgreSQL (und die meisten anderen relationalen Datenbanken) als Anfragesprache verwenden. Sie ist portierbar und leicht zu erlernen. Aber jeder SQL-Befehl muss vom Datenbankserver einzeln ausgeführt werden.

Das bedeutet, dass Ihre Clientanwendung jede Anfrage an den Datenbankserver schickt, wartet, bis sie verarbeitet wurde, die Ergebnisse verarbeitet, einige Berechnungen vornimmt und dann weitere Anfragen an den Server schickt. Dadurch entsteht Interprozesskommunikation und, wenn Server und Client auf verschiedenen Maschinen sind, Verzögerungen durch das Netzwerk.

Mit `PL/pgSQL` können Sie Berechnungen und Anfragen *innerhalb* des Datenbankservers zusammenfassen. Dadurch erhalten Sie die Fähigkeiten einer prozeduralen Sprache und die bekannte Funktionalität von SQL, sparen aber viel Zeit, weil die Verzögerung durch die Client/Server-Kommunikation wegfällt. Das kann die Leistung einer Datenbankverbindung erheblich verbessern.

Mit `PL/pgSQL` können Sie außerdem alle Datentypen, Operatoren und Funktionen von SQL verwenden.

38.1.2 Entwickeln in PL/pgSQL

Hier sind zwei Möglichkeiten, `PL/pgSQL`-Funktionen zu entwickeln:

- Sie verwenden einen Texteditor und laden die Datei mit `psql`.
- Sie verwenden das grafische PostgreSQL-Clientprogramm `PgAccess`.

Eine gute Art, PL/pgSQL-Funktionen zu entwickeln, ist, einfach mit Ihrem Lieblingstexteditor die Funktionen zu schreiben und sie dann in einem anderen Fenster mit psql zu laden. Wenn Sie es so tun wollen, dann ist es zu empfehlen, die Funktionen mit CREATE OR REPLACE FUNCTION zu erzeugen. Dann können Sie die Datei einfach neu laden, um die Funktionsdefinition zu aktualisieren. Zum Beispiel:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Innerhalb von psql können Sie so eine Funktionsdefinitionsdatei mit folgendem Befehl laden:

```
\i dateiname.sql
```

Danach können Sie sofort SQL-Befehle ausführen, um die Funktion zu testen.

Eine andere interessante Möglichkeit, um PL/pgSQL-Funktionen zu entwickeln, ist mit dem grafischen PostgreSQL-Clientprogramm PgAccess. Es nimmt Ihnen einige Arbeit ab, zum Beispiel das Ersetzen von Apostrophen durch Fluchtfolgen, und macht es leicht, Funktionsdefinitionen zu aktualisieren und Funktionen zu debuggen.

38.2 Umgang mit Apostrophen

Da der Code einer Funktion in einer prozeduralen Sprache in CREATE FUNCTION als Zeichenkettenkonstante angegeben wird, müssen Apostrophe im Funktionskörper durch Fluchtfolgen ersetzt werden. Das kann manchmal zu einem ziemlich komplizierten Code führen, besonders wenn Sie eine Funktion schreiben, die andere Funktionen erzeugt, wie im Beispiel in Abschnitt 38.6.4. Die Liste unten gibt Ihnen einen Überblick über die benötigten Apostrophe in verschiedenen Situationen. Halten Sie sich diese Tabelle parat.

- 1 Apostroph
  Am Anfang und Ende eines Funktionskörpers, zum Beispiel:

  ```
  CREATE FUNCTION foo() RETURNS integer AS '...'
      LANGUAGE plpgsql;
  ```

- 2 Apostrophe
  Für Zeichenkettenkonstanten im Funktionskörper, zum Beispiel:

  ```
  a_output := ''Bla'';
  SELECT * FROM benutzer WHERE name=''foobar'';
  ```

  Die zweite Zeile wird dann so interpretiert:

  ```
  SELECT * FROM benutzer WHERE name='foobar';
  ```

- 4 Apostrophe
  Wenn Sie einen Apostroph in einer Zeichenkette im Funktionskörper benötigen, zum Beispiel:

  ```
  a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
  ```

  Der Wert von a_output wäre dann: AND name LIKE 'foobar' AND xyz.

- 6 Apostrophe

  Wenn ein Apostroph in einer Zeichenkette im Funktionskörper an das Ende dieser Zeichenkettenkonstante angrenzt, zum Beispiel:

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

  Der Wert von `a_output` wäre dann: `AND name LIKE 'foobar'`.

- 10 Apostrophe

  Wenn Sie zwei Apostrophe in einer Zeichenkettenkonstante haben wollen (8 Apostrophe) und dies an das Ende dieser Zeichenkettenkonstante angrenzt (2 Apostrophe). Das werden Sie wahrscheinlich nur brauchen, wenn Sie eine Funktion schreiben, die andere Funktionen erzeugt. Zum Beispiel:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

  Der Wert von `a_output` wäre dann:

```
if v_... like ''...'' then return ''...''; end if;
```

38.3 Die Struktur von PL/pgSQL

`PL/pgSQL` ist eine blockstrukturierte Sprache. Der komplette Text einer Funktionsdefinition muss ein *Block* sein. Ein Block ist definiert als:

```
[ <<label>> ]
[ DECLARE
    deklarationen ]
BEGIN
    anweisungen
END;
```

Jede Deklaration und jede Anweisung in einem Block wird mit einem Semikolon abgeschlossen.

Alle Schlüsselwörter und Bezeichner können in Groß- oder Kleinbuchstaben geschrieben werden. Bezeichner werden automatisch in Kleinbuchstaben umgewandelt, wenn Sie nicht in Anführungszeichen stehen.

Es gibt zwei Typen von Kommentaren in `PL/pgSQL`. Ein doppelter Bindestrich (`--`) startet einen Kommentar, der sich bis zum Ende der Zeile erstreckt. Ein `/*` startet einen Blockkommentar, der bis zum nächsten Auftreten von `*/` geht. Blockkommentare können nicht geschachtelt werden, aber ein Doppelstrich-Kommentar kann in einem Blockkommentar stehen und ein Doppelstrich kann die Blockkommentarbegrenzer `/*` und `*/` verstecken.

Jede Anweisung im Anweisungsteil eines Blocks kann ein *Subblock* sein. Subblöcke können verwendet werden, um Anweisungen logisch zu gruppieren oder um lokale Variablen anzulegen.

Die im Deklarationsteil des Blocks deklarierten Variablen werden jedes Mal, wenn der Block ausgeführt wird, auf ihre Vorgabewerte gesetzt, nicht nur einmal pro Funktionsaufruf. Zum Beispiel:

```
CREATE FUNCTION funk1() RETURNS integer AS '
```

```
DECLARE
    menge integer := 30;
BEGIN
    RAISE NOTICE ''Menge ist %'', menge;  -- Menge ist 30
    menge := 50;
    --
    -- Erzeuge Subblock
    --
    DECLARE
        menge integer := 80;
    BEGIN
        RAISE NOTICE ''Menge ist %'', menge;  -- Menge ist 80
    END;

    RAISE NOTICE ''Menge ist %'', menge;  -- Menge ist 50

    RETURN menge;
END;
' LANGUAGE plpgsql;
```

Verwechseln Sie nicht BEGIN/END zum Gruppieren von Anweisungen in PL/pgSQL mit den Datenbankbefehlen zur Transaktionskontrolle. In PL/pgSQL sind BEGIN/END nur zum Gruppieren, sie starten bzw. beenden keine Transaktionen. Funktionen und Triggerprozeduren werden immer in der von der äußeren Anfrage eingerichteten Transaktion ausgeführt; sie können keine Transaktionen starten oder abschließen, da PostgreSQL keine geschachtelten Transaktionen unterstützt.

38.4 Deklarationen

Alle Variablen, die in einem Block verwendet werden, müssen im Deklarationsabschnitt dieses Blocks deklariert werden. (Die einzige Ausnahme ist, dass die Schleifenvariable in einer FOR-Schleife, die durch einen Bereich ganzer Zahlen iteriert, automatisch als Ganzzahlvariable deklariert ist.)

PL/pgSQL-Variablen können jeden SQL-Datentyp haben, zum Beispiel integer, varchar und char.

Hier sind einige Beispiel für Variablendeklarationen:

```
benutzer_id integer;
menge numeric(5);
url varchar;
meinezeile tabellenname%ROWTYPE;
meinfeld tabellenname.spaltenname%TYPE;
einezeile RECORD;
```

Die allgemeine Syntax einer Variablendeklaration ist:

```
name [ CONSTANT ] typ [ NOT NULL ] [ { DEFAULT | := } ausdruck ];
```

Wenn die DEFAULT-Klausel angegeben wird, dann wird der Variable der angegebene Wert zugewiesen, wenn die Ausführung des Blocks beginnt. Wenn die DEFAULT-Klausel weggelassen wird, wird die Variable

mit dem NULL-Wert initialisiert. Die Option `CONSTANT` verhindert, dass der Variablen ein Wert zugewiesen werden kann, womit also der Wert im gesamten Block konstant bleibt. Wenn `NOT NULL` angegeben wird, verursacht die Zuweisung eines NULL-Werts zu einem Laufzeitfehler. Für alle mit `NOT NULL` deklarierten Variablen muss ein vom NULL-Wert verschiedener Vorgabewert angegeben werden.

Der Vorgabewert wird jedes Mal ausgewertet, wenn der Block ausgeführt wird. Wenn zum Beispiel einer Variablen des Typs timestamp der Wert `'now'` zugewiesen wird, enthält die Variable dadurch die Zeit, des Funktionsaufrufs, nicht etwa die Zeit als die Funktion kompiliert wurde.

Beispiele:

```
menge integer DEFAULT 32;
url varchar := ''http://mysite.com'';
benutzer_id CONSTANT integer := 10;
```

38.4.1 Aliasnamen für Funktionsparameter

```
name ALIAS FOR $n;
```

Die der Funktion übergebenen Parameter haben die Bezeichner `$1`, `$2` usw. Den Parameternamen `$n` können wahlweise Aliasnamen zugewiesen werden, um die Lesbarkeit zu erhöhen. Dann kann man auf die Parameter entweder durch den Aliasnamen oder den numerischen Bezeichner verweisen. Einige Beispiele:

```
CREATE FUNCTION mehrwertsteuer(real) RETURNS real AS '
DECLARE
    brutto ALIAS FOR $1;
BEGIN
    RETURN brutto * 0.16;
END;
' LANGUAGE plpgsql;


CREATE FUNCTION instr(varchar, integer) RETURNS integer AS '
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- Berechnungen
END;
' LANGUAGE plpgsql;


CREATE FUNCTION viele_felder(tabellenname) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
' LANGUAGE plpgsql;
```

38.4.2 Typen kopieren

```
variable%TYPE
```

`%TYPE` ergibt den Datentyp einer Variablen oder einer Tabellenspalte. Damit können Sie Variablen deklarieren, die Werte aus der Datenbank aufnehmen sollen. Nehmen wir zum Beispiel an, Sie haben eine Spalte namens `benutzer_id` in einer Tabelle namens `benutzer`. Um eine Variable mit demselben Datentyp wie `benutzer.benutzer_id` zu deklarieren, schreiben Sie:

```
benutzer_id benutzer.benutzer_id%TYPE;
```

Wenn Sie `%TYPE` verwenden, müssen Sie den Datentyp der Struktur, auf die Sie verweisen, nicht kennen, und, was noch viel wichtiger ist, wenn der Datentyp der Struktur sich verändert (zum Beispiel, Sie ändern den Typ von `benutzer_id` von integer in `real`), müssen Sie eventuell die Definition der Funktion nicht ändern.

38.4.3 Zeilentypen

```
name tabellenname%ROWTYPE;
```

Variablen eines zusammengesetzten Typs heißen **Zeilenvariablen** (oder **Zeilentypvariablen**). So eine Variable kann eine ganze Zeile eines Anfrageergebnisses von `SELECT` oder `FOR` speichern, vorausgesetzt, dass die Spalten des Anfrageergebnisses mit dem deklarierten Typ der Variable übereinstimmen. *tabellenname* muss der Name einer bestehenden Tabelle oder Sicht in der Datenbank sein. Auf die einzelnen Felder des Zeilenwerts kann mit der normalen Punktschreibweise zugegriffen werden, zum Beispiel `zeilenvar.feld`.

Gegenwärtig kann eine Zeilenvariablen nur mit der Schreibweise `%ROWTYPE` deklariert werden. Obwohl man vielleicht erwarten könnte, dass ein einfacher Tabellenname als Typdeklaration funktioniert, wird das in `PL/pgSQL`-Funktionen nicht akzeptiert.

Auch Funktionsparameter können zusammengesetzte Typen (komplette Tabellenzeilen) haben. In diesem Fall wäre der entsprechende Bezeichner $*n* eine Zeilenvariable und man könne daraus Felder auswählen, zum Beispiel `$1.benutzer_id`.

Nur benutzerdefinierte Spalten einer Tabellenzeile sind in der Zeilentypvariable vorhanden, nicht die OID oder andere Systemspalten (weil die Zeile aus einer Sicht stammen könnte). Die Felder eines Zeilentyps übernehmen die Feldgrößen und die Präzision für Datentypen wie char(*n*) aus der Tabelle.

Hier ist ein Beispiel für die Verwendung von zusammengesetzten Typen:

```
CREATE FUNCTION verwende_zwei_tabellen(tabellenname) RETURNS text AS '
DECLARE
    in_t ALIAS FOR $1;
    hilfs_t table2name%ROWTYPE;
BEGIN
    SELECT * INTO hilfs_t FROM table2name WHERE ... ;
    RETURN in_t.f1 || hilfs_t.f3 || in_t.f5 || hilfs_t.f7;
END;
' LANGUAGE plpgsql;
```

38.4.4 Record-Variablen

```
name RECORD;
```

Record-Variablen sind ähnlich wie Zeilentypvariablen, haben aber keine vordefinierte Struktur. Sie nehmen jeweils die Struktur einer von einem SELECT- oder FOR-Befehl zugewiesenen Zeile an. Die Struktur einer Record-Variablen kann sich also jedes Mal ändern, wenn ihr ein Wert zugewiesen wird. Als Folge daraus hat eine Record-Variable erst eine Struktur, wenn ihr ein Wert zugewiesen wird, und Versuche, vorher auf ein Feld zuzugreifen, verursachen einen Laufzeitfehler.

Beachten Sie, dass RECORD kein richtiger Datentyp ist, sondern nur ein Platzhalter.

38.5 Ausdrücke

Alle Ausdrücke in PL/pgSQL-Anweisungen werden vom normalen SQL-Auswertungssystem des Servers verarbeitet. Ausdrücke, die scheinbar Konstanten enthalten, können in der Tat Auswertung zur Laufzeit erfordern (z.B. 'now' beim Typ timestamp), weswegen der PL/pgSQL-Parser keine echten konstanten Werte identifizieren kann, außer dem Schlüsselwort NULL. Intern werden alle Ausdrücke mit einer Anfrage der Form

```
SELECT ausdruck
```

durch das SPI-System ausgewertet. Bei der Auswertung werden in dem Ausdruck die Bezeichner von PL/pgSQL-Variablen durch Parameter ersetzt und die eigentlichen Werte aus den Variablen werden im Parameterarray übergeben. Dadurch kann der Anfrageplan des SELECT einmal vorbereitet und dann bei darauf folgenden Auswertungen wiederverwendet werden.

Die Auswertung der Ausdrücke durch den PostgreSQL-Parser hat einige Auswirkungen auf die Interpretation von konstanten Werten. Es gibt zum Beispiel einen feinen Unterschied zwischen diesen beiden Funktionen:

```
CREATE FUNCTION logfunktion1(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtabelle VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE plpgsql;
```

und

```
CREATE FUNCTION logfunktion2(text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
        zeit timestamp;
    BEGIN
        zeit := ''now'';
        INSERT INTO logtabelle VALUES (logtxt, zeit);
        RETURN zeit;
    END;
```

```
' LANGUAGE plpgsql;
```

Im Fall von `logfunktion1` weiß der PostgreSQL-Parser bei der Vorbereitung des Plans für den `INSERT`-Befehl, dass die Zeichenkette `'now'` als Typ timestamp interpretiert werden soll, weil die Zielspalte in der Tabelle `logtabelle` diesen Typ hat. Daher ermittelt er zu diesem Zeitpunkt den Wert und verwendet diese Konstante dann für alle Aufrufe von `logfunktion1` für die Dauer der Sitzung. Das ist logischerweise nicht das, was der Programmierer beabsichtigt hatte.

Im Fall von `logfunktion2` weiß der PostgreSQL-Parser nicht, welchen Typ `'now'` haben soll und gibt daher einen Wert des Typs text mit dem Inhalt `now` zurück. Bei der folgenden Zuweisung zur lokalen Variable `zeit` wird diese Zeichenkette vom `PL/pgSQL`-Interpreter in den Typ timestamp umgewandelt, indem die Funktionen `text_out` und `timestamp_in` aufgerufen werden. Die ermittelte Zeitangabe wird also bei jeder Ausführung aktualisiert, wie vom Programmierer erwartet.

Die veränderliche Natur der Record-Variablen stellt in diesem Zusammenhang ein Problem dar. Wenn Felder einer Record-Variablen in einem Ausdruck oder einer Anweisung verwendet werden, dürfen sich die Datentypen der Felder zwischen Aufrufen desselben Ausdrucks nicht ändern, weil der Ausdruck mit dem Datentyp geplant wird, der beim ersten Aufruf des Ausdrucks verwendet wird. Beachten Sie dies, wenn Sie Triggerprozeduren schreiben, die Ereignisse für mehrere Tabellen bearbeiten können sollen. (Wenn nötig, kann dieses Problem mit `EXECUTE` umgangen werden.)

38.6 Einfache Anweisungen

In diesem und den folgenden Abschnitten beschreiben wir alle Anweisungsarten, die direkt von `PL/pgSQL` verstanden werden. Bei allem, das nicht als einer dieser Anweisungstypen erkannt wird, wird davon ausgegangen, dass es ein SQL-Befehl ist, der vom Datenbankserver selbst ausgeführt werden soll (nach den Einsetzen von etwaigen im Befehl verwendeten `PL/pgSQL`-Variablen). Die SQL-Befehle `INSERT`, `UPDATE` und `DELETE` gelten also zum Beispiel als `PL/pgSQL`-Anweisungen, werden hier aber nicht gesondert aufgezählt.

38.6.1 Zuweisungen

Die Zuweisung eines Werts zu einer Variable oder einem Zeilenfeld wird so geschrieben:

```
bezeichner := ausdruck;
```

Wie oben erklärt, wird der Ausdruck in einer solchen Anweisung durch den SQL-Befehl `SELECT` vom Datenbanksystem ausgewertet. Der Ausdruck muss einen einzelnen Wert ergeben.

Wenn der Ergebnistyp des Ausdrucks nicht mit dem Datentyp der Variable übereinstimmt oder die Variable eine Größen- oder Präzisionsangabe hat (wie char(20)), dann wird der Ergebniswert vom `PL/pgSQL`-Interpreter automatisch umgewandelt, indem die Ausgabefunktion des Ergebnistyps gefolgt von der Eingabefunktion des Variablentyps aufgerufen wird. Beachten Sie, dass die Eingabefunktion eventuell Laufzeitfehler verursachen kann, wenn die Zeichenkettenform des Ergebniswerts für die Eingabefunktion nicht akzeptabel ist.

Beispiele:

```
benutzer_id := 20;
steuer := brutto * 0.16;
```

38.6.2 SELECT INTO

Das Ergebnis eines SELECT-Befehls, der mehrere Spalten (aber nur eine Zeile) ergibt, kann einer Record-Variablen, einer Zeilentypvariablen oder einer Liste von skalaren Variablen zugewiesen werden. Das wird so gemacht:

```
SELECT INTO ziel ausdrücke FROM ...;
```

Hier kann *ziel* eine Record-Variable, eine Zeilentypvariable oder eine Liste von einfachen Variablen und Record-/Zeilenfeldern, durch Kommas getrennt, sein.

Beachten Sie, dass sich dies von der normalen Interpretation von SELECT INTO in PostgreSQL unterscheidet, wo das Ziel von INTO eine neu erzeugte Tabelle ist. Wenn Sie in einer PL/pgSQL-Funktion eine Tabelle aus einem SELECT-Ergebnis erzeugen wollen, verwenden Sie die Syntax CREATE TABLE ... AS SELECT.

Wenn eine Zeile oder eine Variablenliste als Ziel verwendet wird, müssen die von der Anfrage ausgewählten Werte genau mit der Struktur des Ziels übereinstimmen, ansonsten wird ein Laufzeitfehler erzeugt. Wenn das Ziel eine Record-Variable ist, stellt sie sich automatisch auf den Zeilentyp der Ergebnisspalten der Anfrage ein.

Abgesehen von der INTO-Klausel ist der SELECT-Befehl wie der normale SQL-Befehl SELECT und kann alle Fähigkeiten davon verwenden.

Wenn die Anfrage keine Zeilen ergibt, werden den Zielen NULL-Werte zugewiesen. Wenn die Anfrage mehrere Zeilen ergibt, wird die erste Zeile den Zielen zugewiesen und die restlichen werden verworfen. (Beachten Sie, dass “die erste Zeile” nur genau bestimmbar ist, wenn Sie ORDER BY verwendet haben.)

Gegenwärtig kann die INTO-Klausel fast überall in dem SELECT-Befehl auftreten, aber es wird empfohlen, sie wie oben gezeigt direkt nach dem Schlüsselwort SELECT zu setzen. Zukünftige Versionen von PL/pgSQL werden vielleicht weniger flexibel bezüglich der Platzierung der INTO-Klausel sein.

Unmittelbar nach einem SELECT INTO-Befehl können Sie FOUND verwenden, um zu ermitteln, ob die Zuweisung erfolgreich war (das heißt, dass die Anfrage mindestens eine Zeile ergab). Zum Beispiel:

```
SELECT INTO mein_rec * FROM person WHERE personname = meinname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''Person % nicht gefunden'', meinname;
END IF;
```

Um zu testen, ob ein Record- oder Zeilenergebnis den NULL-Wert hat, können Sie die Klausel IS NULL verwenden. Es ist hingegen nicht möglich, festzustellen, ob zusätzliche Zeilen verworfen worden sind. Hier ist ein Beispiel, das den Fall, dass keine Zeilen von der Anfrage zurückgegeben worden sind, korrekt verarbeitet:

```
DECLARE
    benutzer_rec RECORD;
    name varchar;
BEGIN
    SELECT INTO benutzer_rec * FROM benutzer WHERE benutzer_id=3;

    IF benutzer_rec.homepage IS NULL THEN
    -- Wenn der Benutzer keine Homepage eingegeben hat, dann gib "http://" zurück.
        RETURN ''http://'';
    END IF;
END;
```

38.6.3 Einen Ausdruck oder eine Anfrage ohne Ergebnis ausführen

Manchmal möchte man einen Ausdruck oder eine Anfrage auswerten, aber das Ergebnis verwerfen (normalerweise weil man eine Funktion aufruft, die nützliche Nebeneffekte hat, aber keinen brauchbaren Ergebniswert liefert). Um das in PL/pgSQL zu tun, verwendet man den Befehl PERFORM:

```
PERFORM anfrage;
```

Dieser Befehl führt *anfrage*, was ein SELECT-Befehl sein muss, aus und wirft das Ergebnis weg. PL/pgSQL-Variablen werden wie gewohnt in die Anfrage eingesetzt. Außerdem wird die besondere Variable FOUND auf wahr gesetzt, wenn die Anfrage mindestens eine Zeile ergab, und auf falsch, wenn sie keine Zeilen ergab.

Anmerkung

Man mag vielleicht denken, dass man diese Wirkung auch mit SELECT ohne INTO-Klausel erzielen könnte, aber gegenwärtig geht das nur mit PERFORM.

Ein Beispiel:

```
PERFORM create_mv(''cs_session_page_requests_mv'', my_query);
```

38.6.4 Dynamische Befehle ausführen

Häufig werden Sie in PL/pgSQL-Funktionen dynamische Befehle ausführen wollen, dass heißt Befehle, die bei jeder Ausführung eine andere Tabelle oder andere Datentypen verwenden. In so einer Situation wird das Speichern von vorbereiteten Ausführungsplänen in PL/pgSQL nicht funktionieren. Als Lösung können Sie den Befehl EXECUTE verwenden:

```
EXECUTE befehl;
```

befehl ist ein Ausdruck, der eine Zeichenkette (vom Typ text) ergibt, die den auszuführenden Befehl enthält. Diese Zeichenkette wird der SQL-Maschinerie unverändert übergeben.

Beachten Sie insbesondere, dass in der Befehlszeichenkette keine PL/pgSQL-Variablen ersetzt werden. Die Werte von Variablen müssen beim Aufbauen der Befehlszeichenkette eingefügt werden.

Beim Arbeiten mit dynamischen Befehlen werden Sie sicher Probleme mit den Fluchtfolgen für die Apostrophe bekommen. Abschnitt 38.2 enthält eine Übersicht, die Ihnen die Sache etwas leichter machen kann.

Im Gegensatz zu allen anderen Befehlen in PL/pgSQL wird ein mit EXECUTE ausgeführter Befehl nicht nur einmal pro Sitzung vorbereitet. Stattdessen wird der Befehl jedes Mal vorbereitet, wenn der EXECUTE-Befehl ausgeführt wird. Die Befehlszeichenkette kann in der Funktion dynamisch erzeugt werden, um Aktionen mit variablen Tabellen und Spalten auszuführen.

Die Ergebnisse eines SELECT-Befehls werden von EXECUTE weggeworfen und SELECT INTO wird innerhalb von EXECUTE gegenwärtig nicht unterstützt. Die einzige Möglichkeit, an das Ergebnis eines dynamisch erzeugten SELECT-Befehls zu kommen, ist daher die Form FOR-IN-EXECUTE, die weiter unten beschrieben wird.

Ein Beispiel:

```
EXECUTE ''UPDATE tbl SET ''
        || quote_ident(spaltenname)
```

```
          || '' = ''
          || quote_literal(neuerwert)
          || '' WHERE ...'';
```

Dieses Beispiel zeigt die Verwendung der Funktionen `quote_ident(text)` und `quote_literal(text)`. Variablen, die Spalten- und Tabellennamen enthalten, sollten durch die Funktion `quote_ident` geschickt werden. Variablen, die Werte enthalten, die in der konstruierten Befehlszeichenkette als Wertkonstanten agieren, sollten durch die Funktion `quote_literal` geschickt werden. Beide Funktionen sorgen dafür, dass die Werte in Anführungszeichen bzw. Apostrophe eingeschlossen werden und dass im Wert enthaltene Sonderzeichen ordnungsgemäß durch Fluchtzeichen ersetzt werden.

Hier ist ein größeres Beispiel eines dynamischen Befehls und `EXECUTE`:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  -- für die FOR-Schleife
    a_output varchar(4000);
BEGIN
   a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar, varchar, varchar)
                 RETURNS varchar AS ''''
                    DECLARE
                        v_host ALIAS FOR $1;
                        v_domain ALIAS FOR $2;
                        v_url ALIAS FOR $3;
                    BEGIN '';

    -- Schleife durch die Ergebnisse einer Anfrage mit der Konstruktion FOR
<record>.

    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
    a_output := a_output || '' IF v_'' || referrer_keys.kind || '' LIKE ''''''''''
             || referrer_keys.key_string || ''''''''''  THEN RETURN ''''''
             || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE plpgsql;'';

    EXECUTE a_output;
END;
' LANGUAGE plpgsql;
```

38.6.5 Ergebnisstatus

Es gibt mehrere Möglichkeiten, die Auswirkungen eines Befehls zu ermitteln. Die erste Methode ist die Verwendung des Befehls `GET DIAGNOSTICS`, welcher folgende Form hat:

```
GET DIAGNOSTICS variable = wert [ , ... ] ;
```

Mit diesem Befehl können die Werte diverser Statusanzeigen ermittelt werden. Jeder *wert* ist ein Schlüsselwort, das angibt, welcher Statuswert der angegebenen Variablen zugewiesen werden soll. (Die Variable sollte den richtigen Datentyp haben, um den Wert aufnehmen zu können.) Die gegenwärtig verfügbaren Statuswerte sind `ROW_COUNT`, die Anzahl der vom letzten SQL-Befehl bearbeiteten Zeilen, und `RESULT_OID`, die OID der letzten Zeile, die vom letzten SQL-Befehl eingefügt worden ist. Beachten Sie, dass `RESULT_OID` nur nach einem `INSERT`-Befehl einen sinnvollen Wert enthält.

Ein Beispiel:

```
GET DIAGNOSTICS var_integer = ROW_COUNT;
```

Die zweite Möglichkeit, die Auswirkungen eines Befehls festzustellen, ist die besondere Variable mit Namen `FOUND` vom Typ boolean. `FOUND` ist am Anfang jeder `PL/pgSQL` falsch und wird von den folgenden Befehlen gesetzt:

- `SELECT INTO` setzt `FOUND` auf wahr, wenn es mindestens eine Zeile zurückgegeben hat, und auf falsch, wenn es keine Zeilen geliefert hat.
- `PERFORM` setzt `FOUND` auf wahr, wenn die enthaltene Anfrage eine Zeile ergeben hat (welche von `PERFORM` weggeworfen wurde), und auf falsch, wenn sie keine Zeilen ergeben hat.
- Die Befehle `UPDATE`, `INSERT` und `DELETE` setzen `FOUND` auf wahr, wenn mindestens eine Zeile bearbeitet wurde, und auf falsch, wenn keine Zeile bearbeitet wurde.
- `FETCH` setzt `FOUND` auf wahr, wenn es eine Zeile zurückgegeben hat, und auf falsch, wenn keine Zeilen zurückgegeben worden sind.
- Eine `FOR`-Anweisung setzt `FOUND` auf wahr, wenn die Schleife mindestens einmal durchlaufen wurde, ansonsten auf falsch. Das gilt für alle drei Varianten der `FOR`-Anweisung (ganzzahlige Schleifen, Record-Set-Schleifen und dynamische Record-Set-Schleifen). `FOUND` wird erst gesetzt, wenn die `FOR`-Schleife verlassen wird: Innerhalb der Schleife wird `FOUND` durch die `FOR`-Anweisung nicht verändert, aber es könnte durch die Ausführung anderer Anweisungen im Schleifenkörper verändert werden.

`FOUND` ist eine lokale Variable; Änderungen gelten nur für die aktuelle `PL/pgSQL`-Funktion.

38.7 Kontrollstrukturen

Die Kontrollstrukturen sind wahrscheinlich der nützlichste (und wichtigste) Teil von `PL/pgSQL`. Mit den Kontrollstrukturen von `PL/pgSQL` können Sie PostgreSQL-Daten flexibel und vielseitig manipulieren.

38.7.1 Aus einer Funktion zurückkehren

```
RETURN ausdruck;
```

`RETURN` mit einem Ausdruck beendet die Funktion und gibt den Wert von *ausdruck* als Ergebnis der Funktion zurück. Diese Form wird für `PL/pgSQL`-Funktionen, die keine Ergebnismenge zurückgeben, verwendet.

Um einen skalaren Typ zurückzugeben, kann jeder beliebige Ausdruck verwendet werden. Das Ergebnis des Ausdrucks wird automatisch in den Rückgabetyp der Funktion umgewandelt, wie bei Wertzuweisungen beschrieben. Um einen zusammengesetzten (Zeilen-)Wert zurückzugeben, müssen Sie als *ausdruck* eine Zeilen- oder Record-Variable angeben. Wenn die Funktion den Rückgabetyp void hat, kann der Ausdruck weggelassen werden und wird auf jeden Fall ignoriert.

Der Rückgabewert einer Funktion kann nicht undefiniert bleiben. Wenn bei der Ausführung das Ende des äußersten Blocks erreicht wird, ohne dass auf eine `RETURN`-Anweisung gestoßen worden ist, verursacht das einen Laufzeitfehler.

Wenn eine PL/pgSQL-Funktion einen deklarierten Rückgabetyp SETOF *typ* hat, dann muss ein anderes Vorgehen angewendet werden. In diesem Fall werden die einzelnen Elemente mit dem Befehl RETURN NEXT zurückgegeben,

```
RETURN NEXT ausdruck;
```

und der letzte RETURN-Befehl ohne Argumente zeigt an, dass die Funktionsausführung beendet ist. RETURN NEXT kann mit skalaren und zusammengesetzten Typen verwendet werden; im letzteren Fall wird eine ganze "Tabelle" mit Ergebnissen zurückgegeben.

Funktionen, die RETURN NEXT verwenden, sollten auf folgende Art aufgerufen werden:

```
SELECT * FROM meine_funktion();
```

Das heißt, die Funktion wird als Tabellenquelle in der FROM-Klausel verwendet.

RETURN NEXT kehrt nicht wirklich aus der Funktion zurück; es speichert einfach den Wert des Ausdrucks (oder der Record- oder Zeilenvariable, je nach Rückgabetyp) in einem Zwischenspeicher ab. Die Ausführung geht mit der nächsten Anweisung in der PL/pgSQL-Funktion weiter. Wenn weitere RETURN NEXT-Befehle ausgeführt werden, dann wird eine Ergebnismenge aufgebaut. Das letzte RETURN, welches keine Argumente haben darf, verlässt die Funktion.

Anmerkung

Die aktuelle Implementierung von RETURN NEXT in PL/pgSQL speichert die gesamte Ergebnismenge, bevor die Funktion verlassen wird, wie oben beschrieben. Das bedeutet, dass, wenn eine PL/pgSQL-Funktion eine sehr große Ergebnismenge erzeugt, dass dann die Leistung sehr bedürftig sein kann: Daten werden auf der Festplatte zwischengespeichert, um den Hauptspeicher nicht aufzubrauchen, aber die Funktion wird erst verlassen wenn die gesamte Ergebnismenge erzeugt worden ist. In der Zukunft wird es vielleicht einmal möglich sein, PL/pgSQL-Funktionen mit Mengenergebnissen zu definieren, die diese Einschränkung nicht haben. Gegenwärtig wird die Schwelle, an der die Daten auf der Festplatte zwischengespeichert werden, durch den Konfigurationsparameter sort_mem bestimmt. Administratoren, die ausreichend Speicher für große Ergebnismengen haben, sollten die Erhöhung dieses Parameters in Erwägung ziehen.

38.7.2 Auswahlanweisungen

Mit IF-Anweisungen können Sie Befehle aufgrund von Auswahlbedingungen ausführen. PL/pgSQL hat vier Formen von IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE

IF-THEN

```
IF boolean-ausdruck THEN
    anweisungen
END IF;
```

IF-THEN-Anweisungen sind die einfachste Form von IF. Die Anweisungen zwischen THEN und END IF werden ausgeführt, wenn die Bedingung wahr ist, ansonsten werden sie übersprungen.

Beispiel:

```
IF w_benutzer_id <> 0 THEN
    UPDATE benutzer SET email = w_email WHERE benutzer_id = w_benutzer_id;
END IF;
```

IF-THEN-ELSE

```
IF boolean-ausdruck THEN
    anweisungen
ELSE
    anweisungen
END IF;
```

IF-THEN-ELSE-Anweisungen sind eine erweiterte Form von IF-THEN, in der Sie Anweisungen angeben können, die ausgeführt werden sollen, wenn die Bedingung falsch ist.

Beispiele:

```
IF eltern_id IS NULL OR eltern_id = ''''
THEN
    RETURN name;
ELSE
    RETURN hp_dateiname(eltern_id) || ''/'' || name;
END IF;
IF w_anzahl > 0 THEN
    INSERT INTO benutzer_zahl (anzahl) VALUES (w_anzahl);
    RETURN ''t'';
ELSE
    RETURN ''f'';
END IF;
```

IF-THEN-ELSE IF

IF-Anweisungen können geschachtelt werden, wie in diesem Beispiel:

```
IF beispiel_zeile.geschlecht = ''m'' THEN
    anzeige := ''männlich'';
ELSE
    IF beispiel_zeile.geschlecht = ''w'' THEN
        anzeige := ''weiblich'';
    END IF;
END IF;
```

Wenn Sie diese Form verwenden, schachteln Sie eigentlich eine IF-Anweisung im ELSE-Teil einer äußeren IF-Anweisung. Daher benötigen Sie ein END IF für jedes geschachtelte IF und eins für das oberste IF-ELSE. Das ist machbar, wird aber umständlich, wenn man viele Alternativen prüfen muss. Daher gibt es die nächste Form.

IF-THEN-ELSIF-ELSE

```
IF boolean-ausdruck THEN
    anweisungen
[ ELSIF boolean-ausdruck THEN
    anweisungen
[ ELSIF boolean-ausdruck THEN
    anweisungen
    ...]]
[ ELSE
    anweisungen ]
END IF;
```

`IF-THEN-ELSIF-ELSE` ist eine bequemere Methode, um viele Alternativen in einer Anweisung zu prüfen. Formal entspricht es den geschachtelten `IF-THEN-ELSE-IF-THEN`-Anweisungen, benötigt aber nur ein `END IF`.

Hier ist ein Beispiel:

```
IF zahl = 0 THEN
    ergebnis := ''null'';
ELSIF zahl > 0 THEN
    ergebnis := ''positiv'';
ELSIF zahl < 0 THEN
    ergebnis := ''negativ'';
ELSE
    -- Die einzige verbleibende Möglichkeit ist der NULL-Wert.
    ergebnis := ''NULL'';
END IF;
```

38.7.3 Einfache Schleifen

Mit den Anweisungen `LOOP`, `EXIT`, `WHILE` und `FOR` können Sie es einrichten, dass Ihre `PL/pgSQL`-Funktion eine Folge von Befehlen wiederholt ausführt.

LOOP

```
[<<label>>]
LOOP
    anweisungen
END LOOP;
```

`LOOP` definiert eine bedingungslose Schleife, die ausgeführt wird, bis sie von einem `EXIT`- oder `RETURN`-Befehl beendet wird. Das optionale Label kann in `EXIT`-Anweisungen in geschachtelten Schleifen verwendet werden, um anzugeben, welche Schachtelungsebene beendet werden soll.

EXIT

```
EXIT [ label ] [ WHEN ausdruck ];
```

Wenn *label* nicht angegeben wurde, wird die innerste Schleife abgebrochen und als Nächstes die Anweisung nach dem `END LOOP` ausgeführt. Wenn *label* angegeben wurde, muss es das Label der aktuellen oder einer äußeren Ebene einer geschachtelten Schleife oder eines Blocks sein. Die benannte Schleife oder der Block wird beendet und die Ausführung geht nach dem entsprechenden `END` der Schleife bzw. des Blocks weiter.

Wenn `WHEN` vorhanden ist, wird die Schleife nur verlassen, wenn die angegebene Bedingung wahr ist, ansonsten geht die Ausführung mit der Anweisung nach dem `EXIT` weiter.

Beispiele:

```
LOOP
    -- Berechnungen
    IF anzahl > 0 THEN
        EXIT;  -- verlasse Schleife
    END IF;
END LOOP;

LOOP
    -- Berechnungen
    EXIT WHEN anzahl > 0;
END LOOP;

BEGIN
    -- Berechnungen
    IF anzahl > 100000 THEN
        EXIT;  -- ungültig; kann EXIT nicht außerhalb von LOOP verwenden
    END IF;
END;
```

WHILE

```
[<<label>>]
WHILE ausdruck LOOP
   anweisungen
END LOOP;
```

Die `WHILE`-Anweisung wiederholt eine Reihe von Anweisungen so lange, wie der Bedingungsausdruck wahr ist. Die Bedingung wird direkt vor jeder Ausführung des Schleifenkörpers geprüft.

Zum Beispiel:

```
WHILE soll > 0 AND geschenkgutscheine > 0 LOOP
    -- Berechnungen
END LOOP;

WHILE NOT boolean_ausdruck LOOP
    -- Berechnungen
```

```
END LOOP;
```

FOR (Ganzzahl-Variante)

```
[<<label>>]
FOR name IN [ REVERSE ] ausdruck .. ausdruck LOOP
    anweisungen
END LOOP;
```

Diese Form von FOR erzeugt eine Schleife durch einen Bereich ganzer Zahlen. Die Variable *name* wird automatisch als Typ integer definiert und existiert nur in der Schleife. Die zwei Ausdrücke, die die Unter- und Obergrenze des Bereichs angeben, werden einmal am Anfang der Schleife ausgewertet. Der Iterationsschritt ist normalerweise 1, aber -1, wenn REVERSE angegeben worden ist.

Einige Beispiel für FOR-Schleifen mit ganzzahligen Schleifenbedingungen:

```
FOR i IN 1..10 LOOP
    -- Anweisungen
    RAISE NOTICE ''i ist %'', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- Anweisungen
END LOOP;
```

38.7.4 Schleifen durch Anfrageergebnisse

Mit einer anderen Art der FOR-Schleife können Sie durch die Ergebnisse einer Anfrage iterieren und Daten entsprechend manipulieren. Die Syntax ist:

```
[<<label>>]
FOR record_oder_zeile IN anfrage LOOP
    anweisungen
END LOOP;
```

Der Record- oder Zeilenvariable wird nacheinander jede Zeile, die von der Anfrage (ein SELECT-Befehl) zurückgegeben wird, zugewiesen und die Schleife wird für jede Zeile ausgeführt. Hier ist ein Beispiel:

```
CREATE FUNCTION cs_msichten_aktualisieren() RETURNS integer AS '
DECLARE
   msichten RECORD;
BEGIN
    PERFORM cs_log(''Aktualisiere materialisierte Sichten...'');

    FOR msichten IN SELECT * FROM cs_materialisierte_sichten
        ORDER BY sortier_schlüssel LOOP

       -- Jetzt enthält "msichten" einen Datensatz aus cs_materialisierte_sichten.
```

```
        PERFORM cs_log(''Aktualisiere materialisierte Sicht '' ||
                quote_ident(msichten.ms_name) || ''...'');
        EXECUTE ''TRUNCATE TABLE  '' || quote_ident(msichten.ms_name);
        EXECUTE ''INSERT INTO '' || quote_ident(msichten.ms_name) || '' '' ||
                 msichten.ms_anfrage;
    END LOOP;

    PERFORM cs_log(''Aktualisierung der matierialisierten Sichten beendet.'');
    RETURN 1;
END;
' LANGUAGE plpgsql;
```

Wenn die Schleife durch eine EXIT-Anweisung beendet wird, kann man auch nach der Schleife noch auf den letzten zugewiesenen Zeilenwert zugreifen.

Die Anweisung FOR-IN-EXECUTE ist eine andere Möglichkeit, über Datensätze zu iterieren:

```
[<<label>>]
FOR record_oder_zeile IN EXECUTE text_ausdruck LOOP
    anweisungen
END LOOP;
```

Diese Form ist wie die vorige, außer dass die SELECT-Anweisung, die die Zeilen liefert, als Zeichenkettenausdruck angegeben wird, der am Anfang jeder Ausführung der FOR-Schleife ausgewertet und geplant wird. Damit kann ein Programmierer zwischen der besseren Geschwindigkeit einer vorbereiteten Anfrage und der Flexibilität einer dynamischen Anfrage wählen, wie bei einer einfachen EXECUTE-Anweisung.

Anmerkung

Der PL/pgSQL-Parser unterscheidet die zwei Arten der FOR-Schleifen (Ganzzahlbereich oder Anfrageergebnisse) gegenwärtig, indem geprüft wird, ob die Schleifenvariable nach dem FOR als Record- oder Zeilenvariable deklariert worden ist. Wenn nicht, wird davon ausgegangen, dass es eine FOR-Schleife durch einen Ganzzahlbereich ist. Das kann zu recht merkwürdigen Fehlermeldungen führen, wenn das wahre Problem zum Beispiel ein falsch geschriebener Variablenname nach FOR ist.

38.8 Cursor

Anstatt eine ganze Anfrage auf einmal auszuführen, können Sie einen **Cursor** einrichten, der eine Anfrage speichert, und dann die Anfrageergebnisse Zeile für Zeile (oder ein paar Zeilen am Stück) lesen. Ein Grund, das zu tun, ist zu verhindern, dass bei einer großen Zahl von Ergebniszeilen der gesamte Speicher aufgebraucht wird. (Benutzer von PL/pgSQL müssen sich darum jedoch normalerweise keine Sorgen machen, weil FOR-Schleifen intern automatisch einen Cursor verwenden, um Speicherprobleme zu vermeiden.) Eine interessantere Anwendung ist, einen Verweis auf einen Cursor zurückzugeben und demjenigen, der die Funktion aufgerufen hat, zu erlauben, die Zeilen zu lesen. Damit kann man auf effiziente Weise eine große Ergebnismenge aus einer Funktion zurückgeben.

38.8.1 Cursorvariablen deklarieren

Alle Zugriffe auf Cursor gehen in PL/pgSQL über Cursorvariablen, welche immer den besonderen Datentyp refcursor haben. Eine Möglichkeit, eine Cursorvariable zu erzeugen, ist, sie einfach als Variable des Typs refcursor zu deklarieren. Eine andere Möglichkeit ist die Verwendung der Cursordeklarationssyntax, welche im Allgemeinen so aussieht:

```
name CURSOR [ ( argumente ) ] FOR anfrage ;
```

(FOR kann durch IS ersetzt werden, um mit Oracle kompatibel zu sein.) *argumente*, wenn angegeben, ist eine Liste, durch Kommas getrennt, von Paaren der Art *name datentyp*, welche Namen angeben, die in der angegebenen Anfrage durch Parameterwerte ersetzt werden sollen. Die Werte, die tatsächlich für diese Namen eingesetzt werden, werden später angegeben, wenn der Cursor geöffnet wird.

Einige Beispiele:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Alle drei dieser Variablen haben den Typ refcursor, aber die erste kann mit jeder Anfrage verwendet werden, während die zweite schon eine vollständig angegebene Anfrage an sich *gebunden* hat und die letzte eine Anfrage mit Parametern an sich gebunden hat. (key wird durch einen Parameter vom Typ integer ersetzt, wenn der Cursor geöffnet wird.) Die Variable curs1 wird als *ungebunden* bezeichnet, weil sie an keine bestimmte Anfrage gebunden ist.

38.8.2 Cursor öffnen

Bevor ein Cursor verwendet werden kann, um Zeilen abzurufen, muss er mit dem Befehl OPEN *geöffnet* werden. (Das entspricht dem SQL-Befehl DECLARE CURSOR.) PL/pgSQL hat drei Formen der OPEN-Anweisung, wovon zwei ungebundene Cursorvariablen verwenden und die andere eine gebundene Cursorvariable verwendet.

OPEN FOR SELECT

```
OPEN ungebundener-cursor FOR SELECT ...;
```

Die Cursorvariable wird geöffnet und ihr wird die angegebene Anfrage zur Ausführung übergeben. Der Cursor darf nicht schon geöffnet sein und er muss als ungebundene Cursorvariable deklariert worden sein (das heißt einfach als refcursor). Die SELECT-Anfrage wird wie andere SELECT-Befehle in PL/pgSQL bearbeitet: PL/pgSQL-Variablennamen werden ersetzt und der Anfrageplan wird für die eventuelle Wiederverwendung gespeichert.

Ein Beispiel:

```
OPEN curs1 FOR SELECT * FROM foo WHERE wert = mein_wert;
```

OPEN FOR EXECUTE

```
OPEN ungebundener-cursor FOR EXECUTE befehlszeichenkette;
```

Die Cursorvariable wird geöffnet und ihr wird die angegebene Anfrage zur Ausführung übergeben. Der Cursor darf nicht schon geöffnet sein und er muss als ungebundene Cursorvariable deklariert worden sein

(das heißt einfach als refcursor). Die Anfrage wird als Zeichenkettenausdruck angegeben, wie bei einem normalen EXECUTE-Befehl. Wie gewohnt gibt das die Flexibilität, die Anfrage von einem Lauf zum nächsten verändern zu können.

Ein Beispiel:

```
OPEN curs1 FOR EXECUTE ''SELECT * FROM '' || quote_ident($1);
```

Einen gebundenen Cursor öffnen

```
OPEN gebundener-cursor [ ( argument_werte ) ];
```

Diese Form von OPEN wird verwendet, um eine Cursorvariable zu öffnen, wo die Anfrage bei der Deklaration an die Variable gebunden wurde. Eine Liste der tatsächlichen Argumentwerte muss angegeben werden, wenn der Cursor so deklariert wurde, dass er Argumente erfordert (und nur dann). Diese Werte werden in die Anfrage eingesetzt. Der Anfrageplan eines gebundenen Cursors wird immer zwischengespeichert; es gibt keine Entsprechung von EXECUTE für diesen Fall.

Beispiele:

```
OPEN curs2;
OPEN curs3(42);
```

38.8.3 Cursor verwenden

Wenn ein Cursor geöffnet wurde, kann er mit den hier beschriebenen Anweisungen manipuliert werden.

Diese Manipulationen müssen nicht in derselben Funktion passieren, die den Cursor geöffnet hat. Sie können einen Wert vom Typ refcursor aus einer Funktion zurückgeben und es demjenigen, der die Funktion aufgerufen hat, überlassen den Cursor zu verwenden. (Intern ist ein refcursor-Wert einfach eine Zeichenkette mit dem Namen eines so genannten Portals, das die aktive Anfrage eines Cursors enthält. Dieser Name kann herumgereicht, anderen refcursor-Variablen zugewiesen werden und so weiter, ohne das Portal zu stören.)

Alle Portale werden am Transaktionsende automatisch geschlossen. Daher ist ein refcursor-Wert als Verweis auf einen offenen Cursor nur bis zum Ende der Transaktion brauchbar.

FETCH

```
FETCH cursor INTO ziel;
```

FETCH liest eine Zeile aus einem Cursor in ein Ziel, welches eine Zeilenvariable, eine Record-Variable oder eine Liste einfacher Variablen, durch Kommas getrennt, sein könnte, genau wie bei SELECT INTO. Wie bei SELECT INTO kann die besondere Variable FOUND geprüft werden, um zu sehen, ob eine Zeile erhalten wurde oder nicht.

Ein Beispiel:

```
FETCH curs1 INTO zeilenvar;
FETCH curs2 INTO foo, bar, baz;
```

CLOSE

```
CLOSE cursor;
```

CLOSE schließt das Portal, das dem offenen Cursor zugrunde liegt. Das kann verwendet werden, um Ressourcen schon vor dem Transaktionsende freizugeben oder um die Cursorvariable frei zu machen, um sie wieder öffnen zu können.

Ein Beispiel:

```
CLOSE curs1;
```

Cursor zurückgeben

PL/pgSQL-Funktionen können Cursor zurückgeben. Damit kann man mehrere Zeilen oder Spalten aus einer Funktion zurückgeben. Um das zu tun, öffnet die Funktion den Cursor und gibt den Cursornamen aus der Funktion zurück. Derjenige, der die Funktion aufgerufen hat, kann dann mit FETCH Zeilen aus dem Cursor lesen. Der erhaltene Cursor kann auch geschlossen werden; wenn nicht, wird er am Ende der Transaktion automatisch geschlossen.

Der von der Funktion zurückgegebene Cursorname kann vom Aufrufer angegeben oder automatisch erzeugt werden. Das folgende Beispiel zeigt, wie der Aufrufer einen Cursornamen angeben kann:

```
CREATE TABLE test (spalte text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT spalte FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

Das folgende Beispiel verwendet die automatische Cursornamenerzeugung:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT spalte FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
```

```
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

38.9 Fehler und Meldungen

Verwenden Sie die Anweisung `RAISE`, um Meldungen und Fehler zu erzeugen.

```
RAISE level 'format' [, variable [, ...]];
```

Mögliche Werte für *level* sind: `DEBUG` (Meldung in den Serverlog schreiben), `LOG` (Meldung mit höherer Priorität in den Serverlog schreiben), `NOTICE` und `WARNING` (Meldung in den Serverlog schreiben und an den Client schicken, mit unterschiedlichen Prioritäten) und `EXCEPTION` (Fehler erzeugen und die aktuelle Transaktion abbrechen). Ob Meldungen einer bestimmten Priorität an den Client geschickt, in den Serverlog geschrieben werden oder beides, wird von den Konfigurationsparametern `server_min_messages` und `client_min_messages` kontrolliert. Weitere Informationen dazu finden Sie in Abschnitt 16.4.

In der Formatzeichenkette wird `%` durch die Zeichenkettendarstellung des nächsten optionalen Arguments ersetzt. Um ein `%` auszugeben, schreiben Sie `%%`. Beachten Sie, dass die optionalen Argumente gegenwärtig einfache Variablen sein müssen, keine Ausdrücke, und dass das Format eine einfache Zeichenkettenkonstante sein muss.

In diesem Beispiel wird das `%` in der Zeichenkette durch den Wert von `v_job_id` ersetzt.

```
RAISE NOTICE ''Rufe cs_create_job(%) auf'', v_job_id;
```

Dieses Beispiel wird die Transaktion mit der angegebenen Fehlermeldung abbrechen.

```
RAISE EXCEPTION ''Nicht existierende ID --> %'', benutzer_id;
```

PostgreSQL hat kein besonders intelligentes Exception-Modell. Wenn der Parser, Planer, Optimierer oder Executor entscheidet, dass ein Befehl nicht weiter verarbeitet werden kann, dann wird die ganze Transaktion abgebrochen und das System springt zurück in die Hauptschleife und wartet auf den nächsten Befehl von der Clientanwendung.

Man kann sich in diesen Fehlerbehandlungsmechanismus einschalten und Fehler abfangen. Aber es ist gegenwärtig unmöglich, festzustellen, was eigentlich den Abbruch verursachte (Datentypformatfehler, Fließkommafehler, Parsefehler usw.). Und es ist möglich, dass sich der Datenbankserver an diesem Punkt in einem inkonsistenten Zustand befindet, sodass die Rückkehr in den Executor oder das Ausführen weiterer Befehle die gesamte Datenbank verfälschen könnte.

Das Einzige, was `PL/pgSQL` gegenwärtig tut, wenn es einen Abbruch während der Ausführung einer Funktion oder einer Triggerprozedur abfängt, ist einige zusätzliche `NOTICE`-Mitteilungen zu erzeugen, die angeben, in welcher Funktion und wo (Zeilennummer und Anweisungstyp) der Fehler passierte. Der Fehler beendet aber immer die Ausführung der Funktion.

38.10 Triggerprozeduren

PL/pgSQL kann verwendet werden, um Triggerprozeduren zu definieren. Eine Triggerprozedur wird mit dem Befehl CREATE FUNCTION als Funktion ohne Argumente und mit Rückgabetyp trigger erzeugt. Beachten Sie, dass die Funktion ohne Argumente definiert werden muss, selbst wenn sie von CREATE TRIGGER angegebene Argumente empfangen soll. Triggerargumente werden durch TG_ARGV übergeben, wie unten beschrieben.

Wenn eine PL/pgSQL-Funktion als Trigger aufgerufen wird, werden mehrere besondere Variablen automatisch in der äußersten Blockebene erzeugt. Diese sind:

- NEW
  Datentyp RECORD; die Variable enthält die neue Tabellenzeile bei INSERT/UPDATE-Operationen bei Triggern auf Zeilenebene.
- OLD
  Datentyp RECORD; die Variable enthält die alte Tabellenzeile bei DELETE/UPDATE-Operationen bei Triggern auf Zeilenebene.
- TG_NAME
  Datentyp name; die Variable enthält den Namen des ausgeführten Triggers.
- TG_WHEN
  Datentyp text; entweder die Zeichenkette BEFORE oder AFTER, abhängig von der Definition des Triggers.
- TG_LEVEL
  Datentyp text; entweder die Zeichenkette ROW oder STATEMENT, abhängig von der Definition des Triggers.
- TG_OP
  Datentyp text; die Zeichenkette INSERT, UPDATE oder DELETE, welche angibt, für welche Operation der Trigger ausgelöst wurde.
- TG_RELID
  Datentyp oid; die OID der Tabelle, die den Trigger ausgelöst hat.
- TG_RELNAME
  Datentyp name; der Name der Tabelle, die den Trigger ausgelöst hat.
- TG_NARGS
  Datentyp integer; die Anzahl der Argumente, die der Triggerprozedur vom Befehl CREATE TRIGGER übergeben wurden.
- TG_ARGV[]
  Datentyp Array aus text; die Argumente vom CREATE TRIGGER-Befehl. Die Zählung fängt bei 0 an. Ungültige Indizes (kleiner als 0 oder größer als oder gleich tg_nargs) ergeben den NULL-Wert.

Eine Triggerfunktion muss entweder den NULL-Wert oder einen Record- oder Zeilentypwert zurückgeben, der genau die Struktur der Tabelle hat, die den Trigger ausgelöst hat. BEFORE-Trigger können den NULL-Wert zurückgeben, um dem Triggermanager anzuzeigen, dass der Rest der Operation für diese Zeile ausgelassen werden soll (das heißt, weitere Trigger werden nicht ausgelöst und das INSERT/UPDATE/DELETE findet in dieser Zeile nicht statt). Wenn nicht der NULL-Wert zurückgegeben wird, geht die Operation weiter. Wenn ein Zeilenwert zurückgegeben wird, der sich vom ursprünglichen Wert von NEW unterscheidet, verändert sich dadurch die Zeile, die eingefügt oder aktualisiert wird. Sie können einzelne Werte direkt in NEW ersetzen und NEW zurückgeben oder Sie können eine ganz neue Zeile aufbauen und zurückgeben.

Der Rückgabewert eines AFTER-Triggers wird nicht beachtet; Sie können also genauso gut den NULL-Wert zurückgeben. Aber ein AFTER-Trigger kann die ganze Operation abbrechen, indem er einen Fehler auslöst.

Beispiel 38.1 zeigt ein Beispiel für eine Triggerprozedur in PL/pgSQL.

Beispiel 38.1: Eine Triggerprozedur in PL/pgSQL

Dieser Beispieltrigger sorgt dafür, dass, wenn eine Zeile in die Tabelle eingefügt oder in ihr aktualisiert wird, der aktuelle Benutzername und die aktuelle Zeit in die Zeile eingefügt werden. Und er sorgt dafür, dass der Name des Mitarbeiters angegeben wird und dass das Gehalt positiv ist.

```
CREATE TABLE mitarbeiter (
    name text,
    gehalt integer,
    letztes_datum timestamp,
    letzter_benutzer text
);

CREATE FUNCTION mitarbeiter_prüfung() RETURNS trigger AS '
    BEGIN
        -- Prüfe ob Name und Gehalt angegeben wurden.
        IF NEW.name IS NULL THEN
            RAISE EXCEPTION ''Name fehlt'';
        END IF;
        IF NEW.gehalt IS NULL THEN
            RAISE EXCEPTION ''Gehalt für % fehlt'', NEW.name;
        END IF;

        -- Wer arbeitet schon, wenn er selbst zahlen muss?
        IF NEW.gehalt < 0 THEN
            RAISE EXCEPTION ''% kann kein negatives Gehalt haben'', NEW.name;
        END IF;

        -- Zeichne Gehaltslistenveränderungen auf
        NEW.letztes_datum := ''now'';
        NEW.letzter_benutzer := current_user;
        RETURN NEW;
    END;
' LANGUAGE plpgsql;

CREATE TRIGGER mitarbeiter_prüfung BEFORE INSERT OR UPDATE ON mitarbeiter
    FOR EACH ROW EXECUTE PROCEDURE mitarbeiter_prüfung();
```

38.11 Portieren von Oracle PL/SQL

Dieser Abschnitt beschreibt die Unterschiede zwischen der Sprache PL/pgSQL von PostgreSQL und der Sprache PL/SQL von Oracle, um den Entwicklern zu helfen, die Anwendungen von Oracle auf PostgreSQL zu portieren.

PL/pgSQL ist in vielerweise ähnlich mit PL/SQL. Es hat eine Blockstruktur, ist imperativ und alle Variablen müssen deklariert werden. Zuweisungen, Schleifen und Auswahlanweisungen sind ähnlich. Die Hauptunterschiede, die Sie sich beim Portieren von PL/SQL auf PL/pgSQL merken sollten, sind:

- Es gibt keine Vorgabewerte für Parameter in PostgreSQL.
- In PostgreSQL können Funktionen überladen werden. Damit kann man das Fehlen von Vorgabewerten für Parameter leicht ausgleichen.
- In PL/pgSQL müssen Sie keine Cursor verwenden, sondern können die Anfrage gleich in die FOR-Anweisung schreiben. (Siehe Beispiel 38.3.)
- In PostgreSQL müssen Sie Apostrophe im Funktionskörper durch Fluchtfolgen ersetzen. Siehe Abschnitt 38.2.
- Anstelle von Packages können Sie Schemas verwenden, um Ihre Funktionen in Gruppen zu organisieren.

38.11.1Portierungsbeispiele

Beispiel 38.2 zeigt, wie man eine einfache Funktion von PL/SQL auf PL/pgSQL portiert.

Beispiel 38.2: Portierung einer einfachen Funktion von PL/SQL auf PL/pgSQL

Hier ist die Oracle-Funktion in PL/SQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name IN varchar,
                                                  v_version IN varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Gehen wir diese Funktion durch und schauen uns die Unterschiede zu PL/pgSQL an:

- PostgreSQL hat keine benannten Parameter. Sie müssen den Parametern in der Funktion ausdrücklich Aliasnamen geben.
- Oracle kann Parameter der Arten IN, OUT und INOUT haben. INOUT bedeutet zum Beispiel, dass der Parameter einen Wert erhält und einen anderen zurückgibt.
- PostgreSQL hat nur IN-Parameter.
- Das Schlüsselwort RETURN im Funktionsprototyp (nicht im Funktionskörper) ist in PostgreSQL RETURNS.
- In PostgreSQL werden Funktionen mit Apostrophen als Begrenzungszeichen des Funktionskörpers definiert, also müssen Sie Apostrophe im Funktionskörper durch Fluchtfolgen ersetzen.
- Der Befehl /show errors existiert in PostgreSQL nicht.

So würde diese Funktion auf PostgreSQL portiert aussehen:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(varchar, varchar)
RETURNS varchar AS '
DECLARE
    v_name ALIAS FOR $1;
    v_version ALIAS FOR $2;
BEGIN
```

```
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || ''/'' || v_version;
END;
' LANGUAGE plpgsql;
```

Beispiel 38.3 zeigt, wie man eine Funktion, die eine weitere Funktion erzeugt, portiert und wie man mit der dabei entstehenden Problematik mit den Apostrophen umgeht.

Beispiel 38.3: Portierung einer Funktion, die eine weitere Funktion erzeugt, von PL/SQL auf PL/pgSQL

Die folgende Prozedur erhält Ergebniszeilen von einem `SELECT`-Befehl und baut eine große Funktion mit den Anfrageergebnissen in `IF`-Anweisungen, welche der Effizienz halber benötigt wird. Beachten Sie insbesondere die Unterschiede beim Cursor und bei der `FOR`-Schleife.

Dies ist die Oracle-Version:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    a_output VARCHAR(4000);
BEGIN
    a_output :=
              'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
                 v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        a_output := a_output || ' IF v_' || referrer_key.kind || ' LIKE ''' ||
referrer_key.key_string || ''' THEN RETURN ''' || referrer_key.referrer_type ||
'''; END IF;';
    END LOOP;

    a_output := a_output || ' RETURN NULL; END;';
    EXECUTE IMMEDIATE a_output;
END;
/
show errors;
```

So würde diese Funktion in der PostgreSQL-Version aussehen:

```
CREATE FUNCTION cs_update_referrer_type_proc() RETURNS integer AS '
DECLARE
    referrer_keys RECORD;  -- für die FOR-Schleife
    a_output varchar(4000);
BEGIN
   a_output := ''CREATE FUNCTION cs_find_referrer_type(varchar, varchar, varchar)
```

```
                  RETURNS varchar AS ''''
                     DECLARE
                         v_host ALIAS FOR $1;
                         v_domain ALIAS FOR $2;
                         v_url ALIAS FOR $3;
                     BEGIN '';

 -- Schleife durch die Ergebnisse einer Anfrage mit der Konstruktion FOR <record>.

    FOR referrer_keys IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
    a_output := a_output || '' IF v_'' || referrer_keys.kind || '' LIKE ''''''''''
                || referrer_keys.key_string || '''''''''' THEN RETURN ''''''
                || referrer_keys.referrer_type || ''''''; END IF;'';
    END LOOP;

    a_output := a_output || '' RETURN NULL; END; '''' LANGUAGE plpgsql;'';

    -- EXECUTE funktioniert, weil wir keine Variablen einsetzen.  Ansonsten
    -- würde es nicht funktionieren.  Eine anderen Möglichkeit um Funktionen
    -- auszuführen ist die Verwendung von PERFORM.

    EXECUTE a_output;
END;
' LANGUAGE plpgsql;
```

Beispiel 38.4 zeigt, wie man eine Funktion mit OUT-Parametern und Zeichenkettenfunktionen portiert. PostgreSQL hat keine Funktion `instr`, aber Sie können die gleichen Ergebnisse mit anderen Funktionen erreichen. In Abschnitt 38.11.3 finden Sie eine Implementierung von `instr` in PL/pgSQL, die Sie verwenden können, um das Portieren zu erleichtern.

Beispiel 38.4: Portierung einer Prozedur mit Zeichenkettenmanipulation und OUT-Parametern von PL/SQL auf PL/pgSQL

Die folgende PL/SQL-Prozedur wird verwendet, um eine URL zu parsen und mehrere Bestandteile davon zurückzugeben (Host, Pfad, Query). PL/pgSQL-Funktionen können nur einen Wert zurückgeben. Eine Möglichkeit, dieses Problem zu lösen, ist, diese Prozedur in drei verschiedene Funktionen aufzuteilen: eine für den Host, eine für den Pfad und eine für die Query.

Das ist die Oracle-Version:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- dies wird zurückgeben
    v_path OUT VARCHAR,  -- ditto
    v_query OUT VARCHAR) -- ditto
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
```

```
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

So könnte die PL/pgSQL-Funktion aussehen, die den Host-Teil zurückgibt:

```
CREATE OR REPLACE FUNCTION cs_parse_url_host(varchar) RETURNS varchar AS '
DECLARE
    v_url ALIAS FOR $1;
    v_host varchar;
    v_path varchar;
    a_pos1 integer;
    a_pos2 integer;
    a_pos3 integer;
BEGIN
    v_host := NULL;
    a_pos1 := instr(v_url, ''//'');

    IF a_pos1 = 0 THEN
        RETURN '''';  -- leerer Rückgabewert
    END IF;
```

```
    a_pos2 := instr(v_url,''/'',a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := ''/'';
        RETURN v_host;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2 );
    RETURN v_host;
END;
' LANGUAGE plpgsql;
```

Beispiel 38.5 zeigt, wie man eine Prozedur portiert, die diverse Oracle-spezifische Funktionalität verwendet.

Beispiel 38.5: Portierung einer Prozedur von PL/SQL auf PL/pgSQL

Die Oracle-Version:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;(1)
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- Sperre aufgeben(3)
        raise_application_error(-20000, 'Kann neuen Job nicht erzeugen:
                                Ein Job läuft schon.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION WHEN dup_val_on_index THEN NULL; -- egal, wenn er schon existiert(4)
    END;
    COMMIT;
END;
/
show errors
```

Prozeduren wie diese können leicht in PostgreSQL-Funktionen umgewandelt werden, die einen Wert vom Typ integer zurückgeben. Diese Prozedur ist besonders interessant, weil Sie uns einige Sachen zeigt:

- **(1)** Es gibt keine Anweisung PRAGMA in PostgreSQL.

- **(2)** Wenn Sie LOCK TABLE in PL/pgSQL verwenden, wird die Sperre erst wieder freigegeben, wenn die aufrufende Transaktion beendet wird.
- **(3)** Sie können keine Transaktionen in PL/pgSQL-Funktionen haben. Die gesamte Funktion (und andere darin aufgerufene Funktionen) werden in einer einzigen Transaktion ausgeführt und PostgreSQL rollt die Transaktion zurück, wenn etwas schiefgeht.
- **(4)** Die Exception müsste durch eine IF-Anweisung ersetzt werden.

So würden wir diese Prozedur auf PL/pgSQL portieren:

```
CREATE OR REPLACE FUNCTION cs_create_job(integer) RETURNS integer AS '
DECLARE
    v_job_id ALIAS FOR $1;
    a_running_job_count integer;
    a_num integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;
    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0
    THEN
        RAISE EXCEPTION ''Kann neuen Job nicht erzeugen: Ein Job läuft schon.'';
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    SELECT count(*) INTO a_num FROM cs_jobs WHERE job_id=v_job_id;
    IF NOT FOUND THEN  -- wenn letzte Anfrage nichts ergab
    -- Der Job ist nicht in der Tabelle, also fügen wir ihn ein.
    INSERT INTO cs_jobs(job_id, start_stamp) VALUES (v_job_id, current_timestamp);
    RETURN 1;
    ELSE
        RAISE NOTICE ''Job läuft schon.'';(1)
    END IF;

    RETURN 0;
END;
' LANGUAGE plpgsql;
```

38.11.2Andere zu beachtende Dinge

Dieser Abschnitt erklärt einige weitere Dinge, die man beachten sollte, wenn man Funktionen von Oracle PL/SQL auf PostgreSQL portiert.

EXECUTE

Die PL/pgSQL-Version von EXECUTE funktioniert ähnlich wie die PL/SQL-Version, aber Sie müssen daran denken, quote_literal(text) und quote_string(text) zu verwenden, wie in Abschnitt 38.6.4

beschrieben. Konstruktionen der Art `EXECUTE ''SELECT * FROM $1'';` werden nicht funktionieren, wenn Sie diese Funktionen nicht verwenden.

Optimierung von PL/pgSQL-Funktionen

PostgreSQL bietet Ihnen zwei Parameter bei der Erzeugung von Funktionen, die die Ausführung optimieren können: die *Volatility* (ob die Funktion immer das gleiche Ergebnis bei gleichen Argumenten zurückgibt) und die "Striktheit" (ob die Funktion den NULL-Wert zurückgibt, wenn irgendein Argument der NULL-Wert ist). Einzelheiten dazu finden Sie in der Beschreibung von `CREATE FUNCTION`.

Wenn Sie diese Optimierungsattribute verwenden, könnte ein `CREATE FUNCTION`-Befehl etwa so aussehen:

```
CREATE FUNCTION foo(...) RETURNS integer AS '
...
' LANGUAGE plpgsql STRICT IMMUTABLE;
```

38.11.3Anhang

Dieser Abschnitt enthält den Code einer Oracle-kompatiblen `instr`-Funktion, den Sie verwenden können, um das Portieren zu erleichtern.

```
--
-- instr Funktion, die Oracle emuliert
-- Syntax: instr(string1, string2, [n], [m]) wobei [] optionale Parameter anzeigt.
--
-- Sucht in string1 ab dem n-ten Zeichen nach dem m-ten Auftreten von
-- string2.  Wenn n negativ ist, sucht es rückwärts.  Wenn m nicht
-- angegeben wird, dann nimmt es 1 an (Suche fängt beim ersten Zeichen
-- an).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS '
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
' LANGUAGE plpgsql;


CREATE FUNCTION instr(varchar, varchar, varchar) RETURNS integer AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
```

```
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
' LANGUAGE plpgsql;


CREATE FUNCTION instr(varchar, varchar, integer, integer) RETURNS integer AS '
DECLARE
    string ALIAS FOR $1;
    string_to_search ALIAS FOR $2;
    beg_index ALIAS FOR $3;
    occur_index ALIAS FOR $4;
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
```

```
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
' LANGUAGE plpgsql;
```

39

PL/Tcl: Tcl prozedurale Sprache

PL/Tcl ist eine ladbare prozedurale Sprache für das PostgreSQL-Datenbanksystem. Mir Ihr kann man Funktionen und Triggerprozeduren in der Sprache Tcl schreiben.

39.1 Überblick

PL/Tcl bietet die meisten Fähigkeiten, die eine C-Funktion bietet, hat aber einige Einschränkungen.

Eine gute Einschränkung ist, dass alles in einem sicheren Tcl-Interpreter ausgeführt wird. Neben dem begrenzten Befehlsschatz von sicherem Tcl gibt es nur ein paar Befehle, um auf die Datenbank mit SPI zuzugreifen und um Fehlermeldungen mit `elog()` zu erzeugen. Es ist nicht möglich, auf die internen Strukturen des Datenbankservers zuzugreifen oder Zugriff auf das Betriebssystem mit der Benutzerkennung des Datenbankservers zu erhalten. Sie können also jedem normalen Benutzer erlauben, diese Sprache zu verwenden.

Die andere Einschränkung ist implementierungsbedingt und bedeutet, dass man Tcl-Funktionen nicht verwenden kann, um Eingabe- und Ausgabefunktionen für neue Datentypen zu schreiben.

Manchmal ist es wünschenswert, Tcl-Funktionen zu schreiben, die nicht auf das sichere Tcl beschränkt sind. Zum Beispiel möchte man vielleicht eine Tcl-Funktion schreiben, die E-Mails verschickt. Für dieses Fälle gibt es eine Variante von PL/Tcl namens PL/TclU (für *untrusted*). Das ist genau die gleiche Sprache, außer dass der volle Tcl-Interpreter verwendet wird. *Wenn PL/TclU verwendet wird, dann muss es als nicht vertrauenswürdige Sprache installiert werden*, damit nur Datenbank-Superuser damit Funktionen erzeugen können. Der Autor einer PL/TclU-Funktion muss darauf achten, dass die Funktion nichts Unbeabsichtigtes tun kann, weil sie mit den Rechten eines Datenbank-Superusers läuft.

Die dynamische Bibliothek mit den Handlern von `PL/Tcl` und `PL/TclU` wird automatisch gebaut und im PostgreSQL-Bibliotheksverzeichnis installiert, wenn Tcl/Tk-Unterstützung im Konfigurationsschritt der Installation ausgewählt wurde. Um PL/Tcl und/oder PL/TclU in einer bestimmten Datenbank zu installieren, verwenden Sie das Programm `createlang`, zum Beispiel `createlang pltcl` *`dbname`* bzw. `createlang pltclu` *`dbname`*.

39.2 PL/Tcl-Funktionen und Argumente

Um eine Funktion in der Sprache PL/Tcl zu schreiben, verwenden Sie die normale Syntax:

```
CREATE FUNCTION name (argumenttypen) RETURNS ergebnistyp AS '
    # PL/Tcl-Funktionskörper
' LANGUAGE pltcl;
```

`PL/TclU` funktioniert genauso, außer dass die Sprache als `pltclu` angegeben werden muss.

Der Körper der Funktion ist einfach ein Stück Tcl-Skript. Wenn die Funktion aufgerufen wird, werden die Argumentwerte dem Tcl-Skript als Variablen `$1` ... `$`*n* übergeben. Das Ergebnis wird aus dem Tcl-Code normal mit dem Befehl `return` zurückgegeben.

Zum Beispiel könnte eine Funktion, die die größere aus zwei ganzen Zahlen zurückgibt, so definiert werden:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS '
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE pltcl STRICT;
```

Beachten Sie die Klausel `STRICT`, die es uns erspart, über Argumente, die den NULL-Wert haben, nachzudenken: Wenn ein NULL-Wert übergeben wird, dann wird die Funktion gar nicht aufgerufen, sondern gibt automatisch den NULL-Wert zurück.

Wenn in einer Funktion, die nicht strikt ist, der Wert eines Arguments der NULL-Wert ist, dann wird die entsprechende Variable `$`*n* eine leere Zeichenkette enthalten. Um zu ermitteln, ob ein bestimmtes Argument der NULL-Wert ist, verwenden Sie die Funktion `argisnull`. Wenn wir zum Beispiel wollen, dass `tcl_max` mit einem NULL- und einem Nicht-NULL-Argument das Nicht-NULL-Argument zurückgibt, dann könnten wir Folgendes schreiben:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS '
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE pltcl;
```

Wie oben gezeigt wird, kann man einen NULL-Wert aus einer PL/Tcl-Funktion zurückgeben, indem man `return_null` aufruft. Das kann immer gemacht werden, egal ob die Funktion strikt ist oder nicht.

Argumente mit zusammengesetzten Typen werden der Funktion als Tcl-Arrays übergeben. Die Elementnamen des Arrays sind die Attributnamen des zusammengesetzten Typs. Wenn ein Attribut in der Zeile den NULL-Wert hat, taucht es nicht in dem Array auf. Hier ist ein Beispiel:

```
CREATE TABLE mitarbeiter (
    name text,
    gehalt integer,
    alter integer
);
```

```
CREATE FUNCTION überbezahlt(mitarbeiter) RETURNS boolean AS '
    if {200000.0 < $1(gehalt)} {
        return "t"
    }
    if {$1(alter) < 30 && 100000.0 < $1(gehalt)} {
        return "t"
    }
    return "f"
' LANGUAGE pltcl;
```

Gegenwärtig kann man keine zusammengesetzten Typen aus einer PL/Tcl-Funktion zurückgeben.

39.3 Datenwerte in PL/Tcl

Die Argumentwerte, die dem Code der PL/Tcl-Funktion übergeben werden, sind einfach die Eingabeargumente in Textform (als wenn sie mit einem `SELECT`-Befehl angezeigt worden wären). Umgekehrt akzeptiert der Befehl `return` jede Zeichenkette, die als Eingabewert für den deklarierten Rückgabetyp der Funktion akzeptabel ist. Ein PL/Tcl-Programm kann Datenwerte also einfach wie normalen Text behandeln.

39.4 Globale Daten in PL/Tcl

Manchmal ist es nützlich, globale Daten zwischen zwei Aufrufen einer Funktion zu speichern oder Daten zwischen zwei verschiedenen Funktionen zu teilen. Das ist ganz einfach zu machen, weil alle PL/Tcl-Funktionen in einer Sitzung denselben sicheren Tcl-Interpreter verwenden. Also steht jede globale Variable in jedem PL/Tcl-Funktionsaufruf zur Verfügung und bleibt bis zum Ende der SQL-Sitzung bestehen. (Beachten Sie, dass PL/TclU-Funktionen globale Daten gleichermaßen teilen, aber da sie in einem anderen Tcl-Interpreter ausgeführt werden, können sie nicht mit PL/Tcl-Funktionen kommunizieren.)

Um zu verhindern, dass PL/Tcl-Funktionen sich einander unbeabsichtigt stören, wird jeder Funktion mit dem Befehl `upvar` jeweils ein globales Array zur Verfügung gestellt. Der globale Name dieser Variablen ist der interne Name der Funktion und der lokale Name ist `GD`. Es wird empfohlen, dass Sie `GD` für private Daten einer Funktion verwenden. Verwenden Sie normale globale Tcl-Variablen nur für Werte, die wirklich von mehreren Funktionen verwendet werden sollen.

Ein Beispiel für die Verwendung von `GD` finden Sie unten im Beispiel für `spi_execp`.

39.5 Datenbankzugriff aus PL/Tcl

Die folgenden Befehle stehen zur Verfügung, um aus dem Körper einer PL/Tcl-Funktion auf die Datenbank zuzugreifen:

- `spi_exec [-count` *`n`*`] [-array` *`name`*`]` *`befehl`* `[`*`schleifenkörper`*`]`

  Führt einen als Zeichenkette angegebenen SQL-Befehl aus. Wenn der Befehl fehlerhaft ist, wird ein Fehler ausgelöst. Ansonsten ist der Rückgabewert von `spi_exec` die Anzahl der von dem Befehl ver-

arbeiteten Zeilen (ausgewählte, eingefügte, aktualisierte oder gelöschte), oder 0, wenn der Befehl ein anderer Befehl war. Wenn der Befehl ein `SELECT` ist, werden außerdem die Werte der ausgewählten Spalten wie unten beschrieben in Tcl-Variablen angelegt.

Der optionale Wert für `-count` sagt `spi_exec`, wie viele Zeilen der Befehl höchstens verarbeiten soll. Die Wirkung ist vergleichbar mit der, wenn man einen Cursor für eine Anfrage einrichtet und dann `FETCH` *`n`* ausführt.

Wenn der Befehl ein `SELECT` ist, werden die Werte der Ergebnisspalten in Tcl-Variablen angelegt, die nach den Spalten benannt sind. Wenn die Option `-array` angegeben ist, werden die Spaltenwerte stattdessen in einem assoziativen Array abgelegt, mit den Spalten als Arrayindizes.

Wenn der Befehl ein `SELECT` ist und *schleifenkörper* nicht angegeben ist, wird nur die erste Zeile des Ergebnisses in Tcl-Variablen abgelegt; die übrigen Zeilen, falls es noch welche gibt, werden ignoriert. Wenn die Anfrage keine Zeilen ergibt, werden keine Werte abgelegt. (Dieser Fall kann durch das Prüfen des Rückgabewerts von `spi_exec` entdeckt werden.) Zum Beispiel:

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

Dadurch wird in der Tcl-Variablen `$cnt` die Anzahl der Zeilen im Systemkatalog `pg_proc` abgelegt.

Wenn das optionale Argument *schleifenkörper* angegeben wurde, ist es ein Stück Tcl-Skript, das einmal für jede Zeile im Anfrageergebnis ausgeführt wird. (*schleifenkörper* wird ignoriert, wenn der angegebene Befehl kein `SELECT` ist.) Die Spaltenwerte der aktuellen Zeile werden vor jedem Durchlauf in Tcl-Variablen gespeichert. Zum Beispiel:

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "Tabelle $C(relname)"
}
```

Dieser Befehl gibt für jede Zeile in `pg_class` eine Logmeldung aus. Die Schleife funktioniert ähnlich wie andere Schleifenanweisungen in Tcl; insbesondere können `continue` und `break` im Schleifenkörper auf die übliche Art verwendet werden.

Wenn eine Spalte im Anfrageergebnis den NULL-Wert hat, wird die Zielvariable gelöscht anstatt gesetzt.

❑ `spi_prepare` *`anfrage typenliste`*

Bereitet einen Anfrageplan vor und speichert ihn für die spätere Ausführung. Der gespeicherte Plan wird für die Dauer der aktuellen Sitzung erhalten.

Die Anfrage kann Parameter verwenden, das heißt Platzhalter für Werte, die angegeben werden, wenn die Anfrage tatsächlich ausgeführt wird. Im Anfragetext können Sie auf die Parameter mit den Symbolen `$1` ... `$`*`n`* verweisen. Wenn die Anfrage Parameter verwendet, müssen die Parametertypen als Tcl-Liste angegeben werden. (Wenn keine Parameter verwendet werden, schreiben Sie für *typenliste* eine leere Liste.) Gegenwärtig müssen die Parametertypen durch die internen Typnamen aus der Systemtabelle `pg_type` angegeben werden; also zum Beispiel `int4` anstatt `integer`.

Der Rückgabewert von `spi_prepare` ist eine Anfragekennung, die dann von `spi_execp` verwendet werden kann. Siehe unter `spi_execp` für ein Beispiel.

❑ `spi_execp [-count` *`n`*`] [-array` *`name`*`] [-nulls` *`zeichenkette`*`]` *`anfragekennung`*
`[`*`werteliste`*`] [`*`schleifenkörper`*`]`

Führt eine Anfrage aus, die zuvor mit `spi_prepare` vorbereitet wurde. *anfragekennung* ist ein von `spi_prepare` zurückgegebener Wert. Wenn die Anfrage Parameter verwendet, muss *werteliste* angegeben werden. Diese Liste enthält die eigentlichen Werte für die Parameter. Die Liste muss die gleiche Länge haben, wie die zuvor an `spi_prepare` übergebene Parameterliste. Wenn die Anfrage keine Parameter hat, sollten Sie *werteliste* weglassen.

Der optionale Wert für `-nulls` ist eine Zeichenkette aus Leerzeichen und `'n'`-Zeichen, die `spi_execp` mitteilt, welche Parameter NULL-Werte sind. Wenn er angegeben ist, muss er die gleiche Länge haben wie *werteliste*. Wenn er nicht angegeben ist, sind alle Parameter nicht-NULL.

Abgesehen von der Art, wie die Anfrage und ihre Parameter angegeben werden, funktioniert `spi_execp` genauso wie `spi_exec`. Die Optionen `-count`, `-array` und *schleifenkörper* sind dieselben und der Rückgabewert funktioniert auch gleich.

Hier ist ein Beispiel einer PL/Tcl-Funktion, die einen vorbereiteten Plan verwendet:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS '
    if {![ info exists GD(plan) ]} {
        # bereite Plan beim ersten Aufruf vor
        set GD(plan) [ spi_prepare \\
          "SELECT count(*) AS cnt FROM t1 WHERE num >= \\$1 AND num <= \\$2" \\
           [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE pltcl;
```

Beachten Sie, dass jeder Backslash, der von Tcl gesehen werden soll, verdoppelt werden muss, wenn die Funktion eingegeben wird, da der Parser beim Lesen des Befehls `CREATE FUNCTION` die Backslashes auch selbst verarbeitet. Innerhalb des an `spi_prepare` übergebenen Anfragetextes benötigen wir Backslashes, damit die $*n*-Werte unverändert an `spi_prepare` übergeben werden und Tcl nicht versucht, Variablen einzusetzen.

❑ `spi_lastoid`

Gibt die OID der letzten von `spi_exec` oder `spi_execp` eingefügten Zeile zurück, wenn der Befehl ein `INSERT` war, das eine einzige Zeile eingefügt hat. (Wenn nicht, wird 0 zurückgegeben.)

❑ `quote` *`zeichenkette`*

Verdoppelt alle Apostrophe und Backslashes in der angegebenen Zeichenkette. Damit können Sie Zeichenketten preparieren, die in SQL-Befehle eingefügt werden, welche an `spi_exec` oder `spi_prepare` übergeben werden sollen. Denken Sie zum Beispiel an eine Befehlszeichenkette wie diese:

```
"SELECT '$val' AS ret"
```

Wenn die Tcl_Variable `val` zufällig `doesn't` enthält, dann ergibt das den Befehlstext

```
SELECT 'doesn't' AS ret
```

welcher in `spi_exec` oder `spi_prepare` einen Parsefehler auslösen würde. Der Befehlstext sollte vielmehr so aussehen:

```
SELECT 'doesn''t' AS ret
```

Das können Sie in PL/Tcl so konstruieren:

```
"SELECT '[ quote $val ]' AS ret"
```

Ein Vorteil von `spi_execp` ist, dass Sie niemals diese Probleme mit den Parameterwerten haben werden, weil die Parameterwerte nicht als Teil des SQL-Befehls gelesen werden.

- `elog` *`level meldung`*

  Erzeugt eine Log- oder Fehlermeldung. Mögliche Werte für *level* sind: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR` und `FATAL`. Die meisten geben ähnlich wie die C-Funktion `elog` einfach die Meldung aus. `ERROR` erzeugt einen Fehler: Die Ausführung der Funktion wird beendet und die aktuelle Transaktion abgebrochen. `FATAL` bricht die aktuelle Transaktion ab und beendet die aktuelle Sitzung. (Es gibt wahrscheinlich keinen guten Grund, davon in einer PL/Tcl-Funktion Gebrauch zu machen, aber es wird der Vollständigkeit halber angeboten.)

39.6 Triggerprozeduren in PL/Tcl

Es ist möglich, Triggerprozeduren in PL/Tcl zu schreiben. PostgreSQL erfordert, dass eine Prozedur, die als Trigger aufgerufen wird, als Funktion ohne Argumente und mit Rückgabetyp `trigger` deklariert wird.

Die Informationen aus dem Triggermanager werden der Prozedur in den folgenden Variablen übergeben:

- `$TG_name`
  Der Name des Triggers aus dem Befehl `CREATE TRIGGER`.
- `$TG_relid`
  Die OID der Tabelle, die den Trigger ausgelöst hat.
- `$TG_relatts`
  Eine Tcl-Liste mit den Spaltennamen der Tabelle, mit einer leeren Liste vorangestellt. Wenn Sie also eine Spalte in der Liste mit dem Tcl-Befehl `lsearch` suchen, fangen die Elementnummern bei der ersten Spalte mit 1 an, genauso wie Spalten üblicherweise in PostgreSQL nummeriert werden.
- `$TG_when`
  Die Zeichenkette `BEFORE` oder `AFTER`, abhängig vom Typ des Triggers.
- `$TG_level`
  Die Zeichenkette `ROW` oder `STATEMENT`, abhängig vom Typ des Triggers.
- `$TG_op`
  Die Zeichenkette `INSERT`, `UPDATE` oder `DELETE`, abhängig vom Typ des Triggers.
- `$NEW`
  Ein assoziatives Array mit den Werten der neuen Tabellenzeile, bei Triggern für `INSERT` oder `UPDATE`, oder leer bei `DELETE`. Die Arrayindizes sind die Spaltennamen. Spalten, die den NULL-Wert haben, erscheinen nicht in dem Array.
- `$OLD`
  Ein assoziatives Array mit den Werten der alten Tabellenzeile, bei Triggern für `UPDATE` oder `DELETE`, oder leer bei `INSERT`. Die Arrayindizes sind die Spaltennamen. Spalten, die den NULL-Wert haben, erscheinen nicht in dem Array.
- `$args`

  Eine Tcl-Liste mit den Argumenten für die Prozedur, welche im `CREATE TRIGGER`-Befehl angegeben wurden. Auf diese Argumente kann man auch als `$1` ... `$`*`n`* zugreifen.

Der Rückgabewert aus einer Triggerprozedur kann eine der Zeichenketten `OK` oder `SKIP` sein, oder eine vom Tcl-Befehl `array get` zurückgegebene Liste. Wenn der Rückgabewert `OK` ist, wird die Operation, die den Trigger ausgelöst hat (`INSERT`/`UPDATE`/`DELETE`) fortgesetzt. Bei `SKIP` wird die Operation bei die-

ser Zeile stillschweigend übersprungen. Wenn eine Liste zurückgegeben wird, gibt PL/Tcl eine modifizierte Zeile an den Triggermanager zurück, die anstelle der in `$NEW` angegebenen eingefügt wird. (Das funktioniert nur bei `INSERT` und `UPDATE`.) Natürlich ist all dies nur von Bedeutung, wenn der Trigger `BEFORE` und `FOR EACH ROW` ist; ansonsten wird der Rückgabewert ignoriert.

Hier ist ein kleines Beispiel für eine Triggerprozedur, die dafür sorgt, dass eine Spalte in einer Tabelle zählt, wie oft eine Zeile aktualisiert worden ist. Bei neu eingefügten Zeilen wird der Wert mit 0 initialisiert und dann bei jeder Aktualisierung erhöht.

```
CREATE FUNCTION trigfunk_modzahl() RETURNS trigger AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE pltcl;

CREATE TABLE meine_tabelle (num integer, beschreibung text, modzahl integer);

CREATE TRIGGER trig_meine_tabelle_modzahl BEFORE INSERT OR UPDATE ON meine_tabelle
    FOR EACH ROW EXECUTE PROCEDURE trigfunk_modzahl('modzahl');
```

Beachten Sie, dass die Triggerprozedur selbst den Spaltennamen nicht kennt; er wird in den Triggerargumenten angegeben. Dadurch kann die Triggerprozedur mit verschiedenen Tabellen wiederverwendet werden.

39.7 Module und der Befehl unknown

PL/Tcl kann Tcl-Code automatisch laden, wenn er verwendet wird. Es erkennt eine besondere Tabelle, `pltcl_modules`, welche Module mit Tcl-Code enthalten muss. Wenn diese Tabelle existiert, wird das Modul `unknown` unmittelbar nach der Initialisierung des Tcl-Interpreters aus der Tabelle geladen.

Obwohl das Modul `unknown` theoretisch ein beliebiges Initialisierungsskript enthalten könnte, definiert es normalerweise eine Tcl-Prozedur mit dem Namen `unknown`, die jedes Mal aufgerufen wird, wenn Tcl einen Prozedurnamen nicht erkennt. Die Standardversion dieser Prozedur in PL/Tcl versucht, in `pltcl_modules` ein Modul zu finden, das die gesuchte Prozedur definiert. Wenn eine passende Prozedur gefunden wird, dann wird sie in den Interpreter geladen und die Ausführung fährt mit dem ursprünglichen Aufruf der Prozedur fort. Die sekundäre Tabelle `pltcl_modfuncs` enthält einen Index darüber, welche Funktionen in welchen Modulen definiert sind, damit die Suche einigermaßen schnell geht.

Die PostgreSQL-Distribution enthält Skripts, die die Verwaltung dieser Tabellen unterstützen: `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod` sowie den Quellcode für die Standardversion des

Moduls `unknown` in `share/unknown.pltcl`. Dieses Modul muss in jede Datenbank geladen werden, um den automatischen Lademechanismus zu unterstützen.

Die Tabellen `pltcl_modules` und `pltcl_modfuncs` müssen von allen lesbar sein, sollten aber wahrscheinlich einem Datenbankadministrator gehören und nur von diesem beschreibbar sein.

39.8 Tcl-Prozedurnamen

In PostgreSQL kann derselbe Funktionsname für verschiedene Funktionen verwendet werden, solange sie eine unterschiedliche Anzahl Argumente oder unterschiedliche Argumenttypen haben. In Tcl müssen jedoch alle Prozedurnamen unterschiedlich sein. Daher fügt PL/Tcl in die Namen der internen Prozeduren die OID der Funktion aus der Systemtabelle `pg_proc` ein. Daher werden PostgreSQL-Funktionen mit dem gleichen Namen und unterschiedlichen Argumenttypen auch verschiedene Tcl-Prozeduren sein. Das ist für PL/Tcl-Programmierer normalerweise nicht von Belang, könnte aber beim Debuggen sichtbar werden.

40

PL/Perl: Perl prozedurale Sprache

PL/Perl ist eine ladbare prozedurale Sprache, mit der Sie PostgreSQL-Funktionen in der Programmiersprache Perl schreiben können.

Um PL/Perl in einer bestimmte Datenbank zu installieren, verwenden Sie `createlang plperl` *dbname*.

Tipp

Wenn eine Sprache in `template1` installiert wird, steht sie in allen danach erzeugten Datenbank automatisch zur Verfügung.

Anmerkung

Benutzer des Quellcodepakets müssen PL/Perl beim Installationsprozess gesondert aktivieren (siehe Installationsanweisungen). Benutzer von Binärpaketen könnten PL/Perl vielleicht in einem getrennten Unterpaket finden.

40.1 PL/Perl-Funktionen und Argumente

Um eine Funktion in der Sprache PL/Perl zu erzeugen, verwenden Sie die Standardsyntax:

```
CREATE FUNCTION funkname (argumenttypen) RETURNS rückgabetyp AS '
    # PL/Perl-Funktionskörper
' LANGUAGE plperl;
```

Der Körper der Funktion ist normaler Perl-Code.

Argumente und Ergebnisse funktionieren wie in jeder anderen Perl-Subroutine: Argumente werden in `@_` übergeben und ein Ergebniswert wird mit `return` zurückgegeben oder ist der letzte ausgewertete Ausdruck in der Funktion.

Zum Beispiel könnte eine Funktion, die die größere aus zwei ganzen Zahlen zurückgibt, so definiert werden:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS '
    if ($_[0] > $_[1]) { return $_[0]; }
```

```
    return $_[1];
' LANGUAGE plperl;
```

Wenn einer Funktion ein SQL-NULL-Wert übergeben wird, erscheint der Argumentwert in Perl als "undefiniert". Die obige Funktionsdefinition kommt mit NULL-Werten als Eingabe nicht besonders gut zurecht (sie behandelt sie wie 0). Wir könnten der Funktionsdefinition `STRICT` hinzufügen, damit PostgreSQL etwas Besseres macht: Wenn ein NULL-Wert übergeben wird, wird die Funktion gar nicht aufgerufen, sondern ein NULL-Wert wird automatisch zurückgegeben. Andererseits könnten wir auch im Funktionskörper prüfen, ob die Eingabewerte undefiniert sind. Wenn wir zum Beispiel wollen, dass `perl_max` mit einem NULL- und einem Nicht-NULL-Argument das Nicht-NULL-Argument zurückgibt, könnten wir Folgendes schreiben:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS '
    my ($a,$b) = @_;
    if (! defined $a) {
        if (! defined $b) { return undef; }
        return $b;
    }
    if (! defined $b) { return $a; }
    if ($a > $b) { return $a; }
    return $b;
' LANGUAGE plperl;
```

Wie oben gezeigt, kann man einen SQL-NULL-Wert aus einer PL/Perl-Funktion zurückgeben, indem man einen undefinierten Wert zurückgibt. Das kann immer gemacht werden, egal, ob die Funktion strikt ist oder nicht.

Argumente mit zusammengesetzten Typen werden der Funktion als Hash-Referenz übergeben. Die Schlüssel des Hash sind die Attributnamen des zusammengesetzten Typs. Hier ist ein Beispiel:

```
CREATE TABLE mitarbeiter (
    name text,
    grundgehalt integer,
    bonus integer
);

CREATE FUNCTION ma_berechnung(mitarbeiter) RETURNS integer AS '
    my ($ma) = @_;
    return $ma->{''grundgehalt''} + $ma->{''bonus''};
' LANGUAGE plperl;

SELECT name, ma_berechnung(mitarbeiter) FROM mitarbeiter;
```

Gegenwärtig kann man keine zusammengesetzten Typen aus einer PL/Perl-Funktion zurückgeben.

Tipp

Weil der Funktionskörper dem Befehl `CREATE FUNCTION` als SQL-Zeichenkettenkonstante übergeben wird, müssen Sie in Ihrem Perl-Code Apostrophe und Backslashes durch Fluchtfolgen ersetzen, normalerweise, indem Sie sie wie oben verdoppeln. Eine andere Möglichkeit ist es, Apostrophe ganz zu vermeiden und die erweiterten Quote-Operatoren von Perl zu verwenden (`q[]`, `qq[]`, `qw[]`).

40.2 Datenwerte in PL/Perl

Die Argumentwerte, die dem Code der PL/Perl-Funktion übergeben werden, sind einfach die Eingabeargumente in Textform (als wenn sie mit einem `SELECT`-Befehl angezeigt worden wären). Umgekehrt akzeptiert der Befehl `return` jede Zeichenkette, die als Eingabewert für den deklarierten Rückgabetyp der Funktion akzeptabel ist. Ein PL/Perl-Programm kann Datenwerte also einfach wie normalen Text behandeln.

40.3 Datenbankzugriff aus PL/Perl

Eine Perl-Funktion kann auf die Datenbank selbst mit einem experimentellen Modul `DBD::PgSPI` (auch auf CPAN-Mirror-Sites) zugreifen. Dieses Modul stellt eine DBI-kompatible Datenbankhandle namens `$pg_dbh` zur Verfügung, mit der man Anfragen mit der normalen DBI-Syntax ausführen kann.

PL/Perl selbst bietet gegenwärtig nur einen einzigen zusätzlichen Perl-Befehl:

```
elog level, meldung
```

Gibt eine Log- oder Fehlermeldung aus. Mögliche Werte für *level* sind `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` und `ERROR`. `ERROR` erzeugt einen Fehler: Die Ausführung der Funktion wird beendet und die aktuelle Transaktion abgebrochen.

40.4 Trusted und Untrusted PL/Perl

Normalerweise wird PL/Perl als "vertrauenswürdige" Programmiersprache (`TRUSTED`) mit dem Namen `plperl` installiert. In diesem Fall sind bestimmte Perl-Operationen abgeschaltet, um die Sicherheit zu gewährleisten. Generell sind jene Operationen eingeschränkt, die auf die Umgebung zugreifen. Das beinhaltet Operationen mit File-Handles, `require` und `use` (mit externen Modulen). Es ist nicht möglich, auf die internen Strukturen des Datenbankserverprozesses zuzugreifen oder Zugriff auf das Betriebssystem mit der Benutzerkennung des Serverprozesses zu erhalten, wie es C-Funktionen können. Sie können also jedem normalen Datenbankbenutzer die Verwendung dieser Sprache erlauben.

Hier ist ein Beispiel einer Funktion, die nicht funktionieren wird, weil Dateisystemoperationen aus Sicherheitsgründen nicht erlaubt sind:

```
CREATE FUNCTION schlecht() RETURNS integer AS '
    open(TEMP, ">/tmp/geheim");
    print TEMP "Erwischt!\n";
    return 1;
' LANGUAGE plperl;
```

Das Erzeugen dieser Funktion wird funktionieren, aber das Ausführen nicht.

Manchmal ist es wünschenswert, eine Perl-Funktion schreiben zu können, die nicht eingeschränkt ist. Zum Beispiel möchte man vielleicht eine Perl-Funktion schreiben, die E-Mails verschickt. Für diese Fälle kann PL/Perl auch als "nicht vertrauenswürdige" Sprache (*untrusted*) installiert werden (üblicherweise `PL/PerlU` genannt). In dem Fall ist die gesamte Perl-Sprache verfügbar. Wenn man die Sprache mit dem Programm `createlang` installiert, wird durch den Sprachnamen `plperlu` die nicht vertrauenswürdige Variante von PL/Perl ausgewählt.

Der Autor einer PL/PerlU-Funktion muss darauf achten, dass die Funktion nichts Unbeabsichtigtes tun kann, weil sie mit den Rechten eines Datenbank-Superusers läuft. Beachten Sie, dass nur Datenbank-Superuser Funktionen in nicht vertrauenswürdigen Sprachen erzeugen können.

Wenn die obige Funktion von einem Superuser mit der Sprache `plperlu` erzeugt worden wäre, würde das Ausführen funktionieren.

40.5 Fehlende Funktionalität

Die folgende Funktionalität fehlt in PL/Perl, aber entsprechende Beiträge wären willkommen.

- PL/Perl-Funktionen können einander nicht direkt aufrufen (weil Sie in Perl anonyme Subroutinen sind). Es gibt auch keine globalen Variablen.
- PL/Perl kann nicht verwendet werden, um Triggerfunktionen zu schreiben.
- `DBD::PgSPI` oder etwas Ähnliches sollte in die PostgreSQL-Distribution integriert werden.

41

PL/Python: Python prozedurale Sprache

Die prozedurale Sprache `PL/Python` ermöglicht, dass PostgreSQL-Funktionen in der Sprache Python geschrieben werden können.

Um PL/Python in einer bestimmten Datenbank zu installieren, verwenden Sie `createlang plpython` *`dbname`*.

Tipp

Wenn eine Sprache in `template1` installiert wird, steht sie in allen danach erzeugten Datenbanken automatisch zur Verfügung.

Anmerkung

Benutzer des Quellcodepakets müssen PL/Python beim Installationsprozess gesondert aktivieren (siehe Installationsanweisungen). Benutzer von Binärpaketen könnten PL/Python vielleicht in einem getrennten Unterpaket finden.

41.1 PL/Python-Funktionen

Der Python-Code, den Sie schreiben, wird in eine Python-Funktion umgewandelt. Zum Beispiel wird

```
CREATE FUNCTION meinefunktion(text) RETURNS text
    AS 'return args[0]'
    LANGUAGE plpython;
```

in

```
def __plpython_procedure_meinefunktion_23456():
        return args[0]
```

umgewandelt, wobei 23456 die OID der Funktion ist.

Wenn Sie keinen Rückgabewert angeben, gibt Python den Wert *None* zurück. Das Sprachmodul wandelt den Python-Wert *None* in den SQL NULL-Wert um.

Die von PostgreSQL erhaltenen Funktionsparameter sind in der globalen Liste `args` verfügbar. Im Beispiel `meinefunktion` wäre etwa `args[0]` das text-Argument. Bei `meinefunktion2(text, integer)` wäre `args[0]` das text-Argument und `args[1]` das integer-Argument.

Das globale Dictionary `SD` kann verwendet werden, um Daten zwischen Funktionsaufrufen zu speichern. Dies ist eine statische, private Variable. Das globale Dictionary `GD` ist eine öffentliche Variable, auf die alle Python-Funktionen in der Sitzung zugreifen können. Seien Sie damit vorsichtig.

Jede Funktion erhält vom Python-Interpreter eine eigene beschränkte Ausführungsumgebung, so dass die globalen Daten und die Funktionsargumente von `meinefunktion` nicht in `meinefunktion2` sichtbar sind. Die Ausnahme sind die Daten im Dictionary `GD`, wie oben erwähnt.

41.2 Triggerfunktionen

Wenn eine Funktion in einem Trigger verwendet wird, sind im Dictionary `TD` diverse Triggerdaten enthalten. Die Triggerzeilen sind in `TD["new"]` und/oder `TD["old"]`, abhängig vom Trigger-Ereignis. `TD["event"]` enthält das Trigger-Ereignis als Zeichenkette (`INSERT`, `UPDATE`, `DELETE` oder `UNKNOWN`). `TD["when"]` enthält entweder `BEFORE`, `AFTER` oder `UNKNOWN`. `TD["level"]` enthält entweder `ROW`, `STATEMENT` oder `UNKNOWN`. `TD["name"]` enthält den Triggernamen und `TD["relid"]` enthält die OID der Tabelle, die den Trigger ausgelöst hat. Wenn der Trigger mit Argumenten aufgerufen wurde, stehen die Argumente in `TD["args"][0]` bis `TD["args"][(n-1)]`.

Wenn `TD["when"]` gleich `BEFORE` ist, können Sie `None` oder `"OK"` aus der Python-Funktion zurückgeben, um anzuzeigen, dass die Zeile unverändert ist, oder `"SKIP"`, um die Operation abzubrechen, oder `"MODIFY"`, um anzuzeigen, dass Sie die Zeile verändert haben.

41.3 Datenbankzugriff

Das PL/Python-Sprachmodul importiert automatisch ein Python-Modul namens `plpy`. Auf die Funktionen und Konstanten in diesem Modul können Sie im Python-Code über `plpy.`*`foo`* zugreifen. Gegenwärtig bietet `plpy` die Funktionen `plpy.debug("meldung")`, `plpy.log("meldung")`, `plpy.info("meldung")`, `plpy.notice("meldung")`, `plpy.warning("meldung")`, `plpy.error("meldung")` und `plpy.fatal("meldung")`. Diese entsprechen im Großen und Ganzen `elog(`*`LEVEL`*`, "meldung")` in C-Code. `plpy.error` und `plpy.fatal` erzeugen in Wirklichkeit eine Python-Exception, welche, wenn sie nicht abgefangen wird, dazu führt, dass das PL/Python-Modul `elog(ERROR, meldung)` aufruft, nachdem der Python-Interpreter verlassen wird. Das ist wahrscheinlich besser, als direkt aus dem Python-Interpreter herauszuspringen. `raise plpy.ERROR("meldung")` und `raise plpy.FATAL("meldung")` sind gleichbedeutend mit Aufrufen von `plpy.error` bzw. `plpy.fatal`.

Außerdem stellt das Modul `plpy` zwei Funktionen zur Verfügung mit den Namen `execute` und `prepare`. Wenn Sie `plpy.execute` mit einer Anfrage als Argument und wahlweise einer Zeilenhöchstzahl als zweites Argument aufrufen, wird die Anfrage ausgeführt und die Ergebnisse in einem Ergebnisobjekt zurückgegeben. Das Ergebnisobjekt emuliert eine Liste oder ein Dictionary. Auf das Ergebnisobjekt kann man mit Zeilennummer und Spaltenname zugreifen. Zusätzlich hat es diese Methoden: `nrows` gibt die Zahl der von der Anfrage gelieferten Zeilen zurück, und `status` ist die Ergebniswert von `SPI_exec()`. Das Ergebnisobjekt kann modifiziert werden.

Ein Beispiel:

```
rv = plpy.execute("SELECT * FROM meine_tabelle", 5)
```

Dieser Befehl liefert 5 Zeilen aus `meine_tabelle`. Wenn `meine_tabelle` eine Spalte namens `meine_spalte` hat, dann kann man darauf mit

```
foo = rv[i]["meine_spalte"]
```

zugreifen.

Die zweite Funktion, `plpy.prepare`, bereitet den Ausführungsplan für eine Anfrage vor. Sie wird mit einer Anfrage und einer Liste von Datentypen, wenn die Anfrage Parameterverweise enthält, aufgerufen. Zum Beispiel:

```
plan = plpy.prepare("SELECT nachname FROM benutzer WHERE vorname = $1", [ "text" ])
```

`text` ist der Typ der Variable, die Sie für `$1` übergeben werden. Nachdem Sie einen Befehl vorbereitet haben, können Sie ihn mit der Funktion `plpy.execute` ausführen:

```
rv = plpy.execute(plan, [ "name" ], 5)
```

Das dritte Argument gibt die Höchstzahl der zurückgegebenen Zeilen an und ist optional.

In der aktuellen Version führt jeder Datenbankfehler bei der Ausführung einer PL/Python-Funktion zum sofortigen Abbruch der Funktion durch den Server; es ist nicht möglich, Fehler durch die Python-Konstruktion `try ... catch` abzufangen. Ein Syntaxfehler in einem SQL-Befehl, der `plpy.execute` übergeben wurde, würde zum Beispiel die Funktion abbrechen. Dieses Verhalten wird vielleicht in einer zukünftigen Version geändert werden.

Wenn Sie mit dem PL/Python-Modul einen Plan vorbereiten, wird er automatisch gespeichert. Lesen Sie die SPI-Dokumentation (Kapitel 34) als Erklärung dazu. Um daraus über mehrere Funktionsaufrufe hinweg effektiven Nutzen zu ziehen, müssen Sie das globale Dictionary `SD` oder `GD` verwenden (siehe Abschnitt 41.1). Zum Beispiel:

```
CREATE FUNCTION planverwendung() RETURNS trigger AS '
    if SD.has_key("plan"):
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # Rest der Funktion
' LANGUAGE plpython;
```

41.4 Eingeschränkte Umgebung

Die aktuelle Version von `PL/Python` fungiert nur als vertrauenswürdige Sprache (`TRUSTED`); Zugriff auf das Dateisystem und andere lokale Ressourcen ist abgeschaltet. Um genau zu sein, verwendet PL/Python die eingeschränkte Ausführungsumgebung von Python, schränkt diese weiter ein, um den Aufruf von `open` mit Dateien zu verhindern, und erlaubt das Importieren nur von Modulen aus einer bestimmten Liste. Diese Liste ist gegenwärtig: `array`, `bisect`, `binascii`, `calendar`, `cmath`, `codecs`, `errno`, `marshal`, `math`, `md5`, `mpz`, `operator`, `pcre`, `pickle`, `random`, `re`, `regex`, `sre`, `sha`, `string`, `StringIO`, `struct`, `time`, `whrandom` und `zlib`.

Teil VI

Referenz

Die Einträge in diesem Referenzteil stellen eine vollständige, definitive und formale Zusammenfassung ihrer jeweiligen Themen dar. Weitere Informationen über die Verwendung von PostgreSQL in erzählerischer Form, als Tutorial oder mit Beispielen finden Sie in anderen Teilen dieses Buchs.

Die Referenzeinträge gibt es auch im traditionellen „man“-Seiten-Format.

I. SQL-Befehle

Dieser Teil enthält Referenzinformationen über die von PostgreSQL unterstützten SQL-Befehle. Mit „SQL“ meinen wir die Sprache im allgemeinen Sinne; inwiefern die Befehle mit dem SQL-Standard konform oder mit anderen Systemen kompatibel sind, steht auf der jeweiligen Referenzseite.

Inhaltsverzeichnis

II. PostgreSQL-Clientanwendungen

Dieser Teil enthält Referenzinformationen über PostgreSQL-Clientanwendungen und Hilfsprogramme. Nicht alle dieser Programme sind für alle Anwender gedacht; für einige von ihnen benötigen Sie besondere Privilegien. Das gemeinsame Merkmal dieser Anwendungen ist es, dass sie von jedem Host aus aufgerufen werden können, unabhängig davon, wo der Datenbankserver ist. `clusterdb` *(Seite: 786)* – clustert eine PostgreSQL-Datenbank

III. PostgreSQL-Serveranwendungen

Dieser Teil enthält Referenzinformationen über PostgreSQL-Serveranwendungen und Hilfsprogramme. Diese Programme sind nur auf dem Host, wo der Datenbankserver ist, sinnvoll. Andere Hilfsprogramme sind in `Referenz II` *(Seite: 786)*, `PostgreSQL-Clientanwendungen` aufgelistet.

I. SQL-Befehle

Dieser Teil enthält Referenzinformationen über die von PostgreSQL unterstützten SQL-Befehle. Mit „SQL“ meinen wir die Sprache im allgemeinen Sinne; inwiefern die Befehle mit dem SQL-Standard konform oder mit anderen Systemen kompatibel sind, steht auf der jeweiligen Referenzseite.

ABORT

Name

ABORT – bricht die aktuelle Transaktion ab

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Beschreibung

`ABORT` rollt die aktuelle Transaktion zurück und sorgt dafür, dass alle von der Transaktion getätigten Änderungen verworfen werden. Dieser Befehl ist identisch mit dem Befehl `ROLLBACK` aus dem SQL-Standard und ist nur aus historischen Gründen vorhanden.

Parameter

`WORK`
`TRANSACTION`

Optionale Schlüsselwörter ohne jegliche Auswirkung.

Meldungen

`ROLLBACK`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`WARNING: ROLLBACK: no transaction in progress`

Meldung, wenn gegenwärtig keine Transaktion aktiv ist.

Beispiele

Um alle Änderungen abzubrechen:

```
ABORT;
```

Kompatibilität

Dieser *Befehl* ist eine PostgreSQL-Erweiterung, die nur aus historischen Gründen vorhanden ist. Der entsprechende Befehl im SQL-Standard ist ROLLBACK.

Siehe auch

COMMIT *(Seite: 645)*, ROLLBACK *(Seite: 758)*

ALTER DATABASE

Name

ALTER DATABASE – ändert eine Datenbank

Synopsis

```
ALTER DATABASE name SET parameter { TO | = } { wert | DEFAULT }
ALTER DATABASE name RESET parameter
```

Beschreibung

ALTER DATABASE wird verwendet, um die Sitzungsvorgabewerte eines Konfigurationsparameters für eine PostgreSQL-Datenbank einzustellen. Wenn danach eine neue Sitzung in dieser Datenbank gestartet wird, dann wird der angegebene Wert der Vorgabewert für die Sitzung. Die datenbankspezifischen Vorgabewerte haben Vorrang vor Werten, die in postgresql.conf oder auf der Kommandozeile des postmaster-Befehls angegeben wurden.

Nur der Eigentümer der Datenbank oder ein Superuser können die Sitzungsvorgabewerte einer Datenbank ändern.

Parameter

name

Der Name der Datenbank, für die die Sitzungsvorgabewerte geändert werden sollen.

parameter
wert

Setzt den Sitzungsvorgabewert des angegebenen Konfigurationsparameters in dieser Datenbank auf den angegebenen Wert. Wenn *wert* DEFAULT ist oder RESET verwendet wird, wird die datenbankspezifische Parametereinstellung entfernt und in neuen Sitzungen wird wieder der systemweite Vorgabewert verwendet. Mit RESET ALL können alle Einstellungen entfernt werden.

Unter SET *(Seite: 771)* und in Abschnitt SETfinden Sie weitere Informationen über die erlaubten Parameternamen und Werte.

Meldungen

`ALTER DATABASE`

Meldung, wenn die Änderung erfolgreich war.

`ERROR: database "dbname" does not exist`

Fehlermeldung, wenn die angegebene Datenbank nicht existiert.

Hinweise

Mit `ALTER USER` *(Seite: 634)* kann man außerdem einen Sitzungsvorgabewert an einen bestimmten Benutzer statt an eine Datenbank knüpfen. Wenn es einen Konflikt gibt, haben die benutzerspezifischen Einstellungen Vorrang vor den datenbankspezifischen.

Beispiele

Um in der Datenbank `test` Indexscans in der Voreinstellung abzuschalten:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Kompatibilität

Der Befehl `ALTER DATABASE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`ALTER USER` *(Seite: 634)*, `CREATE DATABASE` *(Seite: 660)*, `DROP DATABASE` *(Seite: 713)*, `SET` *(Seite: 771)*

ALTER GROUP

Name

ALTER GROUP – fügt einer Gruppe Benutzer hinzu oder entfernt Benutzer aus einer Gruppe

Synopsis

```
ALTER GROUP gruppenname ADD USER benutzername [, ... ]
ALTER GROUP gruppenname DROP USER benutzername [, ... ]
```

Beschreibung

`ALTER GROUP` wird verwendet, um einer Gruppe Benutzer hinzuzufügen oder Benutzer aus einer Gruppe zu entfernen. Nur Datenbank-Superuser können diesen Befehl verwenden. Wenn man einen Benutzer

einer Gruppe hinzufügt, wird dadurch der Benutzer selbst nicht erzeugt. Wenn man einen Benutzer aus einer Gruppe entfernt, wird der Benutzer dadurch auch nicht gelöscht.

Parameter

gruppenname

Der Name der zu modifizierenden Gruppe.

benutzername

Die Benutzer, die zur Gruppe hinzugefügt oder aus ihr entfernt werden sollen. Die Benutzer müssen schon existieren.

Meldungen

`ALTER GROUP`

Meldung, wenn die Änderung erfolgreich war.

Beispiele

Um Benutzer zu einer Gruppe hinzuzufügen:

```
ALTER GROUP mitarbeiter ADD USER karl, johann;
```

Um Benutzer aus einer Gruppe zu entfernen:

```
ALTER GROUP arbeiter DROP USER betina;
```

Kompatibilität

Der Befehl `ALTER GROUP` ist eine PostgreSQL-Erweiterung. Das Konzept der Rollen im SQL-Standard ist vergleichbar mit Gruppen.

Siehe auch

`CREATE GROUP` *(Seite: 667)*, `DROP GROUP` *(Seite: 717)*

ALTER TABLE

Name

ALTER TABLE -- ändert die Definition einer Tabelle

Synopsis

```
ALTER TABLE [ ONLY ] tabelle [ * ]
    ADD [ COLUMN ] spalte typ [ spalten_constraint [ ... ] ]
ALTER TABLE [ ONLY ] tabelle [ * ]
    DROP [ COLUMN ] spalte [ RESTRICT | CASCADE ]
ALTER TABLE [ ONLY ] tabelle [ * ]
    ALTER [ COLUMN ] spalte { SET DEFAULT wert | DROP DEFAULT }
ALTER TABLE [ ONLY ] tabelle [ * ]
    ALTER [ COLUMN ] spalte { SET | DROP } NOT NULL
ALTER TABLE [ ONLY ] tabelle [ * ]
    ALTER [ COLUMN ] spalte SET STATISTICS ganze_zahl
ALTER TABLE [ ONLY ] tabelle [ * ]
    ALTER [ COLUMN ] spalte SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ALTER TABLE [ ONLY ] tabelle [ * ]
    RENAME [ COLUMN ] spalte TO neue_spalte
ALTER TABLE tabelle
    RENAME TO neue_tabelle
ALTER TABLE [ ONLY ] tabelle [ * ]
    ADD tabellen_constraint
ALTER TABLE [ ONLY ] tabelle [ * ]
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER TABLE tabelle
    OWNER TO neuer_eigentümer
```

Beschreibung

`ALTER TABLE` ändert die Definition einer bestehenden Tabelle. Es gibt mehrere Unterformen:

`ADD COLUMN`

Diese Form fügt eine neue Spalte zur Tabelle hinzu. Die Syntax ist dieselbe wie bei `CREATE TABLE` *(Seite: 686)*.

`DROP COLUMN`

Diese Form löscht eine Spalte aus einer Tabelle. Indexe und Tabellen-Constraints, die die Spalte verwenden, werden automatisch auch gelöscht. Wenn etwas außerhalb der Tabelle von der Spalte abhängt, wie zum Beispiel Fremdschlüssel oder Sichten, dann müssen Sie `CASCADE` angeben.

`SET/DROP DEFAULT`

Diese Form stellt den Vorgabewert einer Spalte ein oder löscht einen Vorgabewert. Die Vorgabewerte werden erst bei nachfolgenden `INSERT`-Befehlen angewendet; die Zeilen, die schon in der Tabelle sind, werden nicht geändert. Vorgabewerte können auch für Sichten gesetzt werden, was dazu führt, dass sie in `INSERT`-Befehle für die Sicht eingesetzt werden, bevor die `ON INSERT`-Regel der Sicht angewendet wird.

`SET/DROP NOT NULL`

Diese Formen ändern, ob die Spalte NULL-Werte akzeptiert oder ablehnt. `SET NOT NULL` können Sie nur verwenden, wenn die Spalte keine NULL-Werte enthält.

`SET STATISTICS`

Diese Form setzt das spaltenspezifische Statistikziel für nachfolgende ANALYZE-Operationen *(Seite: 636)*. Der Wert kann zwischen 0 und 1000 sein oder -1, um die Standardeinstellung zu verwenden.

SET STORAGE

Diese Form setzt den Speichermodus einer Spalte. Dieser kontrolliert, ob die Spalte direkt in der Tabelle gespeichert oder in einer Nebentabelle angelegt werden soll und ob die Daten komprimiert werden sollen. PLAIN muss für Werte mit fester Länge, wie integer, verwendet werden und speichert die Werte in der Haupttabelle und unkomprimiert. MAIN speichert die Werte in der Haupttabelle, aber eventuell komprimiert. EXTERNAL ist für externe, unkomprimierte Daten und EXTENDED ist für externe, komprimierte Daten. EXTENDED ist die Voreinstellung für alle Datentypen, die diesen Modus unterstützen. Wenn Sie EXTERNAL verwenden, dann wird zum Beispiel das Abfragen von Teilzeichenketten aus text-Spalten schneller, aber die Speicheranforderungen erhöhen sich.

RENAME

Die RENAME-Formen ändern den Namen einer Tabelle (oder eines Index, einer Sequenz, einer Sicht) oder den Namen einer einzelnen Spalte in der Tabelle. Sie haben keine Auswirkungen auf die gespeicherten Daten.

ADD *tabellen_constraint*

Diese Form erzeugt einen neuen Constraint in der Tabelle, mit der gleichen Syntax wie bei CREATE TABLE *(Seite: 686)*.

DROP CONSTRAINT

Diese Form löscht Constraints aus einer Tabelle. Gegenwärtig müssen Constraints keine eindeutigen Namen haben, daher könnten mehrere Constraints mit dem angegebenen Namen übereinstimmen. Alle diese Constraints würden dann gelöscht werden.

OWNER

Diese Form ändert den Eigentümer der Tabelle, des Index, der Sequenz oder der Sicht auf den angegebenen Benutzer.

Sie müssen der Eigentümer der Tabelle sein, um ALTER TABLE verwenden zu können, außer für ALTER TABLE OWNER, was nur von Superusern ausgeführt werden kann.

Parameter

tabelle

Der Name der zu ändernden Tabelle (möglicherweise mit Schemaqualifikation). Wenn ONLY angegeben wird, dann wird nur diese Tabelle geändert. Wenn ONLY nicht angegeben wird, werden die Tabelle und alle Kindtabellen (falls vorhanden) geändert. * kann an den Tabellennamen angehängt werden, um anzugeben, dass Kindtabellen geändert werden sollen, aber in der aktuellen Version ist das das Standardverhalten. (In Versionen vor 7.1 war ONLY das Standardverhalten. Das Standardverhalten kann mit dem Konfigurationsparameter sql_inheritance eingestellt werden.)

spalte

Name der neuen oder einer bestehenden Spalte.

typ

Datentyp der neuen Spalte.

neue_spalte

Neuer Name für eine vorhandene Spalte.

neue_tabelle

Neuer Name für die Tabelle.

tabellen_constraint

Neuer Tabellen-Constraint für die Tabelle.

constraint_name

Name eines bestehenden Constraints, der gelöscht werden soll.

neuer_eigentümer

Der Benutzername des neuen Eigentümers der Tabelle.

CASCADE

Löscht automatisch alle Objekte, die von der gelöschten Spalte bzw. dem gelöschten Constraint abhängen (zum Beispiel Sichten, die die Spalte verwenden).

RESTRICT

Weigert sich, die Spalte bzw. den Constraint zu löschen, wenn Objekte davon abhängen. Das ist das Standardverhalten.

Meldungen

ALTER TABLE

Meldung bei erfolgreichem Abschluss der Operation.

ERROR

Meldung, wenn die Spalte oder die Tabelle nicht existiert.

Hinweise

Das Schlüsselwort COLUMN ist nur Dekoration und kann weggelassen werden.

In der gegenwärtigen Implementierung von ADD COLUMN werden die Klauseln DEFAULT und NOT NULL nicht unterstützt. Die neue Spalte hat immer den NULL-Wert in allen Zeilen. Mit der SET DEFAULT-Form von ALTER TABLE können Sie nachher den Vorgabewert einstellen. (Dann sollten Sie vielleicht auch mit UPDATE *(Seite: 782)*den neuen Vorgabewert in die bestehenden Zeilen einsetzen.) Wenn Sie in der Spalte keine NULL-Werte haben wollen, verwenden Sie die Form SET NOT NULL, nachdem Sie in allen Zeilen für die Spalte einen richtigen Wert eingefügt haben.

Die DROP COLUMN-Form löscht die Spalte nicht wirklich, sondern macht sie zunächst unsichtbar. Nachfolgende Einfüge- und Aktualisierungsoperationen in der Tabelle werden in der Spalte einen NULL-Wert speichern. Das Löschen einer Spalte ist daher schnell, verringert aber den Speicherplatzbedarf der Tabelle nicht sofort, weil der Speicher der gelöschten Spalte nicht freigegeben wird. Der Platz wird im Verlauf der Zeit beim Aktualisieren der bestehenden Zeilen wiedergewonnen. Um den Platz sofort wiederzugewinnen, führen Sie erst ein UPDATE aus, das alle Zeilen der Tabelle berührt, und dann VACUUM. Zum Beispiel:

```
UPDATE tabelle SET spalte = spalte;
VACUUM FULL tabelle;
```

Wenn eine Tabelle Kindtabellen hat, ist es nicht erlaubt, Spalten in der Elterntabelle hinzuzufügen oder umzubenennen, ohne das Gleiche auch in den Kindtabellen zu tun. Das heißt, ALTER TABLE ONLY wird nicht erlaubt. Das stellt sicher, dass alle Kindtabellen immer die gleichen Spalten wie die Elterntabelle hat.

Eine rekursive DROP COLUMN-Operation entfernt die Spalte einer Kindtabelle nur, wenn die Kindtabelle diese Spalte nicht von anderen Elterntabellen geerbt hat und nur, wenn die Spalte in der Kindtabelle niemals unabhängig definiert worden ist. Ein nicht rekursives DROP COLUMN (d.h. ALTER TABLE ONLY ... DROP COLUMN) löscht niemals Spalten aus Kindtabellen, sondern markiert sie als unabhängig definiert anstatt geerbt.

Änderungen an irgendwelchen Teilen einer Systemkatalogtabelle sind nicht erlaubt.

Weitere Informationen über gültige Parameterwerte finden Sie in der Beschreibung von `CREATE TABLE`.

Beispiel

Um einer Tabelle eine Spalte vom Typ `varchar` hinzuzufügen:

```
ALTER TABLE händler ADD COLUMN adresse varchar(30);
```

Um eine Spalte aus einer Tabelle zu löschen:

```
ALTER TABLE händler DROP COLUMN adresse RESTRICT;
```

Um eine bestehende Spalte umzubenennen:

```
ALTER TABLE händler RENAME COLUMN adresse TO stadt;
```

Um eine bestehende Tabelle umzubenennen:

```
ALTER TABLE händler RENAME TO lieferanten;
```

Einer Spalte einen NOT-NULL-Constraint hinzuzufügen:

```
ALTER TABLE händler ALTER COLUMN straße SET NOT NULL;
```

Um einen NOT-NULL-Constraint von einer Spalte zu entfernen:

```
ALTER TABLE händler ALTER COLUMN straße DROP NOT NULL;
```

Um einer Tabelle einen Check-Constraint hinzuzufügen:

```
ALTER TABLE händler ADD CONSTRAINT plzchk CHECK (char_length(plz) = 5);
```

Um einen Check-Constraint aus einer Tabelle und allen Kindtabellen zu entfernen:

```
ALTER TABLE händler DROP CONSTRAINT plzchk;
```

Um einer Tabelle einen Fremdschlüssel hinzuzufügen:

```
ALTER TABLE händler ADD CONSTRAINT adrfk FOREIGN KEY (adresse) REFERENCES adressen
(adresse) MATCH FULL;
```

Um einer Tabelle einen (mehrspaltigen) Unique Constraint hinzuzufügen:

```
ALTER TABLE händler ADD CONSTRAINT hdl_id_plz_schlüssel UNIQUE (hdl_id, plz);
```

Um einer Tabelle einen automatisch benannten Primärschlüssel-Constraint hinzuzufügen, wobei beachtet werden muss, dass eine Tabelle immer nur einen Primärschlüssel haben kann:

```
ALTER TABLE händler ADD PRIMARY KEY (hdl_id);
```

Kompatibilität

Die Form `ADD COLUMN` ist mit dem SQL-Standard konform, mit der Ausnahme, dass sie, wie oben beschrieben, keine Vorgabewerte und NOT-NULL-Constraints unterstützt. Die Form `ALTER COLUMN` ist voll konform.

Die Klauseln, um Tabellen, Spalten, Indexe, Sequenzen und Sichten umzubenennen, sind PostgreSQL-Erweiterungen.

ALTER TRIGGER

Name

ALTER TRIGGER – ändert die Definition eines Triggers

Synopsis

```
ALTER TRIGGER trigger ON tabelle RENAME TO neuer_name
```

Beschreibung

`ALTER TRIGGER` ändert die Eigenschaften eines bestehenden Triggers. Die `RENAME`-Klausel ändert den Namen des angegebenen Triggers, ohne die Definition des Triggers selbst zu ändern.

Um die Eigenschaften eines Triggers ändern zu können, müssen Sie der Eigentümer der Tabelle sein, zu der der Trigger gehört.

Parameter

trigger

Der Name des zu ändernden Triggers.

tabelle

Der Name der Tabelle, zu der der Trigger gehört.

neuer_name

Der neue Name für den Trigger.

Meldungen

`ALTER TRIGGER`

Meldung, wenn die Änderung erfolgreich war.

Beispiele

Um einen bestehenden Trigger umzubenennen:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Kompatibilität

Der Befehl `ALTER TRIGGER` ist eine PostgreSQL-Erweiterung.

ALTER USER

Name

ALTER USER – ändert ein Datenbankbenutzerkonto

Synopsis

```
ALTER USER benutzername [ [ WITH ] option [ ... ] ]

wobei option Folgendes sein kann:
    [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'passwort'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | VALID UNTIL 'abstime'

ALTER USER benutzername SET parameter { TO | = } { value | DEFAULT }
ALTER USER benutzername RESET parameter
```

Beschreibung

`ALTER USER` ändert die Attribute eines PostgreSQL-Benutzerkontos. Attribute, die nicht im Befehl erwähnt werden, behalten ihre bisherigen Einstellungen.

Die erste Variante dieses Befehls in der Synopsis ändert bestimmte globale Benutzerprivilegien und Authentifizierungseinstellungen. (Einzelheiten siehe unten.) Nur Datenbank-Superuser können diese Privilegien und die Ablaufzeit des Passworts mit diesem Befehl ändern. Einfache Benutzer können ihr eigenes Passwort ändern.

Die zweite und dritte Variante ändern die Sitzungsvorgabewerte eines Benutzers für den angegebenen Konfigurationsparameter. Wenn der Benutzer danach eine neue Sitzung startet, wird der angegebene Wert der Vorgabewert für die Sitzung. Die benutzerspezifischen Vorgabewerte haben Vorrang vor Werten, die in `postgresql.conf` oder auf der Kommandozeile des `postmaster`-Befehls angegeben wurden. Einfache Benutzer können ihre eigenen Sitzungsvorgabewerte ändern. Superuser können die Sitzungsvorgabewerte eines jeden beliebigen Benutzers ändern.

Parameter

benutzername

Der Name des Benutzers, dessen Attribute geändert werden sollen.

passwort

Das neue Passwort für dieses Benutzerkonto.

ENCRYPTED
UNENCRYPTED

Diese Schlüsselwörter kontrollieren, ob das Passwort in pg_shadow verschlüsselt gespeichert ist. (Siehe unter CREATE USER *(Seite: 702)* für weitere Informationen über diese Option.)

CREATEDB
NOCREATEDB

Diese Klauseln bestimmen, ob ein Benutzer Datenbanken erzeugen darf. Wenn CREATEDB angegeben ist, wird dem Benutzer gestattet, seine eigenen Datenbanken zu erzeugen. Mit NOCREATEDB wird dem Benutzer die Fähigkeit, Datenbanken zu erzeugen, entzogen.

CREATEUSER
NOCREATEUSER

Diese Klauseln bestimmen, ob der Benutzer selbst neue Benutzer erzeugen darf. Diese Option macht den Benutzer auch zum Superuser, der alle Zugriffsbeschränkungen umgehen kann.

abstime

Das Datum (und wahlweise die Zeit), wann das Passwort des Benutzers abläuft.

parameter
wert

Setzt den Sitzungsvorgabewert des angegebenen Konfigurationsparameters für diesen Benutzer auf den angegebenen Wert. Wenn *wert* DEFAULT ist oder RESET verwendet wird, wird die benutzerspezifische Parametereinstellung entfernt und der Benutzer verwendet in neuen Sitzungen wieder den systemweiten Vorgabewert. Mit RESET ALL können alle Einstellungen entfernt werden.

Unter SET *(Seite: 771)* und in Abschnitt SETfinden Sie weitere Informationen über die erlaubten Parameternamen und Werte.

Meldungen

ALTER USER

Meldung, wenn die Änderung erfolgreich war.

ERROR: ALTER USER: user "username" does not exist

Fehlermeldung, wenn der angegebene Benutzer dem Datenbanksystem nicht bekannt ist.

Hinweise

ALTER USER kann nicht verwendet werden, um die Gruppenmitgliedschaft eines Benutzers zu ändern. Verwenden Sie dazu ALTER GROUP *(Seite: 627)*.

Mit ALTER DATABASE *(Seite: 626)* kann man außerdem einen Sitzungsvorgabewert an eine bestimmte Datenbank anstatt an einen Benutzer knüpfen.

Beispiele

Das Passwort eines Benutzers ändern:

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

Das Ablaufdatum eines Passworts ändern:

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Das Ablaufdatum eines Passworts ändern, und zwar, dass es am 4. Mai 2005 mittags, in der Zeitzone, die UTC eine Stunde voraus ist, ablaufen soll:

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 2005 +1';
```

Einem Benutzer die Fähigkeit, andere Benutzer und neue Datenbanken zu erzeugen, geben:

```
ALTER USER miriam CREATEUSER CREATEDB;
```

Kompatibilität

Der Befehl `ALTER USER` ist eine PostgreSQL-Erweiterung. Der SQL-Standard überlässt die Definition der Benutzer der Implementierung.

Siehe auch

`CREATE USER` *(Seite: 702)*, `DROP USER` *(Seite: 730)*, `SET` *(Seite: 771)*

ANALYZE

Name

ANALYZE – sammelt Statistiken über eine Datenbank

Synopsis

```
ANALYZE [ VERBOSE ] [ tabelle [ (spalte [, ...] ) ] ]
```

Beschreibung

`ANALYZE` sammelt Statistiken über den Inhalt von Tabellen in der Datenbank und speichert die Ergebnisse in der Systemtabelle `pg_statistic`. In der Folge verwendet der Anfrageplaner die Statistiken, um bei der Bestimmung des effizientesten Ausführungsplans für Anfragen zu helfen.

Ohne Parameter untersucht `ANALYZE` jede Tabelle in der aktuellen Datenbank. Mit Parameter untersucht `ANALYZE` nur die angegebene Tabelle. Es ist außerdem möglich, eine Liste von Spaltennamen anzugeben, wodurch dann nur Statistiken für jene Spalten gesammelt werden.

Parameter

VERBOSE

Zeigt Fortschrittsmeldungen an.

tabelle

Der Name (möglicherweise mit Schemaqualifikation) einer bestimmten Tabelle, die analysiert werden soll. Ohne diese Angabe werden alle Tabellen in der Datenbank analysiert.

spalte

Der Name einer bestimmten Spalte, die analysiert werden soll. Ohne diese Angabe werden alle Spalten analysiert.

Meldungen

ANALYZE

Der Befehl ist fertig.

Hinweise

Es ist zu empfehlen, ANALYZE in regelmäßigen Abständen oder direkt nach größeren Änderungen im Tabelleninhalt auszuführen. Genaue Statistiken helfen dem Planer bei der Wahl des besten Anfrageplans, wodurch sich die Ausführungszeit der Anfragen verbessert. Ein gebräuchliches Vorgehen ist, VACUUM und ANALYZE *(Seite: 783)* einmal am Tag, zu einer Zeit mit wenig Betrieb, laufen zu lassen.

Im Gegensatz zu VACUUM FULL benötigt ANALYZE nur eine Lesesperre für die Zieltabelle, kann also gleichzeitig mit anderen Aktivitäten in der Tabelle stattfinden.

Bei großen Tabellen verwendet ANALYZE eine zufällige Auswahl von Zeilen aus der Tabelle, anstatt jede einzelne Zeile zu untersuchen. Dadurch können selbst große Tabellen in kurzer Zeit analysiert werden. Beachten Sie jedoch, dass die Statistiken dann nur Näherungswerte sind und sich bei jedem Durchlauf von ANALYZE geringfügig ändern werden, selbst wenn sich der Inhalt der Tabelle nicht geändert hat. Daraus können sich kleine Veränderungen in den von EXPLAIN angezeigten geschätzten Plankosten ergeben.

Die gesammelten Statistiken enthalten meistens eine Liste der häufigsten Werte in jeder Spalte und ein Histogramm mit der ungefähren Datenverteilung in jeder Spalte. Eine oder beide Informationen können weggelassen werden, wenn ANALYZE sie als uninteressant einschätzt (zum Beispiel gibt es in einer Spalte mit Primärschlüssel keine häufigsten Werte) oder der Datentyp der Spalte die passenden Operatoren nicht unterstützt. Weitere Informationen über die Statistiken gibt es in Abschnitt SET.

Der Umfang der Analyse kann mit dem Konfigurationsparameter default_statistics_target eingestellt werden, oder einzeln für jede Spalte mit ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (siehe ALTER TABLE *(Seite: 628)*). Dieser Zielwert bestimmt die maximale Anzahl von Einträgen in der Liste der häufigsten Werte und die Anzahl der „Balken“ im Histogramm. Der Standardzielwert ist 10, aber er kann hoch oder herunter gesetzt werden, wodurch die Genauigkeit des Planers gegen die von ANALYZE benötigte Zeit und den von pg_statistic benötigten Platz eingetauscht würde. Insbesondere wird, wenn man den Parameter auf null setzt, das Sammeln von Statistiken für diese Zeile abgeschaltet. Das könnte bei Spalten sinnvoll sein, die niemals in WHERE-, GROUP BY- oder ORDER BY-Klauseln einer Anfrage auftreten werden, weil der Planer in dem Fall keine Verwendung für Statistiken über diese Spalten hat.

Das größte Statistikziel unter den zu analysierenden Spalten bestimmt die Anzahl der für die Aufstellung der Statistiken zufällig ausgewählten Spalten. Bei der Erhöhung des Zielwerts erhöht sich die von ANALYZE benötigte Zeit und der Speicherbedarf proportional.

Kompatibilität

Der Befehl ANALYZE ist eine PostgreSQL-Erweiterung.

BEGIN

Name

BEGIN – startet einen Transaktionsblock

Synopsis

```
BEGIN [ WORK | TRANSACTION ]
```

Beschreibung

In der Standardeinstellung führt PostgreSQL Transaktionen im „Autocommit“-Modus aus, das heißt, jeder Befehl wird in einer eigenen Transaktion ausgeführt und diese Transaktion wird am Ende des Befehls automatisch abgeschlossen (wenn die Ausführung erfolgreich war, ansonsten wird die Transaktion zurückgerollt). BEGIN startet einen Transaktionsblock, das heißt, dass alle Befehle nach dem BEGIN-Befehl zusammen in einer Transaktion ausgeführt werden, bis ein ausdrückliches COMMIT *(Seite: 645)* oder ROLLBACK *(Seite: 758)* kommt. In Transaktionsblöcken werden Befehle schneller ausgeführt, weil der Start und der Abschluss einer Transaktion erhebliche Prozessorleistung und Festplattenaktivität fordert. Das Ausführen von mehreren Befehlen in einer Transaktion ist auch nützlich, um die Konsistenz zu erhalten, wenn mehrere in Verbindung stehende Tabellen geändert werden, denn andere Sitzungen werden dann die Zwischenzustände, in denen noch nicht alle zusammenhängenden Änderungen getätigt worden sind, nicht sehen können.

Parameter

WORK
TRANSACTION

Optionale Schlüsselwörter ohne jegliche Auswirkung.

Meldungen

BEGIN

Meldung, wenn eine neue Transaktion gestartet wurde.

WARNING: BEGIN: already a transaction in progress

Meldung, wenn bereits eine Transaktion aktiv ist. Das hat aber keine Auswirkung auf die aktuelle Transaktion.

Hinweise

Wenn der Konfigurationsparameter `autocommit` aus ist, ist `BEGIN` nicht nötig: Jeder SQL-Befehl startet dann automatisch eine Transaktion.

Beispiele

Um einen Transaktionsblock zu beginnen:

```
BEGIN;
```

Kompatibilität

Der Befehl `BEGIN` ist eine PostgreSQL-Erweiterung. Im SQL-Standard gibt es keinen `BEGIN`-Befehl; eine Transaktion wird immer implizit gestartet und vom Befehl `COMMIT` oder `ROLLBACK` beendet.

Einige andere Datenbanksysteme bieten auch einen Autocommit-Modus an.

Zufällig wird das Schlüsselwort `BEGIN` in eingebettetem SQL für einen anderen Zweck verwendet. Beachten Sie bei der Portierung von Datenbankanwendungen immer die Transaktionsmodi.

Siehe auch

`COMMIT` *(Seite: 645)*, `ROLLBACK` *(Seite: 758)*, `START TRANSACTION` *(Seite: 779)*

CHECKPOINT

Name

CHECKPOINT – erzwingt einen Checkpoint im Transaktionslog

Synopsis

```
CHECKPOINT
```

Beschreibung

Das WAL-System (Write-Ahead Logging) schreibt von Zeit zu Zeit einen Checkpoint-Eintrag in den Transaktionslog. (Um das automatische Checkpoint-Interval einzustellen, verwenden Sie die Konfigurationsparameter `checkpoint_segments` und `checkpoint_timeout`.) Der Befehl `CHECKPOINT` erzwingt, dass sofort, wenn der Befehl ausgeführt wird, ein Checkpoint durchgeführt wird.

Ein Checkpoint ist ein Punkt in der Transaktionslogfolge, an dem alle Datendateien mit den im Log enthaltenen Daten aktualisiert worden sind. Außerdem werden alle Datendateien auf die Festplatte zurückgeschrieben. Weitere Informationen über das WAL-System finden Sie in Abschnitt SET.

Nur Superuser können `CHECKPOINT` aufrufen. Der Befehl ist nur für die Verwendung in Ausnahmefällen gedacht.

Kompatibilität

Der Befehl `CHECKPOINT` ist eine PostgreSQL-Erweiterung.

CLOSE

Name

CLOSE – schließt einen Cursor

Synopsis

```
CLOSE cursor
```

Beschreibung

`CLOSE` gibt die zu einem offenen Cursor gehörenden Ressourcen frei. Nachdem ein Cursor geschlossen ist, kann er nicht mehr verwendet werden. Ein Cursor sollte geschlossen werden, wenn er nicht mehr benötigt wird.

Jeder offene Cursor wird automatisch geschlossen, wenn eine Transaktion von `COMMIT` oder `ROLLBACK` beendet wird.

Parameter

cursor

Der Name eines offenen Cursors, der geschlossen werden soll.

Meldungen

`CLOSE CURSOR`

Meldung, wenn der Cursor erfolgreich geschlossen wurde.

`WARNING: PerformPortalClose: portal "`*cursor*`" not found`

Diese Warnung wird ausgegeben, wenn der angegebene Cursor nicht deklariert ist oder schon geschlossen wurde.

Hinweise

PostgreSQL hat keinen Befehl `OPEN` zum ausdrücklichen Öffnen eines Cursors; ein Cursor ist geöffnet, sobald er deklariert wurde. Mit dem Befehl `DECLARE` wird ein Cursor deklariert.

Beispiele

Schließe den Cursor `liahona`:

```
CLOSE liahona;
```

Kompatibilität

Der Befehl `CLOSE` ist mit dem SQL-Standard konform.

CLUSTER

Name

CLUSTER – clustert eine Tabelle nach einem Index

Synopsis

```
CLUSTER indexname ON tabellenname
```

Beschreibung

Mit `CLUSTER` wird die in *`tabellenname`* angegebene Tabelle anhand des Index *`indexname`* geclustert. Der Index muss bereits existieren.

Wenn eine Tabelle geclustert wird, wird sie physikalisch anhand des Index umsortiert. Clustern ist eine einmalige Operation: Wenn die Tabelle später aktualisiert wird, werden die Änderungen nicht geclustert. Das heißt, es wird nicht versucht, die neuen oder aktualisierten Zeilen in der Indexreihenfolge einzuordnen. Wenn man es wünscht, kann man die Tabelle regelmäßig neu clustern, indem man den Befehl erneut ausführt.

Parameter

`indexname`

Der Name eines Index.

`tabellenname`

Der Name einer Tabelle (möglicherweise mit Schemaqualifikation).

Meldungen

`CLUSTER`

Die Operation wurde erfolgreich durchgeführt.

Hinweise

Wenn Sie in einer Tabelle nur auf einzelne, zufällig angeordnete Zeilen zugreifen, ist die tatsächliche Reihenfolge der Daten in der Tabelle unwichtig. Wenn Sie aber bestimmte Daten häufiger als andere verwenden und es einen Index gibt, der sie nebeneinander gruppiert, ist `CLUSTER` nützlich. Wenn Sie einen Bereich von indizierten Werten aus einer Tabelle auslesen oder ein indizierter Wert mehrere übereinstimmende Zeilen hat, hilft `CLUSTER`, denn wenn der Index die erste Heap-Seite der ersten übereinstimmenden Zeile gefunden hat, sind alle anderen übereinstimmenden Zeilen wahrscheinlich schon auf derselben Heap-Seite, und dadurch sparen Sie Festplattenzugriffe und beschleunigen die Anfrage.

Während des Cluster-Vorgangs wird eine temporäre Kopie der Tabelle gemacht, welche die Tabellendaten in der Indexreihenfolge enthält. Außerdem werden temporäre Kopien von jedem Index der Tabelle gemacht. Sie müssen daher mindestens so viel Platz frei haben, wie die Tabelle und die Indexe zusammen groß sind.

Weil der Planer Statistiken über die Sortierung von Tabellen speichert, ist es zu empfehlen, für jede neu geclusterte Tabelle `ANALYZE` auszuführen. Ansonsten könnte der Planer ungeeignete Anfragepläne auswählen.

Es gibt auch eine anderen Möglichkeit, um Daten zu clustern. Mit dem Befehl `CLUSTER` wird anhand des angegebenen Index sortiert. Das kann bei großen Tabellen langsam sein, weil die Zeilen in der Indexreihenfolge aus dem Heap gelesen werden. Und wenn die Heap-Tabelle unsortiert ist, sind die Einträge in zufällig verteilten Seiten, und für jede zu verschiebende Zeile muss eine Diskseite gelesen werden. (PostgreSQL hat einen Cache, aber der Großteil einer großen Tabelle wird nicht in den Cache passen.) Die andere Möglichkeit, um eine Tabelle zu clustern, ist

```
CREATE TABLE neue_tabelle AS
    SELECT spaltenliste FROM tabelle ORDER BY spaltenliste;
```

was durch die `ORDER BY`-Klausel die Sortierroutinen von PostgreSQL verwendet; das ist normalerweise viel schneller als ein Indexscan durch unsortierte Daten. Danach löschen Sie die Tabelle, verwenden `ALTER TABLE ... RENAME`, um *`neue_tabelle`* in den alten Namen umzubenennen und erzeugen die Indexe der Tabelle neu. Bei diesem Vorgang gehen aber alle OIDs, Constraints, Fremdschlüsselverbindungen, Privilegien und andere Nebeneigenschaften der Tabelle verloren. All diese Sachen müssten Sie dann selbst neu erzeugen.

Beispiele

Clustere die Tabelle `mitarbeiter` anhand des Index `mitarbeiter_ind`:

```
CLUSTER mitarbeiter_ind ON mitarbeiter;
```

Kompatibilität

Der Befehl `CLUSTER` ist eine PostgreSQL-Erweiterung.

COMMENT

Name

COMMENT – definiert oder ändert den Kommentar von einem Objekt

Synopsis

```
COMMENT ON
{
  TABLE objektname |
  COLUMN tabellenname.spaltenname |
  AGGREGATE aggname (aggtyp) |
  CONSTRAINT constraintname ON tabellenname |
  DATABASE objektname |
  DOMAIN objektname |
  FUNCTION funktionsname (arg1_typ, arg2_typ, ...) |
  INDEX objektname |
  OPERATOR op (linker_operandentyp, rechter_operandentyp) |
  RULE regelname ON tabellenname |
  SCHEMA objektname |
  SEQUENCE objektname |
  TRIGGER triggername ON tabellename |
  TYPE objektname |
  VIEW objektname
} IS 'text'
```

Beschreibung

`COMMENT` speichert einen Kommentar über ein Datenbankobjekt. Kommentare können leicht mit den `psql`-Befehlen `\dd`, `\d+` und `\l+` eingesehen werden. Andere Programme können Kommentare mit denselben eingebauten Funktionen abrufen, die auch `psql` verwendet, nämlich `obj_description` und `col_description`.

Um einen Kommentar zu ändern, führen Sie einen neuen `COMMENT`-Befehl für dasselbe Objekt aus. Pro Objekt wird nur ein Kommentar gespeichert. Um einen Kommentar zu entfernen, schreiben Sie `NULL` anstelle des Textes. Kommentare werden automatisch gelöscht, wenn das Objekt gelöscht wird.

Parameter

objektname
tabellenname.spaltenname
aggname
constraintname
funktionsname
op

regelname
triggername

Der Name des Objekts, für den der Kommentar ist. Die Namen von Tabellen, Aggregatfunktionen, Domänen, Funktionen, Indexen, Operatoren, Sequenzen, Typen und Sichten können eine Schemaqualifikation haben.

text

Der Kommentar.

Meldungen

`COMMENT`

Meldung, wenn der Kommentar erfolgreich geändert wurde.

Hinweise

Es gibt gegenwärtig keinen Sicherheitsmechanismus für Kommentare: Jeder mit einer Datenbank verbundene Benutzer kann alle Kommentare für Objekte in dieser Datenbank sehen (aber nur Superuser können Kommentare von Objekten, die sie nicht besitzen, ändern). Lassen Sie also keine sicherheitsrelevanten Informationen in den Kommentaren.

Beispiele

Erzeuge einen Kommentar für die Tabelle `meine_tabelle`:

```
COMMENT ON TABLE meine_tabelle IS 'Das ist meine Tabelle.';
```

Lösche ihn wieder:

```
COMMENT ON TABLE meine_tabelle IS NULL;
```

Einige weitere Beispiele:

```
COMMENT ON AGGREGATE meine_aggregatfunktion (double precision) IS 'Berechnet
Standardabweichung';
COMMENT ON COLUMN meine_tabelle.meine_spalte IS 'Angestelltennummer';
COMMENT ON DATABASE meine_datenbank IS 'Entwicklungsdatenbank';
COMMENT ON DOMAIN meine_domäne IS 'Domäne für Email-Adressen';
COMMENT ON FUNCTION meine_funktion (timestamp) IS 'Gibt römische Zahl zurück';
COMMENT ON INDEX mein_index IS 'Sorgt für die Eindeutigkeit der
Angestelltennummer';
COMMENT ON OPERATOR ^ (text, text) IS 'Schnittmenge zwischen zwei Texten';
COMMENT ON OPERATOR ^ (NONE, text) IS 'Ein Präfixoperator für Text';
COMMENT ON RULE meine_regel ON meine_table IS 'Zeichnet Aktualisierungen von
Angestelltendaten auf';
COMMENT ON SCHEMA mein_schema IS 'Abteilungsdaten';
COMMENT ON SEQUENCE meine_sequenz IS 'Wird zur Erzeugung von Primärschlüsseln
verwendet';
COMMENT ON TABLE mein_schema.meine_tabelle IS 'Angestellteninformationen';
COMMENT ON TRIGGER mein_trigger ON meine_table IS 'Für referentielle Integrität';
```

```
COMMENT ON TYPE complex IS 'Datentyp für komplexe Zahlen';
COMMENT ON VIEW meine_sicht IS 'Sicht der Abteilungskosten';
```

Kompatibilität

Der Befehl `COMMENT` ist eine PostgreSQL-Erweiterung.

COMMIT

Name

COMMIT – schließt die aktuelle Transaktion ab

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Beschreibung

`COMMIT` schließt die aktuelle Transaktion ab. Alle von der Transaktion getätigten Änderungen werden für andere sichtbar und sind garantiert dauerhaft, falls es zu einem Absturz kommt.

Parameter

`WORK`
`TRANSACTION`

Optionale Schlüsselwörter ohne jegliche Auswirkung.

Meldungen

`COMMIT`

Meldung, wenn die Transaktion erfolgreich abgeschlossen wurde.

`WARNING: COMMIT: no transaction in progress`

Meldung, wenn keine Transaktion offen ist.

Beispiele

Um die aktuelle Transaktion abzuschließen und alle Änderungen dauerhaft zu speichern:

```
COMMIT;
```

Kompatibilität

Der SQL-Standard bestimmt nur die zwei Formen `COMMIT` und `COMMIT WORK`. Ansonsten ist dieser Befehl voll konform.

Siehe auch

`BEGIN` *(Seite: 638)*, `ROLLBACK` *(Seite: 758)*, `START TRANSACTION` *(Seite: 779)*

COPY

Name

COPY – kopiert Daten zwischen Dateien und Tabellen

Synopsis

```
COPY tabelle [ ( spalte [, ...] ) ]
    FROM { 'dateiname' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'trennzeichen' ]
          [ NULL [ AS ] 'null_darstellung' ] ]

COPY tabelle [ ( spalte [, ...] ) ]
    TO { 'dateiname' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'trennzeichen' ]
          [ NULL [ AS ] 'null_darstellung' ] ]
```

Beschreibung

`COPY` kopiert Daten zwischen PostgreSQL-Tabellen und normalen Dateien im Dateisystem. `COPY TO` kopiert den Inhalt einer Tabelle *`in`* eine Datei, `COPY FROM` kopiert Daten *`aus`* einer Datei in eine Tabelle (wo die Daten angehängt werden, wenn schon Daten in der Tabelle sind).

Wenn eine Spaltenliste angegeben ist, kopiert `COPY` nur die Daten in den angegebenen Spalten in oder aus der Datei. Wenn es Spalten in der Tabelle gibt, die nicht in der Spaltenliste stehen, wird `COPY FROM` in diese Spalten die Vorgabewerte einfügen.

Bei COPY mit Dateiname liest oder schreibt der PostgreSQL-Server direkt in einer Datei. Diese Datei muss vom Server aus zugänglich sein und der Dateiname muss vom Gesichtspunkt des Servers aus angegeben sein. Wenn STDIN oder STDOUT angegeben wurden, fließen die Daten über den Client an den Server.

Parameter

tabelle

Der Name einer bestehenden Tabelle (möglicherweise mit Schemaqualifikation).

spalte

Eine optionale Liste mit den zu kopierenden Spalten. Wenn keine Spaltenliste angegeben ist, werden alle Spalten verwendet.

dateiname

Der absolute Dateiname der Eingabe- oder Ausgabedatei.

STDIN

Gibt an, dass die Eingabe von der Clientanwendung kommt.

STDOUT

Gibt an, dass die Ausgabe an die Clientanwendung geht.

BINARY

Schreibt bzw. liest alle Daten im binären Format anstatt im Textformat. Die Optionen DELIMITER und NULL können im binären Modus nicht verwendet werden.

OIDS

Gibt an, dass die OID von jeder Zeile kopiert werden soll. (Es ist ein Fehler, OIDS zu verwenden, wenn die Tabelle keine OIDs hat.)

trennzeichen

Das einzelne Zeichen, das die Spalten in einer Zeile trennt. Der Standardwert ist das Tab-Zeichen.

null_darstellung

Die Zeichenkette, mit der ein NULL-Wert dargestellt werden soll. Der Standardwert ist \N (Backslash-N). Eine leere Zeichenkette wäre zum Beispiel auch eine sinnvolle Wahl.

Anmerkung

Bei COPY FROM werden alle Datenwerte, die mit dieser Zeichenkette übereinstimmen, als NULL-Wert abgespeichert. Sie sollten also darauf achten, dass Sie denselben Wert verwenden, den Sie bei COPY TO verwendet hatten.

Meldungen

COPY

Die Kopieroperation wurde erfolgreich abgeschlossen.

Hinweise

COPY kann nur mit echten Tabellen und keinen Sichten verwendet werden.

Mit dem Schlüsselwort BINARY werden alle Daten im binären Format anstatt im Textformat gelesen bzw. geschrieben. Das ist etwas schneller als der normale Textmodus, aber Dateien im binären Format sind nicht über verschiedene Maschinenarchitekturen portierbar.

Für Tabellen, die von COPY TO gelesen werden, müssen Sie das Privileg SELECT haben; für Tabellen, in die von COPY FROM geschrieben wird, müssen Sie das Privileg INSERT haben.

Im COPY-Befehl genannte Dateien werden direkt vom Server gelesen bzw. geschrieben, nicht von der Client-Anwendung. Deshalb müssen Sie auf der Maschine des Datenbankservers, nicht der des Clients, liegen oder von ihr aus zugänglich sein. Außerdem müssen Sie vom PostgreSQL-Benutzerzugang (der Benutzer, unter dem der Server läuft) lesbar bzw. schreibbar sein. COPY mit einer Datei ist nur für Datenbank-Superuser erlaubt, da es das Lesen bzw. Schreiben aller Dateien, auf die der Server Zugriff hat, erlaubt.

Verwechseln Sie COPY nicht mit der psql-Anweisung \copy. \copy führt COPY FROM STDIN oder COPY TO STDOUT aus und liest bzw. schreibt die Daten dann in einer von psql zugänglichen Datei. Die verfügbaren Dateien und die Zugriffsrechte hängen also in diesem Fall vom Client und nicht vom Server ab.

Es wird empfohlen, dass der in COPY verwendete Dateiname immer ein absoluter Pfad ist. Bei COPY TO wird das vom Server auch durchgesetzt, aber bei COPY FROM können Sie auch einen relativen Pfad angeben. Der Pfad wird relativ zum Arbeitsverzeichnis des Serverprozesses interpretiert (irgendwo unter dem Datenverzeichnis), nicht relativ zum Arbeitsverzeichnis des Clients.

COPY FROM führt Trigger für die Zieltabelle aus und prüft Constraints. Regeln werden jedoch nicht angewendet.

COPY wird beim ersten Fehler abgebrochen. Das sollte bei COPY TO kein Problem sein, aber bei COPY FROM werden in der Zieltabelle schon einige Zeilen enthalten sein. Diese Zeilen sind nicht sichtbar und man kann auf sie auch nicht zugreifen, aber sie verbrauchen trotzdem Speicherplatz. Wenn der Fehler erst spät in einer großen Kopieroperation auftrat, kann dadurch eine ziemliche Menge Platz verschwendet werden. Dann können Sie VACUUM ausführen, um den Platz wieder freizugeben.

Dateiformate

Textformat

Wenn COPY ohne die Option BINARY verwendet wird, werden die Daten als Textdatei mit einer Textzeile pro Datenbankzeile gelesen bzw. geschrieben. Die Spalten einer Zeile werden durch das Trennzeichen getrennt. Die Spaltenwerte selbst sind Zeichenketten, die von der Ausgabefunktion jedes Datentyps erzeugt worden sind bzw. die von der Eingabefunktion verarbeitet werden können. Die angegebene Zeichenkette zur NULL-Darstellung wird für Spaltenwerte, die NULL sind, verwendet. COPY FROM erzeugt einen Fehler, wenn irgendeine Zeile der Eingabedatei mehr oder weniger Zeilen als erwartet enthält. Wenn OIDS angegeben wurde, wird die OID als erste Spalte, vor den Benutzerdatenspalten, gelesen bzw. geschrieben.

Das Ende der Daten kann durch eine Zeile angezeigt werden, die nur Backslash-Punkt (\.) enthält. Die Endmarkierung wird beim Lesen aus einer Datei nicht benötigt, weil das Ende der Datei ebenso gut das Datenende anzeigt; aber eine Endmarkierung muss angegeben werden, wenn Daten vom oder zum Client kopiert werden.

Mit Backslash-Zeichen (\) können in den COPY-Daten Zeichen gesichert werden, die ansonsten als Spalten- oder Zeilentrennzeichen verstanden würden. Insbesondere *muss* vor die folgenden Zeichen ein Backslash gesetzt werden, wenn Sie in einem Spaltenwert auftreten: der Backslash selbst, Newline und das aktuelle Trennzeichen.

Die folgenden besonderen Backslash-Folgen werden von `COPY FROM` erkannt:

Zeichenfolge	Stellt dar
`\b`	Backspace (ASCII 8)
`\f`	Form-Feed (ASCII 12)
`\n`	Newline (ASCII 10)
`\r`	Carriage-Return (ASCII 13)
`\t`	Tab (ASCII 9)
`\v`	Vertikaler Tab (ASCII 11)
`\`*ziffern*	Backslash gefolgt von drei oktalen Ziffern entspricht dem Zeichen mit jenem numerischen Code

`COPY TO` gibt die Backslash-Oktalzahl-Folge gegenwärtig nicht selber aus, aber es verwendet die anderen oben gelisteten Folgen für diese Kontrollzeichen.

Schreiben Sie niemals einen Backslash vor das Datenzeichen `N` oder einen Punkt (`.`). Solche Paare werden als NULL-Darstellung (in der Standardeinstellung) bzw. als Datenendmarkierung missverstanden werden. Jedes andere Zeichen nach einem Backslash, das nicht in der obigen Tabelle erwähnt ist, steht für sich selbst.

Es ist sehr zu empfehlen, dass Anwendungen, die Daten für `COPY` erzeugen, Newline- und Carriage-Return-Zeichen in `\n` bzw. `\r` umwandeln. Gegenwärtig ist es möglich, das Carriage-Return-Zeichen ohne Fluchtfolge und das Newline-Zeichen als Backslash gefolgt von Newline darzustellen, aber diese Schreibweisen werden in der Zukunft nicht mehr akzeptiert werden.

Beachten Sie, dass das Ende einer Zeile im Unix-Stil markiert wird („`\n`“). `COPY FROM` kann zur Zeit noch keine Zeilenenden im DOS- oder Mac-Stil verarbeiten. Das wird wohl in einer zukünftigen Version geändert werden.

Binäres Format

Das von `COPY BINARY` verwendete Format wurde in PostgreSQL 7.1 geändert. Das neue Format besteht aus einem Dateikopf, null oder mehr Tupeln mit den Zeilendaten, und einem Dateiabschluss.

Dateikopf

Der Dateikopf besteht aus 24 Bytes mit festen Feldern, gefolgt von einem Erweiterungsbereich mit variabler Länge. Die festen Felder sind:

Signatur

12-Byte-Folge `PGBCOPY\n\377\r\n\0`. Beachten Sie, dass das Null-Byte unbedingt dazu gehört. (Diese Signatur wurde so entworfen, dass Dateien, die durch nicht 8-Bit-saubere Datenübertragung verändert worden sind, leicht erkannt werden können. Diese Signatur wird verändert durch Filter, die die Zeilenenden umschreiben, fallen gelassene Null-Bytes, verlorene hohe Bits und Paritätsänderungen.)

Integer-Layout-Feld

32-Bit-Konstante 0x01020304 in der Byte-Reihenfolge der Quelle. Ein Leseprozess könnte bei den folgenden Feldern die Byte-Reihenfolge umdrehen, wenn er hier die falsche Reihenfolge entdeckt.

Optionsfeld

32-Bit-Maske, die wichtige Aspekte des Dateiformats angibt. Bits sind von 0 (LSB) bis 31 (MSB) nummeriert. Beachten Sie, dass dieses Feld in der Byte-Reihenfolge der Quelle gespeichert ist, wie alle folgenden Integer-Felder auch. Bits 16–31 sind für wichtige Dateiformataspekte reserviert; ein Leseprozess sollte

abbrechen, wenn er unerwartete Bits gesetzt vorfindet. Bits 0–15 sind für das Signalisieren von Rückwärtskompatibilitätsaspekten im Format reserviert; ein Leseprozess sollte unerwartet gesetzte Bits in diesem Bereich ignorieren. Gegenwärtig ist nur ein Optionsbit definiert, der Rest muss null sein:

Bit 16

wenn 1, dann sind OIDs in der Datei enthalten; wenn 0, dann nicht

Länge des Erweiterungsbereichs

32-Bit-Wert mit der Länge des restlichen Kopfs in Bytes, dieses Feld nicht mit eingeschlossen. Gegenwärtig ist dieses Feld null und das erste Tupel folgt unmittelbar. Durch zukünftige Änderungen im Format könnten weitere Daten im Dateikopf enthalten sein. Ein Leseprozess sollte Erweiterungsdaten, die er nicht verarbeiten kann, ignorieren.

Es ist angedacht, dass der Erweiterungsbereich eine Folge von sich selbst identifizierenden Stücken enthält. Das Optionsfeld soll den Lesern nicht mitteilen, was im Erweiterungsbereich ist. Das genaue Design der Kopferweiterungen wird einer späteren Version überlassen.

Dieses Design ermöglicht sowohl rückwärtskompatible Ergänzungen im Kopfformat (eine Kopferweiterung hinzufügen oder ein niedriges Optionsbit setzen) und nicht rückwärtskompatible Änderungen (ein höheres Optionsbit setzen, um die Änderung anzuzeigen, und eventuell unterstützende Daten im Erweiterungsbereich ablegen).

Tupel

Jedes Tupel beginnt mit einer 16-Bit-Ganzzahl, die die Anzahl der Felder im Tupel zählt. (Gegenwärtig haben alle Tupel in einer Tabelle immer die gleiche Anzahl, aber das muss vielleicht nicht immer der Fall sein.) Dann folgt, wiederholt für jedes Feld im Tupel, eine 16-Bit-Ganzzahl `typlen`, eventuell gefolgt von den Felddaten. Das `typlen`-Feld wird folgendermaßen interpretiert:

0

Das Feld hat den NULL-Wert. Keine Daten folgen.

> 0

Das Feld hat einen Datentyp mit fester Länge. Genau so viele Bytes mit Daten wie angegeben folgen auf das `typlen`-Feld.

-1

Das Feld hat einen Datentyp mit variabler Länge (`varlena`). Die nächsten vier Bytes sind der `varlena`-Kopf, welcher die Länge des Werts, einschließlich der Längenangabe selbst, enthält.

< -1

Reserviert für zukünftige Verwendung.

Bei Feldern, die nicht den NULL-Wert haben, kann der Leseprozess prüfen, ob der `typlen`-Wert mit dem erwarteten `typlen`-Wert der Zielspalte übereinstimmt. Das ist eine einfache, aber sehr nützliche Methode, um zu prüfen, ob die Daten wie erwartet aussehen.

Es gibt keine Ausrichtungslücken oder andere zusätzliche Daten zwischen den Feldern. Beachten Sie auch, dass das Format nicht zwischen Datentypen mit Wertübergabe und mit Referenzübergabe unterscheidet. Das ist beides beabsichtigt: Dadurch kann sich die Portierbarkeit der Datei verbessern (obwohl die Byte-Reihenfolge und Probleme beim Fließkommaformat trotzdem Probleme darstellen können, wenn man eine binäre Datei von einer Maschine auf eine andere überträgt).

Wenn OIDs in der Datei enthalten sind, folgt das OID-Feld unmittelbar nach dem Feld mit der Feldanzahl. Es ist ein normales Feld, außer dass es nicht in der Feldanzahl enthalten ist. Insbesondere hat es auch ein `typlen`-Feld: Dadurch kann man ohne zu viele Probleme mit 4 Byte und 8 Byte großen OIDs umgehen und OIDs könnten auch den NULL-Wert haben, wenn sich das einmal als wünschenswert herausstellen sollte.

Dateiabschluss

Der Dateiabschluss besteht aus einem 16-Bit-Wort mit dem Wert -1. Das kann man einfach vom Feldanzahl-Wort eines Tupels unterscheiden.

Ein Leseprozess sollte einen Fehler auslösen, wenn ein Feldanzahl-Wort weder -1 noch die erwartete Anzahl von Spalten ist. Das bietet einen zusätzlichen Schutz, wenn man aus irgendwelchen Gründen die Synchronisation mit den Daten verliert.

Beispiele

Das folgende Beispiel kopiert eine Tabelle zum Client mit dem senkrechten Strich (|) als Feldtrennzeichen:

```
COPY land TO STDOUT WITH DELIMITER '|';
```

Dieses Beispiel kopiert Daten von einer Datei in die Tabelle `land`:

```
COPY land FROM '/usr1/proj/bray/sql/länderdaten';
```

Hier ist ein Beispiel für passend formatierte Daten, die von `STDIN` in eine Tabelle kopiert werden können (sie müssen bei `STDIN` die Abschlussmarkierung auf der letzten Zeile haben):

```
AF      AFGHANISTAN
AL      ALBANIEN
DZ      ALGERIEN
ZM      SAMBIA
ZW      SIMBABWE
\.
```

Beachten Sie, dass die Leerzeichen in jeder Zeile eigentlich ein Tab-Zeichen sein müssen.

Folgendes sind dieselben Daten im binären Format, erzeugt auf einer Linux/i586-Maschine. Die gezeigten Daten wurden durch das Unix-Hilfsprogramm `od -c` gefiltert. Die Tabelle hat drei Spalten: Die erste ist vom Typ `char(2)`, die zweite ist vom Typ `text` und die dritte ist vom Typ `integer`. Alle Zeilen haben einen NULL-Wert in der dritten Spalte.

```
0000000   P   G   B   C   O   P   Y  \n 377  \r  \n  \0 004 003 002 001
0000020  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 377 377 006  \0  \0  \0
0000040   A   F 377 377 017  \0  \0  \0   A   F   G   H   A   N   I   S
0000060   T   A   N  \0  \0 003  \0 377 377 006  \0  \0  \0   A   L 377
0000100 377  \f  \0  \0  \0   A   L   B   A   N   I   E   N  \0  \0 003
0000120  \0 377 377 006  \0  \0  \0   D   Z 377 377  \f  \0  \0  \0   A
0000140   L   G   E   R   I   E   N  \0  \0 003  \0 377 377 006  \0  \0
0000160  \0   Z   M 377 377  \n  \0  \0  \0   S   A   M   B   I   A  \0
0000200  \0 003  \0 377 377 006  \0  \0  \0   Z   W 377 377  \f  \0  \0
0000220  \0   S   I   M   B   A   B   W   E  \0  \0 377 377
```

Kompatibilität

Der Befehl `COPY` ist eine PostgreSQL-Erweiterung.

Die folgende Syntax wurde in PostgreSQL vor Version 7.3 verwendet und wird weiterhin unterstützt:

```
COPY [ BINARY ] tabelle [ WITH OIDS ]
    FROM { 'dateiname' | STDIN }
    [ [USING] DELIMITERS 'trennzeichen' ]
    [ WITH NULL AS 'null_darstellung' ]

COPY [ BINARY ] tabelle [ WITH OIDS ]
    TO { 'dateiname' | STDOUT }
    [ [USING] DELIMITERS 'trennzeichen' ]
    [ WITH NULL AS 'null_darstellung' ]
```

CREATE AGGREGATE

Name

CREATE AGGREGATE – definiert eine neue Aggregatfunktion

Synopsis

```
CREATE AGGREGATE name (
    BASETYPE = eingabedatentyp,
    SFUNC = übergangsfunktion,
    STYPE = zustandsdatentyp
    [ , FINALFUNC = abschlussfunktion ]
    [ , INITCOND = anfangswert ]
)
```

Beschreibung

`CREATE AGGREGATE` erzeugt eine neue Aggregatfunktion. Einige Aggregatfunktionen für Basistypen, zum Beispiel `min(integer)` und `avg(double precision)`, sind bereits in der Standarddistribution enthalten. Wenn man neue Typen erzeugt hat oder eine noch nicht vorhandene Aggregatfunktion benötigt, kann man diese mit `CREATE AGGREGATE` erzeugen.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE AGGREGATE mein_schema.mein_agg ...`), wird die Aggregatfunktion im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt.

Eine Aggregatfunktion wird durch ihren Namen und ihren Eingabedatentyp identifiziert. Zwei Aggregatfunktionen im selben Schema können denselben Namen haben, wenn sie unterschiedliche Eingabetypen haben. Der Name und der Eingabedatentyp einer Aggregatfunktion muss sich auch von den Namen und Eingabedatentypen aller normalen Funktionen im selben Schema unterscheiden.

Eine Aggregatfunktion wird aus einer oder zwei normalen Funktionen aufgebaut: der Zustandsübergangsfunktion und wahlweise einer Abschlussfunktion. Diese werden folgendermaßen verwendet:

```
übergangsfunktion( interner-zustand, nächster-wert ) ---> nächster-interner-
zustand
abschlussfunktion( interner-zustand ) ---> aggregatwert
```

PostgreSQL erzeugt eine temporäre Variable vom Datentyp *zustandsdatentyp*, um den internen Zustandswert der Aggregatfunktion zu speichern. Bei jedem Datenwert wird die Übergangsfunktion aufgerufen, um den neuen internen Zustandswert zu berechnen. Nachdem alle Daten verarbeitet worden sind, wird die Abschlussfunktion einmal aufgerufen um den Ausgabewert der Aggregatfunktion zu berechnen. Wenn es keine Abschlussfunktion gibt, wird der letzte Zustandswert unverändert zurückgegeben.

Eine Aggregatfunktion kann einen Anfangswert für den internen Zustandswert angeben. Dieser Wert wird als Konstante vom Typ `text` angegeben und wird auch in einer solchen Spalte gespeichert, muss aber eine gültige Konstante für den Datentyp des Zustandswerts darstellen. Wenn kein Anfangswert angegeben ist, fängt der Zustandswert mit dem NULL-Wert an.

Wenn die Übergangsfunktion als „strikt“ deklariert wurde, kann sie nicht mit NULL-Werten als Eingabewert aufgerufen werden. Bei einer solchen Übergangsfunktion verhält sich die Aggregatfunktion bei der Ausführung wie folgt. NULL-Werte in der Eingabe werden ignoriert (die Funktion wird nicht aufgerufen und der vorherige Zustandswert wird beibehalten). Wenn der Anfangswert der NULL-Wert ist, ersetzt der erste vom NULL-Wert verschiedene Eingabewert den Zustandswert, und die Übergangsfunktion wird erst beim zweiten vom NULL-Wert verschiedenen Eingabewert aufgerufen. Das ist für die Implementierung von Aggregatfunktionen wie `max` praktisch. Beachten Sie, dass dieses Verhalten nur funktionieren kann, wenn *zustandsdatentyp* gleich *eingabedatentyp* ist. Wenn diese Typen verschieden sind, müssen Sie einen Anfangswert, der nicht der NULL-Wert ist, angeben oder eine nicht strikte Übergangsfunktion verwenden.

Wenn die Übergangsfunktion nicht strikt ist, wird sie bedingungslos für jeden Eingabewert aufgerufen und muss mit NULL-Werten in sowohl dem Eingabewert als auch im Zustandswert umgehen können. Damit hat der Autor einer Aggregatfunktion volle Kontrolle über den Umgang mit NULL-Werten in einer Aggregatfunktion.

Wenn die Abschlussfunktion als „strikt“ deklariert wurde, wird sie nicht aufgerufen werden, wenn der letzte Zustandswert der NULL-Wert ist; stattdessen wird der NULL-Wert automatisch als Ergebnis zurückgegeben. (Das ist natürlich das normale Verhalten von strikten Funktionen.) Auf jeden Fall hat die Abschlussfunktion die Wahl, den NULL-Wert zurückzugeben. Zum Beispiel gibt die Abschlussfunktion von `avg` den NULL-Wert zurück, wenn gar keine Eingabewerte verarbeitet wurden.

Parameter

name

Der Name der zu erzeugenden Aggregatfunktion.

eingabedatentyp

Der Eingabedatentyp für diese Aggregatfunktion. Hier kann man auch `ANY` angeben, wenn die Aggregatfunktion die Eingabewerte nicht direkt ansieht und daher jeden Datentyp verarbeiten kann (wie zum Beispiel `count(*)`).

übergangsfunktion

Der Name der Zustandsübergangsfunktion, die für jeden Eingabedatenwert aufgerufen werden soll. Das ist normalerweise eine Funktion mit zwei Argumenten, das erste vom Typ *zustandsdatentyp* und das zweite vom Typ *eingabedatentyp*. Bei einer Aggregatfunktion, die die Eingabewerte nicht direkt ansieht, hat die Funktion nur ein Argument vom Typ *zustandsdatentyp*. In jedem Fall muss der Rück-

gabewert der Funktion vom Typ *zustandsdatentyp* sein. Die Funktion nimmt den aktuellen Zustandswert und den aktuellen Eingabewert und berechnet den neuen Zustandswert.

zustandsdatentyp

Der Datentyp für den internen Zustandswert der Aggregatfunktion.

abschlussfunktion

Der Name der Abschlussfunktion, die das Ergebnis der Aggregatfunktion berechnet, nachdem alle Eingabewerte verarbeitet wurden. Die Funktion muss ein einziges Argument vom Typ *zustandsdatentyp* haben. Der Rückgabetyp dieser Funktion definiert gleichzeitig den Rückgabetyp der Aggregatfunktion. Wenn keine Abschlussfunktion angegeben wird, ist das Ergebnis der Aggregatfunktion der letzte Zustandswert und der Rückgabetyp der Aggregatfunktion ist *zustandsdatentyp*.

anfangswert

Der Anfangswert für den Zustandswert. Dieser muss eine Konstante in der Form sein, die vom Datentyp *zustandsdatentyp* akzeptiert wird. Wenn er nicht angegeben ist, fängt der Zustandswert mit dem NULL-Wert an.

Die Parameter von CREATE AGGREGATE können in beliebiger Reihenfolge geschrieben werden, nicht nur in der oben gezeigten.

Meldungen

CREATE AGGREGATE

Meldung, wenn die Aggregatfunktion erfolgreich erzeugt wurde.

Beispiele

Siehe Abschnitt SET.

Kompatibilität

Der Befehl CREATE AGGREGATE ist eine PostgreSQL-Erweiterung. Der SQL-Standard sieht keine benutzerdefinierten Aggregatfunktionen vor.

Siehe auch

DROP AGGREGATE *(Seite: 710)*

CREATE CAST

Name

CREATE CAST – definiert eine neue Typumwandlung

Synopsis

```
CREATE CAST (quelltyp AS zieltyp)
    WITH FUNCTION funktionsname (argtyp)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (quelltyp AS zieltyp)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Beschreibung

`CREATE CAST` definiert eine neue Typumwandlung (englisch *cast*). Eine Typumwandlung gibt an, wie ein Wert von einem Datentyp in einen anderen umgewandelt werden kann. Zum Beispiel:

```
SELECT CAST(42 AS text);
```

Dieser Ausdruck wandelt die Konstante 42 vom Typ `integer` in den Typ `text` um, indem er eine vorher angegebene Funktion aufruft, in diesem Fall `text(int4)`. (Wenn keine passende Typumwandlung definiert wurde, schlägt die Umwandlung fehl.)

Zwei Typen können *binärkompatibel* sein, was heißt, dass sie, ohne eine Funktion aufzurufen, ineinander umgewandelt werden können. Das erfordert, dass zwei einander entsprechende Werte dasselbe interne Format haben müssen. Zum Beispiel sind die Typen `text` und `varchar` binärkompatibel.

Wenn nichts anderes angegeben ist, wird eine Typumwandlung nur bei ausdrücklicher Anforderung angewendet, das heißt durch `CAST(`*x* `AS` *typname*`)`, *x*`::`*typname* oder *typname*(*x*).

Wenn eine Typumwandlung mit `AS ASSIGNMENT` markiert wurde, kann sie implizit angewendet werden, wenn ein Wert einer Spalte mit dem Zieldatentyp zugewiesen wird. Wenn zum Beispiel die Spalte `foo.f1` den Datentyp `text` hat, wird

```
INSERT INTO foo (f1) VALUES (42);
```

funktionieren, wenn die Typumwandlung vom Typ `integer` in den Typ `text` mit `AS ASSIGNMENT` markiert wurde, ansonsten nicht.

Wenn eine Typumwandlung mit `AS IMPLICIT` markiert wurde, kann sie implizit in jedem Zusammenhang verwendet werden, egal ob bei einer Wertzuweisung oder in einem Ausdruck. Zum Beispiel hat der Operator `||` zwei Operanden vom Typ `text`. Daher wird

```
SELECT 'Die Zeit ist ' || now();
```

nur funktionieren, wenn die Typumwandlung vom Typ `timestamp` in den Typ `text` mit `AS IMPLICIT` markiert wurde. Ansonsten müsste man die Typumwandlung ausdrücklich anfordern, zum Beispiel so:

```
SELECT 'Die Zeit ist ' || CAST(now() AS text);
```

Implizite Typumwandlungen sollte man nur mit Vorsicht erzeugen. Bei zu vielen impliziten Typumwandlungspfaden kann PostgreSQL Befehle auf überraschende Weise interpretieren oder Befehle gar nicht auflösen, weil es zu viele mögliche Interpretationen gibt. Eine gute Richtlinie ist, dass man nur Typumwandlungen implizit machen sollten, die zwischen Typen derselben allgemeinen Kategorie umwandeln und alle Informationen erhalten. Zum Beispiel kann eine Umwandlung von `int2` in `int4` ohne Probleme implizit sein, aber die Umwandlung von `float8` in `int4` sollte wahrscheinlich nur bei Wertzuweisungen zugelas-

sen werden. Typumwandlungen zwischen Typen verschiedener Kategorien, wie von `text` in `int4`, sollten am besten nur bei ausdrücklicher Anforderung getätigt werden.

Um eine Typumwandlung erzeugen zu können müssen die den Quell- oder den Zieldatentyp besitzen. Um eine binärkompatible Typumwandlung zu erzeugen, müssen Sie ein Superuser sein. (Diese Einschränkung besteht, weil eine fehlerhafte binärkompatible Typumwandlung ziemlich einfach den Server zum Absturz bringen kann.)

Parameter

`quelltyp`

Der Name des Quelldatentyps der Typumwandlung.

`zieltyp`

Der Name des Zieldatentyps der Typumwandlung.

`funktionsname(argtyp)`

Die Funktion, mit der die Umwandlung durchgeführt werden soll. Der Funktionsname kann eine Schemaqualifikation haben. Wenn nicht, wird die Funktion im Pfad gesucht werden. Der Argumentdatentyp muss mit dem Quelldatentyp identisch sein, der Rückgabedatentyp muss mit dem Zieldatentyp der Typumwandlung übereinstimmen.

`WITHOUT FUNCTION`

Gibt an, dass der Quelldatentyp und der Zieldatentyp binärkompatibel sind, sodass keine Funktion für die Durchführung der Umwandlung benötigt wird.

`AS ASSIGNMENT`

Gibt an, dass die Umwandlung im Zusammenhang mit einer Wertzuweisung angewendet werden darf.

`AS IMPLICIT`

Gibt an, dass die Umwandlung in jedem Zusammenhang implizit angewendet werden kann.

Meldungen

`CREATE CAST`

Meldung, wenn die Typumwandlung erfolgreich erzeugt wurde.

Hinweise

Denken Sie daran, wenn Sie Typen in beide Richtungen umwandeln wollen, müssen Sie Typumwandlungen für beide Richtungen ausdrücklich deklarieren.

Vor PostgreSQL 7.3 war jede Funktion, die den gleichen Namen wie ein Datentyp hatte, diesen Datentyp zurückgab und ein Argument von einem anderen Datentyp hatte, automatisch eine Typumwandlungsfunktion. Diese Konvention wurde abgeschafft wegen der Einführung von Schemas und um binärkompatible Umwandlungen in den Systemkatalogen darstellen zu können. (Die eingebauten Umwandlungsfunktionen folgen nach wie vor diesem Namensschema, aber sie müssen jetzt im Systemkatalog `pg_cast` als Umwandlungsfunktionen angegeben werden.)

Beispiele

Um eine Typumwandlung vom Typ `text` in den Typ `int4` mit der Funktion `int4(text)` zu definieren:

```
CREATE CAST (text AS int4) WITH FUNCTION int4(text);
```

(Diese Umwandlung ist schon im System vordefiniert.)

Kompatibilität

Der Befehl `CREATE CAST` ist konform mit dem SQL-Standard, außer dass der Standard keine Bestimmungen für binärkompatible Typen macht. Die Klausel `AS IMPLICIT` ist ebenfalls eine PostgreSQL-Erweiterung.

Siehe Auch

`CREATE FUNCTION` *(Seite: 664)*, `CREATE TYPE` *(Seite: 697)*, `DROP CAST` *(Seite: 711)*

CREATE CONSTRAINT TRIGGER

Name

CREATE CONSTRAINT TRIGGER – definiert einen neuen Constraint-Trigger

Synopsis

```
CREATE CONSTRAINT TRIGGER name
    AFTER ereignisse ON
    tabelle constraint attribute
    FOR EACH ROW EXECUTE PROCEDURE funktion( argumente )
```

Beschreibung

Der Befehl `CREATE CONSTRAINT TRIGGER` wird in `CREATE TABLE`/`ALTER TABLE` und von `pg_dump` verwendet, um die speziellen Trigger für referenzielle Integrität zu erzeugen. Er ist nicht für allgemeine Verwendung gedacht.

Parameter

name

Der Name des Constraint-Triggers.

ereignisse

Die Ereigniskategorie, für die dieser Trigger ausgelöst werden soll.

tabelle
Der Name der Tabelle (möglicherweise mit Schemaqualifikation), in der die Triggerereignisse geschehen.

constraint
Die eigentliche Constraint-Definition.

attribute
Die Constraint-Attribute.

funktion(argumente)
Die Funktion, die vom Trigger aufgerufen werden soll.

Meldungen

CREATE TRIGGER
Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

CREATE CONVERSION

Name

CREATE CONVERSION – definiert eine neue Zeichensatzkonversion

Synopsis

```
CREATE [DEFAULT] CONVERSION konversionsname
    FOR quellkodierung TO zielkodierung FROM funktionsname
```

Beschreibung

`CREATE CONVERSION` definiert eine neue Zeichensatzkonversion, das heißt eine Umwandlung von einer Zeichensatzkodierung in eine andere. Konversionsnamen können in der Funktion `convert` verwendet werden. Eine Konversion, die als `DEFAULT` markiert wurde, kann auch für die automatische Zeichensatzumwandlung zwischen Client und Server verwendet werden. Dazu müssen zwei Konversionen definiert werden, eine von Kodierung A nach B *und* eine von Kodierung B nach A.

Um eine Konversion erzeugen zu können, müssen Sie das Privileg `EXECUTE` für die Funktion und das Privileg `CREATE` für das Zielschema haben.

Parameter

DEFAULT
Die `DEFAULT`-Klausel gibt an, dass diese Konversion die Standardkonversion für dieses Kodierungspaar ist. Es sollte in jedem Schema nur eine Standardkonversion pro Kodierungspaar geben.

konversionsname

Der Name der Konversion. Der Konversionsname kann eine Schemaqualifikation haben. Wenn nicht, wird die Konversion im aktuellen Schema definiert. Der Konversionsname muss im Schema einmalig sein.

quellkodierung

Der Name der Quellkodierung.

zielkodierung

Der Name der Zielkodierung.

funktionsname

Die Funktion, die die Konversion durchführen soll. Der Funktionsname kann eine Schemaqualifikation haben. Wenn nicht, dann wird die Funktion im Pfad gesucht werden.

Die Funktion muss folgende Signatur haben:

```
funktion(
    integer,  -- Quellkodierungsnummer
    integer,  -- Zielkodierungsnummer
    cstring,  -- Quellzeichenkette (C-Zeichenkette mit Null-Byte am Ende)
    cstring,  -- Zielzeichenkette (C-Zeichenkette mit Null-Byte am Ende)
    integer   -- Länge der Quellzeichenkette
) RETURNS void;
```

Meldungen

`CREATE CONVERSION`

Meldung, wenn die Konversion erfolgreich erzeugt wurde.

Beispiele

Um eine Konversion von der Kodierung `UNICODE` in `LATIN1` zu definieren, die durch die Funktion `meine_funktion` durchgeführt wird:

```
CREATE CONVERSION meine_konv FOR 'UNICODE' TO 'LATIN1' FROM meine_funktion;
```

Kompatibilität

Der Befehl `CREATE CONVERSION` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE FUNCTION` *(Seite: 664)*, `DROP CONVERSION` *(Seite: 712)*

CREATE DATABASE

Name

CREATE DATABASE – erzeugt eine neue Datenbank

Synopsis

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] eigentümer ]
           [ LOCATION [=] 'pfad' ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] kodierung ] ]
```

Beschreibung

`CREATE DATABASE` erzeugt eine neue PostgreSQL-Datenbank.

Normalerweise wird der aktuelle Benutzer der Eigentümer der neuen Datenbank. Superuser können Datenbanken erzeugen, die jemandem anders gehören, indem sie die `OWNER`-Klausel verwenden. Dadurch können Datenbanken für Benutzer, die keine besonderen Privilegien haben, erzeugt werden. Normale Benutzer mit dem Privileg `CREATEDB` können nur Datenbanken mit sich selbst als Eigentümer erzeugen.

Ein alternativer Speicherplatz kann für die Datenbank angegeben werden, um sie zum Beispiel auf einer anderen Festplatte abzulegen. Der Pfad muss mit dem Befehl `initlocation` *(Seite: 845)* vorbereitet worden sein.

Wenn der Pfadname keinen Schrägstrich enthält, wird er als Umgebungsvariable interpretiert, welche dem Serverprozess bekannt sein muss. Dadurch kann der Datenbankadministrator kontrollieren, wo Datenbanken erzeugt werden können. (Eine gebräuchliche Wahl ist zum Beispiel `PGDATA2`.) Wenn der Server mit `ALLOW_ABSOLUTE_DBPATHS` kompiliert wurde (nicht in der Standardeinstellung), werden absolute Pfadnamen, welche einen führenden Schrägstrich haben (z.B. `/usr/local/pgsql/data`), ebenfalls erlaubt.

Normalerweise wird eine neue Datenbank erzeugt, indem die Systemdatenbank `template1` kopiert wird. Eine andere Template-Datenbank kann mit der Klausel `TEMPLATE` *`name`* angegeben werden. Insbesondere kann man mit `TEMPLATE template0` eine „jungfräuliche“ Datenbank erzeugen, die nur die Standardobjekte Ihrer PostgreSQL-Version enthält. Das ist nützlich, wenn Sie vermeiden wollen, dass Objekte, die Sie in `template1` erzeugt haben, kopiert werden.

Mit dem optionalen Parameter `ENCODING` wird die Zeichensatzkodierung für die Datenbank ausgewählt. Wenn er nicht angegeben ist, wird die Kodierung der gewählten Template-Datenbank verwendet.

Parameter

`name`

Der Name der zu erzeugenden Datenbank.

`eigentümer`

Der Name des Datenbankbenutzers, dem die neue Datenbank gehören soll, oder DEFAULT um den Vorgabewert zu verwenden (der Benutzer, der den Befehl ausführt).

pfad

Eine alternative Stelle im Dateisystem, an der die neue Datenbank angelegt werden soll, angegeben als Zeichenkettenkonstante; oder DEFAULT um den Standard zu verwenden.

template

Der Name der Template-Datenbank, aus der die neue Datenbank erzeugt werden soll, oder DEFAULT um das Standardtemplate (template1) zu verwenden.

kodierung

Die Zeichensatzkodierung für die neue Datenbank. Geben Sie eine Zeichenkettenkonstante (z.B. 'SQL_ASCII') oder eine Kodierungsnummer oder DEFAULT, um die Standardkodierung zu verwenden, an.

Die optionalen Parameter können in beliebiger Reihenfolge angegeben werden, nicht nur in der oben gezeigten.

Meldungen

CREATE DATABASE

Meldung, wenn die Datenbank erfolgreich erzeugt wurde.

ERROR: user '*username*' is not allowed to create/drop databases

Sie müssen das besondere Privileg CREATEDB haben, um Datenbanken zu erzeugen. Siehe CREATE USER *(Seite: 702)*.

ERROR: createdb: database "*name*" already exists

Diese Meldung erscheint, wenn eine Datenbank mit dem angegebenen Namen bereits existiert.

ERROR: database path may not contain single quotes

Der Datenbankspeicherplatz *pfad* darf keine Apostrophe enthalten. Das ist nötig, damit die Shell-Befehle, die das Datenbankverzeichnis erzeugen, sicher ausgeführt werden können.

ERROR: CREATE DATABASE: may not be called in a transaction block

Wenn Sie einen expliziten Transaktionsblock offen haben, dann können Sie CREATE DATABASE nicht ausführen. Beenden Sie zuerst den Transaktionsblock.

ERROR: Unable to create database directory '*path*'.
ERROR: Could not initialize database directory.

Diese Meldungen deuten am wahrscheinlichsten auf Dateisystemprobleme, wie fehlende Zugriffsrechte auf das Datenverzeichnis oder eine volle Festplatte, hin. Der Benutzer, unter dem der Datenbankserver läuft, muss Zugriff auf das entsprechende Verzeichnis haben.

Hinweise

Das Programm createdb *(Seite: 788)* führt diesen Befehl aus und kann von der Shell aus aufgerufen werden.

Die Verwendung von alternativen Speicherplätzen, die mit absoluten Pfadnamen angegeben werden, stellt ein Sicherheitsproblem dar. Weitere Informationen finden Sie in Abschnitt SET.

Obwohl es möglich ist, andere Datenbanken außer template1 zu kopieren, indem ihr Name als Template-Datenbank angegeben wird, ist dies (noch) keine generell anwendbare Methode zum Kopieren von

Datenbanken. Wir empfehlen, dass Datenbanken, die als Template verwendet werden sollen, als nicht beschreibbar behandelt werden. Weitere Informationen dazu finden Sie in Abschnitt SET.

Beispiele

Um eine neue Datenbank zu erzeugen:

```
CREATE DATABASE lusiadas;
```

Um eine neue Datenbank im alternativen Verzeichnis `~/private_db` zu erzeugen, führen Sie Folgendes in der Shell aus:

```
mkdir private_db
initlocation ~/private_db
```

Danach führen Sie Folgendes in einer `psql`-Sitzung aus:

```
CREATE DATABASE anderswo WITH LOCATION '/home/olly/private_db';
```

Kompatibilität

Der Befehl `CREATE DATABASE` ist eine PostgreSQL-Erweiterung. Datenbanken entsprechen im SQL-Standard Katalogen, und deren Erzeugung ist der Implementierung überlassen.

Siehe auch

`DROP DATABASE` *(Seite: 713)*, `createdb` *(Seite: 788)*

CREATE DOMAIN

Name

CREATE DOMAIN – definiert eine neue Domäne

Synopsis

```
CREATE DOMAIN name [AS] datentyp
    [ DEFAULT ausdruck ]
    [ NOT NULL | NULL ]
```

Beschreibung

`CREATE DOMAIN` erzeugt eine neue Datendomäne. Der Benutzer, der sie definiert, wird ihr Eigentümer.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE DOMAIN mein_schema.meine_domäne ...`), wird die Domäne im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt. Der Domänenname muss unter allen Typen und Domänen im Schema einmalig sein.

Domänen sind nützlich, um Spalten, die in mehreren Tabellen gleich eingerichtet sind, an einer einzigen Stelle zu konfigurieren. Wenn man zum Beispiel in mehreren Tabellen eine Spalte mit E-Mail-Adressen hat, kann man dafür eine Domäne definieren, anstatt die Constraints in allen Tabellen einzeln einzurichten.

Parameter

`name`

Der Name der zu erzeugenden Domäne (möglicherweise mit Schemaqualifikation).

`datentyp`

Der Datentyp, der der Domäne zugrunde liegt. Dieser kann Arrayangaben enthalten.

`DEFAULT ausdruck`

Diese Klausel gibt einen Vorgabewert für Spalten mit dem Domänendatentyp an. Der Wert kann jeder beliebige Ausdruck ohne Variablen sein (Unteranfragen sind nicht erlaubt). Der Datentyp des Ausdrucks muss mit dem Datentyp der Domäne übereinstimmen. Wenn kein Vorgabewert angegeben wurde, ist der NULL-Wert der Vorgabewert.

Der Vorgabewert wird bei jeder Einfügeoperation verwendet, die keinen Wert für die Spalte angibt. Wenn für eine bestimmte Spalte ein Vorgabewert definiert ist, wird dieser statt dem Vorgabewert der Domäne verwendet. Der Vorgabewert der Domäne hat wiederum Vorrang vor einem etwaigen Vorgabewert des zugrunde liegenden Datentyps.

`NOT NULL`

Wert in dieser Domäne dürfen nicht den NULL-Wert haben.

`NULL`

Werte in dieser Domäne dürfen den NULL-Wert haben. Das ist die Voreinstellung.

Diese Klausel ist nur für die Kompatibilität mit nicht Standard-konformen SQL-Datenbanken gedacht. Sie sollte nicht in neuen Anwendungen verwendet werden.

Meldungen

`CREATE DOMAIN`

Meldung, wenn die Domäne erfolgreich erzeugt wurde.

Beispiele

Dieses Beispiel erzeugt eine Domäne `ländercode` und verwendet sie dann in einer Tabellendefinition:

```
CREATE DOMAIN ländercode char(2) NOT NULL;
CREATE TABLE länderliste (id integer, land ländercode);
```

Kompatibilität

Der Befehl `CREATE DOMAIN` ist mit dem SQL-Standard konform. Allerdings unterstützt PostgreSQL noch keine Check-Constraints für Domänen.

Siehe auch

`DROP DOMAIN` *(Seite: 714)*

CREATE FUNCTION

Name

CREATE FUNCTION – definiert eine neue Funktion

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtyp [, ...] ] )
    RETURNS rückgabetyp
  { LANGUAGE sprachname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'objektdatei', 'linksymbol'
  } ...
    [ WITH ( attribut [, ...] ) ]
```

Beschreibung

`CREATE FUNCTION` definiert eine neue Funktion. `CREATE OR REPLACE FUNCTION` erzeugt entweder eine neue oder ersetzt eine bestehende Funktion.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE FUNCTION mein_schema.meine_funktion ...`), wird die Funktion im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt. Der Name der Funktion darf nicht mit dem Namen einer anderen Funktion mit denselben Argumentdatentypen im selben Schema übereinstimmen. Funktionen mit unterschiedlichen Argumenttypen können jedoch den gleichen Namen haben (das nennt sich *überladen*).

Um die Definition einer bestehenden Funktion zu ändern, verwenden Sie `CREATE OR REPLACE FUNCTION`. Sie können damit aber nicht den Namen oder die Argumenttypen einer Funktion ändern (wenn Sie es versuchen, würden Sie einfach eine neue, getrennte Funktion erzeugen). Außerdem können Sie mit `CREATE OR REPLACE FUNCTION` nicht den Rückgabetyp einer Funktion ändern. Dazu müssen Sie die Funktion löschen und neu erzeugen.

Wenn Sie eine Funktion löschen und neu erzeugen, ist die neue Funktion nicht dasselbe Objekt wie die alte; bestehende Regeln, Sichten, Trigger usw., die die alte Funktion verwenden, sind dann nicht mehr

funktionsfähig. Mit `CREATE OR REPLACE FUNCTION` können Sie eine Funktion ändern, ohne Objekte, die die Funktion verwenden, zu beeinträchtigen.

Der Benutzer, der die Funktion erzeugt, wird ihr Eigentümer.

Parameter

name

Der Name der zu erzeugenden Funktion.

argtyp

Die Datentypen der Funktionsargumente, falls es welche gibt. Die Argumenttypen können Basistypen, zusammengesetzte Typen oder Domänen sein oder den Typ einer bestehenden Tabellenspalte kopieren.

Den Typ einer bestehenden Spalte kopiert man, indem man *tabellenname.spaltennname*`%TYPE` schreibt; das ist manchmal nützlich um die Funktion unabhängig von der Definition einer Tabelle machen zu können.

Je nach Implementierungssprache kann man auch „Pseudotypen“ wie `cstring` verwenden. Ein Pseudotyp bedeutet, dass der eigentliche Argumenttyp entweder unvollständig angegeben ist oder außerhalb der normalen SQL-Datentypen steht.

rückgabetyp

Der Datentyp des Rückgabewerts. Der Rückgabetyp kann ein Basistyp, ein zusammengesetzter Typ oder eine Domäne sein oder den Typ einer bestehenden Tabellenspalte kopieren. Wie man den Typ einer Tabellenspalte kopiert, steht oben unter *argtyp*.

Je nach Implementierungssprache kann man auch „Pseudotypen“ wie `cstring` verwenden. Wenn man `SETOF` vor den Typ schreibt, gibt das an, dass die Funktion eine Ergebnismenge anstatt eines einzelnen Werts zurückgibt.

sprachname

Der Name der Sprache, in der die Funktion implementiert ist. Zur Auswahl stehen `SQL`, `C`, `internal` und die Namen von benutzerdefinierten prozeduralen Sprachen (siehe auch `createlang` *(Seite: 790)*). Aus Kompatibilität mit früheren Versionen kann der Sprachname auch in Apostrophen stehen.

`IMMUTABLE`
`STABLE`
`VOLATILE`

Diese Attribute informieren das System, ob es sicher ist, bei der Laufzeitoptimierung mehrere Aufrufe einer Funktion durch einen einzigen zu ersetzen. Höchstens eines dieser Attribute sollte angegeben werden. Wenn keines angegeben ist, ist die Standardeinstellung `VOLATILE`.

`IMMUTABLE` (unveränderlich) gibt an, dass die Funktion immer das gleiche Ergebnis zurückgibt, wenn sie mit den gleichen Argumentwerten aufgerufen wird; das heißt, sie liest keine Informationen aus der Datenbank oder verwendet sonstige Informationen, die nicht direkt in der Argumentliste vorkommen. Wenn diese Option angegeben wird, kann jeder Aufruf der Funktion mit nur konstanten Argumenten sofort durch das Ergebnis der Funktion ersetzt werden.

`STABLE` (stabil) gibt an, dass die Funktion innerhalb einer Tabellendurchsuchung das gleiche Ergebnis bei gleichen Argumentwerten zurückgibt, aber dass das Ergebnis sich zum nächsten SQL-Befehl ändern kann. Das ist die passende Wahl für Funktionen, die Informationen aus der Datenbank lesen, Parametervariablen verwenden (z.B. die aktuelle Zeitzone) usw. Die Funktionen der Familie `current_timestamp` sind auch stabil, weil ihr Wert sich in einer Transaktion nicht ändert.

`VOLATILE` (flüchtig) gibt an, dass sich der Wert der Funktion jederzeit ändern kann, auch innerhalb einer Tabellendurchsuchung, und dass deswegen keine Optimierungen vorgenommen werden können. Relativ wenige Datenbankfunktionen haben diese Eigenschaft; einige Beispiele sind `random()`, `currval()`,

`timeofday()`. Beachten Sie, dass alle Funktionen mit Nebeneffekten als `VOLATILE` eingeordnet werden müssen, selbst wenn die Ergebnisse vorhersagbar sind, damit die Funktionsaufrufe nicht wegoptimiert werden können; ein Beispiel dafür ist `setval()`.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

`CALLED ON NULL INPUT` (die Standardeinstellung) gibt an, dass die Funktion normal aufgerufen wird, wenn eines ihrer Argumente der NULL-Wert ist. Es liegt dann in der Verantwortung des Autors der Funktion, NULL-Werte zu erkennen und wie gewünscht zu verarbeiten.

`RETURNS NULL ON NULL INPUT` oder `STRICT` gibt an, dass die Funktion immer den NULL-Wert zurückgibt, wenn eines der Argumente der NULL-Wert ist. Wenn dieser Parameter angegeben wird, wird die Funktion gar nicht ausgeführt, wenn eines der Argumente der NULL-Wert ist; stattdessen wird der NULL-Wert automatisch als Ergebnis genommen.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

`SECURITY INVOKER` gibt an, dass die Funktion mit den Privilegien des Benutzers, der sie aufruft, ausgeführt werden soll. (Das ist die Voreinstellung.) `SECURITY DEFINER` gibt an, dass die Funktion mit den Privilegien des Benutzers, der sie erzeugt hat, aufgerufen werden soll.

Das Schlüsselwort `EXTERNAL` ist wegen der Konformität mit dem SQL-Standard vorhanden, kann aber weggelassen werden, weil diese Einstellung, im Gegensatz zum SQL-Standard, nicht nur für externe Funktionen gilt.

`definition`

Eine Zeichenkette, die die Funktion definiert; die Bedeutung hängt von der Sprache ab. Es könnte der Name einer internen Funktion sein, der Pfad zu einer Objektdatei, ein SQL-Befehl oder Text in einer prozeduralen Sprache.

`objektdatei, linksymbol`

Diese Form der `AS`-Klausel wird für dynamisch ladbare C-Funktionen verwendet, wenn der Funktionsname im C-Code nicht derselbe wie der Name der SQL-Funktion ist. Die Zeichenkette *`objektdatei`* ist der Name einer Datei, die das dynamisch zu ladende Objekt enthält, und *`linksymbol`* ist das Link-Symbol der Funktion, das heißt der Name der Funktion im C-Code. Wenn das Link-Symbol weggelassen wird, wird davon ausgegangen, dass die Funktion genauso heißt wie die SQL-Funktion, die gerade erzeugt wird.

`attribut`

Die ältere Methode, um zusätzliche Informationen über die Funktion anzugeben. Die folgenden Attribute können auftreten:

`isStrict`

Gleichbedeutend mit `STRICT` und `RETURNS NULL ON NULL INPUT`

`isCachable`

`isCachable` ist ein obsoletes Äquivalent von `IMMUTABLE`; der Kompatibilität halber wird es immer noch akzeptiert.

Bei den Attributnamen wird Groß- und Kleinschreibung nicht unterschieden.

Meldungen

`CREATE FUNCTION`

Meldung, wenn die Funktion erfolgreich erzeugt wurde.

Hinweise

Weitere Informationen, wie man Funktionen schreibt, erhalten Sie in Abschnitt SET.

Die volle Typsyntax von SQL kann für die Argumenttypen und den Rückgabetyp verwendet werden, aber einige Einzelheiten der Typangabe (z.B. das Präzisionsfeld beim Typ `numeric`) werden von `CREATE FUNCTION` nicht beachtet und müssen von der Implementierung der Funktion verarbeitet werden.

In PostgreSQL können Funktionen *`überladen`* werden; das heißt, derselbe Name kann für mehrere Funktionen verwendet werden, wenn sie verschiedene Argumenttypen haben. Die C-Namen aller Funktionen müssen jedoch unterschiedlich sein, also müssen Sie überladenen C-Funktionen unterschiedliche C-Namen geben (zum Beispiel, indem Sie die Argumenttypen in den Namen einbauen).

Wenn wiederholte Aufrufe von `CREATE FUNCTION` auf dieselbe Objektdatei verweisen, wird die Datei nur einmal geladen. Um die Datei aus dem Speicher zu entfernen und neu zu laden (vielleicht bei der Entwicklungsarbeit), verwenden Sie den Befehl `LOAD` *(Seite: 746)*.

Um eine Funktion definieren zu können, benötigen Sie das Privileg `USAGE` für die Sprache.

Beispiele

Hier ist ein einfaches Beispiel, das Ihnen beim Einstieg helfen kann. Weitere Informationen und Beispiele finden Sie in Abschnitt SET.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Kompatibilität

Ein Befehl `CREATE FUNCTION` ist im SQL-Standard definiert. Die `PostgreSQL`-Version ist ähnlich, aber nicht voll kompatibel. Die Attribute sind nicht portierbar und auch nicht die verschiedenen Sprachen.

Siehe auch

`DROP FUNCTION` *(Seite: 715)*, `LOAD` *(Seite: 746)*

CREATE GROUP

Name

CREATE GROUP – definiert eine neue Benutzergruppe

Synopsis

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

wobei option Folgendes sein kann:

     SYSID gid
   | USER  benutzername [, ...]
```

Beschreibung

`CREATE GROUP` erzeugt eine neue Gruppe im Datenbankcluster. Um diesen Befehl verwenden zu können, müssen Sie ein Datenbank-Superuser sein.

Parameter

name

Der Name der Gruppe.

gid

Mit der `SYSID`-Klausel kann man die PostgreSQL-Gruppen-ID der neuen Gruppe bestimmen. Das ist allerdings nicht notwendig.

Wenn keine ID angegeben ist, wird die höchste bisher vergebene ID plus 1, beginnend bei 1, verwendet.

benutzername

Eine Liste von Benutzern, die in die neue Gruppe aufgenommen werden sollen. Die Benutzer müssen bereits existieren.

Meldungen

`CREATE GROUP`

Meldung, wenn die Gruppe erfolgreich erzeugt wurde.

Beispiele

Um eine leere Gruppe zu erzeugen:

```
CREATE GROUP mitarbeiter;
```

Um eine Gruppe mit Mitgliedern zu erzeugen:

```
CREATE GROUP marketing WITH USER karl, david;
```

Kompatibilität

Der Befehl `CREATE GROUP` ist eine PostgreSQL-Erweiterung. Das Konzept der Rollen im SQL-Standard ist vergleichbar mit Gruppen.

Siehe auch

`ALTER GROUP` *(Seite: 627)*, `DROP GROUP` *(Seite: 717)*

CREATE INDEX

Name

CREATE INDEX – definiert einen neuen Index

Synopsis

```
CREATE [ UNIQUE ] INDEX indexname ON tabelle
    [ USING methode ] ( spalte [ ops_name ] [, ...] )
    [ WHERE prädikat ]

CREATE [ UNIQUE ] INDEX indexname ON tabelle
    [ USING methode ] ( funktionsname( spalte [, ... ]) [ ops_name ] )
    [ WHERE prädikat ]
```

Beschreibung

`CREATE INDEX` erzeugt einen Index namens *`indexname`* für die angegebene Tabelle. Indexe werden hauptsächlich verwendet, um die Leistung der Datenbank zu verbessern. Aber unpassende Verwendung kann auch zu schlechterer Leistung führen.

In der ersten oben gezeigten Syntax sind die Schlüsselfelder des Index die angegebenen Spaltennamen. Mehrere Felder können angegeben werden, wenn die Indexmethode mehrspaltige Indexe unterstützt.

In der zweiten oben gezeigten Syntax wird ein Index über die Ergebnisse der angegebenen Funktion *`funktionsname`*, die für eine oder mehrere Spalten einer einzelnen Tabelle aufgerufen wird, definiert. Diese *`Funktionsindexe`* ermöglichen schnellen Zugriff auf Daten, wenn die Anfragebedingungen die indizierten Daten in Funktionsaufrufen enthalten. Wenn man zum Beispiel einen Funktionsindex über `upper(spalte)` hätte, dann könnte eine Klausel wie `WHERE upper(spalte) = 'FRED'` einen Index verwenden.

PostgreSQL bietet die Indexmethoden B-Tree, R-Tree, Hash und GiST. Die B-Tree-Indexmethode ist eine Implementation der hochparallelen B-Trees von Lehman und Yao. Die R-Tree-Indexmethode ist eine Implementation des normalen R-Trees mit dem quadratischen Split-Algorithmus von Guttman. Der Hash-Index ist eine Implementation des linearen Hashs von Litwin. Benutzer können auch eigene Indexmethoden definieren, aber das ist ziemlich schwierig.

Wenn eine WHERE-Klausel angegeben ist, wird ein *partieller Index* erzeugt. Ein partieller Index ist ein Index, der nur Einträge für einen Teil der Tabelle enthält, normalerweise einen Teil, der irgendwie interessanter ist als der Rest der Tabelle. Wenn Sie zum Beispiel eine Tabelle mit Bestellungen haben, wovon für einige schon die Rechnung gestellt wurde und für andere nicht, und die Bestellungen ohne Rechnung ein kleiner Teil der Tabelle sind, aber die Zeilen mit den meisten Zugriffen, können Sie die Leistung verbessern, wenn Sie den Index nur über diesen Bereich erstellen. Eine weitere mögliche Anwendung ist, WHERE mit UNIQUE zu verwenden, um die Einmaligkeit der Werte in nur einem Teil der Tabelle zu erreichen.

Der Ausdruck in der WHERE-Klausel kann nur Spalten der zugrunde liegenden Tabelle verwenden (aber alle Spalten, nicht nur die, die indiziert werden). Gegenwärtig sind Unteranfragen und Aggregatausdrücke in der WHERE-Klausel nicht erlaubt.

Alle Funktionen und Operatoren, die in der Indexdefinition verwendet werden, müssen als IMMUTABLE (unveränderlich) markiert sein, das heißt, ihre Ergebnisse hängen nur von den Argumentwerten ab und niemals von äußeren Einflüssen (wie zum Beispiel dem Inhalt einer anderen Tabelle oder der aktuelle Zeit). Diese Einschränkung stellt sicher, dass das Verhalten des Index wohldefiniert ist. Um eine benutzerdefinierte Funktion im Index zu verwenden, denken Sie also daran, sie als IMMUTABLE zu markieren.

Parameter

UNIQUE

Damit prüft das System, ob die Tabelle doppelte Werte enthält, sowohl bei der Erzeugung des Index (wenn die Tabelle schon Daten enthält) als auch jedes Mal, wenn Daten hinzugefügt werden. Wenn man Daten einfügt oder aktualisiert und sich dadurch doppelte Werte ergeben würden, ergibt das einen Fehler.

indexname

Der Name des zu erzeugenden Index. Hier kann kein Schemaname angegeben werden; der Index wird immer im selben Schema wie seine Tabelle erzeugt.

tabelle

Der Name der zu indizierenden Tabelle (möglicherweise mit Schemaqualifikation).

method

Der Name der für den Index zu verwendenden Methode. Zur Auswahl stehen btree, hash, rtree und gist. Die Standardmethode ist btree.

spalte

Der Name einer Spalte in der Tabelle.

ops_name

Eine zugehörige Operatorklasse. Einzelheiten siehe unten.

funktionsname

Eine Funktion, die die zu indizierenden Werte zurückgibt.

prädikat

Definiert den Bedingungsausdruck für einen partiellen Index.

Meldungen

CREATE INDEX

Meldung, wenn der Index erfolgreich erzeugt wurde.

Hinweise

Weitere Informationen darüber, wann Indexe verwendet werden, wann sie nicht verwendet werden und in welchen bestimmten Situationen sie nützlich sein können, finden Sie in Abschnitt SET.

Gegenwärtig unterstützen nur die Indexmethoden B-Tree und GiST mehrspaltige Indexe. In der Voreinstellung werden bis zu 32 Felder unterstützt. (Diese Grenze kann beim Kompilieren von PostgreSQL verändert werden.) Nur die B-Tree-Methode unterstützt gegenwärtig die `UNIQUE`-Option.

Eine Indexdefinition kann für jede Spalte eines Index eine *`Operatorklasse`* angeben. Die Operatorklasse bestimmt, welche Operatoren vom Index für diese Spalte verwendet werden sollen. Ein B-Tree-Index für eine Spalte vom Typ `int4` würde die Operatorklasse `int4_ops` verwenden; diese Operatorklasse enthält die Vergleichsfunktionen für Werte vom Typ `int4`. Normalerweise reicht die vorgegebene Operatorklasse für den Datentyp der Spalte aus. Der Hauptgrund, warum es Operatorklassen gibt, ist, dass es für manche Datentypen mehrere sinnvolle Sortierreihenfolgen geben kann. Zum Beispiel könnte man komplexe Zahlen nach Betrag oder nach dem reellen Teil sortieren. Das könnte man erreichen, indem man zwei Operatorklassen erzeugt und die passende wählt, wenn man den Index definiert. Weitere Informationen über Operatorklassen finden Sie in Abschnitt SETund in Abschnitt SET.

Beispiele

Um einen B-Tree-Index für die Spalte `titel` in der Tabelle `filme` zu erzeugen:

```
CREATE UNIQUE INDEX titel_idx ON filme (titel);
```

Kompatibilität

Der Befehl `CREATE INDEX` ist eine PostgreSQL-Erweiterung. Im SQL-Standard gibt es keine Bestimmungen über Indexe.

Siehe auch

`DROP INDEX` *(Seite: 718)*

CREATE LANGUAGE

Name

CREATE LANGUAGE – definiert eine neue prozedurale Sprache

Synopsis

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE sprachname
    HANDLER handler [ VALIDATOR valfunktion ]
```

Beschreibung

Mit `CREATE LANGUAGE` kann ein PostgreSQL-Benutzer eine neue prozedurale Sprache in einer Datenbank registrieren. Danach können Funktion und Triggerprozeduren in dieser neuen Sprache geschrieben werden. Ein Benutzer muss ein PostgreSQL-Superuser sein, um eine neue Sprache registrieren zu können.

`CREATE LANGUAGE` verbindet im Prinzip den Sprachnamen mit einer Handlerfunktion, die dafür verantwortlich ist, die in dieser Sprache geschriebenen Funktionen auszuführen. Weitere Informationen über Sprachhandler finden Sie in Abschnitt SET.

Beachten Sie, dass prozedurale Sprachen immer nur in einer bestimmten Datenbank installiert werden. Um eine Sprache in allen Datenbanken verfügbar zu machen, installieren Sie sie in der Datenbank `template1`.

Parameter

`TRUSTED`

`TRUSTED` gibt an, dass der Handler für diese Sprache sicher ist, das heißt, dass er einem unprivilegierten Benutzer keine Möglichkeit gibt, Zugriffsbeschränkungen zu umgehen. Wenn dieses Schlüsselwort bei der Registrierung der Funktion weggelassen wird, werden nur PostgreSQL-Superuser diese Sprache für neue Funktionen verwenden können.

`PROCEDURAL`

Dieses Wort hat keine Bedeutung.

`sprachname`

Der Name der neuen prozeduralen Sprache. Groß- und Kleinschreibung werden nicht unterschieden. Der Name muss unter den Sprachen in der Datenbank einmalig sein.

Aus Kompatibilität mit früheren Versionen kann der Sprachname auch in Apostrophen stehen.

`HANDLER` *`handler`*

`handler` ist der Name einer vorher registrierten Funktion, die aufgerufen werden wird, um in der Sprache geschriebene Funktionen auszuführen. Der Handler einer prozeduralen Sprache muss in einer kompilierten Sprache wie C mit der Aufrufskonvention Version 1 geschrieben sein und in PostgreSQL als Funktion ohne Argumente und mit Rückgabetyp `language_handler`, einem Platzhaltertyp für Sprachhandler, registriert sein.

`VALIDATOR` *`valfunktion`*

`valfunktion` ist der Name einer vorher registrierten Funktion, die aufgerufen werden wird, wenn eine neue Funktion in der Sprache erzeugt wird, um die Funktion zu überprüfen. Wenn keine Prüffunktion angegeben ist, wird die Funktion bei der Erzeugung nicht überprüft. Die Prüffunktion muss ein Argument vom Typ `oid` haben, welches die OID der zu erzeugenden Funktion enthalten wird, und wird in der Regel den Rückgabetyp `void` haben.

Eine Prüffunktion wird in der Regel den Funktionskörper auf syntaktische Fehler prüfen, kann aber auch andere Eigenschaften der Funktion betrachten, zum Beispiel wenn die Sprache nicht mit bestimmten Argumenttypen umgehen kann. Um einen Fehler anzuzeigen, sollte die Prüffunktion die Funktion `elog()` aufrufen. Der Rückgabewert der Funktion wird ignoriert.

Meldungen

`CREATE LANGUAGE`

Meldung, wenn die Sprache erfolgreich erzeugt wurde.

Hinweise

Dieser Befehl sollte nicht direkt von Anwendern ausgeführt werden. Für die in der PostgreSQL-Distribution gelieferten prozeduralen Sprachen sollte man das Programm `createlang` *(Seite: 790)* verwenden, dass auch den richtigen Handler installiert. (`createlang` ruft `CREATE LANGUAGE` intern auf.)

Vor PostgreSQL-Versionen vor 7.3 musste der Sprachhandler mit dem Platzhaltertyp `opaque` als Rückgabetyp deklariert werden, anstatt wie jetzt `language_handler`. Um alte Sicherungsdateien laden zu können, akzeptiert `CREATE LANGUAGE` Funktionen mit deklariertem Rückgabetyp `opaque`, wird aber eine Hinweismeldung ausgeben und den Rückgabetyp der Funktion in `language_handler` ändern.

Der Systemkatalog `pg_language` (siehe Abschnitt SET) speichert Informationen über die aktuell installierten Sprachen. Außerdem hat `createlang` eine Option, um alle installierten Sprachen anzuzeigen.

Die Definition einer prozeduralen Sprache kann nach der Erzeugung nicht mehr verändert werden, mit Ausnahme der Privilegien.

Um eine prozedurale Sprache verwenden zu können, muss dem entsprechenden Benutzer das Privileg `USAGE` gewährt werden. Das Programm `createlang` gibt automatisch jedem die Erlaubnis, die Sprache zu verwenden, wenn die Sprache als sicher (`TRUSTED`) bekannt ist.

Beispiel

Die folgenden zwei Befehle registrieren erst einen Sprachhandler und dann eine neue prozedurale Sprache.

```
CREATE FUNCTION plbeispiel_call_handler() RETURNS language_handler
    AS '$libdir/plbeispiel'
    LANGUAGE C;
CREATE LANGUAGE plbeispiel
    HANDLER plbeispiel_call_handler;
```

Kompatibilität

Der Befehl `CREATE LANGUAGE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE FUNCTION` *(Seite: 664)*, `DROP LANGUAGE` *(Seite: 719)*, `createlang` *(Seite: 790)*

CREATE OPERATOR

Name

CREATE OPERATOR – definiert einen neuen Operator

Synopsis

```
CREATE OPERATOR name (
    PROCEDURE = funktionsname
    [, LEFTARG = linker_typ ] [, RIGHTARG = rechter_typ ]
    [, COMMUTATOR = kommutator_op ] [, NEGATOR = umkehrungs_op ]
    [, RESTRICT = res_funktion ] [, JOIN = join_funktion ]
    [, HASHES ] [, MERGES ]
    [, SORT1 = linker_sortier_op ] [, SORT2 = rechter_sortier_op ]
    [, LTCMP = kleiner_als_op ] [, GTCMP = größer_als_op ]
)
```

Beschreibung

`CREATE OPERATOR` definiert einen neuen Operator. Der Benutzer, der den Operator definiert, wird sein Eigentümer.

Der Operatorname ist eine Folge von bis zu `NAMEDATALEN`-1 (normalerweise 63) Zeichen aus der folgenden Liste:

+ - * / < > = ~ ! @ # % ^ & | ` ? $

Es gibt allerdings ein paar Einschränkungen für die Operatornamen:

- $ (Dollarzeichen) kann nicht als einzelnes Zeichen ein Operator sein. Es kann allerdings Teil eines Operatornamens aus mehreren Zeichen sein.
- `--` und `/*` können nirgendwo in einem Operatornamen auftauchen, weil sie als der Anfang eines Kommentars angenommen werden.
- Ein Operatorname, der aus mehreren Zeichen besteht, kann nicht auf + oder – enden, es sei denn, der Name enthält mindestens eins der folgenden Zeichen:
  ~ ! @ # % ^ & | ` ? $

  So ist zum Beispiel `@-` ein zulässiger Operatorname, nicht aber `*-`. Diese Einschränkung erlaubt es PostgreSQL, Befehle, die dem SQL-Standard folgen, zu verarbeiten, ohne Leerzeichen zwischen den Tokens zu erfordern.

Der Operator `!=` wird bei der Eingabe immer in `<>` umgewandelt, also sind diese beiden Namen immer gleichbedeutend.

Weitere Informationen über die Erzeugung von benutzerdefinierten Operatoren erhalten Sie in Abschnitt SET.

Parameter

`name`

Der Name des zu definierenden Operators. Siehe oben für gültige Zeichen. Der Name kann eine Schemaqualifikation haben, zum Beispiel `CREATE OPERATOR myschema.+ (...)`. Zwei Operatoren im selben Schema können den gleichen Namen haben, wenn Sie unterschiedliche Operandendatentypen haben. Das nennt sich *`überladen`*.

`funktionsname`

Die Funktion, die den Operator implementiert.

`linker_typ`

Der Datentyp des linken Operanden des Operators. Diese Option wird bei einem linken unären Operator ausgelassen.

rechter_typ

Der Datentyp des rechten Operanden des Operators. Diese Option wird bei einem rechten unären Operator ausgelassen.

kommutator_op

Der Kommutator dieses Operators.

umkehrungs_op

Der Umkehrungsoperator dieses Operators.

res_funktion

Die Auswahlselektivitätsschätzfunktion für diesen Operator.

join_funktion

Die Verbundselektivitätsschätzfunktion für diesen Operator.

HASHES

Gibt an, dass dieser Operator Hash-Verbunde unterstützt.

MERGES

Gibt an, dass dieser Operator Merge-Verbunde unterstützt.

linker_sortier_op

Wenn dieser Operator Merge-Verbunde unterstützt, dann der Kleiner-als-Operator für den Datentyp des linken Operanden.

rechter_sortier_op

Wenn dieser Operator Merge-Verbunde unterstützt, dann der Kleiner-als-Operator für den Datentyp des rechten Operanden.

kleiner_als_op

Wenn dieser Operator Merge-Verbunde unterstützt, dann der Kleiner-als-Operator, der die Operandendatentypen dieses Operators vergleicht.

größer_als_op

Wenn dieser Operator Merge-Verbunde unterstützt, dann der Größer-als-Operator, der die Operandendatentypen dieses Operators vergleicht.

Um einen schemaqualifizierten Operatorname in *kommutator_op* oder den anderen optionalen Argumenten zu verwenden, können Sie den OPERATOR()-Syntax, zum Beispiel

```
COMMUTATOR = OPERATOR(myschema.===)
```

Meldungen

CREATE OPERATOR

Meldung, wenn der Operator erfolgreich erzeugt wurde.

Beispiel

Der folgende Befehl definiert einen neuen Operator, Flächenvergleich, für den Datentyp box:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES,
    SORT1 = <<<,
    SORT2 = <<<
    -- Da Sortieroperatoren angegeben wurden, ist MERGES impliziert.
    -- Für LTCMP und GTCMP werden < bzw. > angenommen.
);
```

Kompatibilität

Der Befehl CREATE OPERATOR ist eine PostgreSQL-Erweiterung. Der SQL-Standard sieht keine benutzerdefinierten Operatoren vor.

Siehe auch

DROP OPERATOR *(Seite: 720)*

CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS – definiert eine neue Operatorklasse für Indexe

Synopsis

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE datentyp USING indexmethode AS
  {  OPERATOR strategienummer operatorname [ ( optyp, op_type ) ] [ RECHECK ]
   | FUNCTION unterst_nummer funktionsname ( argumenttyp [, ...] )
   | STORAGE storage_typ
  } [, ... ]
```

Beschreibung

`CREATE OPERATOR CLASS` erzeugt eine neue Operatorklasse. Eine Operatorklasse definiert, wie ein bestimmter Datentyp mit einem Index verwendet werden kann. Die Operatorklasse gibt an, dass bestimmte Operatoren für diesen Datentyp und diese Indexmethode bestimmte Rollen oder „Strategien“ ausfüllen. Die Operatorklasse gibt auch die von der Indexmethode mit diesem Datentyp verwendeten Unterstützungsprozeduren an. Alle von einer Operatorklasse verwendeten Operatoren und Funktionen müssen definiert sein, bevor die Operatorklasse erzeugt werden kann.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE OPERATOR CLASS mein_schema.meine_opklasse ...`), wird die Operatorklasse im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt. Zwei Operatorklassen im selben Schema können nur den gleichen Namen haben, wenn sie für verschiedene Indexmethoden sind.

Der Benutzer, der die Operatorklasse definiert, wird ihr Eigentümer. Gegenwärtig muss der erzeugende Benutzer ein Superuser sein. (Diese Einschränkung besteht, weil eine fehlerhafte Operatorklassendefinition den Server verwirren oder gar zum Absturz bringen kann.)

`CREATE OPERATOR CLASS` prüft gegenwärtig nicht, ob die Operatorklassendefinition alle von der Indexmethode benötigten Operatoren und Funktionen enthält. Es liegt in der Verantwortung des Benutzers, eine gültige Operatorklasse zu definieren.

Weitere Informationen über Operatorklassen finden Sie in Abschnitt SET.

Parameter

`name`

Der Name der zu erzeugenden Operatorklasse. Der Name kann eine Schemaqualifikation haben.

`DEFAULT`

Wenn angegeben, wird die Operatorklasse die Standardoperatorklasse für ihren Datentyp. Höchstens eine Operatorklasse kann für einen bestimmten Datentyp und eine Indexmethode die Standardoperatorklasse sein.

`datentyp`

Der Datentyp, für den diese Operatorklasse gilt.

`indexmethode`

Der Name der Indexmethode, für die diese Operatorklasse gilt.

`strategienummer`

Die Strategienummer der Indexmethode für einen zur Operatorklasse gehörenden Operator.

`operatorname`

Der Name eines zur Operatorklasse gehörenden Operators (möglicherweise mit Schemaqualifikation).

`op_typ`

Die Operandendatentypen eines Operators oder `NONE` bei einem linken unären oder rechten unären Operator. Die Operandendatentypen können normalerweise weggelassen werden, wenn sie dieselben sind wie der Datentyp der Operatorklasse.

`RECHECK`

Wenn angegeben, ist der Index für diesen Operator „verlustbehaftet“, das heißt, dass Zeilen, die mit diesem Index ermittelt wurden, noch einmal geprüft werden müssen, ob sie die Bedingungsklausel mit diesem Operator wirklich erfüllen.

`unterst_nummer`

Die Unterstützungsprozedurnummer der Indexmethode für eine zur Operatorklasse gehörende Funktion.

funktionsname

Der Name einer Funktion (möglicherweise mit Schemaqualifikation), die eine Unterstützungsprozedur in der Operatorklasse ist.

argumenttyp

Die Argumentdatentypen der Funktion.

storage_typ

Der tatsächlich im Index gespeicherte Datentyp. Normalerweise ist das derselbe wie der Spaltendatentyp, aber bei einigen Indexmethoden (gegenwärtig nur GiST) kann er unterschiedlich sein. Die `STORAGE`-Klausel muss weggelassen werden, wenn die Indexmethode die Angabe eines anderen Typs nicht unterstützt.

Die Klauseln `OPERATOR`, `FUNCTION` und `STORAGE` können in beliebiger Reihenfolge erscheinen.

Meldungen

`CREATE OPERATOR CLASS`

Meldung, wenn die Operatorklasse erfolgreich erzeugt wurde.

Beispiel

Der folgende Beispielbefehl definiert eine GiST-Index-Operatorklasse für den Datentyp `_int4` (Array aus `int4`). Siehe in `contrib/intarray/` für das vollständige Beispiel.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       =       RECHECK,
        OPERATOR        7       @,
        OPERATOR        8       ~,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int4),
        FUNCTION        2       g_int_union (bytea, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Kompatibilität

Der Befehl `CREATE OPERATOR CLASS` ist eine PostgreSQL-Erweiterung.

Siehe auch

DROP OPERATOR CLASS *(Seite: 721)*

CREATE RULE

Name

CREATE RULE – definiert eine neue Umschreiberegel

Synopsis

```
CREATE [ OR REPLACE ] RULE name AS ON ereignis
    TO tabelle [ WHERE bedingung ]
    DO [ INSTEAD ] { NOTHING | befehl | ( befehl ; befehl ... ) }
```

Beschreibung

CREATE RULE definiert eine neue Regel für die angegebene Tabelle oder Sicht. CREATE OR REPLACE RULE erzeugt entweder eine neue Regel oder ersetzt eine bestehende mit demselben Namen für dieselbe Tabelle oder Sicht.

Mit dem PostgreSQL-Regelsystem kann man alternative Aktionen definieren, die bei bestimmten Einfüge-, Aktualisierungs- und Löschoperationen in Datenbanktabellen durchgeführt werden. Grob gesagt, sorgt eine Regel dafür, dass zusätzliche Befehle ausgeführt werden, wenn ein bestimmter Befehl in einer bestimmten Tabelle ausgeführt wird. Als Alternative kann eine Regel auch einen Befehl durch einen anderen ersetzen oder dafür sorgen, dass ein Befehl gar nicht ausgeführt wird. Regeln werden auch zur Implementierung von Sichten verwendet. Wichtig dabei ist, dass eine Regel ein Transformationsmechanismus für Befehle oder ein Befehlsmakro ist. Die Transformation passiert, bevor die Ausführung des Befehls anfängt. Wenn Sie wollen, dass eine Operation unabhängig für jede Zeile ausgeführt wird, sollten Sie wohl eher einen Trigger als eine Regel verwenden. Weitere Informationen über das Regelsystem finden Sie in Abschnitt SET.

Gegenwärtig müssen ON SELECT-Regeln INSTEAD sein, dürfen keine Bedingung haben und müssen einen einzelnen SELECT-Befehl als Aktion haben. Eine ON SELECT-Regel macht also aus einer Tabelle im Prinzip eine Sicht, deren sichtbarer Inhalt die vom SELECT-Befehl zurückgegebenen Zeilen sind, anstatt der eigentliche Inhalt der Tabelle. Es ist aber besserer Stil, den Befehl CREATE VIEW zur Erzeugung von Sichten zu verwenden, anstatt eine richtige Tabelle zu erzeugen und dann dafür eine ON SELECT-Regel zu definieren.

Sie können die Illusion einer aktualisierbaren Sicht erzeugen, indem Sie für eine Sicht Regeln für ON INSERT, ON UPDATE und ON DELETE (oder nur die, die Sie benötigen) erzeugen, um für diese Aktionen die entsprechenden Aktionen in anderen Tabellen zu ersetzen.

Wenn Sie Regeln mit Bedingungen für die Aktualisierung von Sichten verwenden, gibt es einen Haken: Es *muss* für jede Aktion, die Sie für die Sicht erlauben wollen, eine Regel ohne Bedingung und mit INSTEAD geben. Wenn die Regel eine Bedingung hat oder nicht INSTEAD ist, wird das System alle Versuche, die Aktion durchzuführen, ablehnen, weil es denkt, dass es eventuell dazu kommen könnte, dass die Aktion direkt in der Sicht ausgeführt werden wird, was nicht geht. Wenn Regeln mit Bedingung trotzdem für Sie am sinnvollsten sind, können Sie sie nichtsdestotrotz verwenden; erzeugen Sie einfache eine DO INSTEAD

NOTHING-Regel ohne Bedingung, damit das System versteht, dass es niemals eine Aktualisierung direkt in der Sicht vornehmen muss. Die Regeln mit Bedingungen definieren Sie dann ohne INSTEAD; wenn sie angewendet werden, wird der ursprüngliche Befehl durch DO INSTEAD NOTHING ersetzt.

Parameter

name

Der Name der zu erzeugenden Regel. Der Name muss sich von den Namen aller Regeln in derselben Tabelle unterscheiden. Wenn es mehrere Regeln für eine Tabelle und einen Ereignistyp gibt, werden Sie in alphabetischer Reihenfolge ihrem Namen nach angewendet.

ereignis

Das Ereignis ist SELECT, INSERT, UPDATE oder DELETE.

tabelle

Der Name der Tabelle oder der Sicht (möglicherweise mit Schemaqualifikation), für die die Regel gilt.

bedingung

Ein Bedingungsausdruck (Ergebnistyp boolean). Der Bedingungsausdruck kann nur auf die Tabellen NEW und OLD verweisen und darf keine Aggregatfunktionen enthalten.

befehl

Der Befehl oder die Befehle, die die Regelaktion darstellen. Gültige Befehle sind SELECT, INSERT, UPDATE, DELETE oder NOTIFY. Wenn Sie anstatt eines Befehls NOTHING schreiben, wird der ursprüngliche Befehl durch „nichts“ ersetzt, also verworfen.

Innerhalb von *bedingung* und *befehl* können die besonderen Tabellennamen NEW und OLD verwendet werden, um auf die Tabelle, zu der die Regel gehört, zu verweisen. NEW kann in ON INSERT- und ON UPDATE-Regeln verwendet werden und verweist auf die einzufügende bzw. die aktualisierte Zeile. OLD kann in ON UPDATE- und ON DELETE-Regeln verwendet werden und verweist auf die bestehende Zeile, die zu aktualisieren bzw. zu löschen ist.

Meldungen

CREATE RULE

Meldung, wenn die Regel erfolgreich erzeugt wurde.

Hinweise

Um eine Regel definieren zu können, müssen Sie für die betroffene Tabelle das Privileg RULE haben.

Es ist wichtig, rekursive Regeldefinitionen zu vermeiden. Die folgenden beiden Regeldefinitionen zum Beispiel werden zwar von PostgreSQL angenommen, aber der SELECT-Befehl würde einen Fehler verursachen, weil die Anfrage zu oft rekursiv aufgerufen wurde:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
```

```
    DO INSTEAD
SELECT * FROM t1;

SELECT * FROM t1;
```

Wenn eine Regelaktion einen NOTIFY-Befehl enthält, wird der NOTIFY-Befehl gegenwärtig bedingungslos ausgeführt. Das heißt, der NOTIFY-Befehl wird selbst dann ausgeführt, wenn es keine Zeilen gibt, für die die Regel angewendet werden kann. Zum Beispiel würde bei

```
CREATE RULE notify_test AS ON UPDATE TO meine_tabelle DO NOTIFY meine_tabelle;

UPDATE meine_tabelle SET name = 'foo' WHERE id = 42;
```

während eines UPDATE ein NOTIFY-Ereignis gesendet, egal ob es Zeilen mit id = 42 gibt. Der Grund dieser Einschränkung liegt in der Implementierung. In der Zukunft sollte dieser Fehler berichtigt werden.

Kompatibilität

Der Befehl CREATE RULE ist eine PostgreSQL-Erweiterung, wie das gesamte Regelsystem.

Siehe auch

DROP RULE *(Seite: 723)*

CREATE SCHEMA

Name

CREATE SCHEMA – definiert ein neues Schema

Synopsis

```
CREATE SCHEMA schemaname [ AUTHORIZATION benutzername ] [ schemaelement [ ... ] ]
CREATE SCHEMA AUTHORIZATION benutzername [ schemaelement [ ... ] ]
```

Beschreibung

CREATE SCHEMA erzeugt ein neues Schema in der aktuellen Datenbank. Der Name muss sich von allen anderen Schemas in der aktuellen Datenbank unterscheiden.

Ein Schema ist im Prinzip ein Namensraum: Es enthält benannte Objekte (Tabellen, Datentypen, Funktionen und Operatoren), die die gleichen Namen haben können wie andere Objekte in Schemas. Auf benannte Objekte wird zugegriffen, indem man entweder ihre Namen mit dem Schemanamen als Präfix „qualifiziert“ oder die gewünschten Schemas in den Suchpfad einfügt.

Wahlweise kann CREATE SCHEMA Unterbefehle enthalten, um Objekte in dem neuen Schema zu erzeugen. Die Unterbefehle werden im Prinzip genauso wie Befehle behandelt, die getrennt nach der Erzeugung des Schemas ausgeführt werden, außer dass, wenn die AUTHORIZATION-Klausel verwendet wird, alle erzeugte Objekte dann diesem Benutzer gehören.

Um ein Schema zu erzeugen, muss der Benutzer das Privileg CREATE für die aktuelle Datenbank haben. (Superuser umgehen diese Prüfung natürlich.)

Parameter

schemaname

Der Name des zu erzeugenden Schemas. Wenn er weggelassen wird, dann erhält das Schema den Namen des aktuellen Benutzers.

benutzername

Der Name des Benutzers, dem das Schema gehören soll. Wenn er weggelassen wird, gehört das Schema dem Benutzer, der den Befehl ausführt. Nur Superuser können Schemas erzeugen, die anderen Benutzern gehören.

schemaelement

Ein SQL-Befehl, der ein Objekt definiert, das im Schema erzeugt werden soll. Gegenwärtig sind nur die Befehle CREATE TABLE, CREATE VIEW und GRANT innerhalb von CREATE SCHEMA erlaubt. Andere Objekte können in getrennten Befehlen, nachdem das Schema erzeugt worden ist, definiert werden.

Meldungen

CREATE SCHEMA

Meldung, wenn das Schema erfolgreich erzeugt wurde.

ERROR: namespace "*schemaname*" already exists

Meldung, wenn das angegebene Schema bereits existiert.

Beispiele

Dieser Befehl erzeugt ein einfaches Schema:

```
CREATE SCHEMA mein_schema;
```

Dieser Befehl erzeugt ein Schema für den Benutzer fred; das Schema wird ebenfalls fred heißen:

```
CREATE SCHEMA AUTHORIZATION fred;
```

Dieser Befehl erzeugt ein Schema mit einer Tabelle und einer Sicht darin:

```
CREATE SCHEMA hollywood
    CREATE TABLE filme (titel text, kinostart date, preise text[])
    CREATE VIEW gewinner AS
        SELECT titel, kinostart FROM filme WHERE preise IS NOT NULL;
```

Beachten Sie, dass die einzelnen Unterbefehle nicht durch Semikolons getrennt sind.

Das Gleiche wie oben kann auch mit folgenden Befehlen erreicht werden:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.filme (titel text, kinostart date, preise text[]);
CREATE VIEW hollywood.gewinner AS
    SELECT titel, kinostart FROM hollywood.filme WHERE preise IS NOT NULL;
```

Kompatibilität

Der SQL-Standard erlaubt zusätzlich die Klausel `DEFAULT CHARACTER SET` im Befehl `CREATE SCHEMA` sowie mehr Unterbefehlsarten, als gegenwärtig von PostgreSQL akzeptiert werden.

Der SQL-Standard schreibt vor, dass die Unterbefehle in `CREATE SCHEMA` in beliebiger Reihenfolge auftreten können. Die gegenwärtige Implementierung in PostgreSQL kann nicht alle Fälle von Vorwärtsverweisen in den Unterbefehlen verarbeiten; daher kann es manchmal nötig sein, die Befehle umzusortieren, um Vorwärtsverweise zu vermeiden.

Nach dem SQL-Standard ist der Eigentümer eines Schemas der Eigentümer aller Objekte im Schema. PostgreSQL erlaubt, dass Schemas Objekte enthalten können, die anderen Benutzern als dem Schemaeigentümer gehören. Das kann aber nur passieren, wenn der Schemaeigentümer jemandem anders das Privileg `CREATE` gewährt.

Siehe auch

`DROP SCHEMA` *(Seite: 724)*

CREATE SEQUENCE

Name

CREATE SEQUENCE – definiert einen neuen Sequenzgenerator

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE seqname [ INCREMENT inkrement ]
    [ MINVALUE minwert ] [ MAXVALUE maxwert ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Beschreibung

`CREATE SEQUENCE` erzeugt einen neuen Sequenzgenerator. Dazu wird eine neue spezielle einzeilige Tabelle mit dem Namen *seqname* erzeugt und initialisiert. Der Eigentümer des Sequenzgenerators wird der Benutzer sein, der den Befehl ausführt.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE SEQUENCE mein_schema.meine_sequenz ...`), wird die Sequenz im angegebenen Schema erzeugt, ansonsten

wird sie im aktuellen Schema erzeugt. Temporäre Sequenzen existieren in einem besonderen Schema, also darf bei der Erzeugung einer temporären Sequenz kein Schemaname angegeben werden. Der Sequenzname muss sich von den Namen aller Sequenzen, Tabellen, Indexe und Sichten im selben Schema unterscheiden.

Nachdem eine Sequenz erzeugt worden ist, kann man mit den Funktionen `nextval`, `currval` und `setval` auf sie zugreifen und sie bearbeiten. Diese Funktionen sind in Abschnitt SET beschrieben.

Obwohl man eine Sequenz nicht direkt aktualisieren kann, können Sie mit einer Anfrage wie

```
SELECT * FROM seqname;
```

die Parameter und den aktuellen Zustand der Sequenz einsehen. Insbesondere zeigt das Feld `last_value` einer Sequenz den letzten von irgendeiner Sitzung erzeugten Wert. (Natürlich könnte dieser Wert bei der Ausgabe schon veraltet sein, wenn andere Sitzungen währenddessen `nextval` aufrufen.)

Parameter

`TEMPORARY` oder `TEMP`

Wenn angegeben, wird das Sequenzobjekt nur für diese Sitzung erzeugt und wird automatisch entfernt, wenn die Sitzung beendet wird. Bestehende permanente Sequenzen mit demselben Namen sind (in dieser Sitzung) nicht sichtbar, solange die temporäre Sequenz existiert, außer wenn man auf sie mit schemaqualifizierten Namen zugreift.

`seqname`

Der Name der zu erzeugenden Sequenz (möglicherweise mit Schemaqualifikation).

`inkrement`

Die optionale Klausel `INCREMENT` *`inkrement`* gibt an, welcher Wert zum aktuellen Sequenzwert addiert wird, um einen neuen Wert zu ermitteln. Ein positiver Wert ergibt eine wachsende Sequenz, ein negativer eine fallende Sequenz. Der Vorgabewert ist 1.

`minwert`

Die optionale Klausel `MINVALUE` *`minwert`* gibt den niedrigsten Wert an, den die Sequenz erzeugen kann. Die Vorgabewerte sind 1 für wachsende Sequenzen und -263-1 für fallende Sequenzen.

`maxwert`

Die optionale Klausel `MAXVALUE` *`maxwert`* gibt den höchsten Wert an, den die Sequenz erzeugen kann. Die Vorgabewerte sind 263-1 für wachsende Sequenzen und -1 für fallende Sequenzen.

`start`

Mit der optionalen Klausel `START` *`start`* kann die Sequenz mit einer beliebigen Zahl beginnen. Der Standardanfangswert ist *`minwert`* für wachsende Sequenzen und *`maxwert`* für fallende Sequenzen.

`cache`

Die optionale Klausel `CACHE` *`cache`* gibt an, wie viele Sequenzzahlen im Vorab erzeugt und für schnelleren Zugriff im Cache zwischengespeichert werden sollen. Der Mindestwert ist 1 (nur ein Wert wird auf einmal erzeugt, d.h. kein Cache) und das ist auch der Vorgabewert.

`CYCLE`

Die Option `CYCLE` erlaubt der Sequenz beim Erreichen von *`maxwert`* oder *`minwert`* am anderen Ende des Wertebereichs wieder anzufangen. Wenn die Grenze überschritten wird, wird der nächste erzeugte Sequenzwert *`minwert`* bzw. *`maxwert`* sein. Wenn die Grenze erreicht ist, aber `CYCLE` nicht angegeben wurde, dann ergeben darauf folgende Aufrufe von `nextval` einen Fehler.

Meldungen

`CREATE SEQUENCE`

Meldung, wenn der Befehl erfolgreich war.

`ERROR: Relation '`*`seqname`*`' already exists`

Eine Sequenz, Tabelle, Sicht oder ein Index mit dem angegebenen Namen existiert bereits.

`ERROR: DefineSequence: MINVALUE (`*`start`*`) can't be >= MAXVALUE (`*`max`*`)`
`ERROR: DefineSequence: START value (`*`start`*`) can't be < MINVALUE (`*`min`*`)`

Der angegebene Startwert liegt außerhalb des gültigen Bereichs.

`ERROR: DefineSequence: MINVALUE (`*`min`*`) can't be >= MAXVALUE (`*`max`*`)`

Die Minimum- und Maximumwerte widersprechen einander.

Hinweise

Die Arithmetik von Sequenzen wird mit dem Typ `bigint` durchgeführt, also kann der Bereich nicht außerhalb des Bereichs einer 8-Byte-Ganzzahl liegen (-9223372036854775808 bis +9223372036854775807). Wenn auf älteren Plattformen der Compiler 8-Byte-Ganzzahlen nicht unterstützt, haben Sequenzen den Gültigkeitsbereich des Typs `integer` (-2147483648 bis +2147483647).

Es kann unerwartete Ergebnisse geben, wenn die Einstellung für *`cache`* größer als 1 ist und die Sequenz von mehr als einer Sitzung gleichzeitig verwendet wird. Jede Sitzung wird bei einem Zugriff auf die Sequenz mehrere aufeinander folgende Sequenzwerte abrufen und zwischenspeichern und das Feld `last_value` der Sequenz entsprechen setzen. Danach liefern die nächsten *`cache`*-1-Aufrufe von `nextval` einfach die zwischengespeicherten, Werte ab, ohne das Sequenzobjekt zu bemühen. Daher sind alle zwischengespeicherten aber nicht verwendeten Zahlen am Ende der Sitzung verloren, wodurch sich „Löcher“ in der von der Sequenz erzeugten Zahlenfolge ergeben.

Außerdem kann es sein, dass, wenn man alle Sitzungen betrachtet, die Werte nicht in Reihe zurückgeben werden; trotzdem erhalten alle Sitzungen garantiert unterschiedliche Werte. Zum Beispiel: Der Cachewert ist 10. Sitzung A reserviert vielleicht die Werte 1 bis 10 und gibt aus `nextval` 1 zurück. Sitzung B reserviert dann vielleicht die Werte 11 bis 20 und gibt aus `nextval` 11 zurück, bevor Sitzung A aus `nextval` 2 zurückgegeben hat. Mit einer Cache-Einstellung von 1 kann man sicher sein, dass alle `nextval`-Werte der Reihe nach zurückgegeben werden; mit einer Cache-Einstellung größer als 1 sollte man nur davon ausgehen, dass alle Werte unterschiedlich sind, aber nicht der Reihe nach zurückgegeben werden. Außerdem zeigt `last_value` den letzten Wert an, den irgendeine Sitzung reserviert hat, egal, ob dieser schon von `nextval` zurückgegeben wurde.

Eine andere Folge daraus ist, dass ein Aufruf von `setval` bei einer solchen Sequenz von anderen Sitzungen erst berücksichtigt wird, wenn diese alle vorab reservierten Werte aufgebraucht haben.

Beispiele

Der folgende Befehl erzeugt eine wachsende Sequenz namens `zahlenfolge`, die bei 101 anfängt:

```
CREATE SEQUENCE zahlenfolge START 101;
```

Der folgende Befehl ermittelt die nächste Zahl dieser Sequenz:

```
SELECT nextval('zahlenfolge');
```

Hier wird diese Sequenz in einem INSERT-Befehl verwendet:

```
INSERT INTO händler VALUES (nextval('zahlenfolge'), 'irgendwas');
```

Hier wird die Sequenz nach einem COPY FROM aktualisiert:

```
BEGIN;
COPY händler FROM 'eingabedatei';
SELECT setval('zahlenfolge', max(nr)) FROM händler;
END;
```

Kompatibilität

Der Befehl CREATE SEQUENCE ist eine PostgreSQL-Erweiterung.

Siehe auch

DROP SEQUENCE *(Seite: 725)*

CREATE TABLE

Name

CREATE TABLE – definiert eine neue Tabelle

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE tabellenname (
    { spaltenname datentyp [ DEFAULT vorgabeausdruck ] [ spalten_constraint [,
... ] ]
    | tabellen_constraint }  [, ... ]
)
[ INHERITS ( elterntabelle [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
```

wobei *spalten_constraint* Folgendes sein kann:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |
  CHECK (ausdruck) |
  REFERENCES reftabelle [ ( refspalte ) ] [ MATCH FULL | MATCH PARTIAL | MATCH
SIMPLE ]
    [ ON DELETE aktion ] [ ON UPDATE aktion ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

und *tabellen_constraint* Folgendes sein kann:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( spalten_name [, ... ] ) |
  PRIMARY KEY ( spalten_name [, ... ] ) |
  CHECK ( ausdruck ) |
  FOREIGN KEY ( spalten_name [, ... ] ) REFERENCES reftabelle [ ( refspalte [,
... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE aktion ] [ ON UPDATE
aktion ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Beschreibung

`CREATE TABLE` erzeugt eine neue, anfänglich leere Tabelle in der aktuellen Datenbank. Der Eigentümer der Tabelle wird der Benutzer sein, der den Befehl ausführt.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE TABLE mein_schema.meine_tabelle ...`), wird die Tabelle im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt. Temporäre Tabellen existieren in einem besonderen Schema, also darf bei der Erzeugung einer temporären Tabelle kein Schemaname angegeben werden. Der Tabellenname muss sich von den Namen aller Tabellen, Sequenzen, Indexe und Sichten im selben Schema unterscheiden.

`CREATE TABLE` erzeugt außerdem automatisch einen Datentyp, der den zusammengesetzten Datentyp einer Tabellenzeile darstellt. Daher kann eine Tabelle auch nicht denselben Namen wie ein Datentyp im selben Schema haben.

Eine Tabelle kann höchstens 1600 Spalten haben. (In der Praxis ist die Grenze niedriger, weil die Gesamtgröße einer Zeile begrenzt ist.)

Die optionalen Constraint-Klauseln geben Constraints oder Prüfungen an, die von neuen oder aktualisierten Zeilen erfüllt werden müssen, damit die Einfüge- oder Aktualisierungsaktion durchgeführt werden kann. Ein Constraint ist ein SQL-Objekt, das auf verschiedene Weisen gültige Werte für die Tabelle bestimmen kann.

Constraints können auf zwei Arten definiert werden: als Tabellen-Constraints und als Spalten-Constraints. Ein Spalten-Constraint wird als Teil einer Spaltendefinition angegeben. Ein Tabellen-Constraint gehört nicht zu einer bestimmten Spalte und kann mehr als eine Spalte verwenden. Jeder Spalten-Constraint kann auch als Tabellen-Constraint geschrieben werden; ein Spalten-Constraint ist nur eine alternative Schreibweise, wenn der Constraint sich auf nur eine Spalte auswirkt.

Parameter

`[LOCAL] TEMPORARY` oder `[LOCAL] TEMP`

Wenn angegeben, wird die Tabelle als temporäre Tabelle erzeugt. Temporäre Tabellen werden automatisch entfernt, wenn die Sitzung beendet wird. Bestehende permanente Tabellen mit demselben Namen sind in der aktuellen Sitzung nicht sichtbar, solange die temporäre Tabelle existiert, außer wenn man auf sie mit schemaqualifizierten Namen zugreift. Indexe für temporäre Tabellen sind automatisch ebenfalls temporär.

Das Wort `LOCAL` ist optional. Siehe aber unter `Kompatibilität` *(Seite: 693)*.

`tabellenname`

Der Name der zu erzeugenden Tabelle (möglicherweise mit Schemaqualifikation).

spaltenname

Der Name einer in der Tabelle zu erzeugenden Spalte.

datentyp

Der Datentyp der Spalte. Das kann auch eine Arrayangabe sein.

DEFAULT *vorgabeausdruck*

Die DEFAULT-Klausel weist der Spalte, in deren Definition sie auftritt, einen Vorgabewert zu. Der Wert kann ein beliebiger Ausdruck ohne Variablen sein (keine Unteranfragen oder Verweise auf andere Spalten in derselben Tabelle). Der Datentyp des Ausdrucks muss mit dem Datentyp der Spalte übereinstimmen.

Der Vorgabewert wird bei jeder Einfügeoperation verwendet, die keinen Wert für die Spalte angibt. Wenn für eine Spalte kein Vorgabewert definiert ist, dann ist der NULL-Wert der Vorgabewert.

INHERITS (*elterntabelle* [, ...])

Die optionale INHERITS-Klausel gibt eine Liste von Tabellen an, von denen die neue Tabelle alle Spalten erbt. Wenn der gleiche Spaltenname in mehr als einer Elterntabelle existiert und die Datentypen nicht in allen übereinstimmen, ist das ein Fehler. Wenn es keinen Konflikt gibt, werden die doppelten Spalten in der neuen Tabelle zu einer einzelnen verschmolzen. Wenn die Spaltenliste der neuen Tabelle eine Spalte enthält, die auch geerbt ist, muss der Datentyp dabei ebenfalls übereinstimmen und die Spaltendefinitionen werden in eine verschmolzen. Die geerbten und neuen Spaltendefinitionen mit demselben Namen müssen jedoch nicht die gleichen Constraints haben; alle Constraints aus allen Definitionen werden zusammengefasst und alle auf die neue Spalte angewendet. Wenn die neue Tabelle ausdrücklich einen Vorgabewert für die Spalte angibt, hat dieser Vorgabewert vor allen geerbten Vorgabewerten Vorrang. Ansonsten müssen alle Elterntabellen, die einen Vorgabewert angeben, alle denselben haben oder ein Fehler wird ausgelöst.

WITH OIDS
WITHOUT OIDS

Diese optionale Klausel gibt an, ob die Zeilen in der neuen Tabelle OIDs haben sollten. Wenn nichts angegeben ist, werden OIDs erzeugt. (Wenn die Tabelle von einer Tabelle erbt, die OIDs hat, dann wird immer WITH OIDS angenommen, selbst wenn WITHOUT OIDS angegeben wurde.)

Wenn man WITHOUT OIDS angibt, spart man die Erzeugung von OIDs für die Zeilen der Tabelle. Das kann sich bei großen Tabellen lohnen, weil es den OID-Verbrauch verringert und dadurch nicht zum Überlauf des 32-Bit-OID-Zählers beiträgt. Wenn der Zähler einmal übergelaufen ist, sind OIDs nicht mehr einmalig, wodurch ihr Nutzen erheblich verringert wird.

CONSTRAINT *constraint_name*

Ein optionaler Name für einen Spalten- oder Tabellen-Constraint. Wenn kein Name angegeben ist, erzeugt das System einen.

NOT NULL

Diese Spalte darf keine NULL-Werte enthalten.

NULL

Diese Spalte darf NULL-Werte enthalten. Das ist die Voreinstellung.

Diese Klausel ist nur für die Kompatibilität mit Datenbanken, die nicht dem SQL-Standard folgen. Sie sollte nicht in neuen Anwendungen verwendet werden.

UNIQUE (Spalten-Constraint)
UNIQUE (*spaltenname* [, ...]) (Tabellen-Constraint)

Der UNIQUE-Constraint gibt an, dass eine Gruppe von einer oder mehreren verschiedenen Spalten in einer Tabelle in allen Zeilen verschiedene Werte enthalten muss. Das Verhalten eines Tabellen-Unique-Constraints ist dasselbe wie bei einem Spalten-Constraint, mit der zusätzlichen Möglichkeit, dass er mehrere Spalten enthalten kann.

NULL-Werte zählen bei einem Unique Constraint nicht als gleich.

Jeder Tabellen-Unique-Constraint muss eine Spaltengruppe nennen, die sich von den Spaltengruppen aller anderen Unique-Constraints und Primärschlüssel-Constraints unterscheidet. (Ansonsten wäre es einfach nochmal derselbe Constraint.)

`PRIMARY KEY` (Spalten-Constraint)
`PRIMARY KEY ( `*`spaltenname`*` [, ... ] )` (Tabellen-Constraint)

Ein Primärschlüssel-Constraint gibt an, dass eine Spalte oder Spalten in allen Zeilen einer Tabelle verschiedene Nicht-NULL-Werte haben müssen. Im Prinzip ist `PRIMARY KEY` einfach eine Kombination aus `UNIQUE` und `NOT NULL`, aber wenn man eine Gruppe von Spalten als Primärschlüssel identifiziert, dann enthält das auch Informationen über das Design des Schemas, weil ein Primärschlüssel anzeigt, dass diese Spalten als eindeutige Identifikation von Zeilen verwendet werden können.

Eine Tabelle kann nur einen Primärschlüssel haben, egal ob als Spalten- oder als Tabellen-Constraint.

Der Primärschlüssel-Constraint sollte eine Spaltengruppe nennen, die sich von den Spaltengruppen aller Unique-Constraints in derselben Tabelle unterscheidet.

`CHECK (`*`ausdruck`*`)`

Die `CHECK`-Klausel gibt einen Ausdruck an, der ein Ergebnis vom Typ `boolean` zurückgibt, den eine neue oder aktualisierte Zeile erfüllen muss, damit die Einfüge- oder Aktualisierungsoperation zugelassen wird. Ein Check-Constraint, der als Spalten-Constraint angegeben ist, sollte nur den Wert dieser Spalte verwenden; ein Ausdruck, der in einem Tabellen-Constraint vorkommt, kann dagegen mehrere Spalten verwenden.

Gegenwärtig kann ein `CHECK`-Ausdruck keine Unteranfragen enthalten oder auf Variablen außer den Spalten der aktuellen Zeile zugreifen.

`REFERENCES `*`reftabelle`*` [ ( `*`refspalte`*` ) ] [ MATCH `*`match_typ`*` ] [ ON DELETE `*`aktion`*` ] [ ON UPDATE `*`aktion`*` ]` (Spalten-Constraint)
`FOREIGN KEY ( `*`spalte`*` [, ... ] ) REFERENCES `*`reftabelle`*` [ ( `*`refspalte`*` [, ... ] ) ] [ MATCH `*`match_typ`*` ] [ ON DELETE `*`aktion`*` ] [ ON UPDATE `*`aktion`*` ]` (Tabellen-Constraint)

Dieses Klauseln erzeugen einen Fremdschlüssel-Constraint, welcher bestimmt, dass eine Gruppe von Spalten (oder eine Spalte) der neuen Tabelle nur Werte enthalten darf, die mit Werte in den Spalten *`refspalte`* einer anderen Tabelle *`reftabelle`* übereinstimmen. Wenn *`refspalte`* weggelassen wird, dann wird der Primärschlüssel von *`reftabelle`* verwendet. Die Spalten, auf die der Constraint in der anderen Tabelle verweist, müssen die Spalten eines Unique Constraint oder des Primärschlüssels der Tabelle sein.

Ein in diese Spalten eingefügter Wert wird mit den Werten der anderen Tabelle anhand des angegebenen Match-Typs verglichen. Es gibt drei Match-Typen: `MATCH FULL`, `MATCH PARTIAL` und `MATCH SIMPLE`; Letzterer ist die Voreinstellung, wenn nichts angegeben wurde. Bei `MATCH FULL` dürfen keine Spalten eines mehrspaltigen Fremdschlüssels den NULL-Wert haben, außer wenn alle Spalten im Fremdschlüssel den NULL-Wert haben. Bei `MATCH SIMPLE` können einige Spalten im Fremdschlüssel den NULL-Wert haben und andere nicht. `MATCH PARTIAL` ist noch nicht implementiert.

Wenn die Daten in der Tabelle, auf die der Fremdschlüssel verweist, geändert werden, dann werden in der neuen Tabelle bestimmte Aktionen ausgeführt. Die Klausel `ON DELETE` bestimmt, was getan werden soll, wenn eine Zeile in der Tabelle, auf die ein Wert verweist, gelöscht werden soll. Die Klausel `ON UPDATE` bestimmt gleichermaßen, was getan werden soll, wenn eine Zeile, auf die ein Wert verweist, aktualisiert wird und so einen neuen Wert erhält. Die möglichen Aktionen für beide Klauseln sind folgende:

`NO ACTION`

Erzeuge einen Fehler, der anzeigt, dass der Lösch- oder Aktualisierungsvorgang einen Fremdschlüssel verletzen würde. Das ist die Standardaktion.

`RESTRICT`

Das Gleiche wie `NO ACTION`.

`CASCADE`

Lösche alle Zeilen, die auf die zu löschende Zeile verweisen, beziehungsweise ändere alle Zeilen, die auf die zu aktualisierende Zeile verweisen, in den neuen Wert.

`SET NULL`

Setze die Spalten, die auf die zu löschende oder aktualisierende Zeile verweisen, auf den NULL-Wert.

`SET DEFAULT`

Setze die Spalten, die auf die zu löschende oder aktualisierende Zeile verweisen, auf ihren Vorgabewert.

Wenn eine Primärschlüsselspalte oft aktualisiert wird, sollte man eventuell einen Index für die Spalten des Fremdschlüssels erzeugen, damit die Aktionen `NO ACTION` und `CASCADE` für die Fremdschlüsselspalte effizienter ausgeführt werden können.

`DEFERRABLE`
`NOT DEFERRABLE`

Wenn `DEFERRABLE` angegeben ist, kann die Prüfung eines Constraints bis an das Ende der Transaktion verschoben werden. Ein Constraint, der nicht verschoben werden kann, wird sofort nach jedem Befehl geprüft. Wenn die Prüfung verschoben werden kann, kann das mit dem Befehl `SET CONSTRAINTS` *(Seite: 773)* kontrolliert werden. `NOT DEFERRABLE` (nicht verschiebbar) ist die Voreinstellung. Gegenwärtig wird diese Klausel nur bei Fremdschlüssel-Constraints akzeptiert. Alle anderen Constraint-Typen können nicht verschoben werden.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

Wenn ein Constraint verschoben werden kann, dann bestimmt diese Klausel, wann der Constraint in der Standardeinstellung geprüft wird. Im Fall von `INITIALLY IMMEDIATE` (die Voreinstellung) wird er nach jedem Befehl geprüft, im Fall von `INITIALLY DEFERRED` erst am Ende der Transaktion. Mit dem Befehl `SET CONSTRAINTS` *(Seite: 773)* kann die Prüfzeit geändert werden.

Meldungen

`CREATE TABLE`

Meldung, wenn die Tabelle erfolgreich erzeugt wurde.

Hinweise

- Wenn eine Anwendung die OIDs verwendet, um Tabellenzeilen eindeutig zu identifizieren, sollte ein Unique Constraint für die Spalte `oid` dieser Tabelle erzeugt werden, damit die OIDs die Zeilen immer noch eindeutig bestimmen, wenn der Zähler überlaufen sollte. Sie sollten nicht davon ausgehen, dass OIDs in allen Tabellen einmalig sind; wenn Sie eine datenbankweite eindeutige Identifikation von Zeilen benötigen, verwenden Sie dazu die Kombination aus `tableoid` und der Zeilen-OID. (In der Zukunft wird es wahrscheinlich einen getrennten OID-Zähler für jede Tabelle geben, und dann ist es *notwendig*, nicht nur empfehlenswert, `tableoid` für eine datenbankweit eindeutige Identifikation von Zeilen zu verwenden.)

Tipp

`WITHOUT OIDS` sollten Sie nicht verwenden, wenn die Tabelle keine Primärschlüssel hat, weil es ohne OID und eindeutigen Schlüssel schwierig wird, bestimmte Zeilen eindeutig zu identifizieren.

- PostgreSQL erzeugt für jeden Unique Constraint und Primärschlüssel automatisch einen Index, um die Funktion des Constraints zu unterstützen. Es ist also nicht nötig, für Primärschlüsselspalten noch einen ausdrücklichen Index zu erzeugen.

- Unique Constraints und Primärschlüssel werden in der gegenwärtigen Implementierung nicht vererbt. Dadurch wird die Kombination aus Vererbung und Unique Constraint ziemlich unbrauchbar.

Beispiele

Erzeuge eine Tabelle `filme` und eine Tabelle `verleihe`:

```
CREATE TABLE filme (
    code        char(5) CONSTRAINT schlüssel PRIMARY KEY,
    titel       varchar(40) NOT NULL,
    vid         integer NOT NULL,
    prod_datum  date,
    genre       varchar(10),
    länge       interval hour to minute
);
CREATE TABLE verleihe (
     vid    integer PRIMARY KEY DEFAULT nextval('sequenz'),
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Erzeuge eine Tabelle mit einem zweidimensionalen Array:

```
CREATE TABLE array (
    vektor  int[][]
);
```

Definiere die Tabelle `filme` mit einem Tabellen-Unique-Constraint. Tabellen-Unique-Constraints können eine oder mehrere Spalten einer Tabelle umfassen.

```
CREATE TABLE filme (
    code        char(5),
    titel       varchar(40),
    vid         integer,
    prod_datum  date,
    genre       varchar(10),
    länge       interval hour to minute
    CONSTRAINT produktion UNIQUE (prod_datum)
);
```

Definiere eine Tabelle mit Spalten-Check-Constraint:

```
CREATE TABLE verleihe (
    vid     integer CHECK (vid > 100),
    name    varchar(40)
);
```

Definiere eine Tabelle mit Tabellen-Check-Constraint:

```
CREATE TABLE verleihe (
```

```
    vid     integer,
    name    varchar(40)
    CONSTRAINT con1 CHECK (vid > 100 AND name <> '')
);
```

Definiere die Tabelle `filme` mit einem Primärschlüssel-Tabellen-Constraint. Primärschlüssel-Tabellen-Constraints können eine oder mehrere Spalten einer Tabelle umfassen.

```
CREATE TABLE filme (
    code        char(5),
    titel       varchar(40),
    vid         integer,
    prod_datum  date,
    genre       varchar(10),
    länge       interval hour to minute,
    CONSTRAINT code_titel PRIMARY KEY(code, titel)
);
```

Definiere die Tabelle `verleihe` mit einem Primärschlüssel. Die folgenden beiden Beispiele sind gleichbedeutend, das erste verwendet die Tabellen-Constraint-Schreibweise, das zweite die Spalten-Constraint-Schreibweise.

```
CREATE TABLE verleihe (
    vid     integer,
    name    varchar(40),
    PRIMARY KEY(vid)
);
CREATE TABLE verleihe (
    vid     integer PRIMARY KEY,
    name    varchar(40)
);
```

Folgender Befehl weist der Spalte `name` eine Konstante als Vorgabewert zu, sorgt dafür, dass der Vorgabewert der Spalte `vid` der nächste Wert aus einem Sequenzobjekt ist, und richtet es ein, dass der Vorgabewert der Spalte `geändert` die Zeit ist, zu der die Zeile eingefügt wird.

```
CREATE TABLE verleihe (
    name      varchar(40) DEFAULT 'Luso-Film',
    vid       integer DEFAULT nextval('verleih_sequenz'),
    geändert  timestamp DEFAULT current_timestamp
);
```

Definiere die Tabelle `verleihe` mit zwei `NOT NULL`-Spalten-Constraints, von denen einer einen ausdrücklichen Namen erhält:

```
CREATE TABLE verleihe (
    vid     integer CONSTRAINT nicht_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Definiere einen Unique Constraint für die Spalte name:

```
CREATE TABLE verleihe (
    vid     integer,
    name    varchar(40) UNIQUE
);
```

Das Gleiche kann folgendermaßen als Tabellen-Constraint geschrieben werden:

```
CREATE TABLE verleihe (
    vid     integer,
    name    varchar(40),
    UNIQUE (name)
);
```

Kompatibilität

Der Befehl CREATE TABLE ist konform mit dem SQL-Standard, mit den im Folgenden beschriebenen Ausnahmen.

Temporäre Tabellen

Obwohl die Syntax von CREATE TEMPORARY TABLE dem SQL-Standard ähnelt, ist die Auswirkung nicht dieselbe. Im Standard werden temporäre Tabellen einmal definiert und existieren dann automatisch (mit leerem Inhalt) in jeder Sitzung, die sie benötigt. In PostgreSQL muss dagegen jede Sitzung CREATE TEMPORARY TABLE ausführen, wenn sie eine temporäre Tabelle benötigt. Dadurch kann jede Sitzung denselben Namen für temporäre Tabellen mit unterschiedlichen Aufgaben verwenden, wohingegen nach dem SQL-Standard alle Instanzen einer bestimmten temporären Tabelle die gleiche Struktur haben.

Das vom Standard vorgeschriebene Verhalten für temporäre Tabellen wird weithin ignoriert. Das Verhalten von PostgreSQL in diesem Punkt ist vergleichbar mit mehreren anderen SQL-Datenbanken.

Der im Standard angegebene Unterschied zwischen globalen und lokalen temporären Dateien ist in PostgreSQL nicht vorhanden, da diese Unterscheidung vom Konzept des Moduls abhängt, aber PostgreSQL hat keine Module.

Spalten-Check-Constraints

Der SQL-Standard bestimmt, dass CHECK-Spalten-Constraints nur die Spalte, für die sie gelten, verwenden dürfen; nur CHECK-Tabellen-Constraints dürfen auf mehrere Spalten zugreifen. PostgreSQL setzt diese Einschränkung nicht durch; es behandelt Spalten- und Tabellen-Constraints gleich.

NULL-„Constraint“

Der NULL-„Constraint“ (eigentlich ein Nicht-Constraint) ist eine PostgreSQL-Erweiterung des SQL-Standards, die für die Kompatibilität mit anderen Datenbanken gedacht ist (und als Gegenstück für den NOT NULL-Constraint). Da er das Standardverhalten jeder Spalte ausdrückt, ist er nur Dekoration.

Vererbung

Mehrfache Vererbung mit der INHERITS-Klausel ist eine PostgreSQL-Erweiterung. Der SQL-Standard definiert einfache Vererbung mit anderer Syntax und Bedeutung. Vererbung im SQL-Stil wird vom PostgreSQL noch nicht unterstützt.

OIDs

Das PostgreSQL-Konzept der OIDs ist nicht standardisiert.

Siehe auch

ALTER TABLE *(Seite: 628)*, DROP TABLE *(Seite: 726)*

CREATE TABLE AS

Name

CREATE TABLE AS – erzeugt eine neue Tabelle aus den Ergebnissen einer Anfrage

Synopsis

```
CREATE [ [ LOCAL ] { TEMPORARY | TEMP } ] TABLE tabellenname [ (spaltenname [,
...] ) ]
    AS anfrage
```

Beschreibung

CREATE TABLE AS erzeugt eine neue Tabelle und füllt sie mit den Daten, die ein SELECT berechnet hat. Die Tabellenspalten haben die gleichen Namen und Datentypen wie das Ergebnis des SELECT, aber die Spaltennamen können auch ausdrücklich angegeben werden.

CREATE TABLE AS sieht vielleicht so aus, als würde man eine Sicht erzeugen, aber es gibt einen erheblichen Unterschied: Es erzeugt eine neue Tabelle und wertet die Anfrage einmal aus, um die neue Tabelle mit Anfangsdaten zu füllen. Die neue Tabelle verfolgt aber keine späteren Änderungen in den Quelltabellen der Anfrage. Im Gegensatz dazu wird bei einer Sicht der SELECT-Befehl bei jeder Anfrage neu ausgewertet.

Parameter

[LOCAL] TEMPORARY oder [LOCAL] TEMP

Wenn angegeben, wird die Tabelle als temporäre Tabelle erzeugt. Weitere Einzelheiten dazu finden Sie bei CREATE TABLE *(Seite: 686)*.

tabellenname

Der Name der zu erzeugenden Tabelle (möglicherweise mit Schemaqualifikation).

spaltenname

Der Name einer Spalte in der neuen Tabelle. Wenn keine Spaltennamen angegeben sind, werden die Namen der Ergebnisspalten der Anfrage übernommen.

anfrage

Ein Anfragebefehl (das heißt ein SELECT). Einzelheiten zur erlaubten Syntax erhalten Sie unter SELECT *(Seite: 759)*.

Meldungen

Eine Zusammenfassung der möglichen Meldungen erhalten Sie unter CREATE TABLE *(Seite: 686)* und SELECT *(Seite: 759)*.

Hinweise

Dieser Befehl hat dieselbe Funktionalität wie SELECT INTO *(Seite: 769)*, sollte aber bevorzugt verwendet werden, weil er nicht so einfach mit anderen Verwendungszwecken der Syntax SELECT ... INTO verwechselt werden kann.

Kompatibilität

Dieser Befehl wurde Oracle nachempfunden. Es gibt keinen Befehl mit der gleichen Funktionalität im SQL-Standard. Eine Kombination aus CREATE TABLE und INSERT ... SELECT erreicht jedoch dasselbe Ziel mit nur geringem Mehraufwand.

Siehe auch

CREATE TABLE *(Seite: 686)*, CREATE VIEW *(Seite: 704)*, SELECT *(Seite: 759)*, SELECT INTO *(Seite: 769)*

CREATE TRIGGER

Name

CREATE TRIGGER – definiert einen neuen Trigger

Synopsis

```
CREATE TRIGGER name { BEFORE | AFTER } { ereignis [OR ...] }
    ON tabelle FOR EACH ROW
    EXECUTE PROCEDURE funktion ( [ argument [, ...] ] )
```

Beschreibung

CREATE TRIGGER erzeugt einen neuen Trigger. Der Trigger wird zur angegebenen Tabelle gehören und die angegebene Funktion ausführen, wenn bestimmte Ereignisse auftreten.

Ein Trigger kann entweder vor der eigentlichen Operation ausgelöst werden (bevor Constraints geprüft werden und das INSERT, UPDATE oder DELETE versucht wird) oder nachdem die Operation abgeschlossen wurde (nachdem die Constraints geprüft worden sind und das INSERT, UPDATE oder DELETE abge-

schlossen wurde). Wenn der Trigger vor dem Ereignis ausgelöst wird, kann der Trigger die eigentliche Operation für die Zeile auslassen oder die einzufügende Zeile ändern (nur bei `INSERT` und `UPDATE`). Wenn der Trigger nach dem Ereignis ausgelöst wird, sind alle Veränderungen, einschließlich der letzten Einfüge-, Aktualisierungs- oder Löschoperation, im Trigger „sichtbar“.

Wenn mehrere Trigger derselben Art für dasselbe Ereignis definiert sind, werden Sie in alphabetischer Reihenfolge ihrem Namen nach ausgelöst.

`SELECT`-Befehle verändern keine Zeilen und daher kann man keine Trigger für `SELECT`-Ereignisse definieren. Regeln und Sichten sind in solchen Fällen angebrachter.

Weitere Informationen über Trigger finden Sie in Abschnitt SET.

Parameter

`name`

Der Name des neuen Triggers. Er muss sich von den Namen aller Trigger für dieselbe Tabelle unterscheiden.

`ereignis`

Entweder `INSERT`, `UPDATE` oder `DELETE`; dies gibt an, bei welchem Ereignis der Trigger ausgelöst wird. Mehrere Ereignisse können mit `OR` verknüpft werden.

`tabelle`

Der Name der Tabelle (möglicherweise mit Schemaqualifikation), für die der Trigger gilt.

`funktion`

Eine benutzerdefinierte Funktion, die ohne Argumente und mit Rückgabetyp `trigger` deklariert wurde. Diese Funktion wird aufgerufen, wenn der Trigger ausgelöst wird.

`argument`

Argumente, die der Funktion übergeben werden, wenn der Trigger ausgelöst wird. Die Argumente sind Zeichenkettenkonstanten. Einfache Namen und numerische Konstanten sind auch erlaubt, aber sie werden alle in Zeichenketten umgewandelt. Bitte lesen Sie in der Beschreibung der Implementierungssprache der Funktion, wie die Triggerargumente in der Funktion ermittelt werden können; es unterscheidet sich möglicherweise von der Methode für normale Funktionsargumente.

Meldungen

`CREATE TRIGGER`

Meldung, wenn der Trigger erfolgreich erzeugt wurde.

Hinweise

Um einen Trigger erzeugen zu können, muss der Benutzer das Privileg `TRIGGER` für die Tabelle haben.

In PostgreSQL-Versionen vor 7.3 musste die Triggerfunktion mit dem Platzhaltertyp `opaque` als Rückgabetyp deklariert werden, anstatt wie jetzt `trigger`. Um alte Sicherungsdateien laden zu können, akzeptiert `CREATE TRIGGER` Funktionen mit deklariertem Rückgabetyp `opaque`, wird aber eine Hinweismeldung ausgeben und den Rückgabetyp der Funktion in `trigger` ändern.

Beispiele

Abschnitt SET enthält ein vollständiges Beispiel.

Kompatibilität

Der Befehl `CREATE TRIGGER` in PostgreSQL implementiert eine Teilmenge des SQL-Standards. Folgende Funktionalität fehlt:

- SQL erlaubt Trigger, die nur einmal pro Befehl aufgerufen werden (anstatt für jede Tabellenzeile).
- SQL erlaubt Trigger, die nur bei der Aktualisierung von bestimmten Spalten ausgelöst werden (z.B. `AFTER UPDATE OF spalte1, spalte2`).
- SQL erlaubt die Definition von Aliasnamen für die jeweilige „alte“ und „neue“ Zeile oder Tabelle, die dann in der Definition der Triggeraktion verwendet werden kann (z.B. `CREATE TRIGGER ... ON tabelle REFERENCING OLD ROW AS altezeile NEW ROW AS neuezeile ...`). Da man in PostgreSQL die Triggerprozeduren in einer Vielzahl von benutzerdefinierten Sprachen schreiben kann, wird der Datenzugriff je nach Sprache unterschiedlich geregelt.
- PostgreSQL erlaubt als Triggeraktion nur die Ausführung einer benutzerdefinierten Funktion. SQL erlaubt die Ausführung beliebiger SQL-Befehle, wie zum Beispiel `CREATE TABLE`, als Triggeraktion. Diese Beschränkung kann man aber leicht umgehen, indem man eine benuterdefinierte Funktion erzeugt, die die gewünschten Befehle ausführt.

Laut SQL sollen mehrere Trigger in der Reihenfolge ihrer Erzeugung ausgeführt werden. PostgreSQL führt sie in alphabetischer Reihenfolge nach ihren Namen aus, weil es sich damit leichter umgehen lässt.

Siehe auch

`ALTER TRIGGER` *(Seite: 633)*, `CREATE FUNCTION` *(Seite: 664)*, `DROP TRIGGER` *(Seite: 727)*

CREATE TYPE

Name

CREATE TYPE – definiert einen neuen Datentyp

Synopsis

```
CREATE TYPE typname (
    INPUT = eingabefunktion, OUTPUT = ausgabefunktion
      , INTERNALLENGTH = { interne_länge | VARIABLE }
    [ , DEFAULT = vorgabewert ]
    [ , ELEMENT = element ] [ , DELIMITER = trennzeichen ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = ausrichtung ]
    [ , STORAGE = speicherung ]
)
```

```
CREATE TYPE typname AS
    ( attributname datentyp [, ... ] )
```

Beschreibung

`CREATE TYPE` erzeugt einen neuen Datentyp zur Verwendung in der aktuellen Datenbank. Der Benutzer, der den Typ definiert, wird sein Eigentümer.

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE TYPE mein_schema.mein_typ ...`), wird der Typ im angegebenen Schema erzeugt, ansonsten im aktuellen Schema. Der Typname muss sich von den Namen aller bestehenden Typen und Domänen im selben Schema unterscheiden. (Da für jede Tabelle auch ein zugeordneter Datentyp besteht, muss sich der Typname auch von den Namen aller bestehenden Tabellen im Schema unterscheiden.)

Basistypen

Die erste Form von `CREATE TYPE` erzeugt einen neuen Basistyp (skalaren Typ). Dazu müssen vor der Definition des Typs zwei Funktionen registriert werden (mit `CREATE FUNCTION`). Die interne Darstellung des neuen Basistyps wird von *`eingabefunktion`* definiert, welche die externe Darstellung des Typs in eine interne umwandelt, die von den Operatoren und Funktionen für diesen Typ verwendet werden kann. *`ausgabefunktion`* führt die umgekehrte Umwandlung durch. Die Eingabefunktion kann ein Argument vom Typ `cstring` oder drei Argumente der Type `cstring`, `oid` und `integer` haben. Das erste Argument ist der Eingabetext als C-Zeichenkette, das zweite Argument ist der Elementtyp, falls dies ein Arraytyp ist, und das dritte ist der `typmod`-Wert der Zielspalte, falls bekannt. Die Eingabefunktion sollte einen Wert des Datentyps selbst zurückgeben. Die Ausgabefunktion kann ein Argument des neuen Datentyps haben oder zwei Argumente, von denen das zweite vom Typ `oid` ist. Das zweite Argument ist wiederum der Elementtyp bei Arraytypen. Die Ausgabefunktion sollte den Typ `cstring` zurückgeben.

Mittlerweile sollten Sie sich fragen, wie die Eingabe- und Ausgabefunktionen mit Rückgabewerten oder Argumenten des neuen Typs deklariert werden können, wenn Sie vor der Definition des Typs deklariert werden müssen. Die Antwort ist, dass zuerst die Eingabefunktion erzeugt werden muss, dann die Ausgabefunktion und dann der Datentyp. PostgreSQL wird dann zuerst den Namen eines neuen Typs als Rückgabetyp der Eingabefunktion sehen. Dabei erzeugt es eine „Typenhülle“, welche einfach ein Platzhaltereintrag im Systemkatalog ist, und verbindet die Eingabefunktion mit dieser Typenhülle. Genauso wird die Ausgabefunktion mit der (jetzt schon bestehenden) Typenhülle verbunden. Schließlich wird die Hülle von `CREATE TYPE` durch eine vollständige Typendefinition ersetzt und der neue Typ kann verwendet werden.

Ein Basistyp hat entweder eine feste Länge, dann wird *`interne_länge`* auf eine positive ganze Zahl gesetzt, oder eine variable Länge, dann wird *`interne_länge`* durch `VARIABLE` ersetzt. Die interne Darstellung aller Typen mit variabler Länge muss mit einer 4-Byte-Ganzzahl anfangen, die die Gesamtlänge des jeweiligen Werts enthält.

Um anzuzeigen, das der Typ ein Array ist, geben Sie den Typ der Arrayelemente mit dem Schlüsselwort `ELEMENT` an. Um zum Beispiel ein Array aus 4-Byte-Ganzzahlen (`int4`) zu definieren, schreiben Sie `ELEMENT = int4`. Weitere Einzelheiten zu Arraytypen erscheinen unten.

Um das Trennzeichen anzugeben, dass zwischen den Werten in der externen Darstellung von Arrays dieses Typs verwendet wird, setzen Sie *`trennzeichen`* auf ein bestimmtes Zeichen. Das Standardtrennzeichen ist ein Komma (`,`). Beachten Sie, dass das Trennzeichen zum Elementtyp des Arrays gehört, nicht zum Arraytyp selbst.

Wenn der Benutzer wünscht, dass Spalten des neuen Typs nicht den NULL-Wert als Vorgabewert haben, dann kann ein anderer Wert angegeben werden. Geben Sie den Vorgabewert mit dem Schlüsselwort `DEFAULT` an. (Eine für eine bestimmte Spalte angegebene `DEFAULT`-Klausel hat Vorrang vor dem Vorgabewert des Datentyps.)

Die Option PASSEDBYVALUE gibt an, dass dieser Datentyp Wertübergabe statt Referenzübergabe verwendet. Typen, deren interne Darstellung größer ist als der Typ Datum (auf den meisten Maschinen 4 Bytes, auf manchen 8 Bytes), können Wertübergabe nicht verwenden.

Der Parameter *ausrichtung* bestimmt die Ausrichtungserfordernisse des Datentyps. Die erlaubten Werte entsprechen der Ausrichtung auf 1-, 2-, 4- und 8-Byte-Grenzen. Beachten Sie, dass Typen mit variabler Länge mindestens auf 4-Byte-Grenzen ausgerichtet werden müssen, weil sie immer einen int4 als erste Komponente haben.

Der Parameter *speicherung* erlaubt die Auswahl von Speicherungsstrategien für Datentypen mit variabler Länge. (Für Typen mit fester Länge ist nur plain erlaubt.) Wenn plain angegeben ist, werden Werte immer in der Haupttabelle gespeichert und nicht komprimiert. Bei extended wird das System versuchen, lange Datenwerte zu komprimieren und den Wert in einer Nebentabelle zu speichern, wenn er immer noch zu lang ist. Bei external kann der Wert in eine Nebentabelle verschoben werden, es wird aber nicht versucht, ihn zu komprimieren. Bei main kann der Wert komprimiert werden, wird aber bevorzugt nicht in eine Nebentabelle verschoben. (Werte mit dieser Speicherungsstrategie können trotzdem aus der Haupttabelle verschoben werden, wenn die Zeile nicht anders passt, aber sie werden bevorzugt vor Werten mit den Strategien extended und external in der Haupttabelle behalten.)

Zusammengesetzte Typen

Die zweite Form von CREATE TYPE erzeugt einen zusammengesetzten Typ. Ein zusammengesetzter Typ besteht aus einer Liste von Attributnamen und Datentypen. Das ist im Prinzip das Gleiche wie ein Zeilentyp einer Tabelle, aber wenn man CREATE TYPE verwendet, muss man keine wirkliche Tabelle erzeugen, wenn man nur den Typ verwenden möchte. Ein alleinstehender zusammengesetzter Typ ist als Rückgabetyp einer Funktion nützlich.

Arraytypen

Immer wenn ein benutzerdefinierter Basisdatentyp erzeugt wird, erzeugt PostgreSQL automatisch einen dazugehörigen Arraytyp. Der Name dieses Arraytyps ist ein Unterstrich gefolgt vom Namen des Basistyps. Der Parser kennt diese Namenskonvention und wandelt Spaltentypen wie foo[] automatisch in _foo um. Der implizit erzeugte Arraytyp hat variable Länge und verwendet die eingebauten Eingabe- und Ausgabefunktionen array_in und array_out.

Sie könnten sich fragen, warum es eine Option ELEMENT gibt, wenn das System den korrekten Arraytyp automatisch erzeugt. Die Option ELEMENT ist nur nützlich, wenn Sie einen Datentyp mit fester Länge erzeugen, der intern aus einer Anzahl von identischen Elementen besteht, und Sie, neben den anderen Operationen, die Sie für den Datentyp in seiner Gesamtheit vorgesehen haben, auf diese Elemente als Arrayelemente zugreifen wollen. Beim Datentyp name können Sie so zum Beispiel auf die Elemente vom Typ char zugreifen. Bei einem Typ mit 2-D-Punkten könnte man zum Beispiel erlauben, auf die beiden Koordinaten als punkt[0] und punkt[1] zuzugreifen. Diese Funktionalität funktioniert nur bei Datentypen mit fester Länge und nur, wenn dessen interne Form eine Folge von identischen Feldern mit fester Größe ist. Ein Typ mit variabler Länge, auf dessen Elemente mit der Arraysyntax zugegriffen werden soll, muss die allgemeinere, sich aus array_in und array_out ergebende Darstellung verwenden. Aus historischen Gründen (d.h., es ist offensichtlich falsch, aber viel zu spät, um es zu ändern) fangen die Elemente bei Arraytypen mit fester Länge bei 0 an zu zählen, anstatt bei 1 wie bei Arrays mit variabler Länge.

Parameter

typname

Der Name des zu erzeugenden Typs (möglicherweise mit Schemaqualifikation).

interne_länge

Eine numerische Konstante, die die interne Länge des neuen Typs angibt.

eingabefunktion

Der Name einer Funktion, die Daten aus der externen Form des Typs in seine interne Form umwandelt.

ausgabefunktion

Der Name einer Funktion, die Daten aus der internen Form des Typs in eine Form zur Ausgabe umwandelt.

element

Der zu erzeugende Typ ist ein Array; dies gibt den Typ der Arrayelemente an.

trennzeichen

Das Trennzeichen, das zwischen Werten in einem Array aus diesem Typ verwendet werden soll.

vorgabewert

Der Vorgabewert für diesen Datentyp. Wenn nicht angegeben, ist der Vorgabewert der NULL-Wert.

ausrichtung

Die Ausrichtungserfordernisse bei der Speicherung des neuen Datentyps. Mögliche Werte sind `char`, `int2`, `int4` und `double`; die Voreinstellung ist `int4`.

speicherung

Die Speicherungsstrategie für diesen Datentyp. Mögliche Werte sind `plain`, `external`, `extended` und `main`; die Voreinstellung ist `plain`.

attributname

Der Name eines Attributs des zusammengesetzten Datentyps.

datentyp

Der Name eines bestehenden Datentyps.

Meldungen

`CREATE TYPE`

Meldung, wenn der Typ erfolgreich erzeugt wurde.

Hinweise

Der Name eines benutzerdefinierten Typs kann nicht mit einem Unterstrich (_) anfangen und kann nur 62 Zeichen lang sein (oder allgemein ausgedrückt `NAMEDATALEN` – 2, anstatt der `NAMEDATALEN` – 1 Zeichen, die bei anderen Namen erlaubt sind). Typnamen, die mit einem Unterstrich beginnen, sind für intern erzeugte Arraytypnamen reserviert.

In PostgreSQL-Versionen vor 7.3 wurde die Erzeugung einer Typenhülle üblicherweise vermieden, indem bei den Funktionen statt der Verweise auf den noch nicht bestehenden Typ der Pseudotyp `opaque` verwendet wurde. Die Argumente und Ergebnisse vom Typ `cstring` mussten vor Version 7.3 auch als `opaque` deklariert werden. Um alte Sicherungsdateien laden zu können, akzeptiert `CREATE TYPE` Funktionen, die den Typ `opaque` verwenden, wird aber eine Hinweismeldung ausgeben und die Funktionsdeklarationen so ändern, dass sie den korrekten Typ verwenden.

Beispiele

Dieses Beispiel erzeugt den Datentyp `box` und verwendet den Typ dann in einer Tabellendefinition:

```
CREATE TYPE box (
```

```
    INTERNALLENGTH = 16,
    INPUT = meine_box_eingabefunktion,
    OUTPUT = meine_box_ausgabefunktion
);
CREATE TABLE test_box (id integer, beschreibung box);
```

Wenn die interne Struktur von box ein Array von vier Elementen des Typs float4 wäre, könnten wir stattdessen schreiben

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = meine_box_eingabefunktion,
    OUTPUT = meine_box_ausgabefunktion,
    ELEMENT = float4
);
```

Dadurch könnte man auf die Komponenten des Typs mit der Arraysyntax zugreifen. Ansonsten würde sich der Typ genauso wie oben verhalten.

Dieses Beispiel erzeugt einen Large-Object-Typ und verwendet ihn in einer Tabellendefinition:

```
CREATE TYPE largeobj (
    INPUT = lo_datei_in,
    OUTPUT = lo_datei_out,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE large_objects (id integer, obj largeobj);
```

Dieses Beispiel erzeugt einen zusammengesetzten Typ und verwendet ihn in der Definition einer Funktion:

```
CREATE TYPE foo AS (f1 int, f2 text);
CREATE FUNCTION getfoo() RETURNS SETOF foo AS 'SELECT fooid, fooname FROM foo'
LANGUAGE SQL;
```

Weitere Beispiele, einschließlich passender Eingabe- und Ausgabefunktionen, finden Sie in Abschnitt SET.

Kompatibilität

Dieser CREATE TYPE-Befehl ist eine PostgreSQL-Erweiterung. Im SQL-Standard gibt es einen Befehl CREATE TYPE, der sich in den Einzelheiten aber sehr unterscheidet.

Siehe auch

DROP TYPE *(Seite: 729)*

CREATE USER

Name

CREATE USER – definiert ein neues Datenbankbenutzerkonto

Synopsis

```
CREATE USER benutzername [ [ WITH ] option [ ... ] ]
```

wobei *option* Folgendes sein kann:

```
      SYSID uid
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'passwort'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | IN GROUP gruppenname [, ...]
    | VALID UNTIL 'abstime'
```

Beschreibung

`CREATE USER` erzeugt einen neuen Benutzer in einem PostgreSQL-Datenbankcluster. Weitere Informationen über Benutzerverwaltung und Authentifizierung erhalten Sie in Abschnitt SET und Abschnitt SET. Um diesen Befehl verwenden zu können, müssen Sie ein Datenbank-Superuser sein.

Parameter

benutzername

Der Name des Benutzers.

uid

Mit der `SYSID`-Klausel kann man die PostgreSQL-Benutzer-ID des neuen Benutzers bestimmen. Das ist normalerweise nicht notwendig, kann aber nützlich sein, wenn Sie den Eigentümer eines Objekts ohne Eigentümer wiedererzeugen müssen.

Wenn keine ID angegeben ist, wird die höchste bisher vergebene ID plus eins, beginnend bei 100, verwendet.

passwort

Das Passwort des Benutzers. Wenn Sie keine Passwortauthentifizierung verwenden wollen, können Sie diese Option weglassen, aber dann wird der Benutzer nicht verbinden können, wenn Sie doch zur Passwortauthentifizierung wechseln. Das Passwort kann später mit `ALTER USER` *(Seite: 634)* gesetzt oder geändert werden.

`ENCRYPTED`
`UNENCRYPTED`

Diese Optionen bestimmen, ob das Passwort im Systemkatalog verschlüsselt gespeichert wird. (Wenn keines der beiden angegeben wurde, wird das Standardverhalten vom Konfigurationsparameter `password_encryption` bestimmt.) Wenn das angegebene Passwort bereits im verschlüsselten MD5-For-

mat ist, dann wird es so, wie es ist, verschlüsselt gespeichert, egal ob ENCRYPTED oder UNENCRYPTED angegeben wurde (da das System das angegebene verschlüsselte Passwort nicht entschlüsseln kann). Dadurch können verschlüsselte Passwörter aus einer Sicherungsdatei wiederhergestellt werden.

Beachten Sie, dass ältere Clients den MD5-Authentifizierungsmechanismus, der bei verschlüsselt gespeicherten Passwörtern verwendet werden muss, möglicherweise nicht unterstützen.

CREATEDB
NOCREATEDB

Diese Klauseln bestimmen, ob der Benutzer Datenbanken erzeugen darf. Wenn CREATEDB angegeben ist, wird dem Benutzer gestattet, seine eigenen Datenbanken zu erzeugen. Mit NOCREATEDB wird dem Benutzer nicht gestattet, neue Datenbanken zu erzeugen. Wenn die Klausel weggelassen wird, ist die Voreinstellung NOCREATEDB.

CREATEUSER
NOCREATEUSER

Diese Klauseln bestimmen, ob der Benutzer selbst neue Benutzer erzeugen darf. Diese Option macht den Benutzer auch zum Superuser, der alle Zugriffsbeschränkungen umgehen kann. Wenn die Klausel weggelassen wird, dann ist die Voreinstellung NOCREATEUSER.

gruppenname

Ein Name einer Gruppe, in die der neue Benutzer als Mitglied eingefügt werden soll. Es können mehrere Gruppennamen angegeben werden.

abstime

Die Klausel VALID UNTIL gibt die Zeit an, nach der das Passwort des Benutzers nicht mehr gültig ist. Wenn die Klausel weggelassen wird, ist das Passwort des Benutzers unbegrenzt gültig.

Meldungen

CREATE USER

Meldung, wenn das Benutzerkonto erfolgreich erzeugt wurde.

Hinweise

PostgreSQL enthält ein Programm createuser *(Seite: 792)*, das dieselbe Funktionalität wie CREATE USER hat (und es intern auch aufruft), aber von der Shell aus aufgerufen werden kann.

Beispiele

Erzeuge einen Benutzer ohne Passwort:

```
CREATE USER jonathan;
```

Erzeuge einen Benutzer mit einem Passwort:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

Erzeuge einen Benutzer mit einem Passwort, das bis zum Ende von 2004 gültig ist. Nachdem 2005 eine Sekunde alt ist, ist das Passwort nicht mehr gültig.

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Erzeuge einen Benutzer, der Datenbanken erzeugen kann:

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB;
```

Kompatibilität

Der Befehl `CREATE USER` ist eine PostgreSQL-Erweiterung. Der SQL-Standard überlässt die Definition der Benutzer der Implementierung.

Siehe auch

`ALTER GROUP` *(Seite: 627)*, `ALTER USER` *(Seite: 634)*, `DROP USER` *(Seite: 730)*, `createuser` *(Seite: 792)*

CREATE VIEW

Name

CREATE VIEW – definiert eine neue Sicht

Synopsis

```
CREATE [ OR REPLACE ] VIEW name [ ( spaltenname [, ...] ) ] AS anfrage
```

Beschreibung

`CREATE VIEW` definiert eine Sicht einer Anfrage. Die Sicht wird nicht materialisiert. Stattdessen wird die Anfrage jedes Mal ausgeführt, wenn die Sicht in einer Anfrage verwendet wird.

`CREATE OR REPLACE VIEW` ist ähnlich, aber wenn eine Sicht mit demselben Namen schon existiert, wird sie ersetzt. Sie können eine Sicht nur durch eine neue Anfrage ersetzen, die identische Spalten ergibt (d.h. gleiche Spaltennamen und Datentypen).

Wenn ein Schemaname angegeben wurde (zum Beispiel `CREATE VIEW mein_schema.meine_sicht ...`), wird die Sicht im angegebenen Schema erzeugt, ansonsten wird sie im aktuellen Schema erzeugt. Der Sichtname muss sich von den Namen aller Sichten, Tabellen, Sequenzen und Indexe im selben Schema unterscheiden.

Parameter

name

Der Name der zu erzeugenden Sicht (möglicherweise mit Schemaqualifikation).

spaltenname

Namen für die Spalten der Sicht. Wenn keine angegeben werden, werden die Spaltennamen aus der Anfrage ermittelt.

anfrage

Eine Anfrage (das heißt ein SELECT-Befehl), die die Spalten und Zeilen für die Sicht ermittelt.

Weitere Informationen über gültige Anfragen erhalten Sie unter SELECT *(Seite: 759)*.

Meldungen

CREATE VIEW

Meldung, wenn die Sicht erfolgreich erzeugt wurde.

WARNING: Attribute '*spalte*' has an unknown type

Die Sicht wird mit einer Spalte mit unbekanntem Typ erzeugt, wenn Sie den Typ nicht angeben. Folgender Befehl verursacht zum Beispiel diese Warnung:

```
CREATE VIEW vista AS SELECT 'Hallo Welt'
```

Aber dieser Befehl nicht:

```
CREATE VIEW vista AS SELECT text 'Hallo Welt'
```

Hinweise

Gegenwärtig sind Sichten nicht aktualisierbar: Versuche, in der Sicht einzufügen, zu aktualisieren oder zu löschen, werden vom System abgelehnt. Den Effekt von aktualisierbaren Sichten können Sie mit Regeln erreichen, die die Aktualisierungsbefehle usw. in die entsprechenden Aktionen in anderen Tabellen umschreiben. Weitere Informationen dazu finden Sie unter CREATE RULE *(Seite: 679)*.

Beispiele

Erzeuge eine Sicht mit allen Comedy-Filmen:

```
CREATE VIEW comedys AS
    SELECT *
    FROM filme
    WHERE genre = 'Comedy';
```

Kompatibilität

Der SQL-Standard sieht für den Befehl CREATE VIEW weitere Funktionalität vor:

```
CREATE VIEW name [ ( spalte [, ...] ) ]
    AS anfrage
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

Die optionalen Klauseln für den vollen SQL-Befehl sind:

CHECK OPTION

Diese Option hat mit aktualisierbaren Sichten zu tun. Mit dieser Option werden alle INSERT- und UPDATE-Befehle in der Sicht geprüft, ob sie die Bedingungen, die die Sicht definiert, erfüllen (das heißt, dass die neuen Daten durch die Sicht sichtbar wären). Wenn nicht, wird die Aktualisierung abgelehnt.

LOCAL

Prüft die Integrität für diese Sicht.

CASCADE

Prüft die Integrität für diese Sicht und alle abhängigen Sichten. Wenn weder CASCADE noch LOCAL angegeben sind, dann wird CASCADE angenommen.

Die Form CREATE OR REPLACE VIEW ist eine PostgreSQL-Erweiterung.

Siehe auch

DROP VIEW *(Seite: 731)*

DEALLOCATE

Name

DEALLOCATE – gibt einen vorbereiteten Befehl frei

Synopsis

```
DEALLOCATE [ PREPARE ] planname
```

Beschreibung

DEALLOCATE gibt einen zuvor vorbereiteten Befehl frei. Wenn Sie einen vorbereiteten Befehl nicht ausdrücklich freigeben, wird er am Ende der Sitzung automatisch freigegeben.

Weitere Informationen über vorbereitete Befehle erhalten Sie unter PREPARE *(Seite: 752)*.

Parameter

PREPARE

Dieses Schlüsselwort wird ignoriert.

planname

Der Name des freizugebenden vorbereiteten Befehls.

Meldungen

DEALLOCATE

Meldung, wenn der vorbereitete Befehl erfolgreich freigegeben wurde.

Kompatibilität

Der SQL-Standard enthält einen Befehl DEALLOCATE, aber er ist nur für eingebettetes SQL.

DECLARE

Name

DECLARE – definiert einen Cursor

Synopsis

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR anfrage
    [ FOR { READ ONLY | UPDATE [ OF spalte [, ...] ] } ]
```

Beschreibung

Mit DECLARE werden Cursor erzeugt, mit denen aus einer großen Anfrage Stück für Stück eine kleine Anzahl von Zeilen gelesen werden kann. Die Daten werden mit FETCH *(Seite: 737)* aus dem Cursor gelesen, entweder im Textformat oder im binären Format.

Normale Cursor geben Daten im Textformat zurück, wie SELECT. Da Daten intern in einem binären Format gespeichert werden, müssen die Daten vom System erst in das Textformat umgewandelt werden. Wenn die Daten im Textformat zurückgegeben werden, müssen die Clientanwendungen sie in ein binäres Format umwandeln, um sie zu verarbeiten. Außerdem sind die Daten im Textformat oft größer als im binären Format. Binäre Cursor geben die Daten im internen binären Format zurück. Wenn Sie die Daten aber sowieso als Text ausgeben wollen, ersparen Sie sich einige Arbeit im Client, wenn Sie sie vom Server im Textformat anfordern.

Wenn eine Anfrage zum Beispiel den Wert eins aus einer Spalte des Typs integer zurückgibt, würden Sie mit einem normalen Cursor die Zeichenkette 1 erhalten und mit einem binären Cursor einen 4-Byte-Wert mit der internen Darstellung des Werts.

Binäre Cursor sollten nicht sorglos verwendet werden. Viele Anwendungen, wie auch psql, sind nicht auf binäre Cursor vorbereitet und erwarten die Daten im Textformat.

Die Textdarstellung ist Architektur-neutral, aber die binäre Darstellung kann sich zwischen Maschinen mit unterschiedlichen Architekturen unterscheiden. *PostgreSQL kümmert sich bei binären Cursorn nicht um die Umwandlung der Bytereihenfolge oder anderer Formatunterschiede.* Wenn die Client- und die Servermaschine unterschiedliche interne Formate verwenden (z.B. unterschiedliche Bytereihenfolge), dann sollten Sie das binäre Format wahrscheinlich nicht verwenden.

Parameter

cursorname

Der Name des Cursors, zur Verwendung in FETCH-Operationen.

BINARY

Damit gibt der Cursor die Daten im binären Format statt im Textformat zurück.

`INSENSITIVE`

Gibt an, dass die aus dem Cursor zurückgegebenen Daten nicht von Aktualisierungen beeinflusst werden sollen, die während der Existenz des Cursors in den dem Cursor zugrunde liegenden Tabellen getätigt werden. In PostgreSQL ist das bei allen Cursorn der Fall; dieses Schlüsselwort hat keine Auswirkung und ist nur für die Kompatibilität mit dem SQL-Standard vorhanden.

`SCROLL`

Dieses Schlüsselwort gibt an, dass man auf die Zeilen des Cursors außer der Reihe (z.B. rückwärts) zugreifen kann. Zurzeit ist es nur für die Kompatibilität mit dem SQL-Standard und hat in PostgreSQL keine Auswirkung.

`anfrage`

Ein `SELECT`-Befehl, der die Zeilen ermittelt, die der Cursor zurückgibt. Informationen über gültige Anfragen erhalten Sie unter `SELECT` *(Seite: 759)*.

`FOR READ ONLY`
`FOR UPDATE`

`FOR READ ONLY` gibt an, dass aus dem Cursor nur gelesen werden soll. `FOR UPDATE` gibt an, dass der Cursor auch für die Aktualisierung von Tabellen verwendet werden kann. Da aktualisierbare Cursor gegenwärtig in PostgreSQL nicht unterstützt werden, verursacht die Verwendung von `FOR UPDATE` eine Fehlermeldung und die Angabe von `FOR READ ONLY` hat keine Auswirkung.

`spalte`

Die Spalten, die vom Cursor aktualisiert werden sollen. Da aktualisierbare Cursor gegenwärtig in PostgreSQL nicht unterstützt werden, verursacht die Verwendung der Klausel `FOR UPDATE` eine Fehlermeldung.

Meldungen

`DECLARE CURSOR`

Meldung, wenn der Cursor erfolgreich definiert wurde.

`WARNING: Closing pre-existing portal "`*`cursorname`*`"`

Diese Meldung wird ausgegeben, wenn der gleiche Cursorname in der aktuellen Transaktion schon deklariert wurde. Die vorherige Definition wird dann verworfen.

`ERROR: DECLARE CURSOR may only be used in begin/end transaction blocks`

Diese Fehlermeldung wird ausgegeben, wenn der Cursor nicht in einem Transaktionsblock deklariert wird.

Hinweise

Ein Cursor kann nur in der aktuellen Transaktion verwendet werden. Daher ist es nur sinnvoll, `DECLARE` in einem Transaktionsblock (`BEGIN`/`COMMIT`-Paar) aufzurufen.

Der SQL-Standard sieht Cursor nur für eingebettetes SQL vor. Der PostgreSQL-Server implementiert keinen `OPEN`-Befehl für Cursor; ein Cursor ist geöffnet, wenn er deklariert wird. `ECPG`, der Präprozessor für eingebettetes SQL in PostgreSQL, unterstützt jedoch die Cursorkonventionen des SQL-Standards, einschließlich derer für die Befehle `DECLARE` und `OPEN`.

Beispiele

Die Definition eines einfachen Cursors:

```
DECLARE liahona CURSOR FOR SELECT * FROM filme;
```

Weitere Cursorbeispiele finden Sie unter `FETCH` *(Seite: 737)*.

Kompatibilität

Der SQL-Standard sieht Cursor nur für eingebettetes SQL und Module vor. In PostgreSQL können Cursor interaktiv verwendet werden.

Der SQL-Standard erlaubt Cursor, die Tabellendaten aktualisieren können. Cursor in PostgreSQL können nur gelesen werden.

Binäre Cursor sind eine PostgreSQL-Erweiterung.

DELETE

Name

DELETE – löscht Zeilen einer Tabelle

Synopsis

```
DELETE FROM [ ONLY ] tabelle [ WHERE bedingung ]
```

Beschreibung

Der Befehl `DELETE` löscht die Zeilen aus der angegebenen Tabelle, die die `WHERE`-Klausel erfüllen. Wenn keine `WHERE`-Klausel angegeben ist, werden alle Zeilen aus der Tabelle gelöscht. Das Ergebnis ist eine gültige, aber leere Tabelle.

Normalerweise löscht `DELETE` Zeilen aus der angegebenen Tabelle und ihren Untertabellen. Wenn Sie nur aus der angegebenen Tabelle löschen wollen, müssen Sie die Klausel `ONLY` verwenden.

Um aus einer Tabelle zu löschen, müssen Sie das Privileg `DELETE` für die Tabelle haben sowie das Privileg `SELECT` für jede Tabelle, die in der `WHERE`-Klausel verwendet wird.

Parameter

`tabelle`

Der Name einer Tabelle (möglicherweise mit Schemaqualifikation).

`bedingung`

Ein Wertausdruck, der einen Wert vom Typ `boolean` ergibt und bestimmt, welche Zeilen gelöscht werden sollen.

Meldungen

`DELETE` *`zahl`*

Meldung, wenn Zeilen erfolgreich gelöscht wurden. *`zahl`* ist die Anzahl der gelöschten Zeilen. Wenn *`zahl`* 0 ist, wurden keine Zeilen gelöscht.

Hinweise

`TRUNCATE` *(Seite: 780)* ist eine PostgreSQL-Erweiterung, die schneller alle Zeilen aus einer Tabelle löschen kann.

Beispiele

Lösche alle Filme außer Musicals:

```
DELETE FROM filme WHERE genre <> 'Musical';
```

Leere die Tabelle `filme`:

```
DELETE FROM filme;
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform.

DROP AGGREGATE

Name

DROP AGGREGATE – entfernt eine Aggregatfunktion

Synopsis

```
DROP AGGREGATE name ( typ ) [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP AGGREGATE` löscht eine bestehende Aggregatfunktion. Um diesen Befehl ausführen zu können, muss der aktuelle Benutzer der Eigentümer der Aggregatfunktion sein.

Parameter

`name`

Der Name einer bestehenden Aggregatfunktion (möglicherweise mit Schemaqualifikation).

typ

Der Argumentdatentyp der Aggregatfunktion oder *, wenn die Funktion jeden Datentyp akzeptiert.

CASCADE

Löscht automatisch alle Objekte, die von der Aggregatfunktion abhängen.

RESTRICT

Verhindert das Löschen der Aggregatfunktion, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

DROP AGGREGATE

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

ERROR: RemoveAggregate: aggregate '*name*' for type *typ* does not exist

Diese Meldung wird ausgegeben, wenn die angegebene Aggregatfunktion nicht existiert.

Beispiele

Entferne die Aggregatfunktion test_avg für den Typ integer:

```
DROP AGGREGATE test_avg(integer);
```

Kompatibilität

Der Befehl DROP AGGREGATE ist eine PostgreSQL-Erweiterung. Der SQL-Standard sieht keine benutzerdefinierten Aggregatfunktionen vor.

Siehe auch

CREATE AGGREGATE *(Seite: 652)*

DROP CAST

Name

DROP CAST – entfernt eine Typumwandlung

Synopsis

```
DROP CAST (quelltyp AS zieltyp) [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP CAST` entfernt eine zuvor definierte Typumwandlung.

Um eine Typumwandlung löschen zu können, müssen Sie der Eigentümer des Quell- oder des Zieltyps sein. Dies sind die gleichen Privilegien, die beim Erzeugen einer Typumwandlung benötigt werden.

Parameters

`quelltyp`

Der Name des Quelldatentyps der Typumwandlung.

`zieltyp`

Der Name des Zieldatentyps der Typumwandlung.

`CASCADE`
`RESTRICT`

Diese Schlüsselwörter haben keine Auswirkung, weil es keine Abhängigkeiten von Umwandlungen gibt.

Beispiele

Um die Typumwandlung vom Typ `text` in den Typ `int` zu löschen:

```
DROP CAST (text AS int);
```

Kompatibilität

Der Befehl `DROP CAST` ist mit dem SQL-Standard konform.

Siehe auch

`CREATE CAST` *(Seite: 654)*

DROP CONVERSION

Name

DROP CONVERSION – entfernt eine Konversion

Synopsis

```
DROP CONVERSION konversionsname [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP CONVERSION` entfernt eine zuvor definierte Konversion. Um eine Konversion löschen zu können, müssen Sie der Eigentümer der Konversion sein.

Parameters

`konversionsname`

Der Name der Konversion (möglicherweise mit Schemaqualifikation).

`CASCADE`
`RESTRICT`

Diese Schlüsselwörter haben keine Auswirkung, weil es keine Abhängigkeiten von Konversionen gibt.

Beispiele

Um die Konversion namens `test_konv` zu löschen:

```
DROP CONVERSION test_konv;
```

Kompatibilität

Der Befehl `DROP CONVERSION` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE CONVERSION` *(Seite: 658)*

DROP DATABASE

Name

DROP DATABASE – entfernt eine Datenbank

Synopsis

```
DROP DATABASE name
```

Beschreibung

`DROP DATABASE` löscht eine Datenbank. Es entfernt die Katalogeinträge der Datenbank und löscht das Verzeichnis mit den Daten. Nur der Eigentümer der Datenbank kann diesen Befehl ausführen.

`DROP DATABASE` kann nicht rückgängig gemacht werden. Sein Sie vorsichtig!

Parameter

name

Der Name der zu entfernenden Datenbank.

Meldungen

`DROP DATABASE`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`DROP DATABASE: cannot be executed on the currently open database`

Sie können nicht die Datenbank löschen, mit der Sie gerade verbunden sind. Verbinden Sie stattdessen mit `template1` oder einer anderen Datenbank und führen Sie den Befehl noch einmal aus.

`DROP DATABASE: may not be called in a transaction block`

Sie können diesen Befehl nicht in einem Transaktionsblock aufrufen. Beenden Sie zuerst die aktuelle Transaktion.

Hinweise

Dieser Befehl kann nicht ausgeführt werden, während man mit der Zieldatenbank verbunden ist. Daher ist es vielleicht bequemer, stattdessen das Programm `dropdb` *(Seite: 795)* zu verwenden, welches diesen Befehl intern aufruft.

Kompatibilität

Der Befehl `DROP DATABASE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE DATABASE` *(Seite: 660)*, `dropdb` *(Seite: 795)*

DROP DOMAIN

Name

DROP DOMAIN – entfernt eine Domäne

Synopsis

```
DROP DOMAIN name [, ...]  [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP DOMAIN` entfernt eine Domäne. Nur der Eigentümer einer Domäne kann sie entfernen.

Parameter

name

Der Name der zu löschenden Domäne (möglicherweise mit Schemaqualifikation).

`CASCADE`

Löscht automatisch alle Objekte, die von der Domäne abhängen (z.B. Tabellenspalten).

`RESTRICT`

Verhindert das Löschen der Domäne, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP DOMAIN`

Meldung, wenn die Befehl erfolgreich ausgeführt wurde.

`ERROR: RemoveDomain: type '`*name*`' does not exist`

Diese Meldung wird ausgegeben, wenn die angegebene Domäne nicht existiert.

Beispiele

Um die Domäne `box` zu entfernen:

```
DROP DOMAIN box;
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform.

Siehe auch

`CREATE DOMAIN` *(Seite: 662)*

DROP FUNCTION

Name

DROP FUNCTION – entfernt eine Funktion

Synopsis

```
DROP FUNCTION name ( [ typ [, ...] ] ) [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP FUNCTION` entfernt die Definition einer bestehenden Funktion. Um diesen Befehl ausführen zu können, müssen Sie der Eigentümer der Funktion sein. Die Argumenttypen der Funktion müssen angegeben werden, weil es verschiedene Funktion mit dem gleichen Namen, aber unterschiedlichen Argumentlisten geben kann.

Parameter

name

Der Name der zu löschenden Funktion (möglicherweise mit Schemaqualifikation).

typ

Der Datentyp eines Arguments der Funktion.

`CASCADE`

Löscht automatisch alle Objekte, die von der Funktion abhängen (z.B. Operatoren und Trigger).

`RESTRICT`

Verhindert das Löschen der Funktion, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP FUNCTION`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`WARNING: RemoveFunction: Function` *name* `(`*typen*`) does not exist`

Diese Meldung wird ausgegeben, wenn die angegebene Funktion nicht existiert.

Beispiele

Dieser Befehl entfernt die Quadratwurzelfunktion:

```
DROP FUNCTION sqrt(integer);
```

Kompatibilität

Im SQL-Standard ist ein Befehl `DROP FUNCTION` definiert, aber er ist nicht mit dieser Variante kompatibel.

Siehe auch

`CREATE FUNCTION` *(Seite: 664)*

DROP GROUP

Name

DROP GROUP – entfernt eine Benutzergruppe

Synopsis

```
DROP GROUP name
```

Beschreibung

`DROP GROUP` entfernt die angegebene Gruppe. Benutzer in der Gruppe werden nicht gelöscht.

Parameter

name
Der Name der zu entfernenden Gruppe.

Meldungen

`DROP GROUP`
Meldung, wenn die Gruppe erfolgreich entfernt wurde.

Beispiele

Um eine Gruppe zu entfernen:

```
DROP GROUP mitarbeiter;
```

Kompatibilität

Der Befehl `DROP GROUP` ist eine PostgreSQL-Erweiterung. Das Konzept der Rollen im SQL-Standard ist vergleichbar mit Gruppen.

Siehe auch

`ALTER GROUP` *(Seite: 627)*, `CREATE GROUP` *(Seite: 667)*

DROP INDEX

Name

DROP INDEX – entfernt einen Index

Synopsis

```
DROP INDEX indexname [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP INDEX` entfernt einen Index aus dem Datenbanksystem. Um einen Index entfernen zu können, müssen Sie der Eigentümer sein.

Parameter

`indexname`

Der Name des zu entfernenden Index (möglicherweise mit Schemaqualifikation).

`CASCADE`

Löscht automatisch alle Objekte, die von dem Index abhängen.

`RESTRICT`

Verhindert das Löschen des Index, wenn irgendwelche Objekte von ihm abhängen. Das ist die Voreinstellung.

Meldungen

`DROP INDEX`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`ERROR: index "`*`indexname`*`" does not exist`

Meldung, wenn *`indexname`* kein existierender Index ist.

Beispiele

Dieser Befehl entfernt den Index `titel_idx`:

```
DROP INDEX titel_idx;
```

Kompatibilität

Der Befehl `DROP INDEX` ist eine PostgreSQL-Erweiterung. Im SQL-Standard gibt es keine Bestimmungen über Indexe.

Siehe auch

`CREATE INDEX` *(Seite: 669)*

DROP LANGUAGE

Name

DROP LANGUAGE – entfernt eine prozedurale Sprache

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP LANGUAGE` entfernt die Definition einer zuvor registrierten prozeduralen Sprache namens *`name`*.

Parameter

`name`

Der Name der zu entfernenden prozeduralen Sprache. Aus Kompatibilität mit früheren Versionen kann der Sprachname auch in Apostrophen stehen.

`CASCADE`

Löscht automatisch alle Objekte, die von der Sprache abhängen (z.B. Funktionen in der Sprache).

`RESTRICT`

Verhindert das Löschen der Sprache, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP LANGUAGE`

Meldung, wenn die Sprache erfolgreich entfernt wurde.

`ERROR: Language "`*`name`*`" doesn't exist`

Meldung, wenn es in der aktuellen Datenbank keine Sprache mit dem Namen *`name`* gibt.

Beispiele

Dieser Befehl entfernt die prozedurale Sprache `plbeispiel`:

```
DROP LANGUAGE plbeispiel;
```

Kompatibilität

Der Befehl `DROP LANGUAGE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE LANGUAGE` *(Seite: 671)*, `droplang` *(Seite: 797)*

DROP OPERATOR

Name

DROP OPERATOR – entfernt einen Operator

Synopsis

```
DROP OPERATOR name ( linker_typ | NONE , rechter_typ | NONE ) [ CASCADE |
RESTRICT ]
```

Beschreibung

`DROP OPERATOR` entfernt einen bestehenden Operator aus dem Datenbanksystem. Um diesen Befehl ausführen zu können, müssen Sie der Eigentümer des Operators sein.

Parameter

name

Der Name des zu entfernenden Operators (möglicherweise mit Schemaqualifikation).

linker_typ

Der Datentyp des linken Operanden des Operators; `NONE,` wenn der Operator keinen linken Operanden hat.

rechter_typ

Der Datentyp des rechten Operanden des Operators; `NONE,` wenn der Operator keinen rechten Operanden hat.

`CASCADE`

Löscht automatisch alle Objekte, die von dem Operator abhängen.

`RESTRICT`

Verhindert das Löschen des Operators, wenn irgendwelche Objekte von ihm abhängen. Das ist die Voreinstellung.

Meldungen

`DROP OPERATOR`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`ERROR: RemoveOperator: binary operator '`*`name`*`' taking '`*`linker_typ`*`' and '`*`rechter_typ`*`' does not exist`

Diese Meldung wird zurückgegeben, wenn der angegebene binäre Operator nicht existiert.

`ERROR: RemoveOperator: left unary operator '`*`name`*`' taking '`*`linker_typ`*`' does not exist`

Diese Meldung wird zurückgegeben, wenn der angegebene linke unäre Operator nicht existiert.

`ERROR: RemoveOperator: right unary operator '`*`name`*`' taking '`*`rechter_typ`*`' does not exist`

Diese Meldung wird zurückgegeben, wenn der angegebene rechte unäre Operator nicht existiert.

Beispiele

Dieser Befehl entfernt den Potenzierungsoperator `a^b` für den Typ `integer`:

```
DROP OPERATOR ^ (integer, integer);
```

Dieser Befehl entfernt den linken unären Operator `~b` für die bitweise Negation des Typs `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Dieser Befehl entfernt den rechten unären Fakultätsoperator `x!` für den Typ `integer`:

```
DROP OPERATOR ! (integer, none);
```

Kompatibilität

Der Befehl `DROP OPERATOR` ist eine PostgreSQL-Erweiterung. Der SQL-Standard sieht keine benutzerdefinierten Operatoren vor.

Siehe auch

`CREATE OPERATOR` *(Seite: 673)*

DROP OPERATOR CLASS

Name

DROP OPERATOR CLASS – entfernt eine Operatorklasse

Synopsis

```
DROP OPERATOR CLASS name USING indexmethode [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP OPERATOR CLASS` löscht eine bestehende Operatorklasse. Um diesen Befehl ausführen zu können, müssen Sie der Eigentümer der Operatorklasse sein.

Parameter

name

Der Name der zu entfernenden Operatorklasse (möglicherweise mit Schemaqualifikation).

indexmethode

Die Name der Indexmethode, für die die Operatorklasse gilt.

`CASCADE`

Löscht automatisch alle Objekte, die von der Operatorklasse abhängen (z.B. Indexe).

`RESTRICT`

Verhindert das Löschen der Operatorklasse, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP OPERATOR CLASS`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

Beispiele

Dieser Befehl entfernt die B-Tree-Operatorklasse `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Dieser Befehl wird nicht erfolgreich sein, wenn irgendwelche bestehenden Indexe die Operatorklasse verwenden. Geben Sie `CASCADE` an, um diese Indexe zusammen mit der Operatorklasse zu löschen.

Kompatibilität

Der Befehl `DROP OPERATOR CLASS` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE OPERATOR CLASS` *(Seite: 676)*

DROP RULE

Name

DROP RULE – entfernt eine Umschreiberegel

Synopsis

```
DROP RULE name ON relation [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP RULE` entfernt eine Umschreiberegel.

Parameter

`name`
Der Name der zu entfernenden Regel.

`relation`
Der Name der Tabelle oder Sicht (möglicherweise mit Schemaqualifikation), für die die Regel gilt.

`CASCADE`
Löscht automatisch alle Objekte, die von der Regel abhängen.

`RESTRICT`
Verhindert das Löschen der Regel, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP RULE`
Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`ERROR: Rule "`*`name`*`" not found`
Meldung, wenn die angegebene Regel nicht existiert.

Beispiele

Dieser Befehl entfernt eine Regel namens `test_regel`:

```
DROP RULE test_regel ON meine_tabelle;
```

Kompatibilität

Der Befehl `DROP RULE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE RULE` *(Seite: 679)*

DROP SCHEMA

Name

DROP SCHEMA – entfernt ein Schema

Synopsis

```
DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP SCHEMA` entfernt ein Schema aus der Datenbank.

Ein Schema kann nur von seinem Eigentümer oder einem Superuser gelöscht werden. Beachten Sie, dass der Eigentümer ein Schema (und damit alle darin enthaltenen Objekte) auch dann löschen kann, wenn einige Objekte im Schema nicht ihm gehören.

Parameter

name

Der Name des zu entfernenden Schemas.

`CASCADE`

Löscht automatisch alle in dem Schema enthaltenen Objekte (Tabellen, Funktionen usw.).

`RESTRICT`

Verhindert das Löschen des Schemas, wenn es irgendwelche Objekte enthält. Das ist die Voreinstellung.

Meldungen

`DROP SCHEMA`

Meldung, wenn das Schema erfolgreich gelöscht wurde.

`ERROR: Schema "`*name*`" does not exist`

Meldung, wenn das angegebene Schema nicht existiert.

Beispiele

Dieser Befehl entfernt das Schema `meinzeug` aus der Datenbank, zusammen mit allem, was darin enthalten ist:

```
DROP SCHEMA meinzeug CASCADE;
```

Kompatibilität

Der Befehl `DROP SCHEMA` ist voll mit dem SQL-Standard konform, außer dass der Standard nur das Löschen von einem Schema pro Befehl erlaubt.

Siehe auch

`CREATE SCHEMA` *(Seite: 681)*

DROP SEQUENCE

Name

DROP SEQUENCE – entfernt einen Sequenzgenerator

Synopsis

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP SEQUENCE` entfernt Sequenzgeneratoren.

Parameter

name

Der Name der zu löschenden Sequenz (möglicherweise mit Schemaqualifikation).

`CASCADE`

Löscht automatisch alle Objekte, die von der Sequenz abhängen.

`RESTRICT`

Verhindert das Löschen der Sequenz, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP SEQUENCE`
Meldung, wenn die Sequenz erfolgreich gelöscht wurde.

`ERROR: sequence "name" does not exist`
Meldung, wenn die angegebene Sequenz nicht existiert.

Beispiele

Um die Sequenz `serial` zu entfernen:

```
DROP SEQUENCE serial;
```

Kompatibilität

Der Befehl `DROP SEQUENCE` ist eine PostgreSQL-Erweiterung.

Siehe auch

`CREATE SEQUENCE` *(Seite: 683)*

DROP TABLE

Name

DROP TABLE – entfernt eine Tabelle

Synopsis

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP TABLE` löscht Tabellen aus der Datenbank. Nur der Eigentümer kann eine Tabelle löschen. Um alle Zeilen aus einer Tabelle zu entfernen, ohne die Tabelle zu zerstören, verwenden Sie `DELETE`.

`DROP TABLE` löscht immer alle Indexe, Regeln, Trigger und Constraints der Zieltabelle. Um jedoch eine Tabelle zu löschen, auf die ein Fremdschlüssel einer anderen Tabelle verweist, muss `CASCADE` angegeben werden. (`CASCADE` entfernt dann den Fremdschlüssel, aber nicht die ganze andere Tabelle.)

Parameter

name

Der Name der zu löschenden Tabelle (möglicherweise mit Schemaqualifikation).

CASCADE

Löscht automatisch alle Objekte, die von der Tabelle abhängen (z.B. Sichten).

RESTRICT

Verhindert das Löschen der Tabelle, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

DROP TABLE

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

ERROR: table "*name*" does not exist

Meldung, wenn die angegebene Tabelle nicht existiert.

Beispiele

Dieser Befehl entfernt zwei Tabellen, `filme` und `verleihe`:

```
DROP TABLE filme, verleihe;
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform.

Siehe auch

ALTER TABLE *(Seite: 628)*, CREATE TABLE *(Seite: 686)*

DROP TRIGGER

Name

DROP TRIGGER – entfernt einen Trigger

Synopsis

```
DROP TRIGGER name ON tabelle [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP TRIGGER` entfernt eine bestehende Triggerdefinition. Um diesen Befehl auszuführen, muss der aktuelle Benutzer der Eigentümer der Tabelle sein, für die der Trigger definiert ist.

Parameter

`name`

Der Name des zu entfernenden Triggers.

`table`

Der Name der Tabelle (möglicherweise mit Schemaqualifikation), für die der Trigger definiert ist.

`CASCADE`

Löscht automatisch alle Objekte, die von dem Trigger abhängen.

`RESTRICT`

Verhindert das Löschen des Triggers, wenn irgendwelche Objekte von ihm abhängen. Das ist die Voreinstellung.

Meldungen

`DROP TRIGGER`

Meldung, wenn der Trigger erfolgreich gelöscht wurde.

`ERROR: DropTrigger: there is no trigger` *`name`* `on relation "`*`table`*`"`

Meldung, wenn der angegebene Trigger nicht existiert.

Beispiele

Lösche den Trigger `test_trigger` für die Tabelle `meine_tabelle`:

```
DROP TRIGGER test_trigger ON meine_tabelle;
```

Kompatibilität

Der Befehl `DROP TRIGGER` in PostgreSQL ist nicht mit dem SQL-Standard kompatibel. Im SQL-Standard sind die Namen der Trigger nicht an die Tabelle gebunden, daher ist der Befehl einfach `DROP TRIGGER` *`name`*.

Siehe auch

`CREATE TRIGGER` *(Seite: 695)*

DROP TYPE

Name

DROP TYPE – entfernt einen Datentyp

Synopsis

```
DROP TYPE typname [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP TYPE` entfernt einen benutzerdefinierten Datentyp. Nur der Eigentümer des Typs kann ihn entfernen.

Parameter

typename

Der Name des zu entfernenden Datentyps (möglicherweise mit Schemaqualifikation).

`CASCADE`

Löscht automatisch alle Objekte, die von dem Typ abhängen (z.B. Tabellenspalten, Funktionen, Operatoren).

`RESTRICT`

Verhindert das Löschen des Typs, wenn irgendwelche Objekte von ihm abhängen. Das ist die Voreinstellung.

Meldungen

`DROP TYPE`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`ERROR: RemoveType: type '`*typename*`' does not exist`

Meldung, wenn der angegebene Datentyp nicht existiert.

Beispiele

Dieser Befehl entfernt den Datentyp `box`:

```
DROP TYPE box;
```

Kompatibilität

Dieser Befehl ist dem entsprechenden Befehl aus dem SQL-Standard ähnlich, aber beachten Sie, dass `CREATE TYPE` und der Datentyp-Erweiterungsmechanismus in PostgreSQL sich vom SQL-Standard unterscheiden.

Siehe auch

`CREATE TYPE` *(Seite: 697)*

DROP USER

Name

DROP USER – entfernt ein Datenbankbenutzerkonto

Synopsis

```
DROP USER name
```

Beschreibung

Der Befehl `DROP USER` entfernt den angegebenen Benutzer. Er entfernt nicht die Tabellen, Sichten oder anderen Objekte, die dem Benutzer gehören. Wenn dem Benutzer eine Datenbank gehört, dann wird ein Fehler ausgegeben.

Parameter

name
Der Name des zu entfernenden Benutzers.

Meldungen

`DROP USER`
Meldung, wenn der Benutzer erfolgreich gelöscht wurde.

`ERROR: DROP USER: user "`*name*`" does not exist`
Meldung, wenn der angegebene Benutzer nicht existiert.

`DROP USER: user "`*name*`" owns database "`*name*`", cannot be removed`
Dem Benutzer gehört eine Datenbank; Sie müssen diese erst entfernen oder den Eigentümer ändern.

Hinweise

PostgreSQL enthält ein Programm `dropuser` *(Seite: 799)*, das die selbe Funktionalität wie `DROP USER` hat (und es intern auch aufruft), aber von der Shell aus aufgerufen werden kann.

Beispiele

Ein Benutzerkonto löschen:

```
DROP USER jonathan;
```

Kompatibilität

Der Befehl `DROP USER` ist eine PostgreSQL-Erweiterung. Der SQL-Standard überlässt die Definition der Benutzer der Implementierung.

Siehe auch

`ALTER USER` *(Seite: 634)*, `CREATE USER` *(Seite: 702)*, `dropuser` *(Seite: 799)*

DROP VIEW

Name

DROP VIEW – entfernt eine Sicht

Synopsis

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

Beschreibung

`DROP VIEW` löscht eine bestehende Sicht. Um diesen Befehl auszuführen, müssen Sie der Eigentümer der Sicht sein.

Parameter

name

Der Name der zu entfernenden Sicht (möglicherweise mit Schemaqualifikation).

`CASCADE`

Löscht automatisch alle Objekte, die von dem Typ abhängen (z.B. andere Sichten).

`RESTRICT`
Verhindert das Löschen der Sicht, wenn irgendwelche Objekte von ihr abhängen. Das ist die Voreinstellung.

Meldungen

`DROP VIEW`
Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`ERROR: view` *`name`* `does not exist`
Meldung, wenn die angegebene Sicht nicht existiert.

Beispiele

Dieser Befehl entfernt eine Sicht namens `genres`:

```
DROP VIEW genres;
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform.

Siehe auch

`CREATE VIEW` *(Seite: 704)*

END

Name

END – schließt die aktuelle Transaktion ab

Synopsis

```
END [ WORK | TRANSACTION ]
```

Beschreibung

`END` schließt die aktuelle Transaktion ab. Alle von der Transaktion getätigten Änderungen werden für andere sichtbar und sind garantiert dauerhaft, falls es zu einem Absturz kommt. Dieser Befehl ist identisch mit dem Befehl `COMMIT` aus dem SQL-Standard und ist nur aus historischen Gründen vorhanden.

Parameter

`WORK`
`TRANSACTION`

Optionale Schlüsselwörter ohne jegliche Auswirkung.

Meldungen

`COMMIT`

Meldung, wenn die Transaktion erfolgreich abgeschlossen wurde.

`WARNING: COMMIT: no transaction in progress`

Meldung, wenn keine Transaktion offen ist.

Beispiele

Um die aktuelle Transaktion abzuschließen und alle Änderungen dauerhaft zu speichern:

```
END;
```

Kompatibilität

Dieser Befehl ist eine PostgreSQL-Erweiterung, die nur aus historischen Gründen vorhanden ist. Der entsprechende Befehl im SQL-Standard ist `COMMIT`.

Siehe auch

`COMMIT` *(Seite: 645)*, `ROLLBACK` *(Seite: 758)*

EXECUTE

Name

EXECUTE – führt einen vorbereiteten Befehl aus

Synopsis

```
EXECUTE planname [ (parameter [, ...] ) ]
```

Beschreibung

EXECUTE wird verwendet, um einen zuvor vorbereiteten Befehl auszuführen. Da vorbereitete Befehle nur für die Dauer der Sitzung existieren, muss der Befehl in der aktuellen Sitzung mit PREPARE vorbereitet worden sein.

Wenn der PREPARE-Befehl, der den Befehl vorbereitet hat, Parameter vorsah, muss ein Satz kompatibler Parameter an EXECUTE übergeben werden. Beachten Sie, dass vorbereitete Befehle nicht anhand der Anzahl und Typen der Parameter überladen werden können (wie Funktionen); der Name des vorbereiteten Befehls muss in der Datenbanksitzung einmalig sein.

Weitere Informationen über vorbereitete Befehle erhalten Sie unter PREPARE *(Seite: 752)*.

Parameter

planname

Der Name des vorbereiteten Befehls, der ausgeführt werden soll.

parameter

Der tatsächliche Wert für einen Parameter des vorbereiteten Befehls. Dies muss ein Ausdruck sein, der einen Wert ergibt, der mit dem für diese Parameterposition vom PREPARE-Befehl vorgesehenen Parameter kompatibel ist.

Kompatibilität

Der SQL-Standard enthält einen Befehl EXECUTE, aber er ist nur für eingebettetes SQL. Diese Version des EXECUTE-Befehls hat auch eine etwas andere Syntax.

EXPLAIN

Name

EXPLAIN – zeigt den Ausführungsplan eines Befehls

Synopsis

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] befehl
```

Beschreibung

Dieser Befehl zeigt den Ausführungsplan, den der PostgreSQL-Planer für den angegebenen Befehl erzeugt. Der Ausführungsplan zeigt, wie die in dem Befehl verwendeten Tabellen durchsucht werden (sequenzielle Suche, Indexsuche usw.) und wenn mehrere Tabellen verwendet werden, dann die Verbundalgorithmen, die verwendet werden, um die entsprechenden Zeilen aus jeder Eingabetabelle zusammenzubringen.

Der wichtigste Teil der Ausgabe sind die geschätzten Ausführungskosten des Befehls; das ist die Schätzung des Planers, wie lange die Ausführung der Anfrage dauern wird (gemessen in Diskseiten-Fetchs). Gezeigt werden zwei Zahlen: die Startzeit, die benötigt wird, ehe die erste Zeile zurückgegeben werden kann, und die Gesamtzeit, die benötigt wird, um alle Zeilen zurückzugeben. Bei den meisten Befehlen ist die Gesamtzeit entscheidend, aber zum Beispiel bei einer Unteranfrage in einer `EXISTS`-Klausel wählt der Planer die geringste Startzeit statt der geringsten Gesamtzeit (da der Executor sowieso nur eine Zeile lesen wird und dann aufhört). Wenn Sie die Klausel `LIMIT` verwenden, um die Anzahl der zurückgegebenen Zeilen zu beschränken, dann interpoliert der Planer zwischen beiden Kostenangaben, um herauszufinden, welcher Plan wirklich der bessere ist.

Mit der Option `ANALYZE` wird der Befehl auch tatsächlich ausgeführt, nicht nur geplant. Dann wird zusätzlich die in jedem Planknoten verbrachte Gesamtzeit (in Millisekunden) und die Anzahl der tatsächlich zurückgegebenen Zeilen ausgegeben. Das ist nützlich, um festzustellen, ob die Schätzungen des Planers nahe an der Wirklichkeit sind.

Wichtig: Bedenken Sie, dass der Befehl wirklich ausgeführt wird, wenn `ANALYZE` verwendet wird. Das Ergebnis eines `SELECT` wird von `EXPLAIN` verworfen, aber andere Nebenwirkung finden normal statt. Wenn Sie `EXPLAIN ANALYZE` mit einem `INSERT`-, `UPDATE`- oder `DELETE`-Befehl verwenden möchten, ohne die Daten zu verändern, machen Sie Folgendes:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Parameter

`ANALYZE`

Führt den Befehl aus und zeigt die tatsächliche Laufzeit an.

`VERBOSE`

Zeigt die volle interne Darstellung des Planbaums anstatt nur einer Zusammenfassung. Diese Option ist nur für das Debuggen von PostgreSQL nützlich. Das Ausgabeformat der `VERBOSE`-Ausgabe wird vom Konfigurationsparameter `explain_pretty_print` kontrolliert; wenn er an ist, dann wird die Ausgabe zur besseren Lesbarkeit eingerückt.

`befehl`

Ein `SELECT`-, `INSERT`-, `UPDATE`-, `DELETE`-, `EXECUTE`- oder `DECLARE`-Befehl, dessen Ausführungsplan Sie sehen möchten.

Meldungen

`EXPLAIN` gibt den vom PostgreSQL-Planer ermittelten Ausführungsplan des angegebenen Befehls aus.

Hinweise

Weitere Informationen über `EXPLAIN` und die Bedeutung der Kostenschätzung erhalten Sie in Abschnitt SET.

Vor PostgreSQL 7.3 wurde der Ausführungsplan in der Form einer `NOTICE`-Mitteilung ausgegeben. Jetzt erscheint er als Anfrageergebnis (wie eine Tabelle mit einer Spalte vom Typ `text`.)

Beispiele

Ein Ausführungsplan für eine einfache Anfrage mit einer Tabelle mit einer Spalte vom Typ `integer` und 10000 Zeilen:

```
EXPLAIN SELECT * FROM foo;

                      QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

Wenn es einen Index gibt und die Anfrage eine indizierbare `WHERE`-Bedingung hat, könnte `EXPLAIN` einen anderen Plan anzeigen:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                        QUERY PLAN
------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Hier ein Beispiel für einen Ausführungsplan einer Anfrage mit Aggregatfunktion:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Natürlich hängen die gezeigten Zahlen vom tatsächlichen Inhalt der Tabelle ab. Beachten Sie außerdem, dass die Zahlen und sogar die ausgewählten Anfragestrategien je nach PostgreSQL-Version wegen Planerverbesserungen unterschiedlich ausfallen können.

Kompatibilität

Der Befehl `EXPLAIN` ist eine PostgreSQL-Erweiterung.

FETCH

Name

FETCH – liest Zeilen aus einer Anfrage mit einem Cursor

Synopsis

```
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ anzahl | ALL | NEXT | PRIOR ] { FROM |
IN } cursor
```

Beschreibung

`FETCH` liest Zeilen aus einem Cursor. Die Anzahl der zu lesenden Zeilen wird durch den Parameter *`anzahl`* angegeben. Wenn weniger Zeilen im Cursor übrig sind, werden nur die übrigen zurückgegeben. Wenn das Schlüsselwort `ALL` anstelle der Anzahl verwendet wird, werden alle übrigen Zeilen aus dem Cursor gelesen. Zeilen können vorwärts (mit `FORWARD`) oder rückwärts (mit `BACKWARD`) gelesen werden. Wenn keine Richtung angegeben wird, wird vorwärts gelesen.

Parameter

`FORWARD`

Liest die folgende(n) Zeile(n). Das ist die Voreinstellung, wenn keine andere Richtung angegeben ist.

`BACKWARD`

Liest die vorangehende(n) Zeile(n).

`RELATIVE`

Keine Auswirkung; nur für Kompatibilität mit dem SQL-Standard.

`anzahl`

Eine ganze Zahl, die angibt, wie viele Zeilen gelesen werden sollen. Ein negative Zahl vertauscht den Sinn von `FORWARD` und `BACKWARD`.

`ALL`

Liest alle übrigen Zeilen.

`NEXT`

Liest die nächste Zeile; das Gleiche wie `1` als Anzahl.

`PRIOR`

Liest die vorangehende Zeile; das Gleiche wie `-1` als Anzahl.

`cursor`

Der Name eines offenen Cursors.

Meldungen

`WARNING: PerformPortalFetch: portal "`*`cursor`*`" not found`

Meldung, wenn *cursor* nicht vorher deklariert worden ist. Beachten Sie, dass ein Cursor nur in der Transaktion gilt, in der er deklariert wurde.

`WARNING: FETCH/ABSOLUTE not supported, using RELATIVE`

PostgreSQL unterstützt die absolute Positionierung eines Cursors nicht.

`ERROR: FETCH/RELATIVE at current position is not supported`

Nach dem SQL-Standard kann man die Zeile an der „aktuellen Cursorposition“ wiederholt mit der Syntax

```
FETCH RELATIVE 0 FROM cursor.
```

auslesen. PostgreSQL unterstützt das gegenwärtig nicht; die 0 gibt vielmehr an, dass alle Zeilen gelesen werden sollen, wie mit dem Schlüsselwort `ALL`. Wenn das Schlüsselwort `RELATIVE` verwendet wurde, nimmt PostgreSQL an, dass der Benutzer das Verhalten nach dem SQL-Standard wollte, und gibt diese Fehlermeldung aus.

Hinweise

PostgreSQL unterstützt gegenwärtig nicht das Aktualisieren von Daten durch einen Cursor.

Cursor gelten nur in der Transaktion, in der sie deklariert worden sind.

Beispiele

Im folgenden Beispiel wird eine Tabelle mit einem Cursor gelesen.

```
BEGIN WORK;

-- Cursor einrichten:
DECLARE liahona CURSOR FOR SELECT * FROM filme;

-- Die ersten 5 Zeilen aud Cursor liahona lesen:
FETCH FORWARD 5 FROM liahona;

 code  |          titel          | vid | prod_datum |   genre    | länge
-------+-------------------------+-----+------------+------------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama      | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantisch | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantisch | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action     | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama      | 02:28

-- Die vorangegangene Zeile lesen:
FETCH BACKWARD 1 IN liahona;

 code  |  titel  | vid | prod_datum | genre  | länge
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08
```

```
-- Den Cursor schließen und die Transaktion beenden:
CLOSE liahona;
COMMIT WORK;
```

Kompatibilität

Der SQL-Standard definiert FETCH nur für eingebettetes SQL. Die hier beschriebene Variante von FETCH gibt die Daten zurück, als ob sie ein SELECT-Ergebnis wären, anstatt sie in Host-Variablen abzulegen. Abgesehen davon ist FETCH mit dem SQL-Standard kompatibel.

Die Formen von FETCH mit FORWARD und BACKWARD sowie die Formen FETCH *anzahl* und FETCH ALL, bei denen FORWARD implizit ist, sind PostgreSQL-Erweiterungen.

Der SQL-Standard erlaubt nur FROM vor dem Cursornamen; die Möglichkeit, IN zu verwenden, ist eine Erweiterung.

Siehe auch

DECLARE *(Seite: 707)*, MOVE *(Seite: 749)*

GRANT

Name

GRANT – definiert Zugriffsprivilegien

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER } [,...]
        | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tabellenname [, ...]
    TO { benutzername | GROUP gruppenname | PUBLIC } [, ...]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE datenbankname [, ...]
    TO { benutzername | GROUP gruppenname | PUBLIC } [, ...]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funktionsname ([typ, ...]) [, ...]
    TO { benutzername | GROUP gruppenname | PUBLIC } [, ...]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE sprachname [, ...]
    TO { benutzername | GROUP gruppenname | PUBLIC } [, ...]
```

```
GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    TO { benutzername | GROUP gruppenname | PUBLIC } [, ...]
```

Beschreibung

Der Befehl GRANT erteilt Benutzern oder Benutzergruppen bestimmte Privilegien für Datenbankobjekte (Tabellen, Sichten, Sequenzen, Datenbankobjekte, Funktionen, Sprachen oder Schemas). Diese Privilegien werden zu eventuell schon vorhandenen hinzugefügt.

Das Schlüsselwort PUBLIC zeigt an, dass die Privilegien allen Benutzern gewährt werden sollen, einschließlich Benutzern, die erst später erzeugt werden. Man kann sich PUBLIC als eine implizit definierte Gruppe vorstellen, die immer alle Benutzer enthält. Ein Benutzer hat immer die Summe aller Privilegien, die ihm direkt, allen Gruppen, denen er gegenwärtig angehört, und PUBLIC gewährt worden sind.

Dem Eigentümer des Objekts (normalerweise der, der es erzeugt hat) müssen keine Privilegien gewährt werden, da er immer anfänglich alle Privilegien hat. (Der Eigentümer könnte sich jedoch seine eigenen Privilegien aus Sicherheitsgründen entziehen.) Nur der Eigentümer hat die Erlaubnis, Privilegien zu gewähren und zu entziehen; dieses Privileg kann er auch nicht verlieren. Das Recht, das Objekt zu löschen oder es auf irgendeine Art zu verändern, liegt ebenfalls beim Eigentümer und kann nicht weitergegeben oder entzogen werden.

Je nach Art des Objekts hat PUBLIC in der Voreinstellung bestimmte Privilegien für das Objekt. Auf Tabellen und Schemas hat PUBLIC in der Voreinstellung keinen Zugriff, für Datenbanken hat PUBLIC in der Voreinstellung das Privileg TEMP (zur Erzeugung von temporären Tabellen), für Funktionen EXECUTE und für Sprachen USAGE. Der Eigentümer des Objekts kann diese Privilegien natürlich entziehen. (Die beste Sicherheit erreicht man, wenn man das REVOKE in derselben Transaktion ausführt, die das Objekt erzeugt; dann gibt es kein Fenster, in dem ein anderer Benutzer das Objekt verwenden kann.)

Die möglichen Privilegienarten sind:

SELECT

Erlaubt SELECT *(Seite: 759)* aus jeder Spalte der Tabelle, Sicht oder Sequenz und erlaubt COPY TO *(Seite: 646)*. Bei Sequenzen erlaubt dieses Privileg außerdem die Verwendung der Funktion currval.

INSERT

Erlaubt INSERT *(Seite: 743)* in der angegebenen Tabelle und COPY FROM *(Seite: 646)*.

UPDATE

Erlaubt UPDATE *(Seite: 782)* in jeder Spalte der angegebenen Tabelle. SELECT ... FOR UPDATE benötigt dieses Privileg ebenfalls (neben dem Privileg SELECT). Bei Sequenzen erlaubt dieses Privileg außerdem die Verwendung der Funktionen nextval und setval.

DELETE

Erlaubt DELETE *(Seite: 709)* in der angegebenen Tabelle.

RULE

Erlaubt die Erzeugung von Regeln für die Tabelle oder Sicht. (Siehe Befehl CREATE RULE *(Seite: 679)*.)

REFERENCES

Um einen Fremdschlüssel-Constraint erzeugen zu dürfen, muss man dieses Privileg für die Tabelle mit dem Constraint und die Tabelle, auf die er verweist, haben.

TRIGGER

Erlaubt die Erzeugung von Triggern für die angegebene Tabelle. (Siehe Befehl CREATE TRIGGER *(Seite: 695)*.)

CREATE

Bei Datenbanken erlaubt dieses Privileg, dass neue Schemas in der Datenbank erzeugt werden können.

Bei Schemas erlaubt dieses Privileg, dass neue Objekte in dem Schema erzeugt werden können. Um ein bestehendes Objekt umbenennen zu können, müssen Sie das Objekt besitzen *und* dieses Privileg für das übergeordnete Schema haben.

TEMPORARY
TEMP

Erlaubt die Erzeugung von temporären Tabellen während der Benutzung der Datenbank.

EXECUTE

Erlaubt die Verwendung der angegebenen Funktion und aller Operatoren, die auf Grundlage dieser Funktion implementiert sind. Dies ist der einzige Privilegientyp für Funktionen. (Die Syntax funktioniert auch für Aggregatfunktionen.)

USAGE

Bei prozeduralen Sprachen erlaubt dieses Privileg, dass die Sprache zur Erzeugung von neuen Funktionen verwendet werden darf. Dies ist der einzige Privilegientyp für Sprachen.

Bei Schemas erlaubt dieses Privileg, dass man auf Objekte in diesem Schema zugreifen darf (wenn die Privilegien für das Objekt selbst ebenfalls vorhanden sind).

ALL PRIVILEGES

Damit werden alle Privilegien, die für ein Objekt zutreffen, auf einmal gewährt. Das Schlüsselwort PRIVILEGES ist in PostgreSQL optional, nicht jedoch nach dem SQL-Standard.

Die Privilegien, die für andere Befehle benötigt werden, sind auf der Referenzseite des entsprechenden Befehls erläutert.

Hinweise

Datenbank-Superuser können auf alle Objekte zugreifen, egal, ob sie Privilegien dafür haben oder nicht. Das ist mit den Rechten des Benutzers root in Unix-Systemen vergleichbar. Wie bei root sollte man sich nur wenn nötig als Superuser anmelden.

Um Privilegien nur für einige Spalten zu gewähren, müssen Sie in PostgreSQL gegenwärtig eine Sicht erzeugen, die die gewünschten Spalten enthält, und dann Privilegien für diese Sicht gewähren.

Mit dem `psql`-Befehl *(Seite: 819)* `\z` erhalten Sie Informationen über vorhandene Privilegien, zum Beispiel:

```
=> \z meine_tabelle

        Access privileges for database "lusitania"
 Schema |     Table     |           Access privileges
--------+---------------+---------------------------------------
 public | meine_tabelle | {=r,miriam=arwdRxt,"group todos=arw"}
(1 row)
Die von \z gezeigten Einträge sind folgendermaßen zu interpretieren:
              =xxxx -- Privilegien von PUBLIC
         bname=xxxx -- Privilegien eines Benutzers
   group gname=xxxx -- Privilegien einer Gruppe

                  r -- SELECT ("read")
```

```
                    w -- UPDATE ("write")
                    a -- INSERT ("append")
                    d -- DELETE
                    R -- RULE
                    x -- REFERENCES
                    t -- TRIGGER
                    X -- EXECUTE
                    U -- USAGE
                    C -- CREATE
                    T -- TEMPORARY
               arwdRxt -- ALL PRIVILEGES (für Tabellen)
```

So wie oben würde die Ausgabe für die Benutzerin `miriam` aussehen, nachdem die Tabelle `meine_tabelle` erzeugt worden ist und Privilegien wie folgt gewährt worden sind:

```
GRANT SELECT ON meine_tabelle TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON meine_tabelle TO GROUP todos;
```

Wenn die Spalte „Access privileges“ für ein bestimmtes Objekt leer ist, bedeutet das, dass das Objekt die Standardprivilegien hat (die Privilegienspalte hat den NULL-Wert). Die Standardprivilegien enthalten immer alle Privilegien für den Eigentümer und je nach Objekt einige Privilegien für `PUBLIC`, wie oben beschrieben. Das erste `GRANT` oder `REVOKE` für ein Objekt setzt die Standardprivilegien ein (in diesem Beispiel `{=,miriam=arwdRxt}`) und verändert Sie dann wie von dem Befehl gefordert.

Beispiele

Gewähre allen Benutzern das Privileg, in die Tabelle `filme` einfügen zu können:

```
GRANT INSERT ON filme TO PUBLIC;
```

Gewähre dem Benutzer `manuel` alle Privilegien für die Sicht `genres`:

```
GRANT ALL PRIVILEGES ON genres TO manuel;
```

Kompatibilität

Laut dem SQL-Standard ist das Schlüsselwort `PRIVILEGES` in `ALL PRIVILEGES` Pflicht. SQL erlaubt es nicht, in einem Befehl Privilegien für mehr als ein Objekt zu gewähren.

Der SQL-Standard erlaubt Privilegien für einzelne Spalten in Tabellen und erlaubt es, anderen Benutzern zu ermöglichen, ihre Privilegien an andere Benutzer weiterzugeben (`GRANT OPTION`).

```
GRANT privilegien
    ON objekt [ ( spalte [, ...] ) ] [, ...]
    TO { PUBLIC | benutzername [, ...] } [ WITH GRANT OPTION ]
```

Der SQL-Standard sieht das Privileg `USAGE` würde weitere Objektarten vor: Zeichensätze, Kollationen, Translationen und Domänen.

Das Privileg `RULE` und Privilegien für Datenbanken, Schemas, Sprachen und Sequenzen sind PostgreSQL-Erweiterungen.

Siehe auch

REVOKE *(Seite: 756)*

INSERT

Name

INSERT – erzeugt neue Zeilen in einer Tabelle

Synopsis

```
INSERT INTO tabelle [ ( spalte [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { ausdruck | DEFAULT } [, ...] ) | anfrage }
```

Beschreibung

INSERT fügt neue Zeilen in eine Tabelle ein. Man kann einzelne Zeilen einfügen oder mehrere Zeilen aus dem Ergebnis einer Anfrage.

Die Spalten in der Spaltenliste können in beliebiger Reihenfolge aufgelistet werden. Für jede Spalte, die nicht aufgezählt ist, wird der Vorgabewert eingefügt, entweder ein ausdrücklich definierter oder der NULL-Wert.

Wenn die Ausdrücke für jede Spalte nicht den richtigen Datentyp haben, wird versucht, sie automatisch in den richtigen Typ umzuwandeln.

Um in eine Tabelle einfügen zu können, müssen Sie das Privileg INSERT für die Tabelle haben. Wenn Sie die Klausel *anfrage* verwenden, um Zeilen aus einer Anfrage einzufügen, benötigen Sie außerdem das Privileg SELECT für alle Tabellen, die in der Anfrage verwendet werden.

Parameter

tabelle

Der Name einer Tabelle (möglicherweise mit Schemaqualifikation).

spalte

Der Name einer Spalte in *tabelle*.

DEFAULT VALUES

All Spalten werden mit dem Vorgabewert gefüllt.

ausdruck

Ein Ausdruck oder Wert, der der Spalte zugewiesen werden soll.

DEFAULT

Diese Spalte wird mit ihrem Vorgabewert gefüllt.

anfrage

Eine Anfrage (SELECT-Befehl), die die einzufügenden Zeilen liefert. Eine Beschreibung der Syntax erhalten Sie beim SELECT-Befehl.

Meldungen

`INSERT` *`oid`* `1`

Meldung, wenn nur eine Zeile eingefügt wurde. *`oid`* ist die OID der neuen Zeile.

`INSERT 0` *`zahl`*

Meldung, wenn mehr als eine Zeile eingefügt wurde. *`zahl`* ist die Anzahl der eingefügten Zeilen.

Beispiele

Das erste Beispiel fügt eine einzelne Zeile in die Tabelle `filme` ein:

```
INSERT INTO filme
    VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

Im zweiten Beispiel wird die letzte Spalte `länge` weggelassen und erhält daher den NULL-Wert als Vorgabewert:

```
INSERT INTO filme (code, titel, vid, prod_datum, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

Im dritten Beispiel wird die Klausel `DEFAULT` für die Datumsspalten verwendet, anstatt einen Wert anzugeben:

```
INSERT INTO filme
    VALUES ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO filme (code, titel, vid, prod_datum, genre)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

Dieses Beispiel fügt mehrere Zeilen aus der Tabelle `tmp` in die Tabelle `filme` ein:

```
INSERT INTO filme SELECT * FROM tmp;
```

Dieses Beispiel fügt Werte in Arrayspalten ein:

```
-- Erzeuge ein 3x3-Spielfeld für Tic-Tac-Toe
-- (jeder dieser Befehle erzeugt das gleiche Spielfeld)
INSERT INTO tictactoe (spiel, brett[1:3][1:3])
    VALUES (1, '{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (spiel, brett[3][3])
    VALUES (2, '{}');
INSERT INTO tictactoe (spiel, brett)
    VALUES (3, '{{,,},{,,},{,,}}');
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform. Mögliche Einschränkungen in der Klausel *anfrage* sind unter SELECT *(Seite: 759)* beschrieben.

LISTEN

Name

LISTEN – hört auf eine Benachrichtigung

Synopsis

```
LISTEN name
```

Beschreibung

LISTEN registriert die aktuelle Sitzung für die Benachrichtigungsmitteilung *name*.

Immer wenn der Befehl NOTIFY *name* in dieser Sitzung oder einer anderen in derselben Datenbank ausgeführt wird, werden alle Sitzungen, die auf diese Benachrichtigung hören, informiert, und informieren wiederum ihre Clientanwendungen. Weitere Informationen dazu erhalten Sie unter NOTIFY.

Eine Sitzung kann die Registrierung für eine Benachrichtigungen mit dem Befehl UNLISTEN löschen. Die Registrierungen einer Sitzung werden auch automatisch gelöscht, wenn die Sitzung endet.

Wie eine Clientanwendung Benachrichtigungsmitteilung entdecken kann, hängt von der Programmierschnittstelle ab. Mit der Bibliothek libpq führt die Anwendung LISTEN als normalen SQL-Befehl aus und muss dann regelmäßig die Funktion PQnotifies aufrufen, um herauszufinden, ob Benachrichtigungsmitteilungen empfangen wurden. Andere Schnittstellen wie libpgtcl bieten abstraktere Methoden, um mit Benachrichtigungen umzugehen; in libpgtcl sollte der Anwendungsprogrammierer LISTEN und UNLISTEN gar nicht direkt aufrufen. Einzelheiten finden Sie in der Beschreibung der Schnittstelle, die Sie verwenden.

NOTIFY *(Seite: 750)* enthält eine ausführlichere Beschreibung der Verwendung der Befehle LISTEN und NOTIFY.

Parameter

name

Der Name einer Benachrichtigungsmitteilung (ein beliebiger SQL-Bezeichner).

Meldungen

LISTEN

Meldung, wenn die Registrierung erfolgreich abgeschlossen wurde.

WARNING: Async_Listen: We are already listening on *name*

Meldung, wenn diese Sitzung bereits auf diese Benachrichtigungsmitteilung hört.

Beispiele

So sieht es in `psql` aus, wenn man sich für eine Benachrichtigung registriert und sie dann selbst auslöst:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Kompatibilität

Der Befehl `LISTEN` ist eine PostgreSQL-Erweiterung.

LOAD

Name

LOAD – lädt eine dynamische Bibliotheksdatei

Synopsis

```
LOAD 'dateiname'
```

Beschreibung

Dieser Befehl lädt eine dynamische Bibliotheksdatei in den Adressraum des PostgreSQL-Servers. Wenn die Datei vorher schon geladen war, wird sie zuerst entfernt und dann neu geladen. Dieser Befehl ist hauptsächlich nützlich, um eine dynamische Bibliotheksdatei neu zu laden, wenn sie seit dem letzten Laden verändert worden ist. Um aus einer dynamischen Bibliothek Nutzen zu ziehen, müssen die darin enthaltenen Funktionen mit dem Befehl `CREATE FUNCTION` *(Seite: 664)* deklariert werden.

Der Dateiname wird genauso wie bei dynamischen Bibliotheken in `CREATE FUNCTION` *(Seite: 664)* angegeben; insbesondere kann man den Suchpfad verwenden und die Standarddateierweiterung für dynamische Bibliotheken weglassen. Einzelheiten dazu finden Sie in Abschnitt SET.

Kompatibilität

Der Befehl `LOAD` ist eine `PostgreSQL`-Erweiterung.

Siehe auch

`CREATE FUNCTION` *(Seite: 664)*

LOCK

Name

LOCK – sperrt eine Tabelle

Synopsis

```
LOCK [ TABLE ] name [, ...] [ IN sperrmodus MODE ]
```

wobei *sperrmodus* Folgendes sein kann:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Beschreibung

`LOCK TABLE` setzt eine Sperre auf Tabellenebene und wartet, wenn nötig, bis Sperren, die in Konflikt zu dieser stehen, aufgegeben werden. Wenn eine Sperre gesetzt wurde, dann bleibt sie bis zum Ende der aktuellen Transaktion erhalten. (Es gibt keinen Befehl `UNLOCK TABLE`; Sperren werden immer am Transaktionsende aufgegeben.)

PostgreSQL setzt immer die am wenigsten restriktive Sperre, wenn ein Befehl eine verwendete Tabelle automatisch sperrt. `LOCK TABLE` ist für Fälle gedacht, in denen Sie restriktivere Sperren benötigen. Nehmen wir an, Sie wollen eine Transaktion im Isolationsgrad *`Read Committed`* ausführen und müssen sicherstellen, dass die Daten in einer Tabelle während der Transaktion stabil bleiben. Dazu könnten Sie für die Tabelle eine Sperre im Modus `SHARE` setzen, bevor Sie die Anfragen starten. Dadurch werden gleichzeitige Datenänderungen verhindert und den folgenden Lesevorgängen eine stabile Sicht auf die geschriebenen Daten ermöglicht, weil die Sperre im Modus `SHARE` mit der Sperre im Modus `ROW EXCLUSIVE`, welche von Lesevorgängen angefordert wird, in Konflikt steht und Ihr Befehl `LOCK TABLE` *`name`* `IN SHARE MODE` wartet, bis alle Halter einer Sperre des Modus `ROW EXCLUSIVE` abschließen oder zurückrollen. Wenn Sie also die Sperre erst einmal gesetzt haben, dann gibt es keine Transaktionen mehr, die eventuell noch zu schreibende Daten ausstehen haben; des Weiteren müssen neue Schreibvorgänge warten, bis die Sperre aufgegeben wird.

Wenn Sie eine ähnliche Wirkung erzielen wollen und die Transaktion im Isolationsgrad *`Serializable`* ausführen, müssen Sie den Befehl `LOCK TABLE` vor allen Datenmodifikationsbefehlen ausführen. Die Datensicht einer serialisierbaren Transaktion wird eingefroren, wenn der erste Datenmodifikationsbefehl ausgeführt wird. Ein später ausgeführtes `LOCK TABLE` verhindert trotzdem gleichzeitige Schreibvorgänge, aber es kann nicht versichern, dass die Daten, die von der Transaktion gelesen werden, auch die letzten gültigen Daten sind.

Wenn eine Transaktion dieser Art die Daten in der Tabelle ändert möchte, sollte sie statt dem Modus `SHARE` den Modus `SHARE ROW EXCLUSIVE` verwenden. Damit erreicht man, dass nur eine Transaktion dieser Art zur gleichen Zeit laufen kann. Ohne diese Maßnahme ist eine Verklemmung möglich: Zwei Transaktionen könnten beide eine Sperre im Modus `SHARE` setzen und dann wäre es beiden nicht möglich, eine Sperre im Modus `ROW EXCLUSIVE` zu erlangen, die sie für den Schreibvorgang benötigen. (Beachten Sie, dass die eigenen Sperren einer Transaktion niemals mit sich in Konflikt stehen. Also kann eine Transaktion eine `ROW EXCLUSIVE`-Sperre setzen, wenn Sie schon eine `SHARE`-Sperre hält – aber nicht, wenn eine andere Transaktion eine `SHARE`-Sperre hält.) Um Verklemmungen zu vermeiden, sollten Sie dafür sorgen, dass alle Transaktionen, die Sperren für dieselben Objekte in derselben Reihenfolge

anfordern, und dass, wenn mehrere Sperren für ein Objekt im Spiel sind, Transaktionen immer die Sperre im restriktivsten Modus zuerst setzen.

Weitere Informationen über die Sperrmodi und die Verwendung von Sperren finden Sie in Abschnitt SET.

Parameter

name

Der Name der zu sperrenden Tabelle (möglicherweise mit Schemaqualifikation).

Ein Befehl der Form `LOCK a, b;` hat die gleiche Auswirkung wie `LOCK a; LOCK b;`. Die Tabellen werden einzeln in der Reihenfolge, wie sie im `LOCK`-Befehl angegeben wurden, gesperrt.

sperrmodus

Der Sperrmodus gibt an, mit welchen Sperren diese Sperre in Konflikt steht. Die Sperrmodi sind in Abschnitt SET beschrieben.

Wenn kein Sperrmodus angegeben ist, wird `ACCESS EXCLUSIVE`, der restriktivste Sperrmodus, verwendet.

Meldungen

`LOCK TABLE`

Meldung, wenn die Sperre erfolgreich gesetzt wurde.

Hinweise

Für `LOCK ... IN ACCESS SHARE MODE` benötigt man das Privileg `SELECT` für die Zieltabelle. Für alle anderen Formen von `LOCK` benötigt man das Privileg `UPDATE` oder `DELETE`.

`LOCK` ist nur in einem Transaktionsblock (`BEGIN`/`COMMIT`-Paar) sinnvoll, weil die Sperre am Ende einer Transaktion aufgegeben wird. Wenn `LOCK` außerhalb eines Transaktionsblocks aufgerufen wird, bildet es selbst eine Transaktion, also würde die Sperre sofort nach Ende des Befehls wieder aufgegeben werden.

`LOCK TABLE` setzt nur Sperren auf Tabellenebene. Die Modusnamen, die `ROW` (Zeile) enthalten, sind unzutreffend. Diese Modusnamen sollten generell als Hinweis auf die Absicht des Benutzers verstanden werden, in der gesperrten Tabelle Zeilensperren zu setzen. Außerdem ist der Modus `ROW EXCLUSIVE`, keine exklusive sondern eine geteilte Sperre. Merken Sie sich, dass die Sperrmodi, soweit es `LOCK TABLE` betrifft, alle identische Bedeutungen haben und sich nur dadurch unterscheiden, mit welchen Sperrmodi sie in Konflikt stehen.

Beispiele

Dieses Beispiel verwendet eine Sperre im Modus `SHARE` für die Primärschlüsseltabelle, um die Fremdschlüsseltabelle sicher aktualisieren zu können:

```
BEGIN WORK;
LOCK TABLE filme IN SHARE MODE;
SELECT id FROM filme
    WHERE name = 'Star Wars: Episode I – Die dunkle Bedrohung';
-- Hier ROLLBACK, wenn kein Datensatz zurückgegeben wurde
```

```
INSERT INTO filme_benutzerkommentare VALUES
    (_id_, 'Super! Habe lange genug darauf gewartet!');
COMMIT WORK;
```

Dieses Beispiel setzt eine Sperre im Modus SHARE ROW EXCLUSIVE für die Primärschlüsseltabelle, während eine Löschoperation durchgeführt wird:

```
BEGIN WORK;
LOCK TABLE filme IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM filme_benutzerkommentare WHERE id IN
    (SELECT id FROM filme WHERE bewertung < 5);
DELETE FROM filme WHERE bewertung < 5;
COMMIT WORK;
```

Kompatibilität

Im SQL-Standard gibt es keinen Befehl LOCK TABLE. Der SQL-Standard bietet nur SET TRANSACTION, um die Konsistenz im Mehrbenutzerbetrieb zu kontrollieren. PostgreSQL bietet diese Funktionalität ebenfalls; siehe SET TRANSACTION *(Seite: 776)*.

Außer den Sperrmodi ACCESS SHARE, ACCESS EXCLUSIVE und SHARE UPDATE EXCLUSIVE sind die PostgreSQL-Sperrmodi und die Syntax des Befehls LOCK TABLE mit Oracle kompatibel.

MOVE

Name

MOVE – positioniert einen Cursor

Synopsis

```
MOVE [ FORWARD | BACKWARD | RELATIVE ] [ anzahl | ALL | NEXT | PRIOR ] { FROM | IN
} cursor
```

Beschreibung

MOVE positioniert einen Cursor, ohne Daten zu lesen. MOVE funktioniert genauso wie der Befehl FETCH, außer dass es nur den Cursor positioniert und keine Daten zurückgibt.

Einzelheiten zu Syntax und Verwendung finden Sie unter FETCH *(Seite: 737)*.

Beispiele

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM filme;
```

```
-- Die ersten 5 Zeilen überspringen:
MOVE FORWARD 5 IN liahona;

-- Die 6. Zeile aus dem Cursor liahona lesen:
FETCH 1 FROM liahona;
 code  |   titel    | vid | prod_datum | genre  | länge
-------+------------+-----+------------+--------+-------
 P_303 | 48 Stunden | 103 | 1982-10-22 | Action | 01:37

-- Den Cursor schließen und die Transaktion beenden:
CLOSE liahona;
COMMIT WORK;
```

Kompatibilität

Der Befehl `MOVE` ist eine PostgreSQL-Erweiterung.

NOTIFY

Name

NOTIFY – erzeugt eine Benachrichtigung

Synopsis

```
NOTIFY name
```

Beschreibung

Der Befehl `NOTIFY` sendet eine Benachrichtigung an jede Clientanwendung, die zuvor `LISTEN` *`name`* für den angegebenen Benachrichtigungsnamen in der aktuellen Datenbank ausgeführt hat.

Die dem Client bei einer Benachrichtigung übergebenen Informationen bestehen aus dem Benachrichtigungsnamen und der PID des Serverprozesses, der zu der Sitzung gehört, die die Benachrichtigung ausgelöst hat. Es liegt am Datenbankentwickler, die Benachrichtigungsnamen zu bestimmen, die in einer Datenbank verwendet werden, und was jeder Name bedeutet.

Gewöhnlich ist der Name der Benachrichtigung der gleiche wie der Name einer Tabelle in der Datenbank, und die Benachrichtigung sagt im Prinzip aus: „Ich habe diese Tabelle verändert, schaut nach, was neu ist“. Aber die Befehle `NOTIFY` und `LISTEN` erzwingen keine derartigen Zusammenhänge. Ein Datenbankentwickler könnte zum Beispiel mehrere verschiedene Benachrichtigungsnamen vorsehen, um verschiedene Arten von Änderungen in einer einzigen Tabelle anzuzeigen.

`NOTIFY` stellt Prozessen, die auf dieselbe PostgreSQL-Datenbank zugreifen, eine einfache Form von Signalen oder Interprozesskommunikation zur Verfügung. Leistungsfähigere Mechanismen können gebaut

werden, indem Tabellen in der Datenbank verwendet werden, um zusätzliche Daten (anstatt nur den Benachrichtigungsnamen) vom Auslöser an die Zuhörer zu übergeben.

Wenn NOTIFY verwendet werden soll, um Veränderungen in einer bestimmten Tabelle mitzuteilen, dann ist es sinnvoll, den NOTIFY-Befehl von einer Regel ausführen zu lassen, die von entsprechenden Befehlen, die die Tabelle aktualisieren, ausgelöst wird. Dadurch werden die Benachrichtigungen automatisch ausgelöst, wenn die Tabelle geändert wird, und der Anwendungsprogrammierer kann es nicht aus Versehen vergessen.

NOTIFY hat einige wichtige Wechselwirkungen mit SQL-Transaktionen. Erstens, wenn NOTIFY in einer Transaktion ausgeführt wird, werden die Benachrichtigungen erst und nur abgeschickt, wenn die Transaktion erfolgreich abgeschlossen wird. Das ist sinnvoll, denn wenn die Transaktion abgebrochen würde, dann sollten alle darin enthaltenen Befehle, einschließlich des NOTIFY, ohne Auswirkung bleiben. Es kann aber verwirrend sein, wenn man erwartet, dass die Benachrichtigungen sofort abgeschickt werden. Zweitens, wenn eine zuhörende Sitzung eine Benachrichtigung erhält, während sie sich in einer Transaktion befindet, wird die Benachrichtigung erst an den Client weitergegeben, wenn die Transaktion beendet ist (erfolgreich abgeschlossen oder abgebrochen). Der Grund ist wiederum, dass eine Benachrichtigung, die in einer Transaktion ankommt, die später abgebrochen wird, irgendwie zurückgenommen werden sollte – aber der Server kann keine Benachrichtigungen stornieren, wenn Sie schon an den Client gesendet worden ist. Also werden Benachrichtigungen nur zwischen Transaktionen versendet. Die Konsequenz daraus ist, dass Anwendungen, die mit NOTIFY Signale in Echtzeit austauschen wollen, ihre Transaktionen kurz halten sollten.

NOTIFY verhält sich in einer Hinsicht wie Unix-Signale: Wenn Benachrichtigungen mit dem gleichen Namen in kurzen Abständen erzeugt werden, kann es sein, dass die Empfänger nur eine Benachrichtigung bei mehreren Aufrufen von NOTIFY empfangen. Man sollte sich also nicht auf die Anzahl der empfangenen Benachrichtigungen verlassen. Stattdessen sollte man NOTIFY nur verwenden, um Anwendungen, deren Aufmerksamkeit man wünscht, aufzuwecken, und dann ein Datenbankobjekt (zum Beispiel eine Sequenz) verwenden, um festzuhalten, was passiert ist oder wie oft es passiert ist.

Es ist nicht ungewöhnlich, dass ein Client, der NOTIFY ausführt, selbst auf diesen Benachrichtigungsnamen hört. In diesem Fall erhält auch er eine Benachrichtigung wie alle anderen darauf wartenden Sitzungen. Je nach Anwendungsaufbau ergibt sich daraus nutzlose Arbeit, zum Beispiel beim erneuten Lesen einer Datenbanktabelle, die gerade selbst geschrieben wurde. Solche nutzlose Arbeit kann man vermeiden, indem man die PID des Serverprozesses der Sitzung, die die Benachrichtigung erzeugt hat (in der Benachrichtigungsmitteilung enthalten) mit der PID des Serverprozesses der eigenen Sitzung (in libpq verfügbar) vergleicht. Wenn sie gleich sind, ist die Benachrichtigung die eigene zurückgekommene und kann ignoriert werden. (Trotz der Aussagen im vorangegangenen Absatz ist diese Technik sicher. PostgreSQL hält die Selbstbenachrichtigungen von denen aus anderen Sitzungen getrennt, also können Sie keine Benachrichtigungen von anderen Sitzungen verpassen, wenn Sie ihre eigenen ignorieren.)

Parameter

name

Der Name der Benachrichtigungsmitteilung, die ausgelöst werden soll (ein beliebiger SQL-Bezeichner).

Meldungen

NOTIFY

Meldung, wenn der Befehl ausgeführt wurde.

Beispiele

So sieht es in psql aus, wenn man sich für eine Benachrichtigung registriert und sie dann selbst auslöst:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received.
```

Kompatibilität

Der Befehl NOTIFY ist eine PostgreSQL-Erweiterung.

PREPARE

Name

PREPARE – bereitet einen Befehl zur Ausführung vor

Synopsis

```
PREPARE planname [ (datentyp [, ...] ) ] AS befehl
```

Beschreibung

PREPARE erzeugt einen vorbereiteten Befehl. Ein vorbereiteter Befehl ist ein serverseitiges Objekt, das zur Verbesserung der Leistung beitragen kann. Wenn der Befehl PREPARE ausgeführt wird, wird der angegebene Befehl geparst, umgeschrieben und geplant. Wenn danach der Befehl EXECUTE aufgerufen wird, muss der vorbereitete Befehl nur noch ausgeführt werden. Das Parsen, Umschreiben und Planen geschieht also nur einmal anstatt jedes Mal, wenn der Befehl ausgeführt wird.

Vorbereitete Befehle können Parameter enthalten. Das sind Werte, die erst in den Befehl eingesetzt werden, wenn er ausgeführt wird. Um in einem vorbereiteten Befehl Parameter zu verwenden, geben Sie im PREPARE-Befehl eine Liste der Datentypen der Parameter an und verweisen Sie im vorzubereitenden Befehl selbst mit $1, $2 usw. über die Parameterposition auf die Parameter. Wenn der Befehl ausgeführt wird, werden die tatsächlichen Werte für die Parameter im Befehl EXECUTE angegeben. Weitere Informationen dazu erhalten Sie unter EXECUTE *(Seite: 733)*.

Vorbereitete Befehle werden nur in und für die Dauer der aktuellen Sitzung gespeichert. Wenn die Sitzung beendet wird, dann wird der vorbereitete Befehl vergessen und muss also vor der nächsten Verwendung erneut vorbereitet werden. Das bedeutet auch, dass ein vorbereiteter Befehl nicht von mehreren Clientanwendungen gleichzeitig verwendet werden kann; jeder Client kann jedoch den Befehl selbst vorbereiten.

Vorbereitete Befehle ergeben den größten Leistungsvorteil, wenn eine Sitzung ein große Zahl ähnlicher Befehle ausführt. Der Leistungsunterschied wird besonders erheblich sein, wenn die Befehle aufwendig zu planen oder umzuschreiben sind, zum Beispiel Anfragen mit vielen Verbunden oder vielen anzuwendenden Regeln. Wenn die Anfrage ziemlich einfach geplant und umgeschrieben werden kann, aber die Ausführung relativ aufwendig ist, wird der Vorteil eines vorbereiteten Befehls weniger erkennbar sein.

Parameter

planname

Ein beliebiger Name für diesen vorbereiteten Befehl. Er darf in dieser Sitzung nicht schon einmal vergeben sein und wird in der Folge verwendet, um den vorbereiteten Befehl auszuführen oder freizugeben.

datentyp

Der Datentyp eines Parameters des vorbereiteten Befehls. Im Befehl selbst wird auf die Parameter mit `$1`, `$2` usw. verwiesen.

befehl

Ein beliebiger `SELECT`-, `INSERT`-, `UPDATE`- oder `DELETE`-Befehl.

Meldungen

`PREPARE`

Meldung, wenn der Befehl erfolgreich vorbereitet wurde.

Hinweise

In einigen Situationen kann es vorkommen, dass der Anfrageplan für einen vorbereiteten Befehl schlechter ist, als wenn der Befehl normal ausgeführt worden wäre. Das kommt daher, dass beim Planen des Befehls und beim Ermitteln des besten Plans die tatsächlichen Werte für eventuell vorhandene Parameter nicht zur Verfügung stehen. PostgreSQL sammelt Statistiken über die Verteilung der Daten in der Tabelle und kann bei konstanten Werten in Befehlen, das Ergebnis des Befehls schätzen. Wenn Befehle mit Parametern geplant werden, fehlen diese Informationen und daher kann es sein, dass der gewählte Plan nicht der optimale ist. Um den Plan einzusehen, den PostgreSQL für einen vorbereiteten Befehl ausgewählt hat, verwenden Sie `EXPLAIN EXECUTE`.

Weitere Informationen über Befehlsplanung und die von PostgreSQL dafür gesammelten Statistiken finden Sie unter `ANALYZE` *(Seite: 636)*.

Kompatibilität

Der SQL-Standard enthält einen Befehl `PREPARE`, aber er ist nur für eingebettetes SQL. Diese Version des `PREPARE`-Befehls hat auch eine etwas andere Syntax.

REINDEX

Name

REINDEX – baut Indexe neu

Synopsis

```
REINDEX { DATABASE | TABLE | INDEX } name [ FORCE ]
```

Beschreibung

REINDEX baut einen Index anhand der Daten in der Tabelle neu auf und ersetzt die alte Version des Index. Es gibt zwei hauptsächliche Gründe, warum man REINDEX verwenden würde:

- Ein Index wurde zerstört oder verfälscht und enthält keine gültigen Daten mehr. Obwohl das theoretisch nie passieren sollte, kann es aufgrund von Software- oder Hardwarefehlern vorkommen. REINDEX ist eine Methode, um solche Situationen zu bereinigen.
- Der betroffene Index enthält viele tote Indexseiten, die nicht wiederverwendet werden können. Das kann bei B-Tree-Indexen in PostgreSQL bei bestimmten Zugriffsmustern auftreten. Mit REINDEX kann der Platzverbrauch eines Index verringert werden, weil eine neue Version des Index ohne die toten Seiten geschrieben wird. Weitere Informationen dazu finden Sie in Abschnitt SET.

Wenn Sie einen verfälschten Index für eine Benutzertabelle vermuten, können Sie diesen Index oder alle Indexe der Tabelle einfach mit REINDEX INDEX oder REINDEX TABLE neu aufbauen lassen. Eine andere Möglichkeit ist, den Index einfach zu löschen und neu zu erzeugen. Das ist eigentlich auch besser, wenn Sie die Tabelle währenddessen einigermaßen normal weiterverwenden wollen. REINDEX sperrt die Tabelle exklusiv, während CREATE INDEX nur Schreibvorgänge, aber keine Lesevorgänge aussperrt.

Etwas schwieriger wird es bei einem verfälschten Systemindex. In dem Fall ist es wichtig, dass das System den infrage stehenden Index nicht selbst schon verwendet hat. (In solchen Fällen kann es sein, dass Serverprozesse beim Start sofort abstürzen, weil Sie verfälschte Systemindexe zu verwenden versuchen.) Um das System sicher zu reparieren, muss der Server heruntergefahren und eine Sitzung im Einzelbenutzermodus mit den Kommandozeilenoptionen -O und -P gestartet werden. (Diese Optionen erlauben Veränderungen an den Systemtabellen bzw. unterbinden die Verwendung der Systemindexe.) Dann kann man REINDEX DATABASE, REINDEX TABLE oder REINDEX INDEX ausführen, je nachdem, wie viel man reparieren muss. Im Zweifelsfall verwenden Sie REINDEX DATABASE FORCE, um alle Systemindexe in der Datenbank neu aufzubauen. Danach verlassen Sie die Einzelbenutzersitzung und starten den richtigen Server neu.

Informationen über die Verwendung des Einzelbenutzerservers finden Sie unter postgres *(Seite: 852)*.

Parameter

DATABASE

Baut alle Systemindexe in der angegebenen Datenbank neu auf. Indexe für Benutzertabellen sind nicht betroffen. Diese Form von REINDEX kann nur im Einzelbenutzermodus ausgeführt werden (siehe oben).

TABLE

Baut alle Indexe der angegebenen Tabelle neu auf.

INDEX

Baut den angegebenen Index neu auf.

name

Der Name der bestimmten Datenbank, Tabelle oder des Index, der neu indiziert werden soll. Tabellen- und Indexnamen können eine Schemaqualifikation haben.

FORCE

Erzwingt die Erneuerung von Systemindexen. Ohne dieses Schlüsselwort überspringt REINDEX Systemindexe, die nicht als ungültig markiert worden sind. FORCE hat keine Bedeutung bei REINDEX INDEX oder wenn Benutzerindexe erneuert werden.

Meldungen

`REINDEX`
Meldung, wenn die Indexe erfolgreich erneuert wurden.

Beispiele

Baue alle Index für die Tabelle `meine_tabelle` neu auf:

```
REINDEX TABLE meine_tabelle;
```

Baue einen einzelnen Index neu auf:

```
REINDEX INDEX mein_index;
```

Baue alle Systemindexe neu auf (das funktioniert nur im Einzelbenutzermodus):

```
REINDEX DATABASE meine_datenbank FORCE;
```

Kompatibilität

Der Befehl `REINDEX` ist eine PostgreSQL-Erweiterung.

RESET

Name

RESET – setzt einen Konfigurationsparameter auf die Voreinstellung zurück

Synopsis

```
RESET parameter
RESET ALL
```

Beschreibung

`RESET` setzt Konfigurationsparameter auf ihre Voreinstellung zurück. `RESET` ist eine alternative Schreibweise für

```
SET parameter TO DEFAULT
```

Einzelheiten finden Sie unter `SET` *(Seite: 771)*.

Als Voreinstellung gilt der Wert, den der Parameter haben würde, wenn in der aktuellen Sitzung kein `SET` jemals dafür ausgeführt worden wäre. Die tatsächliche Quelle des Werts könnten der eingebaute Vorgabe-

wert, die Konfigurationsdatei, Kommandozeilenoptionen oder datenbank- oder benutzerspezifische Voreinstellungen sein. Einzelheiten dazu finden Sie in Abschnitt SET.

Informationen über das Transaktionsverhalten von RESET finden Sie auf der Referenzseite von SET.

Parameter

parameter

Der Name eines Konfigurationsparameters. Eine Liste finden Sie unter SET *(Seite: 771)*.

ALL

Setzt alle veränderbaren Konfigurationsparameter auf die Voreinstellungen zurück.

Meldungen

Siehe unter SET *(Seite: 771)*.

Beispiele

Setze datestyle auf die Voreinstellung zurück:

```
RESET datestyle;
```

Setze geqo auf die Voreinstellung zurück:

```
RESET geqo;
```

Kompatibilität

Der Befehl RESET ist eine PostgreSQL-Erweiterung.

REVOKE

Name

REVOKE – entfernt Zugriffsprivilegien

Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
[,...]
         | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tabellenname [, ...]
    FROM { benutzername | GROUP gruppenname | PUBLIC } [, ...]
```

```
REVOKE { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE datenbankname [, ...]
    FROM { benutzername | GROUP gruppenname | PUBLIC } [, ...]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION funktionsname ([typ, ...]) [, ...]
    FROM { benutzername | GROUP gruppenname | PUBLIC } [, ...]

REVOKE { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE sprachname [, ...]
    FROM { benutzername | GROUP gruppenname | PUBLIC } [, ...]

REVOKE { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schemaname [, ...]
    FROM { benutzername | GROUP gruppenname | PUBLIC } [, ...]
```

Beschreibung

Mit dem Befehl REVOKE kann der Eigentümer eines Objekts zuvor gewährte Privilegien wieder entziehen. Das Schlüsselwort PUBLIC bezieht sich auf die implizit definierte Gruppe aller Benutzer.

Beachten Sie, dass ein Benutzer immer die Summe aller Privilegien, die ihm direkt, allen Gruppen, denen er gegenwärtig angehört, und PUBLIC gewährt worden sind. Wenn man also zum Beispiel das Privileg SELECT von PUBLIC entzieht, heißt das nicht, dass alle Benutzer das Privileg SELECT verloren haben: Jene, denen es direkt oder über eine Gruppe gewährt worden ist, haben es weiterhin.

Eine Beschreibung der einzelnen Privilegientypen finden Sie unter GRANT *(Seite: 739)*.

Hinweise

Verwenden Sie den psql-Befehl *(Seite: 819)* \z, um die bisher gewährten Privilegien einzusehen. Informationen über das Ausgabeformat finden Sie unter GRANT *(Seite: 739)*.

Beispiel

Entziehe das INSERT-Privileg, das von PUBLIC für die Tabelle filme gehalten wird:

```
REVOKE INSERT ON filme FROM PUBLIC;
```

Entziehe alle Privilegien für die Sicht genres vom Benutzer manuel:

```
  REVOKE ALL PRIVILEGES ON genres FROM manuel;
```

Kompatibilität

Die Kompatibilitätshinweise unter GRANT *(Seite: 739)* gelten analog für REVOKE. Die Syntax ist:

```
REVOKE [ GRANT OPTION FOR ] privilegien
```

```
    ON objekt [ ( spalte [, ...] ) ]
    FROM { PUBLIC | benutzername [, ...] }
    { RESTRICT | CASCADE }
```

Die Schlüsselwörter `RESTRICT` und `CASCADE` werden im Zusammenhang mit der `GRANT OPTION`-Funktionalität verwendet.

Siehe auch

`GRANT` *(Seite: 739)*

ROLLBACK

Name

ROLLBACK – bricht die aktuelle Transaktion ab

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Beschreibung

`ROLLBACK` rollt die aktuelle Transaktion zurück und sorgt dafür, dass alle von der Transaktion getätigten Änderungen verworfen werden.

Parameter

`WORK`
`TRANSACTION`

Optionale Schlüsselwörter ohne jegliche Auswirkung.

Meldungen

`ROLLBACK`

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

`WARNING: ROLLBACK: no transaction in progress`

Meldung, wenn gegenwärtig keine Transaktion aktiv ist.

Beispiele

Um alle Änderungen abzubrechen:

```
ROLLBACK;
```

Kompatibilität

Der SQL-Standard bestimmt nur die zwei Formen `ROLLBACK` und `ROLLBACK WORK`. Ansonsten ist dieser Befehl voll konform.

Siehe auch

`COMMIT` *(Seite: 645)*

SELECT

Name

SELECT – liest Zeilen aus einer Tabelle oder Sicht

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( ausdruck [, ...] ) ] ]
    * | ausdruck [ AS ausgabename ] [, ...]
    [ FROM from_element [, ...] ]
    [ WHERE bedingung ]
    [ GROUP BY ausdruck [, ...] ]
    [ HAVING bedingung [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY ausdruck [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { anzahl | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF tabellenname [, ...] ] ]
```

wobei *from_element* eins der Folgenden sein kann:

```
    [ ONLY ] tabellenname [ * ] [ [ AS ] alias [ ( spaltenalias [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( spaltenalias [, ...] ) ]
   funktionsname ( [ argument [, ...] ] ) [ AS ] alias [ ( spaltenalias [, ...]
| spaltendefinition [, ...] ) ]
    funktionsname ( [ argument [, ...] ] ) AS ( spaltendefinition [, ...] )
    from_element [ NATURAL ] verbundtyp from_element [ ON verbundbedingung |
USING ( spalte [, ...] ) ]
```

Beschreibung

SELECT liest Zeilen aus einer oder mehreren Tabellen. Der prinzipielle Ablauf eines SELECT-Befehls sieht folgendermaßen aus:

1. Alle Elemente in der FROM-Liste werden berechnet. (Jedes Element in der FROM-Liste ist eine echte oder virtuelle Tabelle.) Wenn mehrere Elemente in der FROM-Liste angegeben sind, werden sie per Kreuzverbund verknüpft. (Siehe unten unter Die FROM-Klausel *(Seite: 760)*.)
2. Wenn die WHERE-Klausel angegeben ist, werden alle Zeilen, die die angegebene Bedingung nicht erfüllen, vom Ergebnis entfernt. (Siehe unten unter Die WHERE-Klausel *(Seite: 762)*.)
3. Wenn die GROUP BY-Klausel angegeben ist, werden die Ergebniszeilen in Gruppen unterteilt, die in einem oder mehreren Werten übereinstimmen. Wenn die HAVING-Klausel vorhanden ist, entfernt sie die Gruppen aus dem Ergebnis, die die angegebene Bedingung nicht erfüllen. (Siehe unten unter Die GROUP BY-Klausel *(Seite: 762)* und Die HAVING-Klausel *(Seite: 762)*.)
4. Mit den Operatoren UNION, INTERSECT und EXCEPT kann das Ergebnis von mehreren SELECT-Befehlen in eine einzige Ergebnismenge zusammengefasst werden. Der Operator UNION gibt alle Zeilen zurück, die in einer oder beiden Ergebnismengen enthalten sind. Der Operator INTERSECT gibt alle Zeilen zurück, die in beiden Ergebnismengen enthalten sind. Der Operator EXCEPT gibt alle Zeilen zurück, die in der ersten Ergebnismenge enthalten sind, aber nicht in der zweiten. In allen drei Fällen werden doppelte Zeilen entfernt, wenn ALL nicht angegeben ist. (Siehe unten unter Die UNION-Klausel *(Seite: 763)*, Die INTERSECT-Klausel *(Seite: 763)* und Die EXCEPT-Klausel *(Seite: 763)*.)
5. Die eigentlichen Ergebniszeilen werden von den SELECT-Ausgabeausdrücken aus jeder ausgewählten Zeile berechnet. (Siehe unten unter Die SELECT-Liste *(Seite: 764)*.)
6. Wenn die ORDER BY-Klausel angegeben ist, werden die zurückgegebenen Zeilen in der angegebenen Reihenfolge sortiert. Wenn ORDER BY nicht angegeben ist, werden die Zeilen in der Reihenfolge zurückgegeben, die das System am schnellsten erzeugen kann. (Siehe unten unter Die ORDER BY-Klausel *(Seite: 764)*.)
7. Wenn die Klauseln LIMIT oder OFFSET angegeben sind, gibt der SELECT-Befehl nur eine Teilmenge der Ergebniszeilen zurück. (Siehe unten unter Die LIMIT-Klausel *(Seite: 765)*.)
8. DISTINCT entfernt alle doppelten Zeilen aus dem Ergebnis. DISTINCT ON entfernt Zeilen, die in den angegebenen Ausdrücken übereinstimmen. ALL (die Voreinstellung) gibt alle Zeilen zurück. (Siehe unten unter Die DISTINCT-Klausel *(Seite: 765)*.)
9. Mit der Klausel FOR UPDATE sperrt der SELECT-Befehl die ausgewählten Zeilen gegen gleichzeitige Aktualisierungen. (Siehe unten unter Die FOR UPDATE-Klausel *(Seite: 765)*.)

Um eine Tabelle lesen zu können, müssen Sie das Privileg SELECT für die Tabelle haben. Wenn Sie FOR UPDATE verwenden, benötigen Sie zusätzlich das Privileg UPDATE.

Parameter

Die FROM-Klausel

Die FROM-Klausel gibt eine oder mehrere Quelltabellen für den SELECT-Befehl an. Wenn mehrere Quellen angegeben sind, ist das Ergebnis das kartesische Produkt (der Kreuzverbund) aller Quellen. Aber gewöhnlich werden Bedingungen angegeben, die die zurückgegebenen Zeilen auf eine kleine Teilmenge des Kreuzverbunds beschränken.

Die FROM-Klausel kann folgende Elemente enthalten:

tabellenname

Der Name einer bestehenden Tabelle oder Sicht (möglicherweise mit Schemaqualifikation). Wenn ONLY angegeben ist, wird nur diese Tabelle durchsucht. Wenn ONLY nicht angegeben ist, werden die Tabelle und alle Tabellen, die von dieser erben (falls vorhanden), durchsucht. * kann an den Tabellennamen ange-

hängt werden, um anzuzeigen, dass Tabellen, die von dieser erben, mit durchsucht werden sollen, aber das ist in der aktuellen Version das normale Verhalten. (In Versionen vor 7.1 war `ONLY` die Voreinstellung.) Das voreingestellte Verhalten kann mit dem Konfigurationsparameter `sql_interitance` eingestellt werden.

`alias`

Ein Ersatzname für das `FROM`-Element, das den Aliasnamen enthält. Ein Aliasname kann zur Abkürzung von Namen oder um bei Selbstverbunden (wo dieselbe Tabelle mehrfach verwendet wird) Zweideutigkeiten auszuschließen verwendet werden. Wenn ein Aliasname verwendet wird, versteckt er den eigentlichen Namen der Tabelle oder Funktion; zum Beispiel bei `FROM foo AS f` muss der Rest des `SELECT` auf dieses `FROM`-Element mit `f` verweisen, nicht `foo`. Wenn ein Aliasname angegeben wird, können auch Aliasnamen für die Spalten der Tabelle angegeben werden.

`select`

`SELECT`-Befehle können als Unteranfrage in der `FROM`-Klausel erscheinen. Das verhält sich dann wie eine temporäre Tabelle, die nur für die Dauer des `SELECT`-Befehls besteht. Beachten Sie, dass alle Unteranfragen in Klammern stehen und Aliasnamen haben müssen.

`funktionsname`

Funktionsaufrufe können in der `FROM`-Klausel stehen. (Das ist besonders nützlich bei Funktionen, die Ergebnismengen zurückgeben, aber jede Funktion kann verwendet werden.) Das verhält sich, als ob die Ergebnisse der Funktion in einer temporären Tabelle gespeichert wären, die nur für die Dauer des `SELECT`-Befehls besteht. Ein Aliasname kann auch angegeben werden. Wenn ein Aliasname angegeben wird, kann auch eine Liste von Spaltenaliasnamen Ersatznamen für die Attribute des zusammengesetzten Ergebnistyps der Funktion angeben. Wenn die Funktion mit Rückgabetyp `record` definiert wurde, muss ein Alias oder das Schlüsselwort `AS` angegeben werden, gefolgt von einer Spaltendefinitionsliste der Form `( spaltenname datentyp [, ... ] )`. Die Spaltendefinitionen müssen mit den tatsächlich von der Funktion zurückgegebenen Spalten in Zahl und Typ übereinstimmen.

`verbundtyp`

Einer der folgenden:

- `[ INNER ] JOIN`
- `LEFT [ OUTER ] JOIN`
- `RIGHT [ OUTER ] JOIN`
- `FULL [ OUTER ] JOIN`
- `CROSS JOIN`

Bei den Verbundtypen `INNER` und `OUTER` muss eine Verbundbedingung angegeben werden, und zwar genau eine der folgenden: `NATURAL`, `ON` *`verbundbedingung`* oder `USING (`*`spalte`* `[, ...])`. Die Bedeutungen werden weiter unten beschrieben. Bei `CROSS JOIN` darf keine dieser Klauseln erscheinen.

Eine `JOIN`-Klausel kombiniert zwei `FROM`-Elemente. (Verwenden Sie Klammern, um, wenn nötig, die Reihenfolge der Verschachtelung zu kontrollieren.)

`CROSS JOIN` und `INNER JOIN` erzeugen einfach das kartesische Produkt, genauso, als ob die zwei Elemente direkt in der `FROM`-Liste aufgezählt worden wären. `CROSS JOIN` hat dieselbe Bedeutung wie `INNER JOIN ON (true)`, das heißt, keine Zeilen werden durch eine Bedingung entfernt. Diese Verbundtypen sind nur eine bequemere Schreibweise, da sie nichts machen, was man nicht auch mit einem einfachen `FROM` und `WHERE` erreichen könnte.

`LEFT OUTER JOIN` ergibt alle Zeilen des qualifizierten kartesischen Produkts (d.h. alle kombinierten Zeilen, die die Verbundbedingung erfüllen), plus einmal jede Zeile der linken Tabelle, für die es keine Zeile in der rechten Tabelle gibt, mit der die Verbundbedingung erfüllt werden kann. Diese linke Zeile wird auf die volle Breite der verbundenen Zeile erweitert, indem die Spalten der rechten Tabelle mit NULL-Werten aufgefüllt werden. Beachten Sie, dass nur die Bedingung der `JOIN`-Klausel selbst beim Vergleich der Zeilen verwendet wird. Bedingungen in den äußeren Klauseln werden später angewendet.

Umgekehrt ergibt `RIGHT OUTER JOIN` alle verbundenen Zeilen plus einmal jede Zeile der rechten Tabelle ohne passende Zeile in der linken Tabelle (mit NULL-Werten nach links erweitert). Das ist nur eine alternative Schreibweise, weil man das auch als `LEFT OUTER JOIN` mit den linken und rechten Eingabewerten vertauscht schreiben kann.

`FULL OUTER JOIN` ergibt alle verbundenen Zeilen, plus jede Zeile der linken Tabelle ohne passende rechte Zeile (mit NULL-Werten nach rechts erweitert), plus jede Zeile der rechten Tabelle ohne passende linke Zeile (mit NULL-Werten nach links erweitert).

`ON` *`verbundbedingung`*

`verbundbedingung` ist ein Ausdruck, der einen Wert des Typs `boolean` ergibt (ähnlich wie eine `WHERE`-Klausel), der angibt, welche Zeilen in einem Verbund zueinander gehören sollen.

`USING (`*`spalte`* `[, ...])`

Eine Klausel der Form `USING ( a, b, ... )` ist eine Abkürzung für `ON linke_tabelle.a = rechte_tabelle.a AND linke_tabelle.b = rechte_tabelle.b ...` Außerdem enthält bei `USING` das Ergebnis des Verbunds nur eine Ausgabe für jedes Paar von äquivalenten Spalten, nicht beide.

`NATURAL`

`NATURAL` ist eine Abkürzung für eine `USING`-Liste, die alle Spalten der beiden Tabellen aufzählt, die die gleichen Namen haben.

Die WHERE-Klausel

Die optionale `WHERE`-Klausel hat folgende allgemeine Form:

```
WHERE bedingung
```

wobei *`bedingung`* ein beliebiger Ausdruck ist, der ein Ergebnis des Typs `boolean` hat. Jede Zeile, die diese Bedingung nicht erfüllt, wird aus dem Ergebnis des `SELECT`-Befehls entfernt. Eine Zeile erfüllt die Bedingung, wenn der Ausdruck „wahr“ ergibt, wenn die tatsächlichen Zeilenwerte für etwaige Variablen eingesetzt werden.

Die GROUP BY-Klausel

Die optionale `GROUP BY`-Klausel hat folgende allgemeine Form:

```
GROUP BY ausdruck [, ...]
```

`GROUP BY` fasst alle Zeilen, die den gleichen Wert für die Gruppierungsausdrücke haben, zu jeweils einer Zeile zusammen. *`ausdruck`* kann der Name einer Eingabespalte, die Nummer einer Ausgabespalte (`SELECT`-Liste) oder ein beliebiger Ausdruck aus Eingabespaltenwerten sein. Bei Zweideutigkeiten werden Namen in `GROUP BY` als Eingabespalte anstatt als Ausgabespalte interpretiert.

Aggregatfunktionen berechnen jeweils einen Wert aus allen Zeilen einer Gruppe. (Ohne `GROUP BY` berechnet eine Aggregatfunktion einen einzelnen Wert aus allen ausgewählten Zeilen.) Wenn `GROUP BY` verwendet wird, ist es nicht zulässig, dass Ausdrücke in der `SELECT`-Liste auf ungruppierte Spalten verweisen, außer über Aggregatfunktionen, da es für eine ungruppierte Spalte mehr als einen möglichen Wert geben würde.

Die HAVING-Klausel

Die optionale `HAVING`-Klausel hat die allgemeine Form

```
HAVING bedingung
```

wobei *`bedingung`* genauso wie in der `WHERE`-Klausel funktioniert.

HAVING entfernt Gruppenzeilen, die die Bedingung nicht erfüllen. HAVING unterscheidet sich von WHERE: WHERE filtert einzelne Zeilen vor der Anwendung von GROUP BY, während HAVING die von GROUP BY erzeugten Zeilen filtert. Jede Spalte, die in *bedingung* verwendet wird, muss eindeutig auf eine gruppierte Spalte verweisen, es sei denn, der Verweis steht in einer Aggregatfunktion.

Die UNION-Klausel

Die UNION-Klausel hat diese allgemeine Form:

```
select_befehl UNION [ ALL ] select_befehl
```

select_befehl ist ein beliebiger SELECT-Befehl ohne ORDER BY, LIMIT und FOR UPDATE. (ORDER BY und LIMIT können in einem Unterausdruck stehen, wenn er in Klammern steht. Ohne Klammern werden diese Klauseln auf das Ergebnis von UNION angewendet, nicht auf den rechten Teilausdruck.)

Der Operator UNION berechnet die Vereinigungsmenge der Zeilen, die von beiden SELECT-Befehlen geliefert werden. Eine Zeile ist in der Vereinigungsmenge, wenn sie in mindestens einer der beiden Ergebnismengen erscheint. Die beiden SELECT-Befehle, die die direkten Operanden von UNION sind, müssen die gleiche Anzahl von Spalten ergeben und die einander entsprechenden Spalten müssen kompatible Datentypen haben.

Das Ergebnis von UNION enthält keine doppelten Zeilen, außer wenn die Option ALL angegeben ist. ALL verhindert das Entfernen von Duplikaten.

Mehrere UNION-Operatoren im selben SELECT-Befehl werden von links nach rechts ausgewertet. Klammern können verwendet werden, um die Reihenfolge zu ändern.

Im Ergebnis oder in den Operanden von UNION kann die Klausel FOR UPDATE kann gegenwärtig nicht verwendet werden.

Die INTERSECT-Klausel

Die INTERSECT-Klausel hat diese allgemeine Form:

```
select_befehl INTERSECT [ ALL ] select_befehl
```

select_befehl ist ein beliebiger SELECT-Befehl ohne ORDER BY, LIMIT und FOR UPDATE.

Der Operator INTERSECT berechnet die Schnittmenge der Zeilen, die von beiden SELECT-Befehlen geliefert werden. Eine Zeile ist in der Schnittmenge, wenn sie in beiden Ergebnismengen erscheint.

Das Ergebnis von INTERSECT enthält keine doppelten Zeilen, außer wenn die Option ALL angegeben ist. Wenn ALL angegeben ist, wird eine Zeile, die in der linken Tabelle m-mal und in der rechten Tabelle n-mal erscheint, im Ergebnis min(m,n)-mal erscheinen.

Mehrere INTERSECT-Operatoren im selben SELECT-Befehl werden von links nach rechts ausgewertet, außer wenn Klammern es anders regeln. INTERSECT bindet enger als UNION. Das heißt A UNION B INTERSECT C wird als A UNION (B INTERSECT C) gelesen.

Die EXCEPT-Klausel

Die EXCEPT-Klausel hat diese allgemeine Form:

```
select_befehl EXCEPT [ ALL ] select_befehl
```

select_befehl ist ein beliebiger SELECT-Befehl ohne ORDER BY, LIMIT und FOR UPDATE.

Der Operator EXCEPT berechnet die Menge der Zeilen, die im Ergebnis des linken SELECT-Befehls, aber nicht im Ergebnis des rechten erscheinen.

Das Ergebnis von EXCEPT enthält keine doppelten Zeilen, außer wenn die Option ALL angegeben ist. Wenn ALL angegeben ist, wird eine Zeile, die in der linken Tabelle m-mal und in der rechten Tabelle n-mal erscheint, im Ergebnis max(m-n,0)-mal erscheinen.

Mehrere EXCEPT-Operatoren im selben SELECT-Befehl werden von links nach rechts ausgewertet, außer wenn Klammern es anders regeln. EXCEPT bindet im gleiche Maße wie UNION.

Die SELECT-Liste

Die SELECT-Liste (zwischen den Schlüsselwörtern SELECT und FROM) enthält Ausdrücke, die die Ausgabezeilen des SELECT-Befehls berechnen. Die Ausdrücke können auf die in der FROM-Klausel berechneten Spalten verweisen (und tun das in der Regel auch). Mit der Klausel AS *ausgabename* kann einer Ausgabespalte ein anderer Name gegeben werden. Dieser Name wird hauptsächlich als Spaltenkopf in der Ausgabe verwendet. Er kann aber auch in den Klauseln ORDER BY und GROUP BY verwendet werden, nicht aber in den Klauseln WHERE und HAVING, dort müssen Sie den Ausdruck ausschreiben.

Anstelle eines Ausdrucks kann in der Liste auch * geschrieben werden, als Abkürzung für alle Spalten der ausgewählten Zeilen. Außerdem kann man *tabellenname*.* als Abkürzung für die Spalten aus nur dieser Tabelle schreiben.

Die ORDER BY-Klausel

Die optionale ORDER BY-Klausel hat diese allgemeine Form:

```
ORDER BY ausdruck [ ASC | DESC | USING operator ] [, ...]
```

ausdruck kann der Name oder die Nummer einer Ausgabespalte (SELECT-Liste) oder ein beliebiger Ausdruck aus Eingabespaltenwerten sein.

Die ORDER BY-Klausel sortiert die Ergebniszeilen anhand der angegebenen Ausdrücke. Wenn zwei Zeilen nach dem am weitesten links stehenden Ausdruck gleich sind, werden sie anhand des nächsten Ausdrucks verglichen und so weiter. Wenn sie nach allen angegebenen Ausdrücken gleich sind, werden sie in zufälliger Reihenfolge zurückgegeben.

Eine Nummer verweist auf eine Ergebnisspalte gezählt von links nach rechts. Dadurch kann man auf Spalten verweisen, die keinen eindeutigen Namen haben. Das ist aber niemals absolut notwendig, denn man kann jeder Ergebnisspalte mit der AS-Klausel einen Namen zuweisen.

Man kann in der ORDER BY-Klausel auch beliebige Ausdrücke angeben, einschließlich Spalten, die nicht in der SELECT-Liste erscheinen. Der folgende Befehl ist also gültig:

```
SELECT name FROM verleihe ORDER BY code;
```

Eine Beschränkung ist jedoch, dass eine ORDER BY-Klausel, die auf das Ergebnis von UNION, INTERSECT oder EXCEPT angewendet wird, nur den Namen oder die Nummer von Ausgabespalten angeben darf, keine Ausdrücke.

Wenn ein Ausdruck in ORDER BY sowohl als Name einer Ergebnisspalte als auch als Name einer Eingabespalte interpretiert werden kann, interpretiert ORDER BY ihn als Name der Ausgabespalte. Dies ist das Gegenteil von dem, was GROUP BY in dieser Situation machen würde. Diese Unregelmäßigkeit ist notwendig, um mit dem SQL-Standard kompatibel zu sein.

Wahlweise kann man nach jedem Ausdruck in der ORDER BY-Klausel das Schlüsselwort ASC für aufsteigende Reihenfolge oder DESC für abfallende Reihenfolge schreiben. Wenn keines angegeben ist, dann ist ASC die Voreinstellung. Als Alternative kann man auch einen bestimmten Sortieroperator mit der Klausel USING angeben. ASC entspricht USING < und DESC entspricht USING >.

NULL-Werte werden höher als alle anderen Werten einsortiert. Anders ausgedrückt: Bei aufsteigender Reihenfolge erscheinen NULL-Werte am Ende und bei abfallender Reihenfolge erscheinen NULL-Werte am Anfang.

Die LIMIT-Klausel

Die LIMIT-Klausel besteht aus zwei unabhängigen Klauseln:

```
LIMIT { anzahl | ALL }
OFFSET start
```

anzahl gibt die Anzahl der Zeilen an, die höchstens zurückgegeben werden sollen. *start* gibt die Anzahl der Zeilen an, die übersprungen werden sollen, bevor Zeilen zurückgegeben werden.

Wenn LIMIT verwendet wird, ist es empfehlenswert, eine ORDER BY-Klausel zu verwenden, die die Ergebniszeilen in eine eindeutige Ordnung bringt. Ansonsten erhalten Sie eine nicht vorhersagbare Teilmenge der Ergebniszeilen der Anfrage. Sie könnten zum Beispiel die Zeilen 10 bis 20 anfordern, aber 10 bis 20 nach welcher Reihenfolge? Die Reihenfolge ist unbekannt, wenn ORDER BY nicht verwendet wird.

Der Anfrageoptimierer zieht die LIMIT-Klausel mit in Betracht, wenn er einen Ausführungsplan für eine Anfrage erstellt, was heißt, dass man sehr wahrscheinlich unterschiedliche Pläne (die unterschiedliche Zeilenreihenfolgen ergeben) erhält, wenn man die LIMIT- und OFFSET-Werte verändert. Wenn man folglich die LIMIT- und OFFSET-Werte variiert, um verschiedene Teilmengen eines Anfrageergebnisses auszuwählen, *wird man widersprüchliche Ergebnisse erhalten*, wenn man keine voraussagbare Reihenfolge mit ORDER BY erzwingt. Das ist kein Fehler; das Verhalten folgt aus der Tatsache, dass SQL keine Gewähr dafür gibt, dass das Ergebnis einer Anfrage in einer bestimmten Reihenfolge abgeliefert wird, außer wenn ORDER BY verwendet wird, um die Reihenfolge zu beeinflussen.

Die DISTINCT-Klausel

Wenn DISTINCT angegeben ist, werden doppelte Zeilen aus der Ergebnismenge entfernt. (Eine Zeile wird für jede Gruppe doppelter Zeilen behalten.) ALL ist das Gegenteil: Alle Zeilen werden behalten; das ist auch die Voreinstellung.

DISTINCT ON (*ausdruck* [, ...]) behält nur die erste Zeile jeder Gruppe von Zeilen, bei denen die angegebenen Ausdrücke gleich sind. Die Ausdrücke in DISTINCT ON werden mit denselben Regeln wie in der ORDER BY-Klausel interpretiert (siehe oben). Beachten Sie, dass die „erste Zeile“ einer Menge nicht vorhergesagt werden kann, außer wenn ORDER BY verwendet wird, um sicherzustellen, dass die gewünschte Zeile zuerst erscheint. Zum Beispiel ermittelt folgende Anfrage den neuesten Wetterbericht jedes Orts:

```
SELECT DISTINCT ON (ort) ort, zeit, bericht
    FROM wetterberichte
    ORDER BY ort, zeit DESC;
```

Wenn ORDER BY nicht verwendet worden wäre, um die Zeitangaben an jedem Ort abfallend zu sortieren, würden wir für jeden Ort einen zufällig ausgewählten Bericht erhalten.

Die FOR UPDATE-Klausel

Die FOR UPDATE-Klausel hat diese Form:

```
FOR UPDATE [ OF tabellenname [, ...] ]
```

Durch FOR UPDATE wird jede von dem SELECT-Befehl zurückgegebene Zeile wie bei einer Aktualisierung gesperrt. Dadurch wird verhindert, dass andere Transaktionen diese Zeile ändern oder löschen können, bevor diese Transaktion beendet ist. Das heißt, wenn eine andere Transaktion in diesen Zeilen ein UPDATE, DELETE oder SELECT FOR UPDATE versucht, blockiert sie, bis die aktuelle Transaktion beendet ist. Wenn eine andere Transaktion eine ausgewählte Zeile oder Zeilen schon durch UPDATE, DELETE oder SELECT FOR UPDATE gesperrt hat, wartet SELECT FOR UPDATE, bis die andere Transaktion zu Ende

geht und wird dann die aktualisierte Zeile sperren und zurückgeben (oder gar nichts machen, wenn die Zeile gelöscht wurde). Ausführlichere Informationen über dieses Thema finden Sie in Abschnitt SET.

Wenn bestimmte Tabellen in `FOR UPDATE` aufgezählt sind, werden nur Zeilen, die aus diesen Tabellen kommen, gesperrt; alle anderen Tabellen, die in dem `SELECT`-Befehl verwendet werden, werden ganz normal gelesen.

In Zusammenhängen, wo es nicht klar ist, aus welcher Tabelle eine Zeile kam, kann `FOR UPDATE` nicht verwendet werden, zum Beispiel nicht, wenn Aggregatfunktionen verwendet werden.

`FOR UPDATE` kann vor `LIMIT` stehen, um kompatibel mit PostgreSQL-Versionen vor 7.3 zu sein. Da es aber im Prinzip nach `LIMIT` ausgeführt wird, empfehlen wir, es auch dort hinzuschreiben.

Beispiele

Ein Verbund der Tabellen `filme` und `verleihe`:

```
SELECT f.titel, f.vid, v.name, f.prod_datum, f.genre
    FROM verleihe v, filme f
    WHERE f.vid = v.vid

       titel       | vid |     name     | prod_datum |   genre
-------------------+-----+--------------+------------+------------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantisch
 ...
```

Summiere die Spalte `länge` aller Filme und gruppiere die Ergebnisse nach Genre:

```
SELECT genre, sum(länge) AS gesamt FROM filme GROUP BY genre;

   genre    | gesamt
------------+--------
 Action     |  07:34
 Comedy     |  02:58
 Drama      |  14:28
 Musical    |  06:42
 Romantisch |  04:38
```

Summiere die Spalte `länge` aller Filme, gruppiere die Ergebnisse nach Genre und zeige jene Gruppen, deren Gesamtlänge kleiner als 5 Stunden ist:

```
SELECT genre, sum(länge) AS gesamt
    FROM filme
    GROUP BY genre
    HAVING sum(länge) < interval '5 hours';

   genre    | gesamt
------------+--------
 Comedy     |  02:58
 Romantisch |  04:38
```

Die folgenden zwei Beispiele sind gleichbedeutende Möglichkeiten, die einzelnen Ergebnisse nach dem Inhalt der zweiten Spalte (`name`) zu sortieren:

```
SELECT * FROM verleihe ORDER BY name;
SELECT * FROM verleihe ORDER BY 2;

 vid |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

Dieses Beispiel zeigt, wie man die Vereinigungsmenge der zwei Tabellen `verleihe` und `schauspieler` ermittelt und das Ergebnis auf diejenigen, die mit W beginnen, beschränkt. Duplikate sind nicht erwünscht, also wird das Schlüsselwort `ALL` weggelassen.

```
verleihe:                   schauspieler:
 vid |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...

SELECT verleihe.name
    FROM verleihe
    WHERE verleihe.name LIKE 'W%'
UNION
SELECT schauspieler.name
    FROM schauspieler
    WHERE schauspieler.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
```

```
 Westward
 Woody Allen
```

Dieses Beispiel zeigt, wie eine Funktion in der FROM-Liste verwendet werden kann, mit oder ohne Spaltendefinitionsliste.

```
CREATE FUNCTION verleihe(int) RETURNS SETOF verleihe AS '
    SELECT * FROM verleihe WHERE vid = $1;
' LANGUAGE SQL;

SELECT * FROM verleihe(111);
 vid |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION verleihe_2(int) RETURNS SETOF record AS '
    SELECT * FROM verleihe WHERE vid = $1;
' LANGUAGE SQL;

SELECT * FROM verleihe_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Kompatibilität

Natürlich ist der Befehl SELECT mit dem SQL-Standard kompatibel. Aber es gibt einige Erweiterungen und fehlende Funktionalität.

Weggelassene FROM-Klauseln

In PostgreSQL kann man die FROM-Klausel weglassen. Das hat einfache Anwendung, wie die Auswertung von einfachen Ausdrücken:

```
SELECT 2+2;

 ?column?
----------
        4
```

In einigen anderen SQL-Datenbanken geht das nur, wenn man eine einzeilige Pseudotabelle in dem SELECT-Befehl angibt.

Eine weniger offensichtliche Anwendung ist die Abkürzung von normalen SELECT-Befehlen mit Tabellen:

```
SELECT verleihe.* WHERE verleihe.name = 'Westward';

 vid |   name
-----+----------
 108 | Westward
```

Das funktioniert, weil jede Tabelle, die in anderen Teilen des SELECT-Befehls verwendet, aber nicht in der FROM-Klausel erwähnt wird, implizit in die FROM-Klausel eingetragen wird.

Obwohl diese Abkürzung bequem ist, ist es einfach, sie falsch zu verwenden. Zum Beispiel ist der Befehl

```
SELECT verleihe.* FROM verleihe v;
```

wahrscheinlich ein Fehler; der Benutzer hat wohl eher

```
SELECT v.* FROM verleihe v;
```

gemeint, statt dem bedingungslosen Verbund

```
SELECT verleihe.* FROM verleihe v, verleihe verleihe;
```

der in Wirklichkeit ausgeführt wird. Um bei der Aufdeckung von dieser Art von Fehlern zu helfen, warnt PostgreSQL, wenn Elemente implizit in die FROM-Klausel eingefügt werden, obwohl der SELECT-Befehl schon eine explizite FROM-Klausel enthält.

Das Schlüsselwort AS

Im SQL-Standard ist das optionale Schlüsselwort AS ohne Bedeutung und kann weggelassen werden. Der PostgreSQL-Parser erfordert dieses Schlüsselwort, wenn Ausgabespalten umbenannt werden, weil die Typenerweiterbarkeit sonst zu Syntaxkonflikten führen würde. In der FROM-Klausel ist AS jedoch optional.

Verfügbarer Namensraum in GROUP BY und ORDER BY

Im SQL-Standard kann die ORDER BY-Klausel nur Namen oder Nummern von Ergebnisspalten verwenden, und die GROUP BY-Klausel kann nur Ausdrücke mit den Namen von Eingabespalten verwenden. PostgreSQL erweitert diese Klauseln, indem die jeweils andere Möglichkeit auch verwendet werden kann (aber der Standard geht vor, wenn es Zweideutigkeiten gibt). In PostgreSQL können in beiden Klauseln auch beliebige Ausdrücke verwendet werden. Beachten Sie, dass Namen, die in Ausdrücken verwendet werden, immer als Eingabespalten und nie als Ausgabespalten interpretiert werden.

Nicht standardisierte Klauseln

Die Klauseln DISTINCT ON, LIMIT und OFFSET sind nicht im SQL-Standard definiert.

SELECT INTO

Name

SELECT INTO – erzeugt eine neue Tabelle aus den Ergebnissen einer Anfrage

Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( ausdruck [, ...] ) ] ]
    * | ausdruck [ AS name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] neue_tabelle
    [ FROM from_element [, ...] ]
    [ WHERE bedingung ]
```

```
    [ GROUP BY ausdruck [, ...] ]
    [ HAVING bedingung [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY ausdruck [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { anzahl | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF tabellenname [, ...] ] ]
```

Beschreibung

SELECT INTO erzeugt eine neue Tabelle und füllt sie mit den von einer Anfrage berechneten Daten. Die Daten werden nicht wie bei einem normalen SELECT an den Client zurückgegeben. Die Tabellenspalten haben die gleichen Namen und Datentypen wie das Ergebnis des SELECT.

Parameter

TEMPORARY oder TEMP

Wenn angegeben, wird die Tabelle als temporäre Tabelle erzeugt. Weitere Einzelheiten dazu finden Sie bei CREATE TABLE *(Seite: 686)*.

neue_tabelle

Der Name der zu erzeugenden Tabelle (möglicherweise mit Schemaqualifikation).

Alle weiteren Parameter sind im Einzelnen unter SELECT *(Seite: 759)* beschrieben.

Meldungen

Eine Zusammenfassung der möglichen Meldungen erhalten Sie unter CREATE TABLE *(Seite: 686)* und SELECT *(Seite: 759)*.

Hinweise

CREATE TABLE AS *(Seite: 694)* hat die gleiche Funktionalität wie SELECT INTO. CREATE TABLE AS ist die empfohlene Syntax, weil diese Form von SELECT INTO nicht in ECPG und PL/pgSQL funktioniert, da dort die INTO-Klausel eine andere Bedeutung hat.

Kompatibilität

Der SQL-Standard verwendet SELECT ... INTO, um die ausgewählten Werte in skalaren Variablen in einem Hostprogramm abzulegen. Diese Verwendung findet sich auch in ECPG (siehe Abschnitt SET) und PL/pgSQL (siehe Abschnitt SET). Die Verwendung von SELECT INTO in PostgreSQL, um Tabellen zu erzeugen, ist historisch bedingt. Am besten verwendet man dafür CREATE TABLE AS. (CREATE TABLE AS entspricht auch nicht dem SQL-Standard, aber es erzeugt wohl am wenigsten Verwirrung.)

SET

Name

SET – ändert einen Konfigurationsparameter

Synopsis

```
SET [ SESSION | LOCAL ] variable { TO | = } { wert | 'wert' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { zeitzone | LOCAL | DEFAULT }
```

Beschreibung

Der Befehl SET ändert die Werte von Konfigurationsparametern. Viele der in Abschnitt SET aufgelisteten Parameter können zur Laufzeit mit SET geändert werden. (Aber einige erfordern Superuser-Privilegien und einige andere können nach dem Start der Sitzung nicht mehr geändert werden.) SET wirkt sich nur auf die aktuelle Sitzung aus.

Wenn SET oder SET SESSION in einer Transaktion ausgeführt wird, die später abgebrochen wird, werden die Auswirkungen des SET-Befehls rückgängig gemacht, wenn die Transaktion zurückgerollt wird. (Dieses Verhalten wurde in PostgreSQL Version 7.3 eingeführt. Vorher wurden SET-Befehle nicht nach einem Fehler zurückgerollt.) Wenn die Transaktion zurückgerollt wird, dann bleibt die Auswirkung des SET-Befehls bis zum Ende der Sitzung bestehen, es sei denn ein weiterer SET-Befehl ändert denselben Parameter.

Die Auswirkungen von SET LOCAL bleiben nur bis zum Ende der aktuellen Transaktion bestehen, egal ob sie abgebrochen oder erfolgreich abgeschlossen wird. Ein Sonderfall ist ein SET gefolgt von einem SET LOCAL in derselben Transaktion: Der von SET LOCAL gesetzte Wert bleibt bis zum Ende der Transaktion gesetzt, aber danach (nachdem die Transaktion abgeschlossen wurde) wird der von SET gesetzte aktiv.

Selbst wenn autocommit aus ist, wird durch SET kein neuer Transaktionsblock gestartet. Einzelheiten dazu finden Sie im Abschnitt über autocommit in Abschnitt SET.

Parameter

SESSION

Gibt an, dass der Befehl für die aktuelle Sitzung gelten soll. (Das ist die Voreinstellung, wenn weder SESSION noch LOCAL angegeben wurden.)

LOCAL

Gibt an, dass der Befehl nur für die aktuelle Transaktion gelten soll. Nach COMMIT oder ROLLBACK gilt wieder der Sitzungswert. Beachten Sie, dass SET LOCAL scheinbar ohne Auswirkung bleibt, wenn es außerhalb eines BEGIN-Blocks ausgeführt wird, weil die Transaktion dann sofort zu Ende ist.

variable

Der Name eines Konfigurationsparameters. Die verfügbaren Parameter sind in Abschnitt SET und unten beschrieben.

wert

Der neue Wert des Parameters. Werte können als Zeichenkettenkonstanten, Bezeichner, Zahlen oder als Liste von diesen, getrennt durch Kommas, angegeben werden. DEFAULT kann verwendet werden, um den Parameter wieder auf seinen Vorgabewert zurückzusetzen.

Neben den in Abschnitt SET beschriebenen Konfigurationsparametern gibt es einige wenige, die nur mit dem Befehl SET gesetzt werden können oder die eine besondere Syntax haben:

NAMES

SET NAMES *wert* hat dieselbe Bedeutung wie SET client_encoding TO *wert*.

SEED

Setzt den internen Samen für den Zufallszahlengenerator (die Funktion random). Gültige Wert sind Fließkommazahlen zwischen 0 und 1, die dann mit 231-1 multipliziert werden.

Der Samen kann auch mit der Funktion setseed gesetzt werden:

```
SELECT setseed(wert);
```

TIME ZONE

SET TIME ZONE *wert* hat dieselbe Bedeutung wie SET timezone TO *wert*. Die Syntax SET TIME ZONE erlaubt eine besondere Syntax für die Zeitzonenangaben. Hier sind Beispiele für gültige Zeitzonenangaben:

'PST8PDT'

Die Zeitzone von Berkeley, Kalifornien.

'Portugal'

Die Zeitzone von Portugal.

'Europe/Berlin'

Die Zeitzone von Deutschland.

7

Die Zeitzone 7 Stunden westlich von UTC (entspricht PDT).

INTERVAL '08:00' HOUR TO MINUTE

Die Zeitzone 8 Stunden westlich von UTC (entspricht PST).

LOCAL
DEFAULT

Setzt die Zeitzone auf die lokale Zeitzone (die im Betriebssystem eingestellte).

Siehe Abschnitt SET für weitere Informationen über Zeitzonen.

Meldungen

SET

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

ERROR: '*name*' is not a valid option name

Den Parameter, den Sie setzen wollten, gibt es nicht.

ERROR: '*name*': permission denied

Um bestimmte Einstellungen zu ändern, müssen Sie ein Superuser sein.

ERROR: '*name*' cannot be changed after server start

Einige Parameter können nicht mehr geändert werden, nachdem der Server gestartet wurde.

Hinweise

Die Funktion `set_config` kann auch verwendet werden, um die Werte von Konfigurationsparametern zu setzen. Siehe Abschnitt SET.

Beispiele

Setze den Schemasuchpfad:

```
SET search_path TO mein_schema, public;
```

Setze das Datumsformat auf den traditionellen POSTGRES-Stil mit dem europäischen Modus:

```
SET datestyle TO postgres, european;
```

Setze die Zeitzone für Deutschland (beachten Sie die Apostrophe wegen der Großbuchstaben und Sonderzeichen in der Zeitzonenangabe):

```
SET TIME ZONE 'Europe/Berlin';
SELECT current_timestamp AS heute;

             heute
-------------------------------
 2003-04-30 00:00:42.139638+02
```

Kompatibilität

`SET TIME ZONE` erweitert die vom SQL-Standard definierte Syntax. Der Standard erlaubt nur numerische Zeitzonenangaben, während PostgreSQL flexiblere Zeitzonenangaben erlaubt. Alle anderen `SET`-Funktionen sind PostgreSQL-Erweiterungen.

Siehe auch

`RESET` *(Seite: 755)*, `SHOW` *(Seite: 777)*

SET CONSTRAINTS

Name

SET CONSTRAINTS – setzt den Constraint-Modus der aktuellen Transaktion

Synopsis

```
SET CONSTRAINTS { ALL | constraint [, ...] } { DEFERRED | IMMEDIATE }
```

Beschreibung

`SET CONSTRAINTS` stellt die Auswertung von Constraints in der aktuellen Transaktion ein. Im Modus `IMMEDIATE` (unmittelbar) werden Constraints am Ende jedes Befehls geprüft. Im Modus `DEFERRED` (verschoben) werden Constraints erst am Ende der Transaktion geprüft.

Wenn Sie den Modus eines Constraints auf `IMMEDIATE` setzen, tritt der Modus rückwirkend in Kraft: Alle Datenveränderungen, die erst am Ende der Transaktion geprüft worden wären (Modus `DEFERRED`), werden stattdessen bei der Ausführung des Befehls `SET CONSTRAINTS` geprüft.

Bei der Erzeugung erhält ein Constraint immer eine von drei möglichen Eigenschaften: `INITIALLY DEFERRED` (d.h., er wird in der Standardeinstellung am Ende der Transaktion geprüft), `INITIALLY IMMEDIATE DEFERRABLE` (d.h., er wird in der Standardeinstellung am Ende jedes Befehls geprüft, die Prüfung kann aber an das Ende der Transaktion verschoben werden) oder `INITIALLY IMMEDIATE NOT DEFERRABLE` (d.h., er wird in der Standardeinstellung am Ende jedes Befehls geprüft und kann nicht verschoben werden). Auf die dritte Gruppe hat der Befehl `SET CONSTRAINTS` keine Auswirkung.

Gegenwärtig werden nur Fremdschlüssel-Constraints von dieser Einstellung beeinflusst. Check- und Unique Constraints werden immer nach jedem Befehl geprüft.

Hinweise

Dieser Befehl ändert das Verhalten von Constraints in der aktuellen Transaktion. Wenn Sie den Befehl also außerhalb eines Transaktionsblocks (`BEGIN`/`COMMIT`-Paar) ausführen, hat er keine erkennbare Auswirkung. Wenn Sie das Verhalten eines Constraints ändern wollen, ohne in jeder Transaktion `SET CONSTRAINTS` ausführen zu müssen, geben Sie bei der Erzeugung des Constraints `INITIALLY DEFERRED` oder `INITIALLY IMMEDIATE` an.

Kompatibilität

Dieser Befehl ist mit dem im SQL-Standard definierten Verhalten konform, außer dass er in PostgreSQL nur für Fremdschlüssel-Constraints gilt.

SET SESSION AUTHORIZATION

Name

SET SESSION AUTHORIZATION – setzt den Sitzungsbenutzernamen und den aktuellen Benutzernamen der aktuellen Sitzung

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION benutzername
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Beschreibung

Dieser Befehl setzt den Sitzungbenutzernamen und den aktuellen Benutzernamen der aktuellen SQL-Sitzung auf *benutzername*. Der Benutzername kann als Bezeichner oder als Zeichenkettenkonstante geschrieben werden. Mit diesem Befehl kann man sich zum Beispiel vorübergehend zu einem unprivilegierten Benutzer machen und später auf Superuser zurückschalten.

Der *Sitzungsbenutzername* wird am Anfang auf den vom Client verlangten Benutzernamen gesetzt (welcher möglicherweise authentifiziert wurde). Der *aktuelle Benutzername* ist normalerweise der gleiche, kann sich aber vorübergehend im Rahmen von „setuid“-Funktionen und ähnlichen Vorrichtungen ändern. Der aktuelle Benutzername ist der, der für die Prüfung der Zugriffsrechte herangezogen wird.

Der Sitzungsbenutzername kann nur geändert werden, wenn der anfängliche Sitzungsbenutzer (der *authentifizierte Benutzer*) das Superuser-Privileg hatte. Ansonsten wird dieser Befehl nur akzeptiert, wenn er den authentifizierten Benutzernamen angibt.

Die Optionen `SESSION` und `LOCAL` haben dieselbe Bedeutung wie beim Befehl `SET` *(Seite: 771)*.

Die Formen `DEFAULT` und `RESET` setzen den Sitzungsbenutzernamen und den aktuellen Benutzernamen auf den ursprünglichen authentifizierten Benutzernamen zurück. Diese Formen können von jedem Benutzer ausgeführt werden.

Beispiele

```
SELECT session_user, current_user;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT session_user, current_user;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Kompatibilität

Der SQL-Standard erlaubt einige andere Ausdrücke an der Stelle von *benutzername*, die in der Praxis nicht wichtig sind. PostgreSQL erlaubt die Bezeichnersyntax (`"benutzername"`), aber SQL nicht. Laut SQL ist es nicht erlaubt, diesen Befehl in einer Transaktion auszuführen; PostgreSQL macht diese Einschränkung nicht, weil es dazu keinen Grund gibt. Die Privilegien, die zur Ausführung dieses Befehls nötig sind, werden vom SQL-Standard der Implementierung überlassen.

SET TRANSACTION

Name

SET TRANSACTION – setzt die Charakteristika der aktuellen Transaktion

Synopsis

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL { READ COMMITTED |
SERIALIZABLE }
```

Beschreibung

Der Befehl `SET TRANSACTION` setzt den Transaktionsisolationsgrad der aktuellen Transaktion. Er hat auf nachfolgende Transaktionen keine Auswirkung. Dieser Befehl kann nicht mehr ausgeführt werden, nachdem die ersten Anfrage oder der erste Datenmodifikationsbefehl (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`) in einer Transaktion ausgeführt wurde. `SET SESSION CHARACTERISTICS` setzt den Standard-Transaktionsisolationsgrad für neue Transaktionen in der aktuellen Sitzung. Mit `SET TRANSACTION` kann er für jede Transaktion einzeln geändert werden.

Der Transaktionsisolationsgrad bestimmt, welche Daten eine Transaktion sehen kann, wenn andere Transaktionen gleichzeitig ablaufen.

`READ COMMITTED`

Ein Befehl kann nur Daten aus Transaktionen sehen, die abgeschlossen wurden, bevor der Befehl begann. Das ist die Voreinstellung.

`SERIALIZABLE`

Die aktuelle Transaktion kann nur Daten aus Transaktionen sehen, die abgeschlossen wurden, bevor die erste Anfrage oder der erste Datenmodifikationsbefehl in dieser Transaktion ausgeführt wurde.

Tipp

Intuitiv bedeutet, dass zwei gleichzeitige Transaktionen die Datenbank in dem Zustand hinterlassen, den sie gehabt hätte, wenn die beiden Transaktionen streng hintereinander ausgeführt worden wären.

Weitere Informationen dazu finden Sie in Abschnitt SET.

Hinweise

Der voreingestellte Transaktionsisolationsgrad kann auch mit dem Befehl

```
SET default_transaction_isolation = 'wert'
```

und in der Konfigurationsdatei gesetzt werden. Siehe Abschnitt SET.

Kompatibilität

Beide Befehle sind im SQL-Standard definiert. Im SQL-Standard ist der Standard-Transaktionsisolationsgrad `SERIALIZABLE`; in PostgreSQL ist es `READ COMMITED`, aber das können Sie wie oben beschrieben ändern. In PostgreSQL gibt es nicht die Isolationsgrade `READ UNCOMMITTED` und `REPEATABLE READ`. Wegen des Multiversionsmodells in PostgreSQL ist der Grad `SERIALIZABLE` nicht wirklich serialisierbar. Einzelheiten finden Sie in Abschnitt SET.

Der SQL-Standard sieht zwei weitere Transaktionscharakteristika vor, die mit diesen Befehlen gesetzt werden können: ob die Transaktion nur Lesevorgänge erlaubt und die Größe des Diagnosebereichs. Beide werden von PostgreSQL nicht unterstützt.

SHOW

Name

SHOW – zeigt den Wert eines Konfigurationsparameters

Synopsis

```
SHOW name
SHOW ALL
```

Beschreibung

`SHOW` zeigt den aktuellen Wert eines Konfigurationsparameters. Diese Parameter können durch den Befehl `SET`, die Konfiguratonsdatei `postgresql.conf`, die Umgebungsvariable `PGOPTIONS` (mit `libpq` und auf `libpq` basierenden Anwendungen) oder mit Kommandozeilenoptionen beim Start von `postmaster` gesetzt werden. Einzelheiten dazu finden Sie in Abschnitt SET.

Selbst wenn `autocommit` aus ist, wird durch `SHOW` kein neuer Transaktionsblock gestartet. Einzelheiten dazu finden Sie im Abschnitt über `autocommit` in Abschnitt SET.

Parameter

name

Der Name eines Konfigurationsparameters. Die verfügbaren Parameter sind in Abschnitt SET und auf der Referenzseite von `SET` *(Seite: 771)* beschrieben. Außerdem gibt es Parameter, die nur gezeigt, aber nicht gesetzt werden können:

`SERVER_ENCODING`

Zeigt die Zeichensatzkodierung auf der Serverseite. Dieser Parameter wird bei der Erzeugung der Datenbank festgelegt.

`ALL`

Zeigt die Werte aller Konfigurationsparameter.

Meldungen

`ERROR: Option 'name' is not recognized`
Meldung, wenn *name* kein bekannter Parameter ist.

Hinweise

Die Funktion `current_setting` kann auch verwendet werden, um die Werte von Konfigurationsparametern anzuzeigen. Siehe Abschnitt SET.

Beispiele

Zeigt die aktuelle Einstellung des Parameters `DateStyle`:

```
SHOW DateStyle;
              DateStyle
-------------------------------------
 ISO with US (NonEuropean) conventions
(1 row)
Zeigt die aktuelle Einstellung des Parameters geqo:
SHOW geqo;
 geqo
------
 on
(1 row)
Zeige alle Einstellungen:
SHOW ALL;
             name              |                setting
-------------------------------+---------------------------------------
 australian_timezones          | off
 authentication_timeout        | 60
 checkpoint_segments           | 3
    .
    .
    .
 wal_debug                     | 0
 wal_sync_method               | fdatasync
(94 rows)
```

Kompatibilität

Der Befehl `SHOW` ist eine PostgreSQL-Erweiterung.

Siehe auch

SET *(Seite: 771)*

START TRANSACTION

Name

START TRANSACTION – startet einen Transaktionsblock

Synopsis

```
START TRANSACTION [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
```

Beschreibung

Dieser Befehl beginnt eine neue Transaktion. Wenn ein Isolationsgrad angegeben wurde, hat die neue Transaktion diesen Isolationsgrad, als ob SET TRANSACTION *(Seite: 776)* ausgeführt worden wäre. Ansonsten ist das Verhalten dieses Befehls identisch mit dem Befehl BEGIN *(Seite: 638)*.

Parameter

Die Bedeutung der Parameter finden Sie unter SET TRANSACTION *(Seite: 776)* beschrieben.

Meldungen

START TRANSACTION

Meldung, wenn der Befehl erfolgreich ausgeführt wurde.

WARNING: BEGIN: already a transaction in progress

Meldung, wenn bereits eine Transaktion aktiv war, als der Befehl ausgeführt wurde.

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform; siehe aber auch im Kompatibilitätsabschnitt von SET TRANSACTION *(Seite: 776)*.

TRUNCATE

Name

TRUNCATE – leert eine Tabelle

Synopsis

```
TRUNCATE [ TABLE ] name
```

Beschreibung

`TRUNCATE` entfernt schnell alle Zeilen aus einer Tabelle. Es hat die gleiche Auswirkung wie `DELETE` ohne Bedingung, ist aber schneller, da die Tabelle nicht durchsucht werden muss. Am nützlichsten ist es bei großen Tabellen.

`TRUNCATE` kann nicht in einem Transaktionsblock (`BEGIN`/`COMMIT`-Paar) ausgeführt werden, da es nicht zurückgerollt werden kann.

Parameter

name

Der Name der Tabelle (möglicherweise mit Schemaqualifikation), die geleert werden soll.

Meldungen

`TRUNCATE TABLE`

Meldung, wenn die Tabelle erfolgreich geleert wurde.

Beispiele

Um die Tabelle `riesentabelle` zu leeren:

```
TRUNCATE TABLE riesentabelle;
```

Kompatibilität

Der Befehl `TRUNCATE` ist eine PostgreSQL-Erweiterung.

UNLISTEN

Name

UNLISTEN – beendet das Hören auf eine Benachrichtigung

Synopsis

```
UNLISTEN { name | * }
```

Beschreibung

`UNLISTEN` wird verwendet, um eine bestehende Registrierung für `NOTIFY`-Ereignisse zu löschen. `UNLISTEN` storniert alle bestehenden Registrierungen der aktuellen PostgreSQL-Sitzung für die Benachrichtigung *name*. Wenn als Name * angegeben wurde, werden alle Registrierungen der aktuellen Sitzung gelöscht.

`NOTIFY` *(Seite: 750)* enthält einer ausführlichere Beschreibung der Verwendung der Befehle `LISTEN` und `NOTIFY`.

Parameter

name

Der Name einer Benachrichtigung (ein beliebiger SQL-Bezeichner).

*

Löscht alle Registrierungen dieser Sitzung.

Meldungen

`UNLISTEN`

Meldung, wenn der Befehl ausgeführt wurde.

Hinweise

Sie können eine Registrierung für einen Namen auch löschen, wenn er gar nicht registriert war. Dabei wird keine Warnung und kein Fehler ausgegeben.

Am Ende jeder Sitzung wird automatisch `UNLISTEN *` ausgeführt.

Beispiele

So wird eine Registrierung erzeugt und verwendet:

```
LISTEN virtual;
NOTIFY virtual;
```

```
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

Nachdem `UNLISTEN` ausgeführt wurde, werden darauf folgende `NOTIFY`-Befehle ignoriert:

```
UNLISTEN virtual;
NOTIFY virtual;
-- kein NOTIFY-Ereignis wird empfangen
```

Kompatibilität

Der Befehl `UNLISTEN` ist eine PostgreSQL-Erweiterung.

UPDATE

Name

UPDATE – aktualisiert Zeilen einer Tabelle

Synopsis

```
UPDATE [ ONLY ] tabelle SET spalte = ausdruck [, ...]
    [ FROM fromliste ]
    [ WHERE bedingung ]
```

Beschreibung

`UPDATE` ändert die Werte der angegebenen Spalten in allen Zeilen, die die Bedingung erfüllen. Nur die Spalten, die geändert werden sollen, müssen als Spalten in dem Befehl angegeben werden.

Normalerweise aktualisiert `UPDATE` Zeilen in der angegebenen Tabelle und ihren Untertabellen. Wenn Sie nur in der angegebenen Tabelle aktualisieren wollen, müssen Sie die Klausel `ONLY` verwenden.

Um eine Tabelle zu aktualisieren, müssen Sie das Privileg `UPDATE` für die Tabelle haben sowie das Privileg `SELECT` für jede Tabelle, die in der `WHERE`-Klausel verwendet wird.

Parameter

`tabelle`

Der Name der zu aktualisierenden Tabelle (möglicherweise mit Schemaqualifikation).

`spalte`

Der Name einer Spalte in *`tabelle`*.

`ausdruck`

Ein Ausdruck oder Wert, der der Spalte zugewiesen werden soll.

`fromliste`

Eine Liste von Tabellenausdrücken. Dadurch können Spalten anderer Tabellen in der WHERE-Klausel verwendet werden.

bedingung

Ein Wertausdruck, der einen Wert vom Typ boolean liefert und bestimmt, welche Zeilen aktualisiert werden sollen.

Meldungen

UPDATE *zahl*

Meldung, wenn Zeilen erfolgreich aktualisiert wurden. *zahl* ist die Anzahl der aktualisierten Zeilen. Wenn *zahl* 0 ist, wurden keine Zeilen aktualisiert.

Beispiele

Ändere das Wort Drama in Dramatisch in der Spalte genre der Tabelle filme:

```
UPDATE filme SET genre = 'Dramatisch' WHERE genre = 'Drama';
```

Kompatibilität

Dieser Befehl ist mit dem SQL-Standard konform. Die FROM-Klausel ist eine PostgreSQL-Erweiterung.

VACUUM

Name

VACUUM – säubert und analysiert wahlweise eine Datenbank

Synopsis

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ tabelle ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ tabelle [ (spalte [, ...] ) ] ]
```

Beschreibung

VACUUM gibt den von gelöschten Zeilenversionen in einer Tabelle belegten Speicherplatz frei. PostgreSQL entfernt gelöschte oder durch Aktualisierung hinfällig gewordene Zeilenversionen nicht sofort physikalisch aus der Tabelle; sie müssen durch VACUUM entfernt werden. Deshalb ist es notwendig, regelmäßig VACUUM auszuführen, besonders in Tabellen, die oft aktualisiert werden.

Ohne Parameter bearbeitet VACUUM jede Tabelle in der aktuellen Datenbank. Mit Parameter wird nur die angegebene Tabelle bearbeitet.

VACUUM ANALYZE führt für jede angegebene Tabelle zuerst VACUUM und dann ANALYZE aus. Diese Kombination ist für Wartungsskripte praktisch. Weitere Informationen finden Sie auch unter ANALYZE *(Seite: 636)*.

Ein einfaches VACUUM (ohne FULL) gibt einfach Platz zur Wiederverwendung frei. Diese Form des Befehls kann gleichzeitig mit anderen Lese- und Schreibvorgängen in der Tabelle laufen, da keine exklusive Sperre gesetzt wird. VACUUM FULL ist weitreichender. Unter anderem werden Zeilen zwischen Blöcken verschoben, um die Tabelle auf die geringstmögliche Anzahl von Diskblöcken zu verkleinern. Diese Form ist wesentlich langsamer und benötigt eine exklusive Sperre für jede bearbeitete Tabelle.

Die Option FREEZE sorgt dafür, dass Zeilenversionen so bald wie möglich als „eingefroren“ markiert werden, anstatt zu warten, bis sie ziemlich alt geworden sind. Wenn das gemacht wird, wenn es in derselben Datenbank keine offenen Transaktionen gibt, ist garantiert, dass in der Datenbank alle Zeilenversionen „eingefroren“ sind und keine Probleme mit Transaktionsnummernüberlauf haben werden, egal, wie lange in der Datenbank kein VACUUM ausgeführt wird. FREEZE ist nicht für die regelmäßige Verwendung gedacht, sondern nur für die Vorbereitung einer benutzerdefinierten Templatedatenbank oder anderer Datenbanken, die ausschließlich gelesen werden und nicht regelmäßig mit VACUUM aufgeräumt werden. Einzelheiten dazu finden Sie in Abschnitt SET.

Parameter

FULL

Wählt das „volle“ VACUUM, das mehr Platz wiedergewinnt, aber länger braucht und die Tabelle exklusiv sperrt.

FREEZE

Wählt aggressives „Einfrieren“ von Zeilen.

VERBOSE

Gibt für jede Tabelle einen detaillierten Bericht über die Aktivität aus.

ANALYZE

Aktualisiere, die Statistiken, die vom Planer verwendet werden, um den besten Ausführungsplan für eine Anfrage zu ermitteln.

tabelle

Der Name einer Tabelle (möglicherweise mit Schemaqualifikation), die bearbeitet werden soll. Wenn keine Tabelle angegeben ist, werden alle Tabellen in der Datenbank bearbeitet.

spalte

Der Name einer bestimmten Spalte, die analysiert werden soll. Wenn keine Spalten angegeben sind, werden alle Spalten analysiert.

Meldungen

VACUUM

Der Befehl ist fertig.

INFO: --Relation *tabelle*--

Der Anfang eines Berichts über *tabelle*.

```
INFO: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000,
Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space 586952/
586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.
```

Der Analysebericht für *tabelle*.

`INFO: Index *index*: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.`

Der Analysebericht über einen Index der Tabelle.

Hinweise

Wir empfehlen, dass VACUUM in aktiven Produktionsdatenbanken regelmäßig ausgeführt wird (mindestens jede Nacht), damit gelöschte Zeilen entfernt werden. Außerdem sollte VACUUM ANALYZE ausgeführt werden, wenn eine große Zahl von Zeilen eingefügt oder gelöscht worden ist. Damit werden die Statistiken mit den Ergebnissen aller Änderungen aktualisiert, wodurch der Anfrageplaner beim Planen von Anfragen bessere Entscheidungen treffen kann.

Die Option FULL ist nicht für die regelmäßige Verwendung gedacht, kann aber in besonderen Fällen nützlich sein. Ein Beispiel ist, wenn Sie die meisten Zeilen aus einer Tabelle gelöscht haben und die Tabelle physikalisch verkleinern wollen, um den Speicherplatzverbrauch zu verringern. VACUUM FULL kann eine Tabelle in der Regel besser verkleinern als ein einfaches VACUUM.

Beispiele

So sieht es aus, wenn VACUUM mit einer Tabelle aus der Regressionsdatenbank ausgeführt wird:

```
=> VACUUM VERBOSE ANALYZE onek;
INFO:  --Relation onek--
INFO:  Index onek_unique1: Pages 14; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.11u sec elapsed 0.12 sec.
INFO:  Index onek_unique2: Pages 16; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
INFO:  Index onek_hundred: Pages 13; Tuples 1000: Deleted 3000.
        CPU 0.00s/0.10u sec elapsed 0.10 sec.
INFO:  Index onek_stringu1: Pages 31; Tuples 1000: Deleted 3000.
        CPU 0.01s/0.09u sec elapsed 0.10 sec.
INFO:  Removed 3000 tuples in 70 pages.
        CPU 0.02s/0.04u sec elapsed 0.07 sec.
INFO:  Pages 94: Changed 0, Empty 0; Tup 1000: Vac 3000, Keep 0, UnUsed 0.
        Total CPU 0.05s/0.45u sec elapsed 0.59 sec.
INFO:  Analyzing onek
VACUUM
```

Kompatibilität

Der Befehl VACUUM ist eine PostgreSQL-Erweiterung.

II. PostgreSQL-Clientanwendungen

Dieser Teil enthält Referenzinformationen über PostgreSQL-Clientanwendungen und Hilfsprogramme. Nicht alle dieser Programme sind für alle Anwender gedacht; für einige von ihnen benötigen Sie besondere Privilegien. Das gemeinsame Merkmal dieser Anwendungen ist es, dass sie von jedem Host aus aufgerufen werden können, unabhängig davon, wo der Datenbankserver ist.

clusterdb

Name

clusterdb – clustert eine PostgreSQL-Datenbank

Synopsis

`clusterdb` [*`verbindungsoption`*...] [--table | -t *`tabelle`*] [*`dbname`*]
`clusterdb` [*`verbindungsoption`*...] [--all | -a]

Beschreibung

`clusterdb` ist ein Hilfsprogramm, um Tabellen in einer PostgreSQL-Datenbank neu zu clustern. Es findet Tabellen, die vorher schon geclustert worden sind, und clustert sie erneut mit demselben Index, der beim letzten Mal verwendet wurde. Tabellen, die noch nie geclustert wurden, werden nicht berührt.

`clusterdb` ist ein Shell-Skript-Wrapperprogramm um den SQL-Befehl `CLUSTER` *(Seite: 641)* unter Verwendung des Programms `psql` *(Seite: 819)*. Es macht keinen Unterschied, ob eine Datenbank mit dieser oder einer anderen Methode geclustert wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Es kann sein, dass `clusterdb` mehrmals mit dem PostgreSQL-Server verbinden muss und jedes Mal nach einem Passwort fragt. In solchen Fällen kann es sinnvoll sein, die Datei `$HOME/.pgpass` einzurichten.

Optionen

`clusterdb` akzeptiert die folgenden Kommandozeilenargumente:

`-a`
`--all`

Clustere alle Datenbanken.

`[-d]` *`dbname`*
`[--dbname]` *`dbname`*

Gibt den Namen der Datenbank an, die geclustert werden soll. Wenn dies nicht angegeben ist und `-a` (oder `--all`) nicht verwendet wird, dann wird der Datenbankname aus der Umgebungsvariable `PGDATABASE` gelesen. Wenn diese nicht gesetzt ist, wird der für die Verbindung angegebene Benutzername verwendet.

`-e`
`--echo`

Gibt die Befehle aus, die `clusterdb` erzeugt und an den Server schickt.

`-q`
`--quiet`

Gibt keine Antwort aus.

`-t` *`tabelle`*
`--table` *`tabelle`*

Clustere nur *`tabelle`*.

`clusterdb` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe.

Meldungen

`CLUSTER`

Alles hat geklappt.

`clusterdb: Cluster failed.`

Irgendwas ist schief gelaufen. `clusterdb` ist nur ein Wrapper-Skript. Bei `CLUSTER` *(Seite: 641)* und `psql` *(Seite: 819)* finden Sie eine detailliertere Besprechung von Fehlermeldungen und potenziellen Problemen. Beachten Sie, dass diese Meldung einmal für jede geclusterte Tabelle erscheinen könnte.

Umgebung

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Beispiele

Um die Datenbank `test` zu clustern:

```
$ clusterdb test
```

Um eine einzelne Tabelle `foo` in einer Datenbank namens `xyzzy` zu clustern:

```
$ clusterdb --table foo xyzzy
```

Siehe auch

`CLUSTER` *(Seite: 641)*

createdb

Name

createdb – erzeugt eine neue PostgreSQL-Datenbank

Synopsis

`createdb` [*`option`*...] [*`dbname`*] [*`beschreibung`*]

Beschreibung

`createdb` erzeugt eine neue PostgreSQL-Datenbank.

Normalerweise wird der Datenbankbenutzer, der diesen Befehl ausführt, der Eigentümer der neuen Datenbank. Ein anderer Eigentümer kann jedoch mit der Option `-O` angegeben werden, wenn der ausführende Benutzer entsprechende Privilegien hat.

`createdb` ist ein Shell-Skript-Wrapperprogramm um den SQL-Befehl `CREATE DATABASE` *(Seite: 660)* unter Verwendung des Programms `psql` *(Seite: 819)*. Es macht keinen Unterschied, ob eine Datenbank mit dieser oder einer anderen Methode erzeugt wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Optionen

`createdb` akzeptiert die folgenden Kommandozeilenargumente:

`dbname`

Gibt den Namen der zu erzeugenden Datenbank an. Der Name muss sich von allen PostgreSQL-Datenbanken in dem Cluster unterscheiden. Wenn kein Datenbankname angegeben wird, dann wird eine Datenbank mit dem gleichen Namen wie der aktuelle Systembenutzer erzeugt.

`beschreibung`

Gibt wahlweise einen Kommentar an, der zu der neuen Datenbank gehören soll.

`-D` *`wert`*
`--location` *`wert`*

Gibt einen alternativen Speicherplatz für die Datenbank an. Siehe auch `initlocation` *(Seite: 845)*.

`-e`
`--echo`

Gibt die Befehle aus, die `createdb` erzeugt und an den Server schickt.

`-E` *`kodierung`*
`--encoding` *`kodierung`*

Gibt die Zeichensatzkodierung für die neue Datenbank an.

`-O` *`eigentümer`*
`--owner` *`eigentümer`*

Gibt den Datenbankbenutzer an, dem die neue Datenbank gehören soll.

`-q`
`--quiet`

Gibt keine Antwort aus.

`-T` *`template`*
`--template` *`template`*

Gibt die Templatedatenbank an, aus der die neue Datenbank gebaut werden soll.

Die Optionen `-D`, `-E`, `-O` und `-T` entsprechen Optionen des zugrunde liegenden SQL-Befehls `CREATE DATABASE` *(Seite: 660)*; dort finden Sie weitere Informationen darüber.

`createdb` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe.

Meldungen

`CREATE DATABASE`

Die Datenbank wurde erfolgreich erzeugt.

`createdb: Database creation failed.`

Irgendwas ist schief gegangen.

`createdb: Comment creation failed. (Database was created.)`

Der Kommentar/die Beschreibung für die Datenbank konnte nicht erzeugt werden. Die Datenbank wurde aber schon erzeugt. Die können den SQL-Befehl `COMMENT ON DATABASE` verwenden, um den Kommentar später hinzuzufügen.

Wenn ein Fehler auftritt, wird die Fehlermeldung vom Server angezeigt. Siehe `CREATE DATABASE` *(Seite: 660)* und `psql` *(Seite: 819)* für mögliche Meldungen.

Umgebung

`PGDATABASE`

Wenn gesetzt, dann der Name der zu erzeugenden Datenbank, wenn keiner auf der Kommandozeile angegeben wurde.

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter. `PGUSER` bestimmt auch den Namen der zu erzeugenden Datenbank, wenn keiner auf der Kommandozeile oder durch `PGDATABASE` angegeben wurde.

Beispiele

Um die Datenbank `demo` auf dem Standarddatenbankserver zu erzeugen:

```
$ createdb demo
CREATE DATABASE
```

Das ist die gleiche Anwort, die Sie vom SQL-Befehl `CREATE DATABASE` erhalten hätten.

Um die Datenbank `demo` auf dem Server auf dem Host `eden`, Port 5000, mit der Zeichensatzkodierung `LATIN1` zu erzeugen und sich den im Hintergrund ausgeführten SQL-Befehl anzusehen:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

Siehe auch

`dropdb` *(Seite: 795)*, `CREATE DATABASE` *(Seite: 660)*

createlang

Name

createlang – definiert eine neue prozedurale Sprache in PostgreSQL

Synopsis

createlang [*verbindungsoption*...] *sprachname* [*dbname*]
createlang [*verbindungsoption*...] --list | -l *dbname*

Beschreibung

createlang ist ein Hilfsprogramm um eine neue Programmiersprache in einer PostgreSQL-Datenbank zu definieren. createlang kann alle Sprachen, die zur PostgreSQL-Standarddistribution gehören, verarbeiten, aber keine Sprachen, die anderswo angeboten werden.

Obwohl Sie Server-Programmiersprachen mit mehreren SQL-Befehlen direkt definieren können, wird empfohlen, createlang zu verwenden, da es einige Fehler abfangen kann und viel einfacher zu verwenden ist. Weitere Informationen finden Sie bei CREATE LANGUAGE *(Seite: 671)*.

Optionen

createlang akzeptiert die folgenden Kommandozeilenargumente:

sprachname

Gibt den Namen der zu definierenden prozeduralen Programmiersprache an.

[-d] *dbname*
[--dbname] *dbname*

Gibt die Datenbank an, in der die Sprache definiert werden soll. Wenn keine Datenbank angegeben ist, wird die Datenbank mit dem Namen des aktuellen Systembenutzers verwendet.

-e
--echo

Gibt die SQL-Befehle aus, die im Hintergrund ausgeführt werden.

-l
--list

Zeigt eine Liste aller installierten Sprachen in der Zieldatenbank.

-L *verzeichnis*

Gibt das Verzeichnis an, in dem der Interpreter für die Sprache gefunden werden soll. Dieses Verzeichnis wird normalerweise automatisch gefunden; diese Option ist hauptsächlich zum Debuggen da.

createlang akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

-h *host*
--host *host*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

-p *port*
--port *port*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

-U *benutzername*
--username *benutzername*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe.

Umgebung

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

Die meisten Meldungen sollten klar sein. Wenn nicht, führen Sie `createlang` mit der Option `--echo` aus und sehen Sie unter dem entsprechenden SQL-Befehl wegen Einzelheiten nach. Weitere Möglichkeiten finden Sie auch bei `psql` *(Seite: 819)*.

Hinweise

Mit `droplang` *(Seite: 797)* können Sie eine Sprache entfernen.

`createlang` ist ein Shell-Skript, das `psql` mehrmals aufruft. Wenn Sie es so eingestellt haben, dass zum Verbinden ein Passwwort benötigt wird, werden Sie mehrere Male nach einem Passwort gefragt.

Beispiele

Um die Sprache `pltcl` in die Datenbank `template1` zu installieren:

```
$ createlang pltcl template1
```

Siehe auch

`droplang` *(Seite: 797)*, `CREATE LANGUAGE` *(Seite: 671)*

createuser

Name

createuser – definiert einen neuen PostgreSQL-Benutzerzugang

Synopsis

`createuser` [*option*...] [*benutzername*]

Beschreibung

`createuser` erzeugt einen neuen PostgreSQL-Benutzer. Nur Superuser (Benutzer, bei denen `usesuper` in der Tabelle `pg_shadow` gesetzt ist) können neue Benutzer erzeugen, also muss `createuser` von jemandem aufgerufen werden, der als PostgreSQL-Superuser verbinden kann.

Ein Superuser kann gleichzeitig auch alle Prüfungen der Zugriffsrechte innerhalb der Datenbank umgehen, also sollte man den Superuser-Status nicht leichtfertig vergeben.

`createuser` ist ein Shell-Skript-Wrapperprogramm, um den SQL-Befehl `CREATE USER` *(Seite: 702)* unter Verwendung des Programms `psql` *(Seite: 819)*. Es macht keinen Unterschied, ob ein Benutzer mit dieser oder einer anderen Methode erzeugt wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Optionen

`createuser` akzeptiert die folgenden Kommandozeilenargumente:

`benutzername`

Gibt den Namen des PostgreSQL-Benutzers an, der erzeugt werden soll. Dieser Name muss sich von allen PostgreSQL-Benutzern unterscheiden.

`-a`
`--adduser`

Der neue Benutzer darf wiederum andere Benutzer erzeugen. (Achtung: Eigentlich wird der Benutzer dadurch zum *`Superuser`*. Der Name dieser Option wurde schlecht gewählt.)

`-A`
`--no-adduser`

Der neue Benutzer darf keine anderen Benutzer erzeugen. (Das heißt, der neue Benutzer ist ein normaler Benutzer, kein Superuser.)

`-d`
`--createdb`

Der neue Benutzer darf Datenbanken erzeugen.

`-D`
`--no-createdb`

Der neue Benutzer darf keine Datenbanken erzeugen.

`-e`
`--echo`

Gibt die Befehle aus, die `createuser` erzeugt und an den Server schickt.

`-E`
`--encrypted`

Verschlüsselt das Benutzerpasswort in der Datenbank. Wenn nichts angegeben wird, wird die Voreinstellung verwendet.

`-i` *`zahl`*
`--sysid` *`zahl`*

Damit können Sie dem neuen Benutzer eine selbst gewählte ID-Nummer zuweisen (anstatt vom Server eine erzeugen zu lassen). Das ist nicht erforderlich, aber manche Anwender finden es nützlich.

`-N`
`--unencrypted`

Verschlüsselt das Benutzerpasswort in der Datenbank nicht. Wenn nichts angegeben wird, wird die Voreinstellung verwendet.

`-P`
`--pwprompt`

Wenn diese Option angegeben ist, dann wird `createuser` Sie dazu auffordern, für den neuen Benutzer ein Passwort einzugeben. Das ist nicht notwendig, wenn Sie nicht beabsichtigen, die Passwortauthentifizierung zu verwenden.

`-q`
`--quiet`

Gibt keine Antwort aus.

Wenn der Name oder andere Informationen auf der Kommandozeile nicht angegeben wurden, werden Sie danach gefragt werden und müssen sie interaktiv eingeben.

`createuser` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll (nicht der zu erzeugende Benutzer).

`-W`
`--password`

Erzwingt eine Passworteingabe.

Umgebung

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

`CREATE USER`

Alles klar.

`createuser: creation of user "`*`username`*`" failed`

Irgendwas ist schief gelaufen. Der Benutzer wurde nicht erzeugt.

Wenn ein Fehler auftritt, wird die Fehlermeldung vom Server angezeigt. Sie `CREATE USER` *(Seite: 702)* und `psql` *(Seite: 819)* für mögliche Meldungen.

Beispiele

Um den Benutzer `joe` auf dem Standarddatenbankserver zu erzeugen:

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Um den gleichen Benutzer `joe` auf dem Server auf dem Host `eden`, Port 5000, ohne Nachfragen zu erzeugen und sich den im Hintergrund ausgeführten SQL-Befehl anzusehen:

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

Siehe auch

`dropuser` *(Seite: 799)*, `CREATE USER` *(Seite: 702)*

dropdb

Name

dropdb – entfernt eine PostgreSQL-Datenbank

Synopsis

`dropdb` [*option*...] *dbname*

Beschreibung

`dropdb` zerstört eine bestehende PostgreSQL-Datenbank. Der Benutzer, der diesen Befehl ausführt, muss ein Datenbank-Superuser oder der Eigentümer der Datenbank sein.

`dropdb` ist ein Shell-Skript-Wrapperprogramm, um den SQL-Befehl `DROP DATABASE` *(Seite: 713)* unter Verwendung des Programms `psql` *(Seite: 819)*. Es macht keinen Unterschied, ob eine Datenbank mit dieser oder einer anderen Methode entfernt wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Optionen

`dropdb` akzeptiert die folgenden Kommandozeilenargumente:

`dbname`

Gibt den Namen der Datenbank an, die entfernt werden soll.

`-e`
`--echo`

Gibt die Befehle aus, die `dropdb` erzeugt und an den Server schickt.

`-i`
`--interactive`

Fragt noch einmal nach, bevor irgendetwas zerstört wird.

`-q`
`--quiet`

Gibt keine Antwort aus.

`createdb` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe.

Meldungen

`DROP DATABASE`

Die Datenbank wurde erfolgreich entfernt.

`dropdb: Database removal failed.`

Irgendetwas ist schief gelaufen.

Wenn ein Fehler auftritt, wird die Fehlermeldung vom Server angezeigt. Siehe `DROP DATABASE` *(Seite: 713)* und `psql` *(Seite: 819)* für mögliche Meldungen.

Umgebung

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Beispiele

Um die Datenbank demo auf dem Standarddatenbankserver zu zerstören:

```
$ dropdb demo
DROP DATABASE
```

Um die Datenbank demo auf dem Server auf dem Host eden, Port 5000, mit Nachfragen zu zerstören und sich den im Hintergrund ausgeführten SQL-Befehl anzusehen:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

Siehe auch

createdb *(Seite: 788)*, DROP DATABASE *(Seite: 713)*

droplang

Name

droplang – entfernt eine prozedurale Sprache aus PostgreSQL

Synopsis

droplang [*verbindungsoption*...] *sprachname* [*dbname*]
droplang [*verbindungsoption*...] --list | -l *dbname*

Beschreibung

droplang ist ein Hilfsprogramm, um eine bestehende Programmiersprache aus einer PostgreSQL-Datenbank zu entfernen. droplang kann jede prozedurale Sprache entfernen, selbst solche, die nicht mit PostgreSQL geliefert wurden.

Obwohl Sie Server-Programmiersprachen mit mehreren SQL-Befehlen direkt entfernen können, wird empfohlen, droplang zu verwenden, da es einige Fehler abfangen kann und viel einfacher zu verwenden ist. Weitere Informationen finden Sie bei DROP LANGUAGE *(Seite: 719)*.

Optionen

droplang akzeptiert die folgenden Kommandozeilenargumente:

sprachname

Gibt den Namen der Server-Programmiersprache an, die entfernt werden soll.

`[-d] dbname`
`[--dbname] dbname`

Gibt die Datenbank an, aus der die Sprache entfernt werden soll. Wenn keine Datenbank angegeben ist, wird die Datenbank mit dem Namen des aktuellen Systembenutzers verwendet.

`-e`
`--echo`

Gibt die SQL-Befehle aus, die im Hintergrund ausgeführt werden.

`-l`
`--list`

Zeigt eine Liste aller installierten Sprachen in der Zieldatenbank.

`droplang` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h host`
`--host host`

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p port`
`--port port`

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U benutzername`
`--username benutzername`

Der Benutzername, unter dem verbunden werden soll

`-W`
`--password`

Erzwingt eine Passworteingabe.

Umgebung

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

Die meisten Meldungen sollten klar sein. Wenn nicht, führen Sie `droplang` mit der Option `--echo` aus und sehen Sie unter dem entsprechenden SQL-Befehl wegen Einzelheiten nach. Weitere Möglichkeiten finden Sie auch bei `psql` *(Seite: 819)*.

Hinweise

Mit `createlang` *(Seite: 790)* können Sie eine neue Sprache hinzufügen.

Beispiele

Um die Sprache `pltcl` zu entfernen:

```
$ droplang pltcl dbname
```

Siehe auch

`createlang` *(Seite: 790)*, `DROP LANGUAGE` *(Seite: 719)*

dropuser

Name

dropuser – entfernt einen PostgreSQL-Benutzerzugang

Synopsis

`dropuser` [*option*...] [*benutzername*]

Beschreibung

`dropuser` entfernt einen bestehenden PostgreSQL-Benutzer *und* alle Datenbanken, die dem Benutzer gehörten. Nur Superuser (Benutzer, bei denen `usesuper` in der Tabelle `pg_shadow` gesetzt ist) können PostgreSQL-Benutzer entfernen.

`dropuser` ist ein Shell-Skript-Wrapperprogramm um den SQL-Befehl `DROP USER` *(Seite: 730)* unter Verwendung des Programms `psql` *(Seite: 819)*. Es macht keinen Unterschied, ob ein Benutzer mit dieser oder einer anderen Methode entfernt wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Optionen

`dropuser` akzeptiert die folgenden Kommandozeilenargumente:

`benutzername`

Gibt den Namen des PostgreSQL-Benutzers an, der entfernt werden soll. Wenn kein Name auf der Kommandozeile angegeben wird, werden Sie zur Eingabe aufgefordert werden.

`-e`
`--echo`

Gibt die Befehle aus, die `dropuser` erzeugt und an den Server schickt.

`-i`
`--interactive`

Frage nach Bestätigung, bevor der Benutzer wirklich entfernt wird.

`-q`
`--quiet`

Gibt keine Antwort aus.

`createuser` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll (nicht der zu löschende Benutzer)

`-W`
`--password`

Erzwingt eine Passworteingabe.

Umgebung

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

`DROP USER`

Alles klar.

`dropuser: deletion of user "`*`username`*`" failed`

Irgendetwas ist schief gegangen. Der Benutzer wurde nicht entfernt.

Wenn ein Fehler auftritt, wird die Fehlermeldung vom Server angezeigt. Siehe `DROP USER` *(Seite: 730)* und `psql` *(Seite: 819)* für mögliche Meldungen.

Beispiel

Um den Benutzer `joe` vom Standarddatenbankserver zu entfernen:

```
$ dropuser joe
DROP USER
```

Um den Benutzer `joe` vom Server auf dem Host `eden`, Port 5000, mit Nachfragen zu entfernen und sich den im Hintergrund ausgeführten SQL-Befehl anzusehen:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
```

```
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

Siehe auch

`createuser` *(Seite: 792)*, `DROP USER` *(Seite: 730)*

ecpg

Name

`ecpg` – Präprozessor für eingebettetes SQL in C

Synopsis

`ecpg` [*option*...] *datei*...

Beschreibung

`ecpg` ist der Präprozessor für C-Programme mit eingebettetem SQL. Er wandelt C-Programme mit eingebetteten SQL-Befehlen in normalen C-Code um, indem die SQL-Aufrufe durch spezielle Funktionsaufrufe ersetzt werden. Die Ausgabedateien können dann mit beliebigen C-Compiler-Werkzeugen weiterverarbeitet werden.

`ecpg` wandelt jede auf der Kommandozeile angegebene Eingabedatei in die entsprechende C-Ausgabedatei um. Eingabedateien haben bevorzugt die Erweiterung `.pgc`; in dem Fall wird für den Namen der Ausgabedatei die Erweiterung durch `.c` ersetzt. Wenn die Erweiterung der Eingabedatei nicht `.pgc` ist, wird der Ausgabedateiname ermittelt, indem `.c` an den vollen Dateinamen angehängt wird. Der Name der Ausgabedatei kann auch durch die Option `-o` bestimmt werden.

Diese Referenzseite beschreibt nicht die eingebettete SQL-Sprache. Informationen dazu finden Sie in Abschnitt SET.

Optionen

`ecpg` akzeptiert die folgenden Kommandozeilenoptionen:

`-c`

Erzeugt bestimmten C-Code automatisch aus SQL-Code. Aktuell betrifft das `EXEC SQL TYPE`.

`-D` *`symbol`*

Definiert ein C-Präprozessorsymbol.

`-I` *`verzeichnis`*

Gibt einen zusätzlichen Pfad an, in dem nach Dateien gesucht wird, die mit `EXEC SQL INCLUDE` eingebunden wurden. Voreingestellt sind `.` (aktuelles Verzeichnis), `/usr/local/include`, das PostgreSQL-

Headerdatei-Verzeichnis, welches bei der Compilierung festgelegt wird (Standard: `/usr/local/pgsql/include`) und `/usr/include`, in dieser Reihenfolge.

`-o` *`dateiname`*

Gibt an, dass `ecpg` die gesamte Ausgabe in die angegebene Datei schreiben soll.

`-t`

Schalte Autocommit ein. In diesem Modus wird jeder Befehl automatisch in einer eigenen Transaktion ausgeführt, welche automatisch abgeschlossen wird, außer wenn er in einem ausdrücklichen Transaktionsblock steht. Im Standardmodus werden Transaktionen nur abgeschlossen, wenn `EXEC SQL COMMIT` ausgeführt wird.

`-v`

Gibt zusätzliche Informationen aus, einschließlich der Versionsnummer und dem Pfad für eingebundene Dateien.

`--help`

Zeigt eine kurze Hilfe zum Befehl und beendet dann.

`--version`

Gibt Versionsinformationen aus und beendet dann.

Hinweise

Wenn die verarbeiteten C-Code-Dateien kompiliert werden, muss der Compiler die `ECPG`-Headerdateien im PostgreSQL-Headerdatei-Verzeichnis finden können. Eventuell muss man also die Option `-I` verwenden, wenn man den Compiler aufruft (z.B. `-I/usr/local/pgsql/include`).

Programme, die C-Code mit eingebettetem SQL verwenden, müssen mit der Bibliothek `libecpg` gelinkt werden, zum Beispiel mit den Linkeroptionen `-L/usr/local/pgsql/lib -lecpg`.

Die für die jeweilige Installation gültigen Werte für diese beiden Verzeichnisse können mit `pg_config` *(Seite: 802)* herausgefunden werden.

Beispiel

Wenn Sie eine C-Quelldatei mit eingebettetem SQL haben, die `prog1.pgc` heißt, können Sie daraus mit dieser Befehlsfolge ein ausführbares Programm erzeugen:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_config

Name

pg_config – ermittelt Informationen über die installierte Version von PostgreSQL

Synopsis

pg_config {--bindir | --includedir | --includedir-server | --libdir | --pkglibdir | --configure | --version...}

Beschreibung

Das Hilfsprogramm pg_config gibt Konfigurationsparameter der gegenwärtig installierten Version von PostgreSQL aus. Es ist zum Beispiel vorgesehen für Softwarepakete, die PostgreSQL verwenden, damit sie die erforderlichen Headerdateien und Bibliotheken einfacher finden können.

Optionen

Um pg_config zu verwenden, geben Sie eine oder mehrere der folgenden Optionen an.

--bindir

Gibt den Ort der Benutzerprogramme an. Verwenden Sie das zum Beispiel, um das Programm psql zu finden. Dies ist normalerweise auch der Ort, wo das Programm pg_config selbst sich befindet.

--includedir

Gibt den Ort der C-Headerdateien der Clientschnittstellen aus.

--includedir-server

Gibt den Ort der C-Headerdateien für die Serverprogrammierung aus.

--libdir

Gibt den Ort der Objektcode-Bibliotheken aus.

--pkglibdir

Gibt den Ort der dynamisch ladbaren Module aus oder wo der Server nach ihnen suchen würde. (Andere plattformabhängige Datendateien könnten auch in diesem Verzeichnis installiert sein.)

--configure

Gibt die Optionen aus, die an das configure-Skript übergeben wurden, als PostgreSQL zum Compilieren konfiguriert wurde. Das kann verwendet werden, um die identische Konfiguration zu wiederholen oder um herauszufinden, mit welchen Optionen ein vorkompiliertes Paket gebaut wurde. (Beachten Sie allerdings, dass vorkompilierte Pakete häufig noch herstellerabhängige Patches enthalten.)

--version

Gibt die Version von PostgreSQL aus und beendet.

Wenn mehr als eine Option (außer --version) angegeben wird, werden die Informationen in genau der Reihenfolge ausgegeben, ein Element pro Zeile.

Hinweise

Die Option --includedir-server war neu in PostgreSQL 7.2. In Versionen davor waren die Server-Headerdateien im selben Verzeichnis wie die Client-Headerdateien, welches mit der Option --includedir ermittelt werden konnte, installiert. Um dafür so sorgen, dass Ihr Paket beide Fälle verarbeiten kann, versuchen Sie zuerst die neuere Option und prüfen Sie den Ergebnisstatus und zu sehen, ob es funktioniert hat.

In Versionen vor PostgreSQL 7.1, bevor pg_config entstanden ist, gab es keine gleichwertige Methode, um Konfigurationsinformationen herauszufinden.

Geschichte

Das Programm pg_config erschien zum ersten Mal in PostgreSQL 7.1.

pg_dump

Name

pg_dump – schreibt eine PostgreSQL-Datenbank in eine Skriptdatei oder andere Archivdatei

Synopsis

pg_dump [*option*...] [*dbname*]

Beschreibung

pg_dump ist Hilfsprogramm, das PostgreSQL-Datenbanken in einer Skript- oder Archivdatei speichern kann. Die Skriptdateien bestehen aus reinem Text und enthalten die SQL-Befehle, die notwendig sind, um die Datenbank so wiederherzustellen, wie sie zum Zeitpunkt war, als sie gesichert wurde. Um diese Skripte wiederherzustellen, verwenden Sie psql *(Seite: 819)*. Man kann damit die Datenbank selbst auf anderen Maschinen und anderen Architekturen, und mit einigen Veränderungen auch auf anderen SQL-Datenbankprodukten, wiedererzeugen.

Außerdem gibt es alternative Archivdateiformate, die zusammen mit pg_restore *(Seite: 812)* verwendet werden können, um die Datenbank wiederherzustellen, und die es pg_restore ermöglichen, Objekte selektiv und sogar in veränderter Reihenfolge wiederherzustellen. Die Archivformate wurden so entworfen, dass sie zwischen allen Architekturen portierbar sind.

Wenn man eins der Archivformate zusammen mit pg_restore verwendet, bietet pg_dump einen flexiblen Archivier- und Transfermechanismus. pg_dump kann verwendet werden, um die gesamte Datenbank zu sichern und dann kann pg_restore verwendet werden, um das Archiv einzusehen und auszuwählen, welche Teile der Datenbank wiederhergestellt werden sollen. Das flexibelste Ausgabeformat ist das Format „Custom“ (-Fc). Es ermöglichst die freie Auswahl und das freie Umsortieren aller archivierten Objekte und wird in der Voreinstellung komprimiert. Das Tar-Format (-Ft) ist nicht komprimiert und erlaubt nicht das Umsortieren der Objekte, aber es ist ansonsten recht flexibel; außerdem kann es mit anderen Programmen wie zum Beispiel tar bearbeitet werden.

Während man pg_dump ausführt sollte man darauf achten, ob Warnungen ausgegeben werden (auf der Standardfehlerausgabe), besonders wegen der weiter unten beschriebenen Beschränkungen.

pg_dump erstellt konsistente Sicherungsdateien, selbst wenn die Datenbank zur selben Zeit verändert wird. pg_dump blockiert keine anderen Benutzer, die gerade auf die Datenbank zugreifen (lesend oder schreibend).

Optionen

Die folgenden Kommandozeilenoptionen kontrollieren das Ausgabeformat.

dbname

Gibt den Namen der Datenbank an, die gesichert werden soll. Wenn dies nicht angegeben ist, wird der Datenbankname aus der Umgebungsvariable PGDATABASE gelesen. Wenn diese nicht gesetzt ist, wird der für die Verbindung angegebene Benutzername verwendet.

-a
--data-only

Sichert nur die Daten, nicht das Schema (die Datendefinition).

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie pg_restore ausführen.

-b
--blobs

Sichert auch Large Objects mit.

-c
--clean

Gibt Befehle zum Löschen der Datenbankobjekte vor den Befehlen zum Erzeugen aus.

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie pg_restore ausführen.

-C
--create

Fängt die Ausgabe mit einem Befehl an, der die Datenbank selbst erzeugt und dann mit dieser Datenbank verbindet. (Mit so einem Skript ist es egal, mit welcher Datenbank Sie verbunden sind, bevor Sie es ausführen.)

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie pg_restore ausführen.

-d
--inserts

Sichert die Daten als INSERT-Befehle (anstatt COPY). Das macht die Wiederherstellung sehr langsam, aber es macht die Archive besser auf andere SQL-Datenbankprodukte portierbar.

-D
--column-inserts
--attribute-inserts

Sichert die Daten als INSERT-Befehle mit ausdrücklichen Spaltennamen (INSERT INTO *tabelle* (*spalte*, ...) VALUES ...). Das macht die Wiederherstellung sehr langsam, aber es ist notwendig, wenn Sie die Spaltenreihenfolge verändern wollen.

-f *datei*
--file=*datei*

Sendet die Ausgabe an die angegebene Datei. Wenn diese Option ausgelassen wird, wird die Standardausgabe verwendet.

-F *format*
--format=*format*

Wählt das Format für die Ausgabe. *format* kann eins der Folgenden sein:

p

Gibt eine reine Textdatei aus (Voreinstellung).

t

Gibt ein Tar-Archiv aus, das als Eingabe für pg_restore verwendet werden kann. Mit diesem Archivformat können Schemaelemente umgeordnet oder ausgelassen werden. Es ist auch möglich einzugrenzen, welche Daten bei der Wiederherstellung geladen werden sollen.

c

Gibt ein „Custom“ Archiv aus, das als Eingabe für `pg_restore` verwendet werden kann. Dies ist das flexibelste Format, weil man damit die Daten und die Schemaelemente umsortieren kann. Dieses Format ist in der Voreinstellung außerdem komprimiert.

`-i`
`--ignore-version`

Fährt bei nicht zusammenpassenden Versionen von `pg_dump` und Datenbankserver trotzdem fort.

`pg_dump` kann Datenbanken von früheren PostgreSQL-Versionen verarbeiten, aber sehr alte Versionen (gegenwärtig vor 7.0) werden nicht mehr unterstützt. Verwenden Sie diese Option, wenn Sie die Versionsprüfung umgehen wollen (aber wenn `pg_dump` dann fehlschlägt, sagen Sie nicht, dass Sie nicht gewarnt wurden).

`-o`
`--oids`

Gibt für jede Tabelle die OIDs mit aus. Verwenden Sie diese Option, wenn Ihre Anwendung die OID-Spalten auf irgendeine Art und Weise verwendet (zum Beispiel in Fremdschlüsseln). Ansonsten sollte diese Option nicht verwendet werden.

`-O`
`--no-owner`

Gibt keine Befehle aus, die dafür sorgen, dass die Objekteigentümer mit der ursprünglichen Datenbank übereinstimmen. Normalerweise gibt `pg_dump` (`psql`-spezifische) `\connect`-Befehle aus um die Eigentümer der Schemaobjekte zu setzen. Weitere Informationen darüber finden Sie bei `-R` und `-X use-set-session-authorization`. Beachten Sie, dass die Option `-O` nicht alle Verbindungsversuche mit der Datenbank unterbindet, sondern nur solche, die dazu dienen, die Eigentumsverhältnisse wiederherzustellen.

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie `pg_restore` ausführen.

`-R`
`--no-reconnect`

Verbietet `pg_dump`, ein Skript auszugeben, dass bei der Wiederherstellung erfordern würde, dass die Datenbankverbindung neu aufgebaut wird. Ein typisches Wiederherstellungsskript muss normalerweise mehrere Male neu verbinden, um die ursprünglichen Eigentumsverhältnisse der Objekte einzurichten. Diese Option führt allerdings dazu, dass `pg_dump` die Informationen über die Eigentümer einfach verliert, *es sei denn,* Sie verwenden die Option `-X use-set-session-authorization`.

Ein möglicher Grund, warum Neuverbinden während der Wiederherstellung unerwünscht sein könnte, ist, wenn der Zugriff auf die Datenbank das Eingreifen von Hand erfordert (z.B. Passwörter).

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie `pg_restore` ausführen.

`-s`
`--schema-only`

Sichert nur das Schema (Datendefinition), nicht die Daten.

`-S` *`benutzername`*
`--superuser=`*`benutzername`*

Gibt den Namen des Superusers an, der verwendet werden soll, um die Trigger abzuschalten. Das ist nur von Bedeutung, wenn `--disable-triggers` verwendet wird. (Normalerweise ist es besser `--use-set-session-authorization` zu verwenden und das daraus resultierende Skript als Superuser zu starten.)

`-t` *`tabelle`*
`--table=`*`tabelle`*

Gibt nur die Daten von *`tabelle`* aus.

`-v`
`--verbose`

Mit dieser Option wird pg_dump diverse Fortschrittsmeldungen auf die Standardfehlerausgabe schreiben.

-x
--no-privileges
--no-acl

Gibt die Zugriffsprivilegien nicht mit aus (GRANT/REVOKE-Befehle).

-X use-set-session-authorization
--use-set-session-authorization

Wenn ein von pg_dump erzeugtes Skript (reiner Textmodus) den aktuellen Datenbankbenutzer ändern muss (um zum Beispiel den richtigen Eigentümer einzustellen), dann verwendet es normalerweise den psql-Befehl \connect. Dieser Befehl öffnet in Wirklichkeit eine neue Verbindung, was zum Beispiel eine Passworteingabe von Hand erfordern könnte. Wenn Sie die Option -X use-set-session-authorization verwenden, wird pg_dump anstelle dessen den Befehl SET SESSION AUTHORIZATION *(Seite: 774)* ausgeben. Das hat den gleichen Endeffekt, aber es erzwingt, dass der Benutzer, der die Datenbank mit dem erzeugten Skript wiederherstellen möchte, ein Datenbank-Superuser ist. Diese Option schaltet im Prinzip auch die Option -R aus.

Da SET SESSION AUTHORIZATION *(Seite: 774)* ein standardisierter SQL-Befehl ist, wohingegen \connect nur in psql funktioniert, erhöhen Sie mit dieser Option theoretisch auch die Portierbarkeit des ausgegebenen Skripts.

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie pg_restore ausführen.

-X disable-triggers
--disable-triggers

Diese Option ist nur von Bedeutung, wenn Sie nur die Daten, nicht das Schema sichern. Sie weist pg_dump an, Befehle auszugeben, welche Trigger für die Zieltabellen vorübergehend ausschalten, während die Daten geladen werden. Verwenden Sie diese Option, wenn Sie Fremdschlüssel oder andere Trigger für die betroffenen Tabellen haben, die Sie bei der Wiederherstellung nicht aktiv haben wollen.

Gegenwärtig müssen die für --disable-triggers ausgegebenen Befehle als Superuser ausgeführt werden. Sie sollten also mit -S einen Superusernamen angeben oder besser noch --use-set-session-authorization verwenden und dann das erzeugte Skript als Superuser starten. Wenn Sie weder die eine noch die andere Option angeben, muss das gesamte Skript als Superuser ausgeführt werden.

Diese Option ist nur für das reine Textformat von Bedeutung. Bei den anderen Formaten können Sie diese Option angeben, wenn Sie pg_restore ausführen.

-Z *0..9*
--compress=*0..9*

Gibt das Komprimierungsniveau an, das bei Archivformaten, die Komprimierung unterstützen, verwendet werden soll. (Gegenwärtig unterstützt nur das Archivformat „Custom“ die Komprimierung.)

Die folgenden Kommandozeilenoptionen kontrollieren die Datenbankverbindungsparameter.

-h *host*
--host *host*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet. Wenn kein Host angegeben wird, wird die Umgebungsvariable PGHOST verwendet, wenn Sie gesetzt ist, ansonsten wird eine Verbindung über eine Unix-Domain-Socket versucht.

-p *port*
--port *port*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet. Wenn kein Port angegeben wird, dann wird die Umgebungsvariable PGPORT verwendet, wenn Sie gesetzt ist, ansonsten wird die eingebaute Standardportnummer verwendet.

`-U` *benutzername*
`--username` *benutzername*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe. Das sollte automatisch geschehen, wenn der Server Passwortauthentifizierung erfordert.

Lange Optionen gibt es nur auf einigen Plattformen.

Umgebung

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

`pg_dump` führt intern `SELECT`-Befehle aus. Wenn Sie Probleme bei der Ausführung von `pg_dump` haben, versichern Sie sich, dass Sie, zum Beispiel mit `psql` *(Seite: 819)*, Daten aus der Datenbank lesen können.

Hinweise

Wenn in Ihrem Datenbankcluster die Datenbank `template1` verändert wurde, sollten Sie aufpassen, dass Sie die Ausgabe von `pg_dump` in einer wirklich leeren Datenbank wiederherstellen; ansonsten werden Sie wahrscheinlich Fehler verursachen, weil schon in `template1` vorhandene Objekte wiederhergestellt werden würden. Um eine leere Datenbank ohne lokale Veränderungen zu erzeugen, erstellen Sie sie aus `template0`, nicht aus `template1`, zum Beispiel:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

`pg_dump` hat ein paar Beschränkungen:

- Wenn nur eine einzelne Tabelle ausgegeben oder das reine Textformat verwendet wird, kann `pg_dump` Large Objects nicht ausgeben. Large Objects müssen mit der gesamten Datenbank und in einem der Nicht-Text-Formate gesichert werden.
- Wenn nur die Daten gesichert werden sollen und die Option `--disable-triggers` verwendet wird, gibt `pg_dump` Befehle aus, um Trigger für die betroffenen Tabellen abzuschalten, bevor die Daten eingefügt werden, und um sie danach wieder einzuschalten. Wenn die Wiederherstellung zwischendurch unterbrochen wird, könnten die Systemkataloge im falschen Zustand zurückgelassen werden.

Mitglieder von Tar-Archiven müssen weniger als 8 GB groß sein. (Das ist eine im Tar-Dateiformat verankerte Beschränkung.) Daher kann dieses Format nicht verwendet werden, wenn die Textdarstellung einer Tabelle diese Größe überschreitet. Die Gesamtgröße eines Tar-Archivs und der anderen Ausgabeformate ist nicht begrenzt, außer möglicherweise durch das Betriebssystem.

Beispiele

Um eine Datenbank zu sichern:

```
$ pg_dump meinedb > db.out
```

Um diese Datenbank wiederherzustellen:

```
$ psql -d datenbank -f db.out
```

Um eine Datenbank namens `meinedb` mit den Large Objects in eine `tar`-Datei zu sichern:

```
$ pg_dump -Ft -b meinedb > db.tar
```

Um diese Datenbank (mit Large Objects) in einer bestehenden Datenbank namens `neuedb` wiederherzustellen:

```
$ pg_restore -d neuedb db.tar
```

Geschichte

Das Programm `pg_dump` erschien zum ersten Mal in `Postgres95` Version 0.02. Die Formate neben dem Textformat wurden in `PostgreSQL` Version 7.1 eingeführt.

Siehe auch

`pg_dumpall` *(Seite: 809)*, `pg_restore` *(Seite: 812)*, `psql` *(Seite: 819)*

pg_dumpall

Name

pg_dumpall – schreibt einen PostgreSQL-Datenbankcluster in eine Skriptdatei

Synopsis

`pg_dumpall` [*option*...]

Beschreibung

`pg_dumpall` ist ein Hilfsprogramm, das alle PostgreSQL-Datenbanken eines Clusters in einer einzigen Skriptdatei speichern kann. Die Skriptdatei enthält die SQL-Befehle, die als Eingabe für `psql` *(Seite: 819)* verwendet werden können, um die Datenbanken wiederherzustellen. Intern ruft `pg_dumpall` für jede Datenbank im Cluster `pg_dump` *(Seite: 804)* auf. Außerdem sichert `pg_dumpall` globale Objekte, die zu allen Datenbanken gehören. (`pg_dump` sichert diese Objekte nicht.) Gegenwärtig sind das die Informationen über Datenbankbenutzer und Gruppen.

Folglich ist pg_dumpall eine integrierte Lösung, um Sicherungskopien von Ihren Datenbanken anzufertigen. Beachten Sie aber ein Einschränkung: Es kann keine Large Objects sichern, da pg_dump solche Objekte nicht in Textdateien sichern kann. Wenn Sie Datenbanken mit Large Objects haben, dann sollten Sie diese mit pg_dump in einem Nicht-Text-Format sichern.

Da pg_dumpall Tabellen aus allen Datenbanken liest, müssen Sie höchstwahrscheinlich als Datenbank-Superuser verbinden, um eine komplette Sicherungsdatei erzeugen zu können. Außerdem werden sie Superuser-Privilegien benötigen, um das erzeugte Skript ausführen zu können, damit Sie Benutzer, Gruppen und Datenbank erzeugen dürfen.

Das SQL-Skript wird auf den Standardausgabestrom geschrieben. Mit Shell-Operatoren können Sie es in eine Datei umleiten.

pg_dumpall muss mehrmals mit dem PostgreSQL-Server verbunden werden und es kann sein, dass es jedes Mal nach einem Passwort fragt. In solchen Fällen kann es sinnvoll sein, die Datei $HOME/.pgpass einzurichten.

Optionen

Die folgenden Kommandozeilenoptionen kontrollieren das Ausgabeformat.

-c
--clean

Gib tSQL-Befehle aus, um die Datenbanken zu entfernen, bevor Sie erzeugt werden.

-d
--inserts

Sichert die Daten als INSERT-Befehle (anstatt COPY). Das macht die Wiederherstellung sehr langsam, aber es macht die Archive besser auf andere SQL-Datenbankprodukte portierbar.

-D
--column-inserts
--attribute-inserts

Sichert die Daten als INSERT-Befehle mit ausdrücklichen Spaltennamen (INSERT INTO *tabelle* (*spalte*, ...) VALUES ...). Das macht die Wiederherstellung sehr langsam, aber es ist notwendig, wenn Sie die Spaltenreihenfolge verändern wollen.

-g
--globals-only

Sichert nur globale Objekte (Benutzer und Gruppen), keine Datenbanken.

-i
--ignore-version

Fährt bei nicht zusammenpassenden Versionen von pg_dumpall und Datenbankserver trotzdem fort.

pg_dumpall kann Datenbanken von früheren PostgreSQL-Versionen verarbeiten, aber sehr alte Versionen (gegenwärtig vor 7.0) werden nicht mehr unterstützt. Verwenden Sie diese Option, wenn Sie die Versionsprüfung umgehen wollen (aber wenn pg_dumpall dann fehlschlägt, sagen Sie nicht, dass Sie nicht gewarnt wurden).

-o
--oids

Gibt für jede Tabelle die OIDs mit aus. Verwenden Sie diese Option, wenn Ihre Anwendung die OID-Spalten auf irgendeine Art und Weise verwendet (zum Beispiel in Fremdschlüsseln). Ansonsten sollte diese Option nicht verwendet werden.

-v
--verbose

Mit dieser Option wird pg_dumpall diverse Fortschrittsmeldungen auf die Standardfehlerausgabe schreiben.

Die folgenden Kommandozeilenoptionen kontrollieren die Datenbankverbindungsparameter.

-h *host*
--host *host*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet. Wenn kein Host angegeben wird, wird die Umgebungsvariable PGHOST verwendet, wenn Sie gesetzt ist, ansonsten wird eine Verbindung über eine Unix-Domain-Socket versucht.

-p *port*
--port *port*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet. Wenn kein Port angegeben wird, dann wird die Umgebungsvariable PGPORT verwendet, wenn Sie gesetzt ist, ansonsten wird die eingebaute Standardportnummer verwendet.

-U *benutzername*
--username *benutzername*

Der Benutzername, unter dem verbunden werden soll.

-W
--password

Erzwinge eine Passworteingabe. Das sollte automatisch geschehen, wenn der Server Passwortauthentifizierung erfordert.

Lange Optionen gibt es nur auf einigen Plattformen.

Umgebung

PGHOST
PGPORT
PGUSER

Standardverbindungsparameter

Hinweise

Da pg_dumpall intern pg_dump aufruft, werden sich einige Meldungen auf pg_dump beziehen.

Beispiele

Um alle Datenbanken zu sichern:

```
$ pg_dumpall > db.out
```

Um diese Datenbank wiederherzustellen, verwenden Sie zum Beispiel:

```
$ psql -f db.out template1
```

(Es ist nicht wichtig, mit welcher Datenbank Sie hier verbinden, da die von pg_dumpall erzeugte Skriptdatei die passenden Befehle enthält, um die gesicherten Datenbanken zu erzeugen und mit ihnen zu verbinden.)

Siehe auch

pg_dump *(Seite: 804)*, psql *(Seite: 819)*. Schauen Sie da nach, um Einzelheiten zu möglichen Fehlermeldungen zu erhalten.

pg_restore

Name

pg_restore – stellt eine PostgreSQL-Datenbank aus einer mit pg_dump erzeugten Archivdatei wieder her

Synopsis

pg_restore [*option*...] [*dateiname*]

Beschreibung

pg_restore ist ein Hilfsprogramm, das eine PostgreSQL-Datenbank aus einer mit pg_dump *(Seite: 804)* erzeugten Archivdatei (außer im reinen Textformat) wiederherstellen kann. Es führt die Befehle aus, die notwendig sind, um die Datenbank so wiederherzustellen, wie sie zum Zeitpunkt war, als sie gesichert wurde. Die Archivdateien ermöglichen es pg_restore auch, auszuwählen, was wiederhergestellt werden soll, und sogar die Objekte in veränderter Reihenfolge wiederherzustellen. Die Archivformate wurden so entworfen, dass sie zwischen allen Architekturen portierbar sind.

pg_restore kann in zwei Modi arbeiten: Wenn ein Datenbankname angegeben wird, wird das Archiv direkt in die Datenbank geladen. Large Objects können nur mit einer direkten Datenbankverbindung wiederhergestellt werden. Ansonsten wird eine Skriptdatei erzeugt (entweder auf die Standardausgabe oder in eine Datei geschrieben), die die SQL-Befehle enthält, um die Datenbank wiederherzustellen, ähnlich der von pg_dump mit dem Textformat erzeugten Skripts. Einige Optionen, die die Skriptausgabe kontrollieren, sind daher analog zu Optionen von pg_dump.

pg_restore kann logischerweise nur Informationen wiederherstellen, die in der Archivdatei enthalten sind. Wenn zum Beispiel das Archiv mit der Option erzeugt wurden, die die Daten als INSERT-Befehle ausgibt, kann pg_restore die Daten nicht mit COPY-Befehlen laden lassen.

Optionen

pg_restore akzeptiert die folgenden Kommandozeilenargumente:

dateiname

Gibt die Archivdatei an, die wiederhergestellt werden soll. Wenn nicht angeben, wird die Standardeingabe verwendet.

-a
--data-only

Stellt nur die Daten wieder her, nicht das Schema (die Datendefinitionen).

-c
--clean

Entfernt die Datenbankobjekte, bevor sie erzeugt werden.

`-C`
`--create`

Erzeugt die Datenbank, bevor Daten in sie geladen werden. (Wenn diese Option verwendet wird, wird die durch `-d` genannte Datenbank nur verwendet, um den Befehl `CREATE DATABASE` am Anfang auszuführen. Alle Daten werden in der im Archiv genannten Datenbank wiederhergestellt.)

`-d` *`dbname`*
`--dbname=`*`dbname`*

Verbindet mit Datenbank *`dbname`* und lädt die Daten direkt in die Datenbank.

`-f` *`dateiname`*
`--file=`*`dateiname`*

Gibt die Ausgabedatei für das erzeugte Skript oder die mit `-l` erzeugte Liste an. Wenn nichts angegeben wird, wird die Standardausgabe verwendet.

`-F` *`format`*
`--format=`*`format`*

Gibt das Format der Archivdatei an. Normalerweise bestimmt `pg_restore` das Format automatisch, es muss also nicht angegeben werden. Wenn es angegeben wird, muss es eins der folgenden Werte sein:

`t`

Das Archiv ist ein `Tar`-Archiv. Mit diesem Archivformat können Schemaelemente umgeordnet oder ausgelassen werden. Es ist auch möglich, einzugrenzen, welche Daten bei der Wiederherstellung geladen werden sollen.

`c`

Das Archiv ist im „Custom“-Format von `pg_dump`. Dies ist das flexibelste Format, weil man damit die Daten und die Schemaelemente umsortieren kann. Dieses Format ist in der Voreinstellung außerdem komprimiert.

`-i`
`--ignore-version`

Ignoriert Datenbankversionsprüfungen.

`-I` *`index`*
`--index=`*`index`*

Stellt nur die Definition des benannten Index wieder her.

`-l`
`--list`

Listet den Inhalt des Archivs. Die Ausgabe dieser Operation kann mit der Option `-L` verwendet werden, um Objekte bei der Wiederherstellung auszuwählen oder umzusortieren.

`-L` *`listendatei`*
`--use-list=`*`listendatei`*

Stellt nur die in *`listendatei`* aufgezählten Objekte wieder her, in der Reihenfolge, in der sie in der Datei auftauchen. Zeilen können verschoben und auskommentiert werden, indem ein `;` an den Zeilenanfang gestellt wird. (Siehe Beispiele unten.)

`-N`
`--orig-order`

Stellt die Objekte in der Originalreihenfolge wieder her. Normalerweise gibt `pg_dump` die Objekte in einer Reihenfolge, die für `pg_dump` günstig ist, aus und erzeugt das Archiv dann in einer modifizierten OID-Reihenfolge. Diese Option übergeht die OID-Reihenfolge.

`-o`
`--oid-order`

Stellt die Objekte in OID-Reihenfolge wieder her. Normalerweise gibt pg_dump die Objekte in einer Reihenfolge, die für pg_dump günstig ist, aus und erzeugt das Archiv dann in einer modifizierten OID-Reihenfolge. Diese Option erzwingt eine strikte OID-Reihenfolge.

-O
--no-owner

Unterdrückt alle Versuche, die ursprünglichen Eigentumsverhältnisse der Objekte wiederherzustellen. Die Objekte gehören dann zu dem Benutzer, unter dem verbunden wurde. Siehe auch bei -R und -X use-set-session-authorization.

-P *funktionsname(argtyp [, ...])*
--function=*funktionsname(argtyp [, ...])*

Stellt nur die benannte Funktion wieder her.

-r
--rearrange

Stellt die Objekte in modifizierter OID-Reihenfolge wieder her. Normalerweise gibt pg_dump die Objekte in einer Reihenfolge, die für pg_dump günstig ist, aus und erzeugt das Archiv dann in einer modifizierten OID-Reihenfolge. Die meisten Objekte werden in der OID-Reihenfolge wiederhergestellt, aber einige Objekte (z.B. Regeln und Indexe) werden am Ende des Vorgangs wiederhergestellt, ungeachtet ihrer OIDs. Diese Option ist die Voreinstellung.

-R
--no-reconnect

Beim Wiederherstellen eines Archivs muss pg_restore typischerweise mehrere Male mit der Datenbank unter verschiedenen Benutzernamen verbinden, um die Eigentumsverhältnisse richtig zu setzen. Wenn das unerwünscht ist (zum Beispiel weil das Eingreifen von Hand erforderlich wäre (Passwörter)), hält diese Option pg_restore davon ab, jemals neu zu verbinden. (Ohne direkte Datenbankverbindung wird bei einer Neuverbindung der psql-Befehl *(Seite: 819)* \connect ausgegeben.) Diese Option führt allerdings dazu, dass pg_restore die Informationen über die Eigentümer einfach verliert, *es sei denn,* Sie verwenden die Option -X use-set-session-authorization.

-s
--schema-only

Stellt nur das Schema (die Datendefinitionen) wieder her, nicht die Daten. Sequenzwerte werden zurückgesetzt.

-S *benutzername*
--superuser=*benutzername*

Gibt den Namen des Superusers an, der verwendet werden soll, um die Trigger abzuschalten. Das ist nur von Bedeutung, wenn --disable-triggers verwendet wird.

-t *tabelle*
--table=*tabelle*

Stellt nur die Definition und/oder die Daten der benannten Tabelle wieder her.

-T *trigger*
--trigger=*trigger*

Stellt nur den benannten Trigger wieder her.

-v
--verbose

Wenn diese Option verwendet wird, wird pg_restore diverse Fortschrittsmeldungen auf die Standardfehlerausgabe schreiben.

-x
--no-privileges
--no-acl

Stellt keine Zugriffsprivilegien wieder her (GRANT/REVOKE-Befehle).

`-X use-set-session-authorization`
`--use-set-session-authorization`

Wenn bei der Wiederherstellung eines Archivs der aktuelle Datenbankbenutzer geändert werden muss (um zum Beispiel den richtigen Eigentümer einzustellen), muss normalerweise eine neue Verbindung zur Datenbank geöffnet werden, was zum Beispiel eine Passworteingabe von Hand erfordern könnte. Wenn Sie die Option `-X use-set-session-authorization` verwenden, wird `pg_restore` anstelle dessen den Befehl `SET SESSION AUTHORIZATION` *(Seite: 774)* verwenden. Das hat den gleichen Endeffekt, aber es erzwingt, dass der Benutzer, der das Archiv wiederherstellen möchte, ein Datenbank-Superuser ist. Diese Option schaltet im Prinzip auch die Option `-R` aus.

`-X disable-triggers`
`--disable-triggers`

Diese Option ist nur von Bedeutung, wenn Sie nur die Daten, nicht das Schema wiederherstellen. Sie weist `pg_restore` an, Befehle auszugeben, welche Trigger für die Zieltabellen vorübergehend ausschalten, während die Daten geladen werden. Verwenden Sie diese Option, wenn Sie Fremdschlüssel oder andere Trigger für die betroffenen Tabellen haben, die Sie bei der Wiederherstellung nicht aktiv haben wollen.

Gegenwärtig müssen die für `--disable-triggers` ausgegebenen Befehle als Superuser ausgeführt werden. Sie sollten also mit `-S` einen Superusernamen angeben, oder besser noch `--use-set-session-authorization` verwenden und `pg_restore` als PostgreSQL-Superuser ausführen.

`pg_restore` akzeptiert außerdem die folgenden Kommandozeilenoptionen für Verbindungsparameter:

`-h` *`host`*
`--host` *`host`*

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet. Wenn kein Host angegeben wird, wird die Umgebungsvariable `PGHOST` verwendet, wenn Sie gesetzt ist, ansonsten wird eine Verbindung über eine Unix-Domain-Socket versucht.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet. Wenn kein Port angegeben wird, wird die Umgebungsvariable `PGPORT` verwendet, wenn Sie gesetzt ist, ansonsten wird die eingebaute Standardportnummer verwendet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe. Das sollte automatisch geschehen, wenn der Server Passwortauthentifizierung erfordert.

Lange Optionen gibt es nur auf einigen Plattformen.

Umgebung

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Meldungen

Wenn eine direkte Datenbankverbindung durch die Option `-d` bestimmt wird, führt `pg_restore` intern SQL-Befehle aus. Wenn Sie Probleme bei der Ausführung von `pg_restore` haben, versichern Sie sich, dass Sie, zum Beispiel mit `psql` *(Seite: 819)*, Daten aus der Datenbank lesen können.

Hinweise

Wenn in Ihrem Datenbankcluster die Datenbank `template1` verändert wurde, sollten Sie aufpassen, dass Sie die Ausgabe von `pg_restore` in eine wirklich leere Datenbank laden; ansonsten werden Sie wahrscheinlich Fehler verursachen, weil schon in `template1` vorhandene Objekte erneut erzeugt werden würden. Um eine leere Datenbank ohne lokale Veränderungen zu erzeugen, erstellen Sie sie aus `template0`, nicht aus `template1`, zum Beispiel:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Die Beschränkungen von `pg_restore` werden im Folgenden beschrieben.

- Wenn Daten in eine bestehende Tabelle wiederhergestellt werden und die Option `--disable-triggers` verwendet wird, führt `pg_restore` Befehle aus, um Trigger für die betroffenen Tabellen abzuschalten, bevor die Daten eingefügt werden, und um sie danach wieder einzuschalten. Wenn die Wiederherstellung zwischendurch unterbrochen wird, könnten die Systemkataloge im falschen Zustand zurückgelassen werden.
- `pg_restore` wird keine Large Objects für eine einzelne Tabelle wiederherstellen. Wenn ein Archiv Large Objects enthält, dann werden alle Large Objects wiederhergestellt.

Siehe auch unter `pg_dump` *(Seite: 804)* bezüglich Einzelheiten über die Beschränkungen von `pg_dump`.

Beispiele

Um eine Datenbank namens `meinedb` mit den Large Objects in eine `tar`-Datei zu sichern:

```
$ pg_dump -Ft -b meinedb > db.tar
```

Um diese Datenbank (mit Large Objects) in einer bestehenden Datenbank namens `neuedb` wiederherzustellen:

```
$ pg_restore -d neuedb db.tar
```

Um die Datenbankobjekte umzusortieren, muss zuerst das Inhaltsverzeichnis des Archivs ausgegeben werden:

```
$ pg_restore -l archivdatei > archivliste
```

Das Verzeichnis besteht aus einem Kopfeintrag und einer Zeile für jedes Objekt, zum Beispiel:

```
;
; Archive created at Fri Jul 28 22:28:36 2000
;     dbname: birds
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
```

```
;     Format: CUSTOM
;
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semikolons beginnen einen Kommentar, und die Zahlen am Anfang jeder Zeile beziehen sich auf die interne Nummer jedes Objekts im Archiv.

Die Zeilen in der Datei können auskommentiert, gelöscht oder umsortiert werden. Zum Beispiel:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

Das könnte als Eingabe für `pg_restore` verwendet werden, und damit würden nur die Objekte Nummer 10 und 6, in dieser Reihenfolge, wiederhergestellt werden:

```
$ pg_restore -L archivliste archivdatei
```

Geschichte

Das Programm `pg_restore` erschien zum ersten Mal in PostgreSQL 7.1.

Siehe auch

`pg_dump` *(Seite: 804)*, `pg_dumpall` *(Seite: 809)*, `psql` *(Seite: 819)*

pgtclsh

Name

pgtclsh – PostgreSQL `Tcl`-Shell-Client

Synopsis

`pgtclsh` [*`dateiname`* [*`argument`*...]]

Beschreibung

`pgtclsh` ist eine `Tcl`-Shell, die um Funktionen für den Zugriff auf PostgreSQL-Datenbanken erweitert wurde. (Im Prinzip ist es `tclsh` mit `libpgtcl` geladen.) Wie bei der normalen `Tcl`-Shell ist das erste Argument eine Skriptdatei und etwaige folgende Argumente werden dem Skript übergeben. Wenn keine Skriptdatei angegeben wird, dann ist die Shell interaktiv.

Eine `Tcl`-Shell mit `Tk`- und PostgreSQL-Funktionen steht unter dem Namen `pgtksh` *(Seite: 818)* zur Verfügung.

Siehe auch

`pgtksh` *(Seite: 818)*, Abschnitt SET (Beschreibung von `libpgtcl`), tclsh

pgtksh

Name

pgtksh – PostgreSQL `Tcl/Tk`-Shell-Client

Synopsis

`pgtksh` [*`dateiname`* [*`argument`*...]]

Beschreibung

`pgtksh` ist eine `Tcl/Tk`-Shell, die um Funktionen für den Zugriff auf PostgreSQL-Datenbanken erweitert wurde. (Im Prinzip ist es `wish` mit `libpgtcl` geladen.) Wie bei `wish`, der normalen `Tcl/Tk`-Shell, ist das erste Argument eine Skriptdatei und etwaige folgende Argumente werden dem Skript übergeben. Bestimmte Optionen könnten anstelle dessen auch von den Bibliotheken des `X Window System` verarbeitet werden. Wenn keine Skriptdatei angegeben wird, ist die Shell interaktiv.

Eine einfache `Tcl`-Shell mit PostgreSQL-Funktionen steht unter dem Namen `pgtclsh` *(Seite: 818)* zur Verfügung.

Siehe auch

pgtclsh *(Seite: 818)*, Abschnitt SET (Beschreibung von libpgtcl), tclsh, wish

psql

Name

psql – interaktives PostgreSQL-Terminal

Synopsis

psql [*option*...] [*dbname* [*benutzername*]]

Beschreibung

psql ist eine interaktive PostgreSQL-Terminalanwendung. Mit ihr können Sie interaktive Anfragen eingeben, sie an PostgreSQL schicken und die Anfrageergebnisse sehen. Alternativ kann die Eingabe aus einer Datei kommen. Zusätzlich gibt es eine Reihe von Metabefehlen und Shell-Skript-ähnliche Funktionalität, um das Schreiben von Skripten und das Automatisieren von verschiedenartigen Aufgaben zu erleichtern.

Optionen

-a
--echo-all

Gibt alle Zeilen auf dem Bildschirm aus, sobald Sie gelesen wurden. Das ist bei Skripts sinnvoller als im interaktiven Modus. Gleichbedeutend kann man die Variable ECHO auf all setzen.

-A
--no-align

Wählt den unausgerichteten Ausgabemodus. (Der Standardausgabemodus ist sonst der ausgerichtete.)

-c *befehl*
--command *befehl*

Gibt an, dass psql eine Befehlszeichenkette, *befehl*, ausführen und dann beenden soll. Das ist in Shell-Skripts nützlich.

befehl muss entweder komplett vom Server zu verarbeiten sein (das heißt, er verwendet keine Funktionen von psql) oder ein einzelner Metabefehl sein. Sie können also nicht SQL-Befehle und psql-Metabefehle vermischen. Um das zu erreichen, können Sie die Zeichenkette per Pipe an psql übergeben, etwa so: echo "\x \\ select * from foo;" | psql.

-d *dbname*
--dbname *dbname*

Gibt die Datenbank an, mit der verbunden werden soll. Das ist das Gleiche, als wenn man *dbname* als erstes Kommandozeilenargument nach den Optionen angibt.

-e
--echo-queries

Zeige alle Befehle, die an den Server gesendet werden. Gleichbedeutend kann man die Variable `ECHO` auf `queries` setzen.

`-E`
`--echo-hidden`

Zeige die Anfragen, die von `\d` und anderen Metabefehlen erzeugt werden. Das können Sie verwenden, wenn Sie ähnliche Funktionalität in Ihre eigene Programme einbauen wollen. Gleichbedeutend kann man die Variable `ECHO_HIDDEN` innerhalb von `psql` setzen.

`-f` *`dateiname`*
`--file` *`dateiname`*

Verwende die Datei *`dateiname`* als Quelle für Befehle, anstatt Befehle interaktiv einzulesen. Nachdem die Datei verarbeitet wurde, wird `psql` beendet. Das ist in vieler Hinsicht mit dem internen Befehl `\i` vergleichbar.

Wenn *`dateiname`* gleich – (Bindestrich) ist, wird die Standardeingabe gelesen.

Diese Option unterscheidet sich geringfügig von einer Konstruktion wie `psql <` *`dateiname`*. Generell machen beide, was Sie erwarten würden, aber mit `-f` erhalten Sie ein paar nette Zugaben, wie Fehlermeldungen mit Zeilennummern. Außerdem wird durch diese Option der Programmstart eventuell etwas beschleunigt. Andererseits ergibt die Variante mit den Shell-Operatoren zur Eingabeumleitung theoretisch genau die gleiche Ausgabe, als ob Sie alles von Hand eingegeben hätten.

`-F` *`trennzeichen`*
`--field-separator` *`trennzeichen`*

Verwendet *`trennzeichen`* als Feldtrennzeichen. Das ist gleichbedeutend mit den Befehlen `\pset fieldsep` oder `\f`.

`-h` *`hostname`*
`--host` *`hostname`*

Gibt den Hostnamen der Maschine an, auf der der Server läuft. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-H`
`--html`

Schaltet das HTML-Tabellenformat ein. Das ist gleichbedeutend mit dem Befehl `\pset format html` oder`\H`.

`-l`
`--list`

Listet alle verfügbaren Datenbanken und beendet dann. Andere Optionen außer Verbindungsparametern werden ignoriert. Dies ist ähnlich dem internen Befehl `\list`.

`-o` *`dateiname`*
`--output` *`dateiname`*

Schreibet alle Anfrageergebnisse in *`dateiname`*. Das ist gleichbedeutend mit dem Befehl `\o`.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet. Wenn diese Option nicht verwendet wird, wird der Wert der Umgebungsvariablen `PGPORT` verwendet oder, wenn diese nicht gesetzt ist, der beim Compilieren festgelegte Port, normalerweise 5432.

`-P` *`zuweisung`*
`--pset` *`zuweisung`*

Ermöglicht Ihnen, Ausgabeoptionen im Stil des Befehls `\pset` auf der Kommandozeile anzugeben. Beachten Sie, dass Sie hier zwischen Name und Wert ein Gleichheitszeichen anstatt eines Leerzeichens schreiben müssen. Um also das Ausgabeformat auf LaTeX zu setzen, können Sie schreiben `-P format=latex`.

`-q`
`--quiet`

Gibt an, dass `psql` seine Arbeit still erledigen soll. Normalerweise gibt es eine Willkommensmeldung und andere Informationen aus. Wenn diese Option verwendet wird, dann passiert nichts davon. Das ist zusammen mit der Option `-c` nützlich. Innerhalb von `psql` können Sie das Gleiche erreichen, indem Sie die Variable `QUIET` setzen.

`-R` *`trennzeichen`*
`--record-separator` *`trennzeichen`*

Verwendet *`trennzeichen`* als Trennzeichen für Datensätze. Das ist gleichbedeutend mit dem Befehl `\pset recordsep`.

`-s`
`--single-step`

Verwendet den Einzelschrittmodus. Das bedeutet, dass beim Benutzer nachgefragt wird, bevor ein Befehl an den Server geschickt wird, mit der Wahlmöglichkeit, die Ausführung abzubrechen. Verwenden Sie dies, um in Skripts nach Fehlern zu suchen.

`-S`
`--single-line`

Verwendet den Einzelzeilenmodus, in dem das Zeilenende einen SQL-Befehl abschließt, wie ein Semikolon.

Anmerkung

Dieser Modus ist für jene, die darauf bestehen, aber es wird nicht unbedingt empfohlen, ihn zu verwenden. Insbesondere wenn Sie SQL-Befehle und Metabefehle auf einer Zeile vermischen, ist die Ausführungsreihenfolge für den unerfahrenen Benutzer vielleicht nicht immer klar.

`-t`
`--tuples-only`

Schaltet das Ausgeben von Spaltennamen, Fußzeilen mit Zeilenzahlen usw. ab. Das entspricht vollständig dem Metabefehl `\t`.

`-T` *`tabellenattribute`*
`--table-attr` *`tabellenattribute`*

Erlaubt es Ihnen, Attribute anzugeben, die in das `table`-Tag in der HTML-Ausgabe gestellt werden sollen. Für Einzelheiten siehe `\pset`.

`-u`

Damit fragt `psql` nach Benutzername und Passwort, bevor es mit der Datenbank verbindet.

Diese Option wird nicht mehr empfohlen, weil Sie im Konzept Fehler aufweist. (Das Fragen nach einem Benutzernamen und das Fragen nach einem Passwort, weil es der Server erfordert, sind eigentlich zwei verschiedene Angelegenheiten.) Anstelle dessen sollten Sie die Optionen `-U` und `-W` verwenden.

`-U` *`benutzername`*
`--username` *`benutzername`*

Verbindet mit der Datenbank als der angegebene Benutzer anstatt als der voreingestellte Benutzer. (Sie müssen dazu natürlich die Rechte haben.)

`-v` *`zuweisung`*
`--set` *`zuweisung`*
`--variable` *`zuweisung`*

Führt eine Variablenzuweisung aus, wie der interne Befehl `\set`. Beachten Sie, dass Sie zwischen Namen und Wert, falls vorhanden, auf der Kommandozeile ein Gleichheitszeichen schreiben müssen. Um die Variable zu löschen, lassen Sie das Gleichheitszeichen weg. Um eine Variable ohne Wert nur zu setzen, schreiben Sie das Gleichheitszeichen, aber lassen Sie den Wert weg. Diese Zuweisungen geschehen in

einer frühen Phase des Startvorgangs, sodass für interne Zwecke reservierte Variablen später überschrieben werden könnten.

`-V`
`--version`

Zeigt die Version von `psql`.

`-W`
`--password`

Verlangt, dass `psql` nach einem Passwort fragen sollte, bevor es mit der Datenbank verbindet. Diese Einstellung bleibt für die gesamte Sitzung erhalten, selbst wenn Sie die Datenbankverbindung mit dem Metabefehl `\connect` ändern.

In der aktuellen Version fragt `psql` automatisch nach einem Passwort, wenn der Server Passwortauthentifizierung verlangt. Da diese Funktionalität aber auf ziemlich wackeligem Code beruht, kann die automatische Erkennung auf mysteriöse Art und Weise fehlschlagen, daher die Option um die Eingabe zu erzwingen. Wenn kein Passwort eingegeben wird und der Server Passwortauthentifizierung verlangt, wird der Verbindungsversuch scheitern.

`-x`
`--expanded`

Schaltet den erweiterten Tabellenformatiermodus an. Das ist gleichbedeutend mit dem Befehl `\x`.

`-X`,
`--no-psqlrc`

Liest die Startdatei `~/.psqlrc` nicht.

`-?`
`--help`

Zeigt eine Hilfe zu den Kommandozeilenargumenten von `psql`.

Lange Optionen gibt es nur auf einigen Plattformen.

Rückgabewert

`psql` gibt 0 an die Shell zurück, wenn es normal beendet wurde, 1, wenn ein eigener Fehler (Speicher aufgebraucht, Datei nicht gefunden) den Abbruch verursachte, 2, wenn die Verbindung zum Server abgebrochen wurde und die Sitzung nicht interaktiv war, und 3, wenn ein Fehler auftrat und die Variable `ON_ERROR_STOP` gesetzt war.

Verwendung

Mit einer Datenbank verbinden

`psql` ist eine normale PostgreSQL-Clientanwendung. Um mit einer Datenbank zu verbinden, müssen Sie den Namen der Zieldatenbank, den Hostnamen und die Portnummer des Servers und den Benutzernamen, unter dem Sie verbinden wollen, wissen. Diese Parameter können Sie `psql` mit Kommandozeilenoptionen mitteilen, nämlich `-d`, `-h`, `-p` bzw. `-U`. Wenn ein Argument gefunden wird, das zu keiner Option gehört, wird es als Datenbankname interpretiert (oder als Benutzername, wenn der Datenbankname auch angegeben ist). Nicht alle diese Optionen müssen angegeben werden, es gibt Vorgabewerte. Wenn Sie den Hostnamen weglassen, verbindet `psql` über eine Unix-Domain-Socket mit einem Server auf der lokalen Maschine. Die Standardportnummer wird bei der Compilierung festgelegt. Da der Datenbankserver die gleiche Standardportnummer hat, müssen Sie den Port meistens nicht angeben. Der vorgegebene Benutzername ist Ihr Unix-Benutzername, ebenso der vorgegebene Datenbankname. Beachten Sie, dass Sie nicht einfach mit jeder Datenbank unter jedem Benutzernamen verbinden können. Der Datenbankadministrator sollte Sie über Ihre Zugriffsrechte informiert haben. Um sich etwas Tippen zu ersparen, können

Sie einige Umgebungsvariablen auf die passenden Werte setzen: `PGDATABASE` (Datenbank), `PGHOST` (Hostname), `PGPORT` (Portnummer) und `PGUSER` (Benutzername).

Wenn die Verbindung aus irgendeinem Grund nicht aufgebaut werden konnte (z.B. fehlende Privilegien, kein Server läuft auf dem Zielhost usw.), gibt `psql` einen Fehler zurück und wird beendet.

SQL-Befehle eingeben

Im normalen Operationsmodus gibt `psql` eine Eingabeaufforderung aus, die den Namen der Datenbank, mit der `psql` gerade verbunden ist, gefolgt von den Zeichen `=>` enthält. Zum Beispiel:

```
$ psql testdb
Welcome to psql 7.3.3, the PostgreSQL interactive terminal.

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

testdb=>
```

An der Eingabeaufforderung kann der Benutzer SQL-Befehle eingeben. Gewöhnlich werden die eingegebenen Zeilen an den Server geschickt, wenn das Semikolon am Ende des Befehls erreicht wird. Das Zeilenende beendet einen Befehl nicht. Befehle können somit über mehrere Zeilen verteilt werden, um sie übersichtlicher zu gestalten. Nachdem der Befehl abgesendet wurde und wenn er ohne Fehler war, werden die Ergebnisse des Befehls auf dem Bildschirm angezeigt.

Immer wenn ein Befehl ausgeführt wird, schaut `psql` auch nach, ob eine durch `NOTIFY` *(Seite: 750)* (in Verbindung mit `LISTEN` *(Seite: 745)*) erzeugte asynchrone Benachrichtigung angekommen ist.

Metabefehle

Alles was Sie in `psql` eingeben, das mit einem Backslash anfängt (der nicht in Anführungszeichen steht), ist ein `psql`-Metabefehl, der von `psql` selbst verarbeitet wird. Diese Befehle machen `psql` interessant für die Administration oder für das Schreiben von Skripts. Metabefehle werden auch Backslash-Befehle genannt.

Das Format eines `psql`-Befehls ist der Backslash, direkt gefolgt von einem Befehlswort, danach eventuelle Argumente. Die Argumente werden vom Befehlswort und voneinander durch ein oder mehrere Leerzeichen getrennt.

Um Leerzeichen, Tabs oder Zeilenumbrüche in ein Argument zu schreiben, können Sie halbe Anführungszeichen (Apostrophe) um das Argument setzen. Um ein Apostroph in so ein Argument zu schreiben, setzen Sie davor einen Backslash. Außerdem werden bei jedem Text, der in halben Anführungszeichen steht, die folgenden C-ähnliche Fluchtfolgen ersetzt: `\n` (Newline-Zeichen), `\t` (Tab), `\`*`ziffern`*, `\0`*`ziffern`* und `\0x`*`ziffern`* (das Zeichen mit dem angegebenen dezimalen, oktalen bzw. hexadezimalen Code).

Wenn ein Argument, das nicht in halben Anführungszeichen steht, mit einem Doppelpunkt (`:`) beginnt, wird es als `psql`-Variablen interpretiert und der Wert der Variable wird als Argument genommen.

Argumente, die zwischen Rückwärtsapostrophen (`` ` ``) stehen, werden als Befehl an die Shell übergeben. Die Ausgabe dieses Befehls (mit dem abschließenden Newline-Zeichen, falls vorhanden, entfernt) wird als Wert des Arguments genommen. Die oben aufgeführten Fluchtfolgen gelten auch in diesem Fall.

Einige Befehle haben einen SQL-Bezeichner (z.B. einen Tabellennamen) als Argument. Diese Argumente folgen den Syntaxregeln von SQL: Buchstaben ohne Anführungszeichen werden in Kleinbuchstaben

umgewandelt und mit Anführungszeichen bleiben die Buchstaben erhalten und die Bezeichner können Leerzeichen enthalten. Innerhalb von Anführungszeichen ergeben doppelte Anführungszeichen ein einziges im daraus resultierenden Namen. Aus `FOO"BAR"BAZ` wird zum Beispiel `fooBARbaz` und aus `"Ein komischer"" Name"` wird `Ein komischer" Name`.

Das Verarbeiten der Argumente ist zu Ende, wenn ein weiterer Backslash (der nicht in Anführungszeichen steht) gefunden wird. Dort fängt ein neuer Metabefehl an. Wenn die spezielle Zeichenfolge `\\` (zwei Backslashes) gefunden wird, ist das das Ende der Argumente, und was danach folgt, sind wieder normale SQL-Befehle. Dadurch können SQL-Befehle und `psql`-Befehle frei auf einer Zeile vermischt werden. Aber die Argumente eines Metabefehls können auf keinen Fall über das Ende einer Zeile hinausgehen.

Die folgenden Metabefehle stehen zur Verfügung:

`\a`

Wenn das aktuelle Tabellenausgabeformat das nicht ausgerichtete ist, wechselt es auf das ausgerichtete. Ansonsten wird es auf das nicht ausgerichtete gesetzt. Dieser Befehl wird wegen der Kompatibilität mit früheren Versionen behalten. Eine allgemeingültigere Lösung finden Sie unter `\pset`.

`\cd` [*`verzeichnis`*]

Ändert das aktuelle Arbeitsverzeichnis in *`verzeichnis`*. Ohne Argument wird in das Home-Verzeichnis des aktuellen Benutzers gewechselt.

Tipp

Um Ihr aktuelles Arbeitsverzeichnis auszugeben, verwenden Sie `\!pwd`.

`\C` [*`titel`*]

Setzt den Titel einer Tabelle, die als Ergebnis einer Anfrage ausgeben wird, oder löscht einen solchen Titel. Dieser Befehl ist gleichbedeutend mit `\pset title` *`title`*. (Der Name dieses Befehls kommt von *`caption`*, weil er früher nur für HTML-Tabellen galt.)

`\connect` (oder `\c`) [*`dbname`* [*`benutzername`*]]

Baut eine Verbindung mit einer neuen Datenbank und/oder unter einem anderen Benutzernamen auf. Die bisherige Verbindung wird geschlossen. Wenn *`dbname`* gleich – ist, wird der aktuelle Datenbankname verwendet.

Wenn *`benutzername`* weggelassen wird, wird der aktuelle Benutzername verwendet.

Als Sonderregel gilt, wenn `\connect` ohne Argumente aufgerufen wird, dann verbindet es mit der Standarddatenbank als der Standardbenutzer (genauso, als wenn Sie `psql` ohne Argumente aufgerufen hätten).

Wenn der Verbindungsversuch scheitert (falscher Benutzername, Zugriff verweigert usw.), wird die vorhergehende Verbindung beibehalten, aber nur wenn `psql` im interaktiven Modus ist. Bei der Ausführung eines nicht interaktiven Skripts wird die Verarbeitung sofort mit einem Fehler abgebrochen. Diese Unterscheidung wurde gemacht, damit Benutzer nicht für Tippfehler bestraft werden, aber andererseits als Sicherheitsvorkehrung, damit Skripts nicht aus Versehen die falsche Datenbank bearbeiten.

`\copy` *`tabelle`* `[ (` *`spaltenliste`* `) ] { from | to }` *`dateiname`* `| stdin | stdout [ with ] [ oids ] [ delimiter [as] '`*`zeichen`*`' ] [ null [as] '`*`zeichenkette`*`' ]`

Führt eine clientseitige Kopieraktion durch. Diese Operation führt den SQL-Befehl `COPY` *(Seite: 646)* aus, aber anstatt dass der Server die angegebene Datei liest oder schreibt, liest oder schreibt `psql` die Datei und leitet die Daten vom lokalen System an den Server. Das bedeutet, dass die verfügbaren Dateien und die Zugriffsrechte sich auf das lokale System beziehen und nicht auf den Server, und damit keine SQL-Superuser-Privilegien erforderlich sind.

Die Syntax dieses Befehls ist ähnlich dem SQL-Befehl `COPY`. (Siehe die Beschreibung dort.) Beachten Sie, dass deswegen besondere Syntaxregeln für den Befehl `\copy` gelten. Insbesondere können keine Variablen eingesetzt und keine Backslash-Fluchtfolgen verwendet werden.

Tipp

Diese Operation ist nicht so effizient wie `COPY` auf dem Server, weil alle Daten durch die Client/Server-Verbindung geschickt werden müssen. Für große Datenmengen könnte die andere Methode besser sein.

Anmerkung

Beachten Sie die unterschiedliche Interpretation von `stdin` und `stdout` bei client- und serverseitigen Kopieraktionen: Auf der Clientseite beziehen diese sich immer auf den Standardeingabe bzw. -ausgabe von `psql`. Auf der Serverseite kommt `stdin` immer daher, wo das `COPY` selbst herkam (zum Beispiel ein mit der Option `-f` ausgeführtes Skript), und `stdout` verweist auf das Anfrageausgabeziel (mit dem Metabefehl `\o` eingerichtet).

`\copyright`

Zeigt Urheberrechtsinformationen über `PostgreSQL`.

`\d` [*`muster`*]

Zeigt für jede Relation (Tabelle, Sicht, Index oder Sequenz), die mit *`muster`* übereinstimmt, alle Spalten, deren Typen und besondere Attribute wie `NOT NULL` oder Vorgabewerte, falls vorhanden. Zugehörige Indexe, Constraints, Regeln und Trigger werden auch gezeigt, ebenso die Sichtdefinition, falls die Relation eine Sicht ist. (Das Format des Musters wird weiter unten beschrieben.)

Die Befehlsform `\d+` ist identisch, außer dass eventuell zu den Tabellenspalten gehörende Kommentare mit angezeigt werden.

Anmerkung

Wenn `\d` ohne das Argument *`muster`* verwendet wird, ist es gleichbedeutend mit `\dtvs`, was eine Liste aller Tabellen, Sichten und Sequenzen zeigt. Das ist einfach eine nützliche Abkürzung.

`\da` [*`muster`*]

Listet alle verfügbaren Aggregatfunktionen, zusammen mit den Argumentdatentypen. Wenn *`muster`* angegeben wurde, werden nur die übereinstimmenden Aggregatfunktionen angezeigt.

`\dd` [*`muster`*]

Zeigt die Beschreibung aller Objekte, die mit *`muster`* übereinstimmen, oder aller sichtbaren Objekte, wenn kein Argument angegeben wurde. In jedem Fall werden nur Objekte, die eine Beschreibung haben, gelistet. („Objekte“ sind Aggregatfunktionen, Funktionen, Operatoren, Typen, Relationen (Tabellen, Sichten, Indexe, Sequenzen, Large Objects), Regeln und Trigger.) Zum Beispiel:

```
=> \dd version
                    Object descriptions
   Schema   |  Name   |  Object  |        Description
------------+---------+----------+---------------------------
 pg_catalog | version | function | PostgreSQL version string
(1 row)
```

Beschreibungen für Objekte können mit dem SQL-Befehl `COMMENT` erzeugt werden.

`\dD` [*`muster`*]

Listet alle verfügbaren Domänen. Wenn *muster* angegeben wurde, werden nur die übereinstimmenden Domänen angezeigt.

`\df [ muster ]`

Listet alle verfügbaren Funktionen, zusammen mit den Argument- und Rückgabetypen. Wenn *muster* angegeben wurde, werden nur die übereinstimmenden Funktionen angezeigt. Wenn die Form `\df+` verwendet wird, werden zusätzliche Informationen über jede Funktion angezeigt, zum Beispiel die Sprache und die Beschreibung.

Anmerkung

Um die Übersichtlichkeit des Ergebnisses zu bewahren, zeigt `\df` keine Eingabe- und Ausgabefunktionen an. Das heißt konkret, dass Funktionen, die den Typ `cstring` als Argument- oder Rückgabetyp haben, ignoriert werden.

`\distvS [ muster ]`

Das ist nicht der wirkliche Name des Befehls: Die Buchstaben i, s, t, v, S stehen für Index, Sequenz, Tabelle, Sicht (englisch *view*) bzw. Systemtabelle. Sie können einen oder mehrere dieser Buchstaben angeben, in beliebiger Reihenfolge, um eine Liste aller entsprechenden Objekte zu erhalten. Der Buchstabe S beschränkt die Ausgabe auf Systemobjekte; ohne S werden keine Systemobjekte ausgegeben. Wenn + an den Befehlsnamen angehängt wird, wird jedes Objekt mit seiner zugehörigen Beschreibung, falls vorhanden, gelistet.

Wenn *muster* angegeben wurde, werden nur Objekte aufgelistet, deren Name mit dem Muster übereinstimmt.

`\dl`

Dieser Befehl ist gleichbedeutend mit `\lo_list`, welcher eine Liste der Large Objects anzeigt.

`\do [ muster ]`

Listet alle verfügbaren Operatoren, mit den Operanden- und Ergebnistypen. Wenn *muster* angegeben wurde, werden nur Operatoren aufgelistet, deren Namen mit dem Muster übereinstimmen.

`\dp` [*muster*]

Zeigt eine Liste aller verfügbaren Tabellen mit den zugehörigen Zugriffsprivilegien. Wenn *muster* angegeben wurde, werden nur Tabellen aufgelistet, deren Name mit dem Muster übereinstimmt.

Zugriffsprivilegien werden mit den Befehlen `GRANT` *(Seite: 739)* und `REVOKE` *(Seite: 756)* gesetzt. Weitere Informationen finden Sie bei `GRANT` *(Seite: 739)*.

`\dT [ muster ]`

Listet alle Datentypen bzw. nur die, die mit *muster* übereinstimmen. Die Befehlsform `\dT+` zeigt zusätzliche Informationen an.

`\du [ muster ]`

Listet alle Datenbankbenutzer bzw. nur die, die mit *muster* übereinstimmen.

`\edit` (oder `\e`) [*dateiname*]

Wenn *dateiname* angegeben wurde, wird die Datei mit einem Editor bearbeitet; nachdem der Editor verlassen wird, wird der Inhalt der Datei in den Eingabepuffer zurückkopiert. Wenn kein Argument angegeben wurde, wird der aktuelle Eingabepuffer in eine temporäre Datei kopiert, die auf die gleiche Art bearbeitet wird.

Der neue Eingabepuffer wird entsprechend den normalen Regeln von `psql` neu eingelesen, wobei der gesamte Puffer als einzelne Zeile behandelt wird. (Sie können also auf diese Art keine Skripts schreiben.) Das heißt, wenn der Eingabepuffer mit einem Semikolon endet (oder um ganz genau zu sein eins enthält), wird er sofort ausgeführt. Ansonsten wartet die Eingabe einfach weiter im Puffer.

Tipp

`psql` sucht in den Umgebungsvariablen `PSQL_EDITOR`, `EDITOR` und `VISUAL` (in dieser Reihenfolge) nach einem Editor. Wenn keine dieser Variablen gesetzt ist, wird `/bin/vi` verwendet.

`\echo` *`text`* [...]

Gibt die Argumente auf der Standardausgabe aus, getrennt durch ein Leerzeichen und abgeschlossen durch ein Zeilenende. Das kann nützlich sein, um Informationen in die Ausgabe von Skripts einzustreuen. Zum Beispiel:

```
=> \echo `date`
Sam Mär 22 17:14:36 CET 2003
```

Wenn das erste Argument ein `-n` ist, das nicht in Anführungszeichen steht, wird am Ende keine neue Zeile begonnen.

Tipp

Wenn Sie den Befehl `\o` verwenden, um die Ergebnisse Ihrer Anfragen umzuleiten, finden Sie vielleicht anstelle dieses Befehls `\qecho` nützlich.

`\encoding` [*`kodierung`*]

Setzt die Zeichensatzkodierung des Clients. Ohne Argument wird die aktuelle Kodierung angezeigt.

`\f` [*`zeichenkette`*]

Setzt das Feldtrennzeichen für das unausgerichtete Tabellenausgabeformat. Die Voreinstellung ist der senkrechte Strich (|). Weitere Möglichkeiten, um das Ausgabeformat zu ändern, werden unter `\pset` beschrieben.

`\g` [{ *`dateiname`* | `|`*`befehl`* }]

Sendet den aktuellen Eingabepuffer an den Server und speichert wahlweise die Ausgabe in *`dateiname`* oder schickt sie per Pipe an eine Unix-Shell, die *`befehl`* ausführt. Ein einfaches `\g` ist im Prinzip gleichbedeutend mit einem Semikolon. Ein `\g` mit Argument ist eine Alternative zu dem Befehl `\o`, wenn die Umleitung nur das eine Mal erfolgen soll.

`\help` (oder `\h`) [*`befehl`*]

Gibt Syntaxhilfe für den angegebenen SQL-Befehl. Wenn *`befehl`* nicht angegeben wurde, gibt `psql` eine Liste aller Befehle aus, für die Syntaxhilfe zur Verfügung steht. Wenn *`befehl`* ein Stern (*) ist, wird die Syntaxhilfe für alle SQL-Befehle gezeigt.

Anmerkung

Um das Eingeben zu erleichtern, können Befehle, die aus mehreren Wörtern bestehen, ohne Anführungszeichen eingegeben werden. Es ist also in Ordnung, **`\help alter table`** einzugeben.

`\H`

Schaltet das HTML-Tabellenausgabeformat an. Wenn das HTML-Format bereits an ist, wird zurück in das normale ausgerichtete Textformat geschaltet. Weitere Ausgabeformatoptionen können Sie mit `\pset` setzen. Dieser Befehl ist eine Abkürzung davon, die auch wegen der Kompatibilität mit früheren Versionen beibehalten wird.

`\i` *`dateiname`*

Liest die Eingabe aus der Datei *dateiname* und verarbeitet sie, als ob sie von der Tastatur eingegeben worden wäre.

Anmerkung

Wenn Sie die gelesenen Zeilen auf dem Bildschirm ausgegeben haben wollen, müssen Sie die Variable ECHO auf all setzen.

\l (oder \list)

Listet alle Datenbanken auf dem Server mit Eigentümern und der Zeichensatzkodierung. Wenn Sie ein + an den Befehlsnamen anhängen, werden außerdem die Beschreibungen der Datenbanken gezeigt.

\lo_export *loid dateiname*

Liest das Large Object mit der OID *loid* aus der Datenbank und schreibt es in die Datei *dateiname*. Beachten Sie, dass sich das etwas von der Serverfunktion lo_export unterscheidet, welche auf das Dateisystem und mit den Benutzerrechten des Servers zugreift.

Tipp

Verwenden Sie \lo_list, um die OID eines Large Object herauszufinden.

Anmerkung

Die Beschreibung der Variablen LO_TRANSACTION enthält wichtige Informationen, die auf alle Large-Object-Operationen Auswirkung haben.

\lo_import *dateiname* [*kommentar*]

Speichert die Datei in ein PostgreSQL-Large-Object. Wahlweise kann dem Objekt ein Kommentar zugeordnet werden. Beispiel:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'ein Bild von mir'
lo_import 152801
```

Die Antwort zeigt an, dass das Large Object die Objekt-ID 152801 erhalten hat, was man sich vielleicht merken sollte, wenn man auf das Objekt jemals wieder zugreifen will. Aus diesem Grund wird empfohlen, jedem Objekt einen Kommentar zuzuordnen. Die Kommentare können dann mit dem Befehl \lo_list angesehen werden.

Beachten Sie, dass sich dieser Befehl von der Serverfunktion lo_import unterscheidet, weil er auf das lokale Dateisystem unter dem lokalen Benutzerzugang zugreift, anstatt auf das Serverbetriebssystem unter dem dortigen Benutzerzugang.

Anmerkung

Die Beschreibung der Variablen LO_TRANSACTION enthält wichtige Informationen, die auf alle Large-Object-Operationen Auswirkung haben.

\lo_list

Zeigt eine Liste aller Large Objects in einer PostgreSQL-Datenbank, zusammen mit eventuell zugehörigen Kommentaren.

\lo_unlink *loid*

Löscht das Large Object mit der OID *loid* aus der Datenbank.

Tipp

Verwenden Sie `\lo_list`, um die OID eines Large Objects herauszufinden.

Anmerkung

Die Beschreibung der Variable `LO_TRANSACTION` enthält wichtige Informationen, die auf alle Large-Object-Operationen Auswirkung haben.

`\o` [{*`dateiname`* | `|`*`befehl`*}]

Speichert zukünftige Anfrageergebnisse in der Datei *`dateiname`* oder schickt sie per Pipe an eine Unix-Shell, die *`befehl`* ausführt. Wenn keine Argumente angegeben sind, wird die Anfrageausgabe auf die Standardausgabe zurückgesetzt.

„Anfrageergebnisse“ sind alle Tabellen, Antworten auf Befehle und Hinweismeldungen vom Datenbankserver sowie die Ausgaben der Metabefehle, die die Datenbank abfragen (wie zum Beispiel `\d`), aber keine Fehlermeldungen.

Tipp

Um Textausgaben in die Anfrageergebnisse einzustreuen, verwenden Sie `\qecho`.

`\p`

Gibt den aktuellen Eingabepuffer auf der Standardausgabe aus.

`\pset` *`parameter`* [*`wert`*]

Dieser Befehl setzt Ausgabebeoptionen für die Anfrageergebnistabellen. *`parameter`* bestimmt die zu setzende Option. Die Bedeutung von *`wert`* hängt von der jeweiligen Option ab.

Die einstellbaren Ausgabeoptionen sind:

`format`

Setzt das Ausgabeformat. Möglichkeiten sind: `unaligned` (nicht ausgerichtet), `aligned` (ausgerichtet), `html` oder `latex`. Eindeutige Abkürzungen sind erlaubt. (Das heißt, ein Buchstabe ist genug.)

Im „nicht ausgerichteten“ Format werden alle Spalten einer Zeile getrennt durch das aktuelle Feldtrennzeichen auf eine Bildschirmzeile geschrieben. Dieses Ausgabeformat ist dafür gedacht, dass andere Programme es einfach einlesen können (zum Beispiel mit Tabulatorzeichen oder Kommas als Trennzeichen). „Ausgerichtet“ ist das normale Standardtextformat, das zum Lesen durch menschliche Anwender gedacht ist. Die „HTML“- und „LaTeX“-Modi geben Tabellen aus, die in Dokumente der entsprechenden Markup-Sprachen eingebettet werden können. Sie ergeben selbst keine vollständigen Dokumente! (Das mag bei HTML nicht so dramatisch sein, aber in LaTeX müssen Sie ein komplettes Dokument darum herum setzen.)

`border`

Das zweite Argument muss eine Zahl sein. Generell gilt, je höher die Zahl, desto mehr Rahmen und Linien wird die Tabelle haben, aber Genaueres hängt vom konkreten Format ab. Im HTML-Modus wird dies direkt in das Attribut `border=...` übertragen, in den anderen Formaten sind nur die Werte 0 (kein Rahmen), 1 (interne Trennlinien) und 2 (Tabellenrahmen) sinnvoll.

`expanded` (oder `x`)

Schaltet zwischen dem normalen und dem erweiterten Format um. Im erweiterten Format hat die Ausgabe zwei Spalten mit dem Spaltennamen links und den Daten rechts. Dieser Modus ist nützlich, wenn die Daten im normalen, „horizontalen“ Modus nicht auf den Bildschirm passen würden.

Der erweiterte Modus wird von allen vier Ausgabeformaten unterstützt.

`null`

Das zweite Argument ist eine Zeichenkette, die ausgegeben werden soll, wenn eine Spalte den NULL-Wert hat. In der Voreinstellung wird nichts ausgegeben, was aber leicht mit einer leeren Zeichenkette verwechselt werden kann. Daher könnte man zum Beispiel `\pset null '(null)'` wählen.

`fieldsep`

Gibt das Feldtrennzeichen für den nicht ausgerichteten Ausgabemodus an. Damit kann man zum Beispiel durch Kommas oder Tabulatorzeichen getrennte Ausgaben erzeugen, was von anderen Programmen leichter eingelesen werden kann. Um einen Tabulator als Trennzeichen zu setzen, geben Sie `\pset fieldsep '\t'` ein. Das Standardfeldtrennzeichen ist `'|'` (ein senkrechter Strich).

`footer`

Schaltet die Ausgabe der Standardfußzeile `(x Zeilen)` bzw. `(x rows)` ein oder aus.

`recordsep`

Gibt das Satztrennzeichen (Zeilentrennzeichen) für den nicht ausgerichteten Ausgabemodus an. Die Voreinstellung ist das Newline-Zeichen.

`tuples_only` (oder `t`)

Schaltet um zwischen der Nur-Tupel- und der vollen Ausgabe. Die volle Ausgabe zeigt zusätzliche Informationen wie Spaltenköpfe, Titel und diverse Fußzeilen. Im Nur-Tupel-Modus werden nur die eigentlichen Tabellendaten gezeigt.

`title` [*`text`*]

Setzt den Titel für im Folgenden ausgegebenen Tabellen. Damit können Sie der Ausgabe sinnvolle Beschreibungen geben. Wenn kein Argument angegeben ist, dann wird der Titel gelöscht.

`tableattr` (oder `T`) [*`text`*]

Damit können Sie beliebige Attribute in das `table`-Tag in der HTML-Ausgabe stellen. Das könnte man zum Beispiel für `cellpadding` oder `bgcolor` verwenden. Beachten Sie aber, dass `border` hier eher nicht angegeben werden sollte, da das schon von `\pset border` erledigt wird.

`pager`

Schaltet die Verwendung eines Pagers für Anfrageergebnisse und Hilfeanzeigen in `psql` an oder aus. Wenn die Umgebungsvariable `PAGER` gesetzt ist, wird die Ausgabe per Pipe an das angegebene Programm geschickt. Ansonsten wird ein plattformabhängiges Standardprogramm (zum Beispiel `more`) verwendet.

Auf jeden Fall verwendet `psql` den Pager nur, wenn es angebracht erscheint. Das heißt unter anderem, dass die Ausgabe auf ein Terminal geht und die Tabelle normalerweise nicht auf den Bildschirm passen würde. Wegen der modularen Aufteilung der Ausgaberoutinen ist es nicht immer möglich, die genaue Anzahl der ausgegebenen Zeilen vorherzusagen. Daher kann es sein, dass `psql` bei der Verwendung des Pagers nicht besonders genau erscheint.

Ein paar Beispiele, wie diese unterschiedlichen Formate aussehen, gibt es im Abschnitt `Beispiele` *(Seite: 838)*.

Tipp

Es gibt diverse Abkürzungen für den Befehl `\pset`. Siehe unter `\a`, `\C`, `\H`, `\t`, `\T` und `\x`.

Anmerkung

Es ist ein Fehler, `\pset` ohne Argumente aufzurufen. Zukünftig könnte damit der aktuelle Zustand aller Ausgabeoptionen angezeigt werden.

`\q`

Verlässt das psql-Programm.

\qecho *text* [...]

Dieser Befehl ist identisch mit \echo, außer dass die Ausgabe an die mit \o gesetzte Anfrageergebnisausgabe geht.

\r

Löscht den Eingabepuffer.

\s [*dateiname*]

Gibt die Befehlszeilengeschichte aus oder speichert sie in *dateiname*. Wenn *dateiname* ausgelassen wird, wird die Geschichte auf der Standardausgabe angezeigt. Diese Option ist nur verfügbar, wenn psql für die GNU-History-Bibliothek konfiguriert wurde.

Anmerkung

In der aktuellen Version ist es nicht mehr notwendig, die Befehlszeilengeschichte zu speichern, da das am Programmende automatisch geschieht. Die Befehlsgeschichte wird auch beim Start von psql automatisch wieder geladen.

\set [*name* [*wert* [...]]]

Setzt die interne Variable *name* auf *wert* oder, wenn mehr als ein Wert angegeben ist, auf die Verknüpfung aller Werte. Wenn kein zweites Argument angegeben ist, wird die Variable einfach gesetzt, ohne Wert. Um eine Variable wieder zu löschen, verwenden Sie den Befehl \unset.

Gültige Variablennamen können Zeichen, Ziffern und Unterstriche enthalten. Einzelheiten finden Sie im Abschnitt Variablen *(Seite: 832)* unten.

Obwohl es Ihnen frei steht, jede Variable auf jeden beliebigen Wert zu setzen, werden einige Variablen von psql besonders behandelt. Diese werden im Abschnitt über Variablen beschrieben.

Anmerkung

Dieser Befehl hat nichts mit dem SQL-Befehl SET *(Seite: 771)* zu tun.

\t

Schaltet die Ausgabe von Spaltenköpfen und Fußzeilen mit der Zeilenzahl an oder aus. Dieser Befehl ist gleichbedeutend mit \pset tuples_only und wird der Bequemlichkeit halber angeboten.

\T *tabellenattribute*

Damit können Sie Attribute in das table-Tag im HTML-Tabellenausgabemodus stellen. Dieser Befehl ist gleichbedeutend mit \pset tableattr *tabellenattribute*.

\timing

Schaltet die Anzeige, wie lange ein SQL-Befehl gedauert hat (in Millisekunden) an oder aus.

\w {*dateiname* | |*befehl*}

Schreibt den gegenwärtigen Eingabepuffer in die Datei *dateiname* oder schickt ihn per Pipe an eine Unix-Shell, die *befehl* ausführt.

\x

Schaltet den erweiterten Tabellenformatiermodus ein oder aus. Dieser Befehl ist gleichbedeutend mit \pset expanded.

\z [*muster*]

Zeigt eine Liste aller verfügbaren Tabellen mit den zugehörigen Zugriffsprivilegien. Wenn *muster* angegeben wurde, werden nur Tabellen aufgelistet, deren Name mit dem Muster übereinstimmt.

Zugriffsprivilegien werden mit den Befehlen GRANT *(Seite: 739)* und REVOKE *(Seite: 756)* gesetzt. Weitere Informationen finden Sie bei GRANT *(Seite: 739)*.

Dieser Befehl ist gleichbedeutend mit \dp.

\! [*befehl*]

Startet eine Unix-Shell oder führt *befehl* in einer Unix-Shell aus. Die Argumente werden nicht weiter interpretiert; die Shell sieht sie so, wie sie sind.

\?

Zeigt Hilfe über die Metabefehle an.

Die verschiedenen \d-Befehle akzeptieren einen *muster*-Parameter, der die Namen der anzuzeigenden Objekte angibt: * steht für eine beliebige Folge von Zeichen und ? steht für ein einzelnes beliebiges Zeichen. (Diese Schreibweise ist mit den Dateinamenmustern der Unix-Shells vergleichbar.) Fortgeschrittene Anwender können auch reguläre Ausdrücke wie Zeichenklassen verwenden, zum Beispiel [0-9] für eine beliebige Ziffer. Um eines dieser Musterzeichen nicht als Sonderzeichen zu behandeln, setzen Sie es in Anführungszeichen.

Ein Muster, das einen Punkt, der nicht in Anführungszeichen steht, enthält, wird als Schemanamenmuster gefolgt von einem Objektnamenmuster interpretiert. Zum Beispiel zeigt \dt foo*.bar* alle Tabellen, deren Name mit foo anfängt, in Schemas, deren Name mit bar anfängt. Wenn kein Punkt in der Angabe vorkommt, dann findet das Muster nur Objekte, die im aktuellen Schemasuchpfad sichtbar sind.

Wenn der Parameter *muster* ganz weggelassen wird, dann zeigen die \d-Befehle alle Objekte, die im aktuellen Schemasuchpfad sichtbar sind. Um alle Objekte in der Datenbank anzuzeigen, müssen Sie das Muster *.* verwenden.

Fortgeschrittene Funktionen

Variablen

psql bietet die Fähigkeit, ähnlich wie in Unix-Shells Variablen in Befehle einzusetzen. Variablen sind einfache Paare aus Name und Wert, wobei der Wert eine beliebig lange Zeichenkette sein kann. Um eine Variable zu setzen, verwenden Sie den psql-Metabefehl \set:

```
testdb=> \set foo bar
```

Dieser Befehl setzt die Variable foo auf den Wert bar. Um den Inhalt einer Variablen abzurufen, schreiben Sie vor den Namen einen Doppelpunkt und verwenden Sie ihn als Argument eines Metabefehls:

```
testdb=> \echo :foo
bar
```

Anmerkung

Für die Argumente von \set gelten dieselben Ersetzungsregeln wie bei anderen Befehlen. Sie können somit interessante Referenzen wie zum Beispiel \set :foo 'wert' konstruieren, was vergleichbar mit „Soft Links“ oder „variablen Variablen“ aus Perl bzw. PHP wäre. Leider (oder glücklicherweise?) kann man mit diesen Konstruktionen nichts Sinnvolles anfangen. Andererseits ist \set bar :foo eine vollkommen zulässige Methode, um eine Variable zu kopieren.

Wenn Sie \set ohne zweites Argument aufrufen, wird die Variable einfach gesetzt, hat aber keinen Wert. Um eine Variable zu löschen, verwenden Sie den Befehl \unset.

Die Namen der internen psql-Variablen können aus Buchstaben, Zahlen und Unterstrichen bestehen, in beliebiger Reihenfolge und Anzahl. Eine Reihe normaler Variablen werden von psql besonders behandelt. Sie stehen für bestimmte Optionseinstellungen, die geändert werden können, indem der Wert der Variablen verändert wird, oder sie enthalten Statusinformationen. Obwohl Sie diese Variablen auch für andere Zwecke verwenden können, ist das nicht zu empfehlen, weil das Verhalten des Programms dann ziemlich schnell ziemlich merkwürdig werden könnte. Alle besonderen Variablen haben Namen, die nur aus Großbuchstaben (und möglicherweise Zahlen und Unterstrichen) bestehen. Um Kompatibilität auch in der Zukunft zu gewährleisten, sollten Sie solche Variablen selbst nicht verwenden. Eine Liste aller intern verwendeten Variablen folgt.

DBNAME

Der Name der Datenbank, mit der Sie gegenwärtig verbunden sind. Diese Variable wird jedes Mal gesetzt, wenn Sie mit einer Datenbank verbinden (einschließlich beim Programmstart), kann aber gelöscht werden.

ECHO

Wenn Sie diese Variable auf all setzen, werden alle eingegebenen oder aus einem Skript gelesenen Zeilen auf der Standardausgabe ausgegeben, bevor Sie verarbeitet oder ausgeführt werden. Um dieses Verhalten beim Programmstart auszuwählen, verwenden Sie die Option -a. Wenn die Variable auf queries gesetzt ist, gibt psql nur die Anfragen aus, die an den Server gesendet werden. Die Kommandozeilenoption dafür ist -e.

ECHO_HIDDEN

Wenn diese Variable gesetzt ist und ein Metabefehl eine Anfrage an die Datenbank sendet, wird die Anfrage zuerst ausgegeben. Dadurch können Sie die Interna von PostgreSQL studieren und ähnliche Funktionalität in Ihre eigenen Programme einbauen. Wenn Sie die Variable auf den Wert noexec setzen, werden die Anfragen nur gezeigt, aber nicht tatsächlich an den Server gesendet und ausgeführt.

ENCODING

Die aktuelle Client-Zeichensatzkodierung.

HISTCONTROL

Wenn diese Variable auf ignorespace gesetzt ist, werden Zeilen, die mit einem Leerzeichen anfangen, nicht in die Geschichtsliste eingetragen. Wenn die Variable den Wert ignoredups hat, werden Zeilen, die mit der vorangegangenen Geschichtszeile übereinstimmen, nicht eingetragen. Der Wert ignoreboth verbindet beide Optionen. Wenn die Variable nicht gesetzt ist oder auf einen anderen Wert als die oben angegebenen gesetzt ist, werden alle im interaktiven Modus eingegebenen Zeilen in der Befehlsgeschichte gespeichert.

Anmerkung

Die Funktion wurde schamlos von Bash abgekupfert.

HISTSIZE

Die Anzahl der Befehle, die in der Befehlsgeschichte gespeichert werden soll. Die Voreinstellung ist 500.

Anmerkung

Die Funktion wurde schamlos von Bash abgekupfert.

HOST

Der Name der Datenbankserverhosts, mit dem Sie gegenwärtig verbunden sind. Diese Variable wird jedes Mal gesetzt, wenn Sie mit einer Datenbank verbinden (einschließlich beim Programmstart), kann aber gelöscht werden.

IGNOREEOF

Wenn die Variable nicht gesetzt ist und ein Dateiendezeichen (EOF, normalerweise **Strg**+**D**) an eine interaktive Sitzung von `psql` gesendet wird, wird das Programm beendet. Wenn die Variable auf einen numerischen Wert gesetzt wird, müssen so viele Dateiendezeichen gesendet werden, bis das Programm beendet wird. Wenn die Variable gesetzt ist, aber keinen Wert hat, dann ist die Voreinstellung 10.

Anmerkung

Die Funktion wurde schamlos von `Bash` abgekupfert.

`LASTOID`

Der Wert der letzten betroffenen OID, zurückgegeben von einem `INSERT`- oder `lo_insert`-Befehl. Die Gültigkeit dieser Variablen ist nur garantiert, bis das Ergebnis des nächsten SQL-Befehls ausgegeben worden ist.

`LO_TRANSACTION`

Wenn Sie die PostgreSQL-Large-Object-Schnittstelle verwenden, um Daten, die nicht in eine Zeile passen, gesondert zu speichern, dann müssen alle Operationen in einem Transaktionsblock ausgeführt werden. (Weitere Informationen dazu finden Sie in der Anleitung zur Large-Object-Schnittstelle.) Da `psql` nicht feststellen kann, ob bereits eine Transaktion aktiv ist, wenn Sie einen der internen Befehle (`\lo_export`, `\lo_import`, `\lo_unlink`) aufrufen, muss es irgendetwas tun. Das könnte sein, eine eventuell aktive Transaktion zurückzurollen, eine solche Transaktion abzuschließen oder gar nichts zu tun. Im letzteren Fall müssen Sie selbst einen `BEGIN`/`COMMIT`-Block erzeugen oder die Ergebnisse werden nicht vorhersehbar sein. (In der Regel würde aber die gewünschte Aktion auf jeden Fall nicht ausgeführt werden.)

Um das gewünschte Verhalten auszuwählen, setzen Sie diese Variable auf `rollback` (Transaktion zurückrollen), `commit` (Transaktion abschließen) oder `nothing` (nichts tun). Die Voreinstellung ist, die Transaktion zurückzurollen. Wenn Sie nur ein oder ein paar Large Objects laden wollen, dann ist das in Ordnung. Wenn Sie aber beabsichtigen, viele Large Objects auf einmal zu übertragen, sollten Sie selbst einen Transaktionsblock um alle Befehle setzen.

`ON_ERROR_STOP`

Wenn in einem nicht interaktiven Skript ein Fehler auftritt, wie zum Beispiel ein fehlerhafter SQL-Befehl oder interner Metabefehl, dann geht in der Voreinstellung die Verarbeitung einfach weiter. Das ist das traditionelle Verhalten von `psql`, aber in manchen Fällen ist es nicht erwünscht. Wenn diese Variable gesetzt ist, wird die Verarbeitung des Skripts sofort abgebrochen. Wenn das Skript von einem anderen Skript aufgerufen wurde, wird dieses ebenso beendet. Wenn das äußerste Skript nicht aus einer interaktiven `psql`-Sitzung heraus aufgerufen wurde, sondern mit der Option `-f`, gibt `psql` den Fehlercode 3 zurück, um diesen Fall von schweren Fehlern (Fehlercode 1) zu unterscheiden.

`PORT`

Der Datenbankserverport, mit dem Sie gegenwärtig verbunden sind. Diese Variable wird jedes Mal gesetzt, wenn Sie mit einer Datenbank verbinden (einschließlich beim Programmstart), kann aber gelöscht werden.

`PROMPT1`
`PROMPT2`
`PROMPT3`

Diese Variablen geben an, wie die von `psql` ausgegebenen Eingabeaufforderungen aussehen sollen. Siehe `Einstellen der Eingabeaufforderung` *(Seite: 835)* unten.

`QUIET`

Diese Variable entspricht der Kommandozeilenoption `-q`. Im interaktiven Modus ist sie wahrscheinlich nicht besonders nützlich.

`SINGLELINE`

Diese Variable wird von der Kommandozeilenoption `-S` (Einzelzeilenmodus) gesetzt. Sie können Sie zu Laufzeit setzen oder löschen.

SINGLESTEP

Diese Variable entspricht der Kommandozeilenoption -s (Einzelschrittmodus).

USER

Der Name des Benutzers, unter dem Sie gegenwärtig verbunden sind. Diese Variable wird jedes Mal gesetzt, wenn Sie mit einer Datenbank verbinden (einschließlich beim Programmstart), kann aber gelöscht werden.

SQL-Interpolierung

Ein weiteres nützliches Feature von psql ist, dass Sie Variablen in normale SQL-Befehle einsetzen („interpolieren“) können. Die Syntax dafür ist wiederum, dass Sie vor den Variablennamen einen Doppelpunkt setzen:

```
testdb=> \set foo 'meine_tabelle'
testdb=> SELECT * FROM :foo;
```

Dies würde dann die Tabelle meine_tabelle abfragen. Der Wert der Variablen wird unverändert kopiert, selbst wenn er unausgeglichene Anführungszeichen oder Metabefehle enthält. Sie müssen also darauf achten, dass er dort, wo Sie Ihn hinschreiben, auch Sinn hat. Die Variableninterpolation wird aber nicht innerhalb von Anführungszeichen durchgeführt.

Eine verbreitete Anwendung dieser Funktionalität ist, dass man sich auf die eben eingefügte OID in den darauf folgenden Befehlen beziehen kann, wenn man OIDs als Fremdschlüssel verwendet. Eine andere Verwendung dieses Mechanismus ist, den Inhalt einer Datei in eine Tabellenspalte zu kopieren. Erst laden Sie die Datei in eine Variable und dann fahren Sie wie schon oben fort.

```
testdb=> \set inhalt '\'' `cat meine_datei.txt` '\''
testdb=> INSERT INTO meine_tabelle VALUES (:inhalt);
```

Ein mögliches Problem an diesem Ansatz ist, dass meine_datei.txt Apostrophen enthalten könnte. Diese müssen durch Fluchtfolgen ersetzt werden, damit sie keine Syntaxfehler verursachen, wenn die zweite Zeile verarbeitet wird. Das könnte mit dem Programm sed erledigt werden:

```
testdb=> \set inhalt '\'' `sed -e "s/'/\\\\\\'/g" < meine_datei.txt` '\''
```

Beachten Sie die korrekte Anzahl der Backslashes (6)! Sie können das folgendermaßen auflösen: Nachdem psql diese Zeile verarbeitet hat, übergibt es sed -e "s/'/\\\'/g" < meine_datei.txt an die Shell. Die Shell verarbeitet den Inhalt der doppelten Anführungszeichen nochmal selbst und führt dann sed mit den Argumenten -e und s/'/\\'/g aus. Wenn sed das liest, ersetzt es die zwei Backslashes durch einen und führt dann die Ersetzung durch. Vielleicht hatten Sie mal gedacht, dass es toll ist, dass alle Unix-Befehle das gleiche Fluchtzeichen haben. Und das ignoriert noch die Tatsache, dass Sie die Backslashes auch noch durch Fluchtfolgen ersetzen müssten, da diese in SQL-Zeichenkettenkonstanten auch noch besonders interpretiert werden. In dem Fall sollten Sie die Datei vielleicht extern vorbereiten.

Da Doppelpunkte in normalen SQL-Befehlen vorkommen können, gilt folgende Regel: Wenn die Variable nicht gesetzt ist, wird die Zeichenfolge „Doppelpunkt+Name“ nicht verändert. Auf jeden Fall können Sie vor den Doppelpunkt einen Backslash stellen, um die Interpretation als Variable zu unterbinden. (Die Syntax mit dem Doppelpunkt ist der SQL-Standard für eingebettete Sprachen, wie ECPG. Die Doppelpunkt-Syntax für Arraystücke und Typumwandlungen sind jeweils PostgreSQL-Erweiterungen, daher der Konflikt.)

Einstellen der Eingabeaufforderung

Die von psql ausgegebenen Eingabeaufforderungen können Sie an Ihren Geschmack anpassen. Die drei Variablen PROMPT1, PROMPT2 und PROMPT3 enthalten Zeichenketten und besondere Fluchtfolgen, die das

Erscheinungsbild der Eingabeaufforderungen beschreiben. Prompt 1 ist die normale Eingabeaufforderung, wenn `psql` auf die Eingabe eines neuen Befehls wartet. Prompt 2 wird ausgegeben, wenn bei der Befehlseingabe auf weitere Eingaben gewartet wird, weil der Befehl nicht durch ein Semikolon oder ein Anführungszeichen nicht durch ein zweites abgeschlossen wurde. Prompt 3 wird ausgegeben, wenn Sie den SQL-Befehl `COPY` ausführen und Sie die Zeilenwerte am Terminal eingeben sollen.

Der Wert der entsprechenden Prompt-Variable wird unverändert ausgegeben, außer wo ein Prozentzeichen auftritt. Abhängig vom nächsten Zeichen wird bestimmter anderer Text dafür eingesetzt. Die vorgesehenen Sonderwerte sind:

`%M`

Der vollständige Hostname (mit Domainname) des Datenbankservers, oder `[local]` wenn die Verbindung über eine Unix-Domain-Socket ist, oder `[local:`*`/verzeichnis/name`*`]` wenn die Unix-Domain-Socket nicht im eingebauten Standardverzeichnis ist.

`%m`

Der Hostname des Datenbankservers, am ersten Punkt abgeschnitten, oder `[local]` wenn die Verbindung über eine Unix-Domain-Socket ist.

`%>`

Die Portnummer des Datenbankservers.

`%n`

Der Benutzername, unter dem Sie verbunden sind (nicht Ihr lokaler Benutzername).

`%/`

Der Name der aktuellen Datenbank.

`%~`

Wie `%/`, aber die Ausgabe ist `~` (Tilde), wenn die Datenbank Ihre Standarddatenbank ist.

`%#`

Wenn der aktuelle Benutzer ein Datenbank-Superuser ist, dann `#`, ansonsten `>`.

`%R`

In Prompt 1 normalerweise `=`, aber `^` im Einzelzeilenmodus und `!`, wenn die Sitzung nicht mit der Datenbank verbunden ist (was nur passieren kann, wenn `\connect` scheitert). In Prompt 2 wird die Zeichenfolge durch `-`, `*`, ein Apostroph oder ein Anführungszeichen ersetzt, abhängig davon, ob `psql` auf weitere Eingaben wartet, weil der Befehl noch nicht abgeschlossen wurde, weil Sie in einem `/* ... */` Kommentar sind bzw. weil Sie innerhalb eines Anführungszeichenblocks sind. In Prompt 3 ergibt diese Folge gar nichts.

`%`*`ziffern`*

Wenn *`ziffern`* mit `0x` anfängt, werden die restlichen Zeichen als hexadezimale Ziffern interpretiert und das Zeichen mit dem angegebenen Code wird eingesetzt. Wenn die erste Ziffer `0` ist, werden die Zeichen als Oktalzahl interpretiert und das entsprechende Zeichen eingesetzt. Ansonsten wird eine Dezimalzahl angenommen.

`%:`*`name`*`:`

Der Wert der `psql`-Variablen *`name`*. Siehe im Anschnitt `Variablen` *(Seite: 832)* wegen Einzelheiten.

`%` `` ` ``*`befehl`*`` ` ``

Die Ausgabe von *`befehl`*, ähnlich wie bei der normalen Befehlsausführung in Rückwärtsapostrophen.

Um ein Prozentzeichen in die Eingabeaufforderung einzufügen, schreiben Sie `%%`. Die Voreinstellungen sind `'%/%R%# '` für Prompt 1 und 2, sowie `'>> '` für Prompt 3.

Anmerkung

Die Funktion wurde schamlos von `tcsh` abgekupfert.

Eingabezeilenbearbeitung

`psql` unterstützt die `Readline`-Bibliothek für das bequeme Bearbeiten der Eingabezeilen und zum Abrufen von alten Befehlen. Die Befehlsgeschichte wird in einer Datei namens `.psql_history` in Ihrem Home-Verzeichnis gespeichert und beim Start von `psql` neu geladen. Eingaben von Befehlen können auch mit der Tab-Taste vervollständigt werden, aber der Komplettierungsmechanismus ist kein vollständiger SQL-Parser. Wenn Sie die Tab-Vervollständigung aus irgendwelchen Gründen nicht mögen, können Sie sie ausschalten, indem Sie Folgendes in eine Datei namens `.inputrc` in Ihrem Home-Verzeichnis schreiben:

```
$if psql
set disable-completion on
$endif
```

(Diese Funktionalität wird nicht von `psql`, sondern von `Readline` zur Verfügung gestellt. Weitere Informationen finden Sie dort in der Anleitung.)

Umgebung

`HOME`

Verzeichnis für die Initialisierungsdatei (`.psqlrc`) und die Datei für die Befehlsgeschichte (`.psql_history`).

`PAGER`

Wenn die Anfrageergebnisse nicht auf den Bildschirm passen, werden die per Pipe durch diesen Befehl gesendet. Typische Werte sind `more` oder `less`. Die Voreinstellung ist plattformabhängig. Die Verwendung des Pagers kann mit dem Befehl `\pset` abgeschaltet werden.

`PGDATABASE`

Die Datenbank, mit der verbunden werden soll, wenn keine andere Datenbank angegeben wurde.

`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Editorprogramm für den Befehl `\e`. Die Variablen werden in der angegebenen Reihenfolge durchsucht; die erste, die gesetzt ist, wird verwendet.

`SHELL`

Der Befehl, der durch `\!` ausgeführt wird.

`TMPDIR`

Das Verzeichnis für temporäre Dateien. Die Voreinstellung ist `/tmp`.

Dateien

- Vor dem Start versucht `psql`, die Datei `$HOME/.psqlrc` zu lesen und die darin enthaltenen Befehle auszuführen. Diese Datei könnte verwendet werden, um den Client oder Server nach Geschmack einzurichten (mit den Befehlen `\set` bzw. `SET`).
- Die Befehlsgeschichte wird in der Datei `$HOME/.psql_history` gespeichert.

Hinweise

- In einem früheren Leben konnte in `psql` das erste Argument eines einbuchstabigen Metabefehls direkt nach dem Befehl stehen, ohne Leerzeichen dazwischen. Der Kompatibilität halber wird das zum Teil noch unterstützt, aber die Einzelheiten werden wir hier nicht erläutern, da diese Form nicht mehr empfohlen wird. Wenn Sie seltsame Meldungen erhalten, erinnern Sie sich daran. Zum Beispiel

```
testdb=> \foo
Field separator is "oo".
```

  würde man wohl nicht erwarten.

- `psql` funktioniert nur mit Servern der gleichen Version reibungslos. Das bedeutet nicht, dass andere Kombination völlig unbrauchbar sind, aber es könnten einige subtile und nicht ganz so subtile Probleme auftreten. Metabefehle werden mit Servern anderer Versionen mit besonders hoher Wahrscheinlichkeit nicht funktionieren.
- Wenn man während eines Kopiervorgangs an den Server **Strg+C** drückt, ist das gezeigte Verhalten nicht besonders ideal. Wenn Sie eine Meldung wie COPY state must be terminated first sehen, dann erneuern Sie einfach die Verbindung mit `\c - -`.

Beispiele

Das erste Beispiel zeigt, wie man einen Befehl über mehrere Zeilen verteilen kann. Beachten Sie, wie sich die Eingabeaufforderung verändert:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE TABLE
```

Schauen Sie sich die Tabellendefinition nochmal an:

```
testdb=> \d my_table
            Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Jetzt ändern wir die Eingabeaufforderung in etwas Interessanteres:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
```

```
peter@localhost testdb=>
```

Nehmen wir an, Sie haben die Tabelle mit Daten gefüllt und wollen sie sich ansehen:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

Sie können das Aussehen dieser Tabelle mit dem Befehl `\pset` verändern:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
```

```
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Als Alternative gibt es die kurzen Befehle:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

vacuumdb

Name

vacuumdb – säubert und analysiert eine PostgreSQL-Datenbank

Synopsis

vacuumdb [*verbindungsoption*...] [--full | -f] [--verbose | -v] [--analyze | -z] [--table | -t *tabelle* [(*spalte* [,...])]] [*dbname*]
vacuumdb [*verbindungsoptionen*...] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]

Beschreibung

vacuumdb ist ein Hilfsprogramm, um PostgreSQL-Datenbanken zu säubern. vacuumdb erstellt auch interne Statistiken, die vom PostgreSQL-Anfrageoptimierer verwendet werden.

vacuumdb ist ein Shell-Skript-Wrapperprogramm, um den SQL-Befehl VACUUM *(Seite: 783)* unter Verwendung des Programms psql *(Seite: 819)*. Es macht keinen Unterschied, ob in einer Datenbank VACUUM mit

dieser oder einer anderen Methode durchgeführt wird. `psql` muss von dem Skript gefunden werden und ein Datenbankserver muss auf dem ausgewählten Host laufen. Außerdem gelten etwaige Voreinstellungen und Umgebungsvariablen, die von `psql` und der Clientbibliothek `libpq` verwendet werden.

Es kann sein, dass `vacuumdb` mehrmals mit dem PostgreSQL-Server verbinden muss und jedes Mal nach einem Passwort fragt. In solchen Fällen kann es sinnvoll sein, die Datei `$HOME/.pgpass` einzurichten.

Optionen

`vacuumdb` akzeptiert die folgenden Kommandozeilenargumente:

`-a`
`--all`

Führe `VACUUM` in allen Datenbanken aus.

`[-d] dbname`
`[--dbname] dbname`

Gibt den Namen der Datenbank an, die gesäubert oder analysiert werden soll. Wenn dies nicht angegeben ist und `-a` (oder `--all`) nicht verwendet wird, wird der Datenbankname aus der Umgebungsvariablen `PGDATABASE` gelesen. Wenn diese nicht gesetzt ist, wird der für die Verbindung angegebene Benutzername verwendet.

`-e`
`--echo`

Gibt die Befehle aus, die `vacuumdb` erzeugt und an den Server schickt.

`-f`
`--full`

Führt ein „volles“ Vacuum durch (`VACUUM FULL`).

`-q`
`--quiet`

Gibt keine Antwort aus.

`-t tabelle [ (spalte [,...]) ]`
`--table tabelle [ (spalte [,...]) ]`

Säubert oder analysiert nur *`tabelle`*. Spaltennamen können nur zusammen mit der Option `--analyze` angegeben werden.

Tipp

Wenn Sie Spalten angeben, müssen Sie die Klammern sicherlich vor der Shell verstecken. (Siehe Beispiele unten.)

`-v`
`--verbose`

Gibt während der Verarbeitung detaillierte Informationen aus.

`-z`
`--analyze`

Berechnet Statistiken für den Optimierer.

`vacuumdb` akzeptiert außerdem die folgenden Kommandozeilenargumente für Verbindungsparameter:

`-h host`
`--host host`

Gibt den Hostnamen der Maschine, auf der der Datenbankserver läuft, an. Wenn der Wert mit einem Schrägstrich anfängt, wird er als Verzeichnis für die Unix-Domain-Socket verwendet.

`-p` *`port`*
`--port` *`port`*

Gibt den TCP-Port oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der Server auf Verbindungen wartet.

`-U` *`benutzername`*
`--username` *`benutzername`*

Der Benutzername, unter dem verbunden werden soll.

`-W`
`--password`

Erzwingt eine Passworteingabe.

Meldungen

`VACUUM`

Alles hat geklappt.

`vacuumdb: Vacuum failed.`

Irgendwas ist schief gelaufen. `vacuumdb` ist nur ein Wrapper-Skript. Bei `VACUUM` *(Seite: 783)* und `psql` *(Seite: 819)* finden Sie eine detailliertere Besprechung von Fehlermeldungen und potenziellen Problemen.

Umgebung

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Standardverbindungsparameter

Beispiele

Um die Datenbank `test` zu säubern:

```
$ vacuumdb test
```

Um eine Datenbank namens `bigdb` zu säubern und für den Optimierer zu analysieren:

```
$ vacuumdb --analyze bigdb
```

Um eine einzelne Tabelle `foo` in einer Datenbank namens `xyzzy` zu säubern und eine einzelne Spalte `bar` dieser Tabelle für den Optimierer zu analysieren:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

Siehe auch

`VACUUM` *(Seite: 783)*

III. PostgreSQL-Serveranwendungen

Dieser Teil enthält Referenzinformationen über PostgreSQL-Serveranwendungen und Hilfsprogramme. Diese Programme sind nur auf dem Host, wo der Datenbankserver ist, sinnvoll. Andere Hilfsprogramme sind in Referenz II *(Seite: 786)*, PostgreSQL-Clientanwendungen aufgelistet.

initdb

Name

initdb – erzeugt einen neuen PostgreSQL-Datenbankcluster

Synopsis

initdb [*option*...] --pgdata | -D *verzeichnis*

Beschreibung

initdb erzeugt einen neuen PostgreSQL-Datenbankcluster. Ein Datenbankcluster ist eine Sammlung von Datenbanken, die von einer einzelnen Serverinstanz verwaltet wird.

Das Erzeugen eines neuen Datenbankclusters teilt sich auf in das Erzeugen der Verzeichnisse, in dem die Datenbankdaten abgelegt werden, das Erzeugen der geteilten Katalogtabellen (Tabellen, die zum gesamten Cluster gehören anstatt zu einer bestimmten Datenbank) und das Erzeugen der Datenbank template1. Wenn Sie später eine neue Datenbank erzeugen, wird alles aus der Datenbank template1 kopiert. Sie enthält die Katalogtabellen, in denen Dinge wie die eingebauten Typen eingetragen sind.

initdb initialisiert die Standardlocale und den Standardzeichensatz eines Datenbankclusters. Einige Locale-Kategorien stehen für die Lebensdauer des Cluster fest, also muss die richtige Wahl getroffen werden, wenn initdb ausgeführt wird. Andere Locale-Kategorien können auch später, wenn der Server läuft, verändert werden. initdb schreibt die Einstellungen für letztere Locale-Kategorien in die Konfigurationsdatei postgresql.conf, damit sie die Voreinstellung sind, aber Sie können sie ändern, indem Sie diese Datei bearbeiten. Um die Locale einzustellen, die initdb verwendet, schauen Sie unter der Beschreibung der Option --locale nach. Die Zeichensatzkodierung kann für jede Datenbank separat bei ihrer Erzeugung festgelegt werden. initdb bestimmt die Kodierung für die Datenbank template1, welche als Vorgabe für alle anderen Datenbanken dient. Um die Standardkodierung zu ändern, verwenden Sie die Option --encoding.

initdb muss als der Benutzer ausgeführt werden, dem der Serverprozess gehören soll, weil der Server Zugriff auf die von initdb erzeugten Dateien und Verzeichnisse haben muss. Da der Server nicht als root ausgeführt werden darf, können Sie initdb auch nicht als root ausführen. (Wenn Sie es versuchen, wird es sich weigern.)

Obwohl initdb versuchen wird, das angegebene Datenverzeichnis zu erzeugen, wird es oft dazu keine Rechte haben, da das dem Datenverzeichnis übergeordnete Verzeichnis oft dem root-Benutzer gehört. Um eine solche Gliederung einzurichten, erzeugen Sie das Datenverzeichnis als root, verwenden chown, um den Datenbankbenutzerzugang als Eigentümer einzusetzen, melden sich mit su unter dem Datenbankbenutzerzugang an und schließlich führen Sie initdb als dieser Benutzer aus. ,

Optionen

`-D` *verzeichnis*
`--pgdata=`*verzeichnis*

Diese Option gibt an, in welchem Verzeichnis der Datenbankcluster abgelegt werden soll. Das ist die einzige Information, die `initdb` benötigt, aber Sie können diese Angabe weglassen, wenn Sie die Umgebungsvariable `PGDATA` eingerichtet haben, was auch nützlich sein kanin, weil der Datenbankserver (`postmaster`) das Datenbankverzeichnis später durch dieselbe Variable finden kann.

`-E` *kodierung*
`--encoding=`*kodierung*

Wählt die Zeichensatzkodierung der Templatedatenbank. Das wird auch die Standardkodierung aller später erzeugten Datenbanken sein, wenn dann keine andere Kodierung bestimmt wird. Der Vorgabewert ist `SQL_ASCII`.

`--locale=`*locale*

Setzt die Standardlocale für den Datenbankcluster. Wenn diese Option nicht angegeben wird, wird die Locale aus der Umgebung, in der `initdb` ausgeführt wird, übernommen.

`--lc-collate=`*locale*
`--lc-ctype=`*locale*
`--lc-messages=`*locale*
`--lc-monetary=`*locale*
`--lc-numeric=`*locale*
`--lc-time=`*locale*

Wie `--locale`, aber nur die Locale in der angegebenen Kategorie wird gesetzt.

`-U` *benutzername*
`--username=`*benutzername*

Wählt den Benutzernamen des Datenbank-Superusers. Wenn kein Name angegeben wird, wird der Name des effektiven Benutzers, der `initdb` ausführt, verwendet. Es ist nicht wirklich wichtig, was der Name des Superusers ist, aber man konnte zum Beispiel den gebräuchlichen Namen `postgres` wählen, obwohl der Betriebssystembenutzername ein anderer ist.

`-W`
`--pwprompt`

Führt dazu, dass `initdb` um die Eingabe eines Passworts für den Datenbank-Superuser bittet. Wenn Sie nicht beabsichtigen, die Passwortauthentifizierung zu verwenden, ist dies nicht wichtig. Ansonsten werden Sie die Passwort-Authentifizierungsmethode erst verwenden können, wenn Sie ein Passwort eingerichtet haben.

Außerdem gibt es einige selten verwendete Optionen:

`-d`
`--debug`

Gibt Debugmeldungen des Bootstrap-Servers und einige andere Meldungen, die für die breite Allgemeinheit weniger interessant sind, aus. Der Bootstrap-Server ist das Programm, das von `initdb` verwendet wird, um die Katalogtabellen zu erzeugen. Diese Option erzeugt eine gewaltige Menge extrem langweiliger Meldungen.

`-L` *verzeichnis*

Gibt an, wo `initdb` die Eingabedateien zum Initialisieren des Datenbankclusters finden soll. Das ist normalerweise nicht notwendig. Wenn Sie das Verzeichnis doch ausdrücklich angeben müssen, wird das Ihnen mitgeteilt werden.

`-n`
`--noclean`

Wenn `initdb` feststellt, dass es wegen eines Fehlers die Initialisierung des Datenbankclusters nicht zu Ende führen kann, löscht es normalerweise alle Dateien, die es bis zu diesem Zeitpunkt erzeugt hat. Diese Option unterbindet dieses Löschen und ist daher nützlich zur Fehleranalyse.

Umgebung

`PGDATA`

Gibt das Verzeichnis an, in dem der Datenbankcluster abgelegt werden soll; kann durch die Option `-D` übergangen werden.

Siehe auch

`postgres` *(Seite: 852)*, `postmaster` *(Seite: 855)*

initlocation

Name

initlocation – erzeugt einen sekundären PostgreSQL-Datenbankspeicherbereich

Synopsis

`initlocation` *`verzeichnis`*

Beschreibung

`initlocation` erzeugt einen sekundären Speicherbereich für PostgreSQL-Datenbanken. Die Verwaltung und Verwendung von sekundären Speicherbereichen wird unter `CREATE DATABASE` *(Seite: 660)* besprochen. Wenn das Argument keinen Schrägstrich enthält, wird angenommen, dass es sich auf eine Umgebungsvariable bezieht. Siehe Beispiele am Ende.

Um diesen Befehl verwenden zu können, müssen Sie sich unter dem Datenbankbenutzerzugang anmelden (zum Beispiel mit `su`).

Beispiele

Um eine Datenbank an einem alternativen Speicherplatz zu erzeugen, setzen Sie zunächst eine Umgebungsvariable:

```
$ export PGDATA2=/opt/postgres/data
```

Halten Sie den `postmaster`-Prozess an und starten Sie Ihn so neu, dass er die Umgebungsvariable `PGDATA2` sieht. Das System muss so eingerichtet werden, dass der `postmaster PGDATA2` bei jedem Start sieht. Schließlich führen Sie Folgendes aus:

```
$ initlocation PGDATA2
$ createdb -D PGDATA2 testdb
```

Wenn Sie absolute Pfade zulassen, könnten Sie alternativ schreiben:

```
$ initlocation /opt/postgres/data
$ createdb -D /opt/postgres/data/testdb testdb
```

ipcclean

Name

ipcclean – entfernt Shared Memory und Semaphore eines abgebrochenen PostgreSQL-Servers

Synopsis

`ipcclean`

Beschreibung

`ipcclean` entfernt alle Shared-Memory-Segmente und Semaphorgruppen, die dem aktuellen Benutzer gehören. Es ist dafür gedacht, nach einem abgestürzten PostgreSQL-Server (`postmaster` *(Seite: 855)*) aufzuräumen. Beachten Sie, dass Shared Memory und Semaphore auch freigegeben werden, wenn der Server direkt neu gestartet wird, und daher ist dieses Programm nicht besonders nützlich.

Nur der Datenbankadministrator sollte dieses Programm ausführen, da es bei der Verwendung im Mehrbenutzerbetrieb zu bizarrem Verhalten (d.h. Abstürze) führen kann. Wenn dieses Programm ausgeführt wird, während ein Server läuft, werden das Shared Memory und die Semaphore, die zu diesem Serverprozess gehören, gelöscht, was für diesen Server ziemlich schwerwiegende Folgen haben würde.

Hinweise

Dieses Skript ist notdürftig zusammengeschustert, aber in den vielen Jahren, seit es geschrieben wurde, hat sich niemand eine gleichermaßen effektive und portierbare Lösung einfallen lassen. Da der `postmaster` beim Start mittlerweile selbst aufräumen kann, ist es unwahrscheinlich, dass `ipcclean` weiter verbessert werden wird.

Dieses Skript macht bestimmte Annahmen über das Ausgabeformat des Hilfsprogramms `ipcs`, die nicht auf allen Betriebssystemen stimmen müssen. Daher kann es sein, dass es auf Ihrem Betriebssystem nicht funktioniert.

pg_controldata

Name

pg_controldata – zeigt Kontrollinformationen eines PostgreSQL-Datenbankclusters an

Synopsis

`pg_controldata` [*`datenverzeichnis`*]

Beschreibung

`pg_controldata` gibt Informationen aus, die von `initdb` initialisiert wurden, zum Beispiel die Katalogversion und die Serverlocale. Es zeigt auch Informationen über WAL und Checkpoints. Diese Informationen beziehen sich auf den gesamten Datenbankcluster, nicht auf eine einzelne Datenbank.

Dieses Hilfsprogramm kann nur von dem Benutzer ausgeführt werden, der den Cluster initialisiert hatte, weil es das Datenverzeichnis lesen können muss. Das Datenverzeichnis kann auf der Kommandozeile oder durch die Umgebungsvariable `PGDATA` angegeben werden.

Umgebung

`PGDATA`
Standarddatenverzeichnis

pg_ctl

Name

pg_ctl – startet oder stoppt einen PostgreSQL-Server

Synopsis

`pg_ctl` start [-w] [-s] [-D *`datenverzeichnis`*] [-l *`dateiname`*] [-o *`optionen`*] [-p *`pfad`*]
`pg_ctl` stop [-W] [-s] [-D *`datenverzeichnis`*] [-m s[mart] | f[ast] | i[mmediate]]
`pg_ctl` restart [-w] [-s] [-D *`datenverzeichnis`*] [-m s[mart] | f[ast] | i[mmediate]] [-o *`optionen`*]
`pg_ctl` reload [-s] [-D *`datenverzeichnis`*]
`pg_ctl` status [-D *`datenverzeichnis`*]

Beschreibung

`pg_ctl` ist ein Hilfsprogramm, mit dem man den PostgreSQL-Server (`postmaster` *(Seite: 855)*) starten, stoppen oder neu starten, oder den Status eines laufenden Servers anzeigen kann. Ein `postmaster` kann auch direkt gestartet werden, aber `pg_ctl` übernimmt Aufgaben wie die Umleitung der Logausgabe und

das Abtrennen vom Terminal und der Prozessgruppe, und es bietet handliche Optionen zum kontrollierten Herunterfahren.

Im Modus `start` wird ein neuer Server gestartet. Der Server wird im Hintergrund gestartet, die Standardeingabe wird auf `/dev/null` umgeleitet. Die Standardausgabe und die Standardfehlerausgabe werden entweder an eine Logdatei angehängt, wenn die Option `-l` verwendet wird, oder auf die Standardausgabe (nicht die Standardfehlerausgabe) von `pg_ctl` umgeleitet. Wenn keine Logdatei ausgewählt wird, sollte die Standardausgabe von `pg_ctl` in eine Datei umgeleitet werden oder per Pipe mit einem anderen Prozess, zum Beispiel ein Logrotierprogramm, verbunden werden, ansonsten wird der `postmaster` seine Ausgabe auf das kontrollierende Terminal schreiben (aus dem Hintergrund) und wird die Prozessgruppe der Shell nicht verlassen.

Im Modus `stop` wird der im angegebenen Datenverzeichnis laufende Server heruntergefahren. Drei verschiedene Modi zum Herunterfahren können mit der Option `-m` ausgewählt werden: Der Modus *`Smart`* wartet, bis alle Clients ihre Verbindungen getrennt haben. Das ist der Standardmodus. Im Modus *`Fast`* wird nicht darauf gewartet, dass die Clients beenden. Alle aktiven Transaktionen werden zurückgerollt und die Clientverbindungen vom Server getrennt, danach wird der Server heruntergefahren. Der Modus *`Immediate`* bricht alle Serverprozesse ohne sauberes Herunterfahren ab. Das führt zu einem Wiederherstellungsdurchlauf beim Neustart.

Der Modus `restart` führt im Prinzip ein Stopp gefolgt von einem Start aus. Damit können Sie zum Beispiel Kommandozeilenoptionen des `postmaster`-Programms ändern.

Der Modus `reload` schickt einfach das Signal SIGHUP an den `postmaster`-Prozess, wodurch dieser seine Konfigurationsdateien (`postgresql.conf`, `pg_hba.conf` usw.) neu liest. Dadurch können in Konfigurationsdateien Optionen geändert werden, die keinen kompletten Neustart erfordern, um aktiv werden zu können.

Der Modus `status` prüft, ob ein Server gerade im angegebenen Datenverzeichnis läuft, und wenn ja, dann wird die PID und die bei dessen Aufruf verwendete Kommadozeile ausgegeben.

Optionen

`-D` *`datenverzeichnis`*

Gibt den Ort der Datenbankdateien im Dateisystem an. Wenn diese Option ausgelassen wird, wird die Umgebungsvariable `PGDATA` verwendet.

`-l` *`dateiname`*

Hängt die Logausgabe des Servers an die angegebene Datei an. Wenn die Datei nicht existiert, wird sie neu erstellt. Die Umask wird auf 077 gesetzt, sodass auf die Logdatei normal nicht von anderen Benutzern zugegriffen werden kann.

`-m` *`modus`*

Gibt den Modus zum Herunterfahren aus. Möglichkeiten sind `smart`, `fast` oder `immediate`, oder jeweils der erste Buchstabe von einem dieser drei.

`-o` *`optionen`*

Gibt Optionen an, die direkt an den `postmaster`-Befehl übergeben werden.

Die Parameter müssen in der Shell normalerweise in Anführungszeichen gesetzt werden, damit sie als Gruppe ankommen.

`-p` *`pfad`*

Gibt den Pfad zur `postmaster`-Programmdatei an. In der Voreinstellung wird der `postmaster` aus demselben Verzeichnis wie `pg_ctl` oder, wenn das nicht funktioniert, aus dem eingebauten Installationsver-

zeichnis genommen. Es ist nur nötig, diese Option zu verwenden, wenn Sie etwas Ungewöhnliches machen und Fehler erhalten, weil die `postmaster`-Programmdatei nicht gefunden wurde.

`-s`

Gibt nur Fehler aus, keine sonstigen Informationsmeldungen.

`-w`

Wartet, bis der Start oder das Herunterfahren abgeschlossen ist. Die Wartezeit läuft nach 60 Sekunden ab. Diese Option ist die Voreinstellung beim Herunterfahren.

`-W`

Wartet nicht, bis der Start oder das Herunterfahren abgeschlossen ist. Diese Option ist die Voreinstellung beim Starten oder Neustarten.

Umgebung

`PGDATA`

Standarddatenverzeichnis

Weitere finden Sie unter `postmaster` *(Seite: 855)*.

Dateien

Wenn es im Datenverzeichnis die Datei `postmaster.opts.default` gibt, wird der Inhalt dieser Datei als Optionen an den `postmaster`-Befehl übergeben, außer wenn die Option `-o` dies unterbindet.

Hinweise

Das Warten auf den Abschluss des Starts ist eine nicht genau definierte Operation und kann fehlschlagen, wenn die Zugriffskontrolle so eingerichtet ist, dass ein lokales Clientprogramm nicht ohne manuellen Eingriff verbinden kann. Man sollte diese Option vermeiden.

Beispiele

Den Server starten

Um einen Server zu starten:

```
$ pg_ctl start
```

Um den Server zu starten und zu warten, bis der Startvorgang abgeschlossen ist:

```
$ pg_ctl -w start
```

Um einen Server am Port 5433 und ohne `fsync` zu starten, verwenden Sie:

```
$ pg_ctl -o "-F -p 5433" start
```

Den Server stoppen

```
$ pg_ctl stop
```

hält den Server an. Mit der Option `-m` können Sie bestimmen, *`wie`* das Herunterfahren vonstatten geht.

Den Server neu starten

Den Server neu zu starten, ist was das Gleiche wie den Server anzuhalten und dann wieder zu starten, außer dass `pg_ctl` die vom vorangegangenen Lauf verwendeten Kommandozeilenoptionen speichert und wiederverwendet. Um den Server in der einfachsten Form neu zu starten, verwenden Sie:

```
$ pg_ctl restart
```

Um den Server zu starten und zu warten, bis er herunter und wieder hoch gefahren wurde:

```
$ pg_ctl -w restart
```

Um auf Port 5433 und ohne `fsync` neu zu starten:

```
$ pg_ctl -o "-F -p 5433" restart
```

Den Serverstatus anzeigen

Hier eine Beispielausgabe von `pg_ctl`:

```
$ pg_ctl status
pg_ctl: postmaster is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postmaster '-D' '/usr/local/pgsql/data' '-p' '5433' '-
B' '128'
```

Das wäre die Kommandozeile, die beim Neustart wiederverwendet werden würde.

Sieh auch

`postmaster` *(Seite: 855)*

pg_resetxlog

Name

pg_resetxlog – setzt den Write-Ahead-Log und andere Kontrollinformationen eines PostgreSQL-Datenbankclusters zurück

Synopsis

`pg_resetxlog` [-f] [-n] [-o *`oid`*] [-x *`xid`*] [-l *`datei-id,seg`*] *`datenverzeichnis`*

Beschreibung

pg_resetxlog löscht den Write-Ahead-Log (WAL) und wahlweise weitere Kontrollinformationen (in der Datei pg_control). Diese Funktionalität wird manchmal benötigt, wenn diese Dateien verfälscht worden sind. Der Befehl sollte nur als letzter Ausweg verwendet werden, wenn der Server wegen solchen verfälschten Dateien nicht mehr startet.

Nachdem dieser Befehl ausgeführt wurde, sollte es möglich sein, den Server zu starten, aber bedenken Sie, dass die Datenbank inkonsistente Daten wegen teilweise abgeschlossenen Transaktionen enthalten könnte. Sie sollte, sofort Ihre Daten sichern (mit pg_dump), initdb ausführen und die Daten wieder einladen. Danach sollten Sie nach Inkonsistenzen suchen und je nach Bedarf berichtigen.

Dieses Hilfsprogramm kann nur von dem Benutzer ausgeführt werden, der den Cluster initialisiert hatte, weil es das Datenverzeichnis lesen können muss. Aus Sicherheitsgründen muss das Datenverzeichnis auf der Kommandozeile angegeben werden. pg_resetxlog verwendet die Umgebungsvariable PGDATA nicht.

Wenn sich pg_resetxlog beschwert, dass es keine gültigen Daten aus pg_control lesen kann, können Sie mit der Option -f (*force*) erzwingen, dass trotzdem fortgesetzt wird. In dem Fall werden für die fehlenden Daten plausible Werte eingesetzt. Bei den meisten Feldern wird das auch klappen, aber eventuell müssen Sie bei der nächsten OID, der nächsten Transaktionsnummer, der WAL-Startadresse und den Locale-Einstellungen selbst nachhelfen. Die ersten drei dieser Werte können mit den unten besprochenen Kommandozeilenoptionen angegeben werden. Die Quelle für die Schätzung der Localefelder ist die Umgebung von pg_resetxlog selbst; also achten Sie darauf, dass LANG und so weiter mit der Umgebung übereinstimmen, in der initdb ausgeführt wurde. Wenn Sie keine korrekten Werte für alle Felder ermitteln können, können Sie -f trotzdem verwenden, aber die wiederhergestellte Datenbank muss mit noch viel mehr Vorsicht als sonst behandelt werden: Das sofortige Sichern und Neuladen ist unbedingt erforderlich. Führen Sie *keine* Daten modifizierenden Operationen in der Datenbank aus, bevor Sie sie sichern; solche Aktionen würden die Zerstörung höchstwahrscheinlich noch schlimmer machen.

Mit den Optionen -o, -x und -l können die nächste OID, die nächste Transaktionsnummer bzw. die WAL-Startadresse manuell gesetzt werden. Das ist nur notwendig, wenn pg_resetxlog die richtigen Werte nicht aus pg_control lesen kann. Ein sicherer Wert für die nächste Transaktionsnummer kann ermittelt werden, indem man in das Verzeichnis pg_clog unter dem Datenverzeichnis schaut, dort den numerisch größten Dateinamen findet, eins addiert und dann mit 1048576 multipliziert. Beachten Sie, dass die Dateinamen in hexadezimal sind. Am einfachsten ist es normalerweise, wenn Sie den Optionswert auch in hexadezimal angeben. Wenn zum Beispiel 0011 der größte Eintrag in pg_clog ist, dann können Sie -x 0x1200000 verwenden. (Die fünf Nullen am Ende ergeben den richtigen Multiplikationsfaktor.) Die WAL-Startadresse sollte numerisch größer sein als jede Datei, die gegenwärtig im Verzeichnis pg_xlog unter dem Datenverzeichnis existiert. Die Adressen sind auch in hexadezimal und haben zwei Teile. Wenn zum Beispiel der größte Eintrag in pg_xlog 000000FF0000003A ist, dann können Sie -l 0xFF,0x3B verwenden. Es gibt keinen vergleichbar einfachen Weg, die nächste OID, die größer ist als die größte in der Datenbank, zu ermitteln, aber glücklicherweise ist es nicht entscheidend, die nächste OID richtig zu bestimmen.

Die Option -n gibt an, dass pg_resetxlog nur die aus pg_control ermittelten Daten ausgeben soll und dann beenden soll, ohne etwas zu verändern. Das ist hauptsächlich ein Debug-Werkzeug, aber es kann auch nützlich sein, um diese Werte zu prüfen, bevor man pg_resetxlog wirklich ausführt.

Hinweise

Dieser Befehl darf nicht ausgeführt werden, während ein Server läuft. pg_resetxlog wird sich weigern zu starten, wenn es im Datenverzeichnis eine Sperrdatei des Servers findet. Wenn der Server abgestürzt ist, kann eine Sperrdatei zurückgelassen worden sein; in diesem Fall können Sie die Sperrdatei entfernen, um das Ausführen von pg_resetxlog zu ermöglichen. Aber bevor Sie das tun, sollten Sie sich doppelt und dreifach versichern, dass kein postmaster- oder anderer Serverprozess mehr läuft.

postgres

Name

postgres – führt einen PostgreSQL-Server im Einzelbenutzermodus aus

Synopsis

postgres [-A 0 | 1] [-B *puffer*] [-c *name*=*value*] [-d *debugniveau*] [-D *datenverzeichnis*] [-e] [-E] [-f s | i | t | n | m | h] [-F] [-i] [-N] [-o *dateiname*] [-O] [-P] [-s | -t pa | pl | ex] [-S *sortierspeicher*] [-W *sekunden*] [--*name*=*wert*] *datenbank*
postgres [-A 0 | 1] [-B *puffer*] [-c *name*=*wert*] [-d *debugniveau*] [-D *datenverzeichnis*] [-e] [-f s | i | t | n | m | h] [-F] [-i] [-o *dateiname*] [-O] [-p *datenbank*] [-P] [-s | -t pa | pl | ex] [-S *sortierspeicher*] [-v *protokoll*] [-W *sekunden*] [--*name*=*wert*]

Beschreibung

Das Programm postgres ist der eigentliche PostgreSQL-Serverprozess, der Anfragen verarbeitet. Normalerweise wird es nicht direkt ausgeführt; anstelle dessen wird ein Mehrbenutzerserver (siehe postmaster *(Seite: 855)*) gestartet.

Die zweite Form oben ist wie postgres vom postmaster *(Seite: 855)* aus gestartet (nur theoretisch, da postmaster und postgres eigentlich dasselbe Programm sind); direkt sollte es so nicht aufgerufen werden. Die erste Form startet den Server direkt im interaktiven Einzelbenutzermodus. Dieser Modus wird hauptsächlich für die Initialisierung durch initdb *(Seite: 843)* verwendet. Manchmal kann er auch zum Debuggen oder zur Katastrophenbereinigung verwendet werden.

Wenn postgres von der Shell im interaktiven Modus aufgerufen wird, dann kann der Benutzer Anfragen eingeben und die Ergebnisse werden auf dem Bildschirm ausgegeben werden, aber in einer Form, die eher für Entwickler als für Endanwender brauchbar ist. Beachten Sie auch, dass der Einzelbenutzermodus nicht wirklich für das Debuggen des Servers geeignet ist, da die Interprozesskommunikation und die Sperren nicht dem realistischen Ausmaß entsprechen.

Wenn ein Einzelbenutzerserver ausgeführt wird, wird der Sitzungsbenutzer auf den Benutzer mit der ID 1 gesetzt. Dieser Benutzer muss nicht tatsächlich existieren, daher kann ein Einzelbenutzerserver verwendet werden, um von Hand bestimmte versehentliche Schäden an den Systemkatalogen zu reparieren. Im Einzelbenutzermodus erhält der Benutzer mit der ID 1 automatisch Superuser-Rechte.

Optionen

Wenn postgres durch postmaster *(Seite: 855)* aufgerufen wird, übernimmt es alle Optionen davon. Zusätzlich können postgres-spezifische Optionen vom postmaster mit der Option -o übergeben werden.

Sie können das Eingeben dieser Optionen vermeiden, indem Sie eine Konfigurationsdatei einrichten. Einzelheiten dazu finden Sie in Abschnitt SET. Einige (sichere) Optionen können auch von der verbindenden Clientanwendung gesetzt werden, wie genau, hängt von der Anwendung ab. Wenn zum Beispiel die Umgebungsvariable PGOPTIONS gesetzt ist, werden Anwendungen auf libpq-Basis diese Zeichenkette an den Server schicken, welcher Sie dann als postgres-Kommandozeilenoptionen interpretiert.

Allgemeine Optionen

Die Optionen `-A`, `-B`, `-c`, `-d`, `-D`, `-F` und `--`*`name`* haben die gleiche Bedeutung wie bei `postmaster` *(Seite: 855)*, außer dass `-d 0` verhindert, dass das Logniveau vom `postmaster` in `postgres` übernommen wird.

`-e`

Setzt den Standarddatumsstil auf „Europäisch“, was bedeutet, dass der Tag vor dem Monat kommt (anstatt Monat vor Tag), wenn zweideutige Datumseingaben interpretiert werden und bei bestimmten Ausgabeformaten. Weitere Informationen finden sich in Abschnitt SET.

`-o` *`dateiname`*

Sendet die Serverlogausgabe in die Datei *`dateiname`*. Wenn `postgres` unter `postmaster` läuft, wird diese Option ignoriert und die von `postmaster` übernommene Standardfehlerausgabe wird verwendet.

`-P`

Ignoriere Systemindexe beim Durchsuchen und Aktualisieren von Systemtabellen. Diese Option muss verwendet werden, wenn der Befehl `REINDEX` auf Systemtabellen/-indexe angewendet wird.

`-s`

Gib am Ende jedes Befehls Zeitmessungen und andere Informationen aus. Das kann zum Benchmarking und um die Zahl der Puffer abzustimmen nützlich sein.

`-S` *`sortierspeicher`*

Gibt an, wie viel Speicher von internen Sortier- und Hash-Vorgängen verwendet werden darf, bevor sie auf temporäre Dateien ausweichen. Der Wert wird in Kilobyte angegeben und die Voreinstellung ist 1024 (also 1 MB). Beachten Sie, dass bei komplexen Anfragen mehrere Sortiervorgänge parallel ablaufen können und dass dann jeder einzelne so viel Speicher verwenden kann, wie dieser Wert angibt, bevor er anfängt, Daten in temporäre Dateien auszulagern.

Optionen für den Einzelbenutzermodus

`datenbank`

Gibt die Datenbank an, auf die zugegriffen werden soll. Wenn nichts angegeben wird, dann wird der Benutzername verwendet.

`-E`

Gibt alle Befehle aus, bevor Sie ausgeführt werden.

`-N`

Schaltet die Verwendung des Neue-Zeile-Zeichens als Befehlstrennzeichen ab.

Semiinterne Optionen

Es gibt einige weitere Optionen, die hauptsächlich zum Debuggen da sind. Diese werden hier nur zur Verwendung durch PostgreSQL-Entwickler aufgelistet. *`Die Verwendung dieser Optionen wird nicht empfohlen.`* Ferner kann jede dieser Optionen in einer zukünftigen Version verschwinden oder geändert werden.

`-f { s | i | m | n | h }`

Verbietet die Verwendung bestimmter Plantypen: `s` und `i` schalten sequenzielle Scans bzw. Indexscans ab; `n`, `m` und `h` schalten Nested-Loop-Verbunde, Merge-Verbunde bzw. Hash-Verbunde ab.

`-i`

Verhindert die Ausführung einer Anfrage, aber zeigt den Planbaum.

`-O`

Erlaubt die Veränderung der Strukturen von Systemtabellen. Das wird von `initdb` verwendet.

Anmerkung

Weder sequenzielle Scans noch Nested-Loop-Verbunde können vollständig abgeschaltet werden; die Optionen `-fs` und `-fn` sorgen einfach dafür, dass der Planer diese Plantypen nicht verwendet, wenn er eine Alternative hat.

`-p` *`datenbank`*

Zeigt an, dass dieser Prozess von einem `postmaster` gestartet wurde und gibt die zu verwendende Datenbank an.

`-t pa[rser] | pl[anner] | e[xecutor]`

Gib für jede Anfrage Zeitmessungen für das entsprechende größere Systemmodul aus. Diese Option kann nicht zusammen mit der Option `-s` verwendet werden.

`-v` *`protokoll`*

Gibt die Versionsnummer des für diese Sitzung zu verwendenden Client/Server-Protokolls an.

`-W` *`sekunden`*

Sobald diese Option entdeckt wird, schläft der Prozess für die angegebene Anzahl von Sekunden. Das gibt Entwicklern Zeit ,um einen Debugger für den Serverprozess zu starten.

Umgebung

`PGDATA`

Standarddatenverzeichnis

Weitere Variablen, die wenig Einfluss im Einzelbenutzermodus haben, werden bei `postmaster` *(Seite: 855)* beschrieben.

Hinweise

Um eine laufende Anfrage abzubrechen, verwenden Sie das Signal SIGINT. Um `postgres` aufzufordern, die Konfigurationsdatei neu zu lesen, verwenden Sie das Signal SIGHUP. Der `postmaster` verwendet das Signal SIGTERM, um einen `postgres`-Prozess aufzufordern, normal zu beenden, und SIGQUIT, um ohne normales Herunterfahren zu beenden. Diese sollten von Endanwendern *`nicht`* verwendet werden.

Verwendung

Starten Sie einen Einzelbenutzerserver mit einem Befehl wie:

```
postgres -D /usr/local/pgsql/data andere-optionen meine_datenbank
```

Geben Sie mit `-D` den richtigen Pfad zum Datenbankverzeichnis an oder setzen Sie die Umgebungsvariable `PGDATA` richtig. Geben Sie außerdem den Namen einer bestimmten Datenbank an, mit der Sie arbeiten wollen.

Normalerweise behandelt ein Einzelbenutzerserver ein Neue-Zeile-Zeichen (Newline) als Ende der Befehlseingabe; es gibt keine intelligente Behandlung von Semikolons wie in `psql`. Um einen Befehl über mehrere Zeilen zu verteilen, müssen Sie vor jedem Zeilenende außer dem letzten einen Backslash eingeben.

Wenn Sie aber die Kommandozeilenoption `-N` verwenden, wird ein Befehl durch das Zeilenende nicht abgeschlossen. In diesem Fall liest der Server die Standardeingabe bis zur Dateiendemarkierung (EOF) und verarbeitet die Eingabe dann als eine einzige Befehlzeichenkette. Backslash-Neue-Zeile-Folgen werden in diesem Fall nicht besonders behandelt.

Um die Sitzung zu beenden, geben Sie EOF ein (üblicherweise **Strg**+**D**). Wenn Sie `-N` verwendet haben, müssen zwei aufeinander folgende EOFs eingegeben werden, um zu beenden.

Beachten Sie, dass ein Einzelbenutzerserver keine besonderen Fähigkeiten zur Befehlszeilenbearbeitung anbietet (keine Befehlsgeschichte zum Beispiel).

Siehe auch

`initdb` *(Seite: 843)*, `ipcclean` *(Seite: 846)*, `postmaster` *(Seite: 855)*

postmaster

Name

postmaster – PostgreSQL-Mehrbenutzer-Datenbankserver

Synopsis

`postmaster` [-A 0 | 1] [-B *`puffer`*] [-c *`name`*=*`wert`*] [-d *`debugniveau`*] [-D *`datenverzeichnis`*] [-F] [-h *`hostname`*] [-i] [-k *`verzeichnis`*] [-l] [-N *`max-verbindungen`*] [-o *`extra-optionen`*] [-p *`port`*] [-S] [--*`name`*=*`wert`*] [-n | -s]

Beschreibung

`postmaster` ist der PostgreSQL-Mehrbenutzer-Datenbankserver. Wenn eine Clientanwendung auf eine Datenbank zugreifen will, verbindet sie (lokal oder über ein Netzwerk) mit einem laufenden `postmaster`. Der `postmaster` startet dann einen separaten Serverprozess („`postgres`“ *(Seite: 852)*), der diese Verbindung betreut. Der `postmaster` verwaltet auch die Kommunikation zwischen den Serverprozessen.

In der normalen Einstellung startet der `postmaster` im Vordergrund und gibt Logmeldungen auf der Standardfehlerausgabe aus. In richtigen Anwendungen sollte der `postmaster` als Hintergrundprozess gestartet werden, vielleicht beim Booten.

Ein `postmaster` verwaltet immer die Daten von genau einem Datenbankcluster. Ein Datenbankcluster ist eine Sammlung von Datenbanken, die an einer gemeinsamen Stelle im Dateisystem abgelegt sind. Wenn ein `postmaster` gestartet wird, muss er wissen, wo die Datenbankclusterdateien (der so genannte Datenbereich) liegen. Das wird mit der Kommandozeilenoption `-D` oder mit der Umgebungsvariable `PGDATA` getan; es gibt keine Voreinstellung. Mehrere `postmaster`-Prozesse können gleichzeitig auf einem System laufen, solange sie verschiedene Datenbereiche und verschiedene Kommunikationsports (siehe unten) verwenden. Ein Datenbereich wird mit `initdb` *(Seite: 843)* erzeugt.

Optionen

`postmaster` akzeptiert die folgenden Kommandozeilenargumente. Eine detaillierte Besprechung dieser Optionen finden Sie in Abschnitt SET. Sie können außerdem das Eingeben der meisten dieser Optionen vermeiden, indem Sie eine Konfigurationsdatei einrichten

`-A 0|1`

Schaltet *`Assertion`*-Prüfungen zur Laufzeit ein. Das ist ein Hilfsmittel, um Programmierfehler zu entdecken. Diese Option ist nur verfügbar, wenn Sie bei der Compilierung eingeschaltet wurde. Wenn das der Fall ist, dann ist die Voreinstellung an.

`-B` *`puffer`*

Setzt die Anzahl der vom Datenbankserver verwendeten Puffer im Shared Memory. Die Voreinstellung ist 64 Puffer, wo jeder Puffer 8 kB groß ist.

`-c` *`name=wert`*

Setzt den benannten Konfigurationsparameter. Eine Liste der Parameter und Beschreibungen dazu finden Sie in Abschnitt SET. Die meisten anderen Kommandozeilenoptionen sind eigentlich nur Kurzformen von solchen Parameterzuweisungen. Die Option `-c` kann mehrfach verwendet werden, um mehrere Parameter zu setzen.

`-d` *`debugniveau`*

Setzt das Debugniveau. Je höher dieser Wert ist, desto mehr Debugmeldungen werden in den Serverlog geschrieben. Gültige Werte gehen von 1 bis 5.

`-D` *`datenverzeichnis`*

Gibt das Datenverzeichnis an. Siehe Besprechung oben.

`-F`

Schaltet `fsync`-Aufrufe ab, wodurch eine bessere Leistung erreicht wird, aber das Risiko des Datenverlusts bei Systemabstürzen steigt. Diese Option hat die gleiche Bedeutung wie in `postgresql.conf` `fsync=false` zu setzen. Lesen Sie die detaillierte Beschreibung, bevor Sie diese Option verwenden!

Die umgekehrte Wirkung können Sie mit `--fsync=true` erreichen.

`-h` *`hostname`*

Gibt den Hostnamen oder die IP-Adresse an, auf der `postmaster` auf ankommende Verbindungen von Clientanwendungen hören soll. In der Voreinstellung wird auf allen konfigurierten Adressen gehört (einschließlich localhost).

`-i`

Erlaubt Clientverbindungen über TCP/IP (Internet-Domain). Ohne diese Option werden nur Verbindungen über Unix-Domain-Sockets angenommen. Diese Option hat die gleiche Bedeutung wie in `postgresql.conf` `tcpip_socket=true` zu setzen.

Die umgekehrte Wirkung können Sie mit `--tcpip-socket=false` erreichen.

`-k` *`verzeichnis`*

Gibt das Verzeichnis für die Unix-Domain-Socket an, auf der `postmaster` auf ankommende Verbindungen von Clientanwendungen hören soll. Die Voreinstellung ist normalerweise `/tmp`, kann aber beim Compilieren geändert werden.

`-l`

Ermöglicht sichere Verbindungen über SSL. Die Option `-i` muss auch angegeben werden. Außerdem muss SSL beim Compilieren angeschaltet worden sein, um diese Option verwenden zu können.

`-N` *`max-verbindungen`*

Setzt, wie viele Verbindungen dieser `postmaster` maximal entgegen nimmt. In der Voreinstellung ist dieser Wert 32, aber er kann so hoch gesetzt werden, wie Ihr System unterstützt. (Beachten Sie, dass die Option `-B` mindestens zweimal so hoch wie `-N` sein muss. Schauen Sie auch in Abschnitt SET nach, wo die Systemanforderungen für eine große Anzahl von Client-Verbindungen besprochen werden.)

`-o` *`extra-optionen`*

Die in *`extra-optionen`* angegebenen Optionen im Kommandozeilenstil werden an alle von diesem `postmaster` gestarteten Serverprozesse übergeben. Siehe `postgres` *(Seite: 852)* über die Möglichkeiten. Wenn die Optionszeichenkette Leerzeichen enthält, muss die gesamte Zeichenkette in Anführungszeichen gesetzt werden.

`-p` *`port`*

Gibt die TCP-Portnummer oder die Dateierweiterung der lokalen Unix-Domain-Socket an, wo der `postmaster` auf Verbindungen hören soll. Der Vorgabewert ist der Wert der Umgebungsvariablen `PGPORT` oder, wenn `PGPORT` nicht gesetzt ist, ein während der Compilierung festgelegter Wert (normalerweise 5432). Wenn Sie nicht den vorgegebenen Port verwenden, müssen alle Clientanwendungen den gleichen Port entweder mit Kommandozeilenoptionen oder durch `PGPORT` angeben.

`-S`

Gibt an, dass der `postmaster`-Prozess im stillen Modus starten soll. Das heißt, er trennt sich vom kontrollierenden Terminal des Benutzers, startet seine eigene Prozessgruppe und leitet die Standardausgabe und die Standardfehlerausgabe an `/dev/null` um.

Durch diese Option werden alle Logmeldungen verworfen, was Sie wahrscheinlich nicht wirklich wollen, da es dadurch sehr schwer wird, Probleme zu analysieren. Weiter unten finden Sie eine bessere Methode, um den `postmaster` im Hintergrund zu starten.

Die umgekehrte Wirkung können Sie mit `--silent-mode=false` erreichen.

`--name=wert`

Setzt den benannten Konfigurationsparameter; eine kürzere Form von `-c`.

Zwei weitere Kommandozeilenoptionen stehen zur Verfügung, um Probleme zu analysieren, durch die ein Serverprozess auf nicht normale Weise abgebrochen wird. Das normale Vorgehen in solchen Situationen ist, alle anderen Serverprozesse zu benachrichtigen, dass Sie abbrechen müssen, und dann das Shared Memory und die Semaphore neu zu initialisieren. Das ist notwendig, weil ein fehlerhafter Serverprozess vor seinem Abbruch geteilte Statusinformationen verfälscht haben könnte. Diese Optionen wählen alternative Verhalten von `postmaster` in dieser Situation. *`Beide Optionen sind nicht für den normalen Betrieb vorgesehen.`*

Diese beiden besonderen Optionen sind:

`-n`

`postmaster` wird die geteilten Datenstrukturen nicht neu initialisieren. Ein gut informierter Systemprogrammierer kann dann mit einem Debugger den Zustand des Shared Memory und der Semaphore untersuchen.

`-s`

`postmaster` wird alle anderen Serverprozesse mit dem Signal SIGSTOP anhalten, aber nicht abbrechen. Dadurch können Systemprogrammierer von allen Serverprozessen von Hand Coredumps einsammeln.

Umgebung

`PGCLIENTENCODING`

Standardzeichensatzkodierung der Clients. (Die Clients können dies individuell ändern.) Dieser Wert kann auch in der Konfigurationsdatei gesetzt werden.

`PGDATA`

Standarddatenverzeichnis

PGDATESTYLE

Vorgabewert der Laufzeit-Konfigurationsoption `datestyle`. (Die Verwendung dieser Umgebungsvariablen ist nicht mehr empfohlen.)

PGPORT

Standardport (besser in der Konfigurationsdatei zu setzen)

TZ

Zeitzone des Servers

andere

Andere Umgebungsvariablen können verwendet werden, um alternative Datenspeicherplätze festzulegen. Siehe Abschnitt SET für weitere Informationen.

Meldungen

`semget: No space left on device`

Wenn Sie diese Meldung sehen, müssen Sie in Ihrem Kernel wahrscheinlich Shared Memory und Semaphore konfigurieren, wie in Abschnitt SET beschrieben wird. Wenn Sie mehrere Instanzen von `postmaster` auf einer einzigen Maschine laufen haben oder Ihr Kernel besonders kleine Grenzen für Shared Memory und/oder Semaphore hat, müssen Sie Ihren Kernel wahrscheinlich umkonfigurieren, um diese Parameter zu erhöhen.

Tipp

Sie können das Umkonfigurieren des Kernels eventuell verschieben, wenn Sie mit der Option `-B` den Shared-Memory-Verbrauch und/oder mit `-N` den Semaphorverbrauch von PostgreSQL verringern.

`StreamServerPort: cannot bind to port`

Wenn Sie diese Meldungen sehen, versichern Sie sich, dass kein anderer `postmaster` schon die gleiche Portnummer verwendet. Am einfachsten finden Sie dies mit dem folgenden Befehl heraus:

```
$ ps ax | grep postmaster
```

oder

```
$ ps -e | grep postmaster
```

je nach Betriebssystem.

Wenn Sie sich sicher sind, dass keine anderen `postmaster`-Prozesse laufen und Sie diesen Fehler immer noch erhalten, versuchen Sie mit der Option `-p` einen anderen Port zu wählen. Sie könnten diesen Fehler auch sehen, wenn Sie den `postmaster` beenden und sofort am selben Port neu starten; in diesem Fall müssen Sie einfach einige Sekunden warten, bis das Betriebssystem den Port geschlossen hat, und dann neu probieren. Schließlich könnten Sie diesen Fehler auch erhalten, wenn Sie eine Portnummer angeben, die das Betriebssystem als reserviert betrachtet. Viele Unix-Varianten reservieren zum Beispiel Portnummern unter 1024 und erlauben nur dem Unix-Superuser, sie zu verwenden.

Hinweise

Wenn nur irgend möglich, sollten Sie *nicht* das Signal SIGKILL verwenden, um den `postmaster` zu beenden. Das verhindert, dass der `postmaster` bestimmte Systemressourcen (z.B. Shared Memory und Semaphore) vor dem Beenden freigibt.

Um den `postmaster` normal zu beenden, können die Signale SIGTERM, SIGINT oder SIGQUIT verwendet werden. Durch das erste wartet der Serve,r bis alle Clients beendet haben, bevor er selbst beendet, durch das zweite bricht der Server alle Clientverbindungen sofort ab und durch das dritte beendet der Server sofort ohne ordentliches Herunterfahren, wodurch beim Neustart ein Wiederherstellungsdurchlauf nötig sein wird.

Das Hilfsprogramm `pg_ctl` *(Seite: 847)* kann verwendet werden, um den `postmaster` bequem und sicher zu starten und zu stoppen.

Die `--` Optionen funktionieren nicht auf FreeBSD oder OpenBSD. Verwenden Sie anstelle dessen `-c`. Das ist ein Fehler in den betroffenen Betriebssystemen; wenn er nicht berichtigt wird, dann wird eine zukünftige Version von PostgreSQL eine eigene Lösung anbieten.

Beispiele

Um den `postmaster` mit allen Standardwerten im Hintergrund zu starten, geben Sie ein:

```
$ nohup postmaster >logdatei 2>&1 </dev/null &
```

Um den `postmaster` mit einem bestimmten Port zu starten:

```
$ postmaster -p 1234
```

Dieser Befehl startet den `postmaster` so, dass er durch Port 1234 kommuniziert. Um mit diesem `postmaster` von `psql` aus zu verbinden, müssen Sie Folgendes ausführen:

```
$ psql -p 1234
```

Oder setzen Sie die Umgebungsvariable `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Konfigurationsparameter können auf folgende Arten gesetzt werden:

```
$ postmaster -c sort_mem=1234
$ postmaster --sort-mem=1234
```

Beide Formen haben Vorrang vor etwaigen Einstellungen von `sort_mem` in `postgresql.conf`. Wie sie sehen, können Unterstriche in Parameternamen auf der Kommandozeile auch als Bindestriche geschrieben werden.

Tipp

Außer bei kurzzeitigen Experimenten ist es in der Praxis wohl besser, die Einstellungen in der Datei `postgresql.conf` zu bearbeiten, anstatt sich auf die Kommandozeilenoptionen zu verlassen.

Siehe auch

`initdb` *(Seite: 843)*, `pg_ctl` *(Seite: 847)*

Teil VII

Anhänge

Interna der Datums- und Zeitunterstützung

PostgreSQL verwendet einen internen heuristischen Parser zur Interpretation von Datums- und Zeiteingaben. Daten und Zeiten werden als Zeichenketten eingegeben und werden in getrennte Felder aufgeteilt, die vorläufige Feststellungen darüber enthalten, welche Information in dem Feld sein könnte. Jedes Feld wird analysiert und bekommt entweder einen numerischen Wert zugewiesen, wird ignoriert oder wird abgelehnt. Der Parser hat interne Tabellen für alle Textfelder, einschließlich Monatsnamen, Wochentagsnamen und Zeitzonen.

Dieser Anhang enthält Informationen über den Inhalt dieser Tabellen und beschreibt die Schritte des Parsers bei der Entschlüsselung von Datums- und Zeiteingaben.

A.1 Eingabeinterpretation von Datum und Zeit

Alle Eingabewerte von Datums- und Zeittypen werden nach folgender Prozedur entschlüsselt.

1. Zerteile die Eingabezeichenkette in Tokens und kategorisiere jedes Token als Zeichenkette, Zeit, Zeitzone oder Zahl.
   a. Wenn ein numerisches Token einen Doppelpunkt (`:`) enthält, dann ist es eine Zeitangabe. Schließe alle folgenden Ziffern und Doppelpunkte mit ein.
   b. Wenn ein numerisches Token einen Bindestrich (`-`), Schrägstrich (`/`) oder zwei oder mehr Punkte (`.`) enthält, dann ist es eine Datumsangabe, welche einen Monatsnamen als Text enthalten könnte.
   c. Wenn ein Token durchweg numerisch ist, ist es entweder ein einzelnes Feld oder ein zusammenhängendes Datum nach ISO 8601 (z.B. `19990113` für 13. Januar 1999) oder eine zusammenhängende Zeitangabe (z.B. `141516` für 14:15:16).
   d. Wenn ein Token ein Plus- (+) oder Minuszeichen (`-`) enthält, dann ist es entweder eine Zeitzone oder ein Sonderfeld.
2. Wenn das Token eine Textzeichenkette ist, suche eine passende Interpretation.
   a. Führe eine binäre Tabellensuche nach dem Token entweder als Sonderwert (z.B. `today`), Wochentag (z.B. `Thursday`), Monat (z.B. `January`) oder Dekorationswort (z.B. `at`, `on`) durch.

      Setze die Feldwerte und die Bitmasken für die Felder. Zum Beispiel, bei `today` setze Jahr, Monat und Tag, und bei `now` zusätzlich Stunde, Minute, Sekunde.

   b. Wenn nicht gefunden, führe eine binäre Tabellensuche durch, um das Token als Zeitzone zu interpretieren.
   c. Wenn nicht gefunden, erzeuge einen Fehler.
3. Das Token ist eine Zahl oder ein numerisches Feld.
   a. Wenn mehr als 4 Ziffern vorhanden sind und wenn noch keine anderen Datumsfelder gelesen wurden, dann interpretiere sie als "zusammenhängendes Datum" (z.B. `19990118`). 8 und 6 Ziffern werden als Jahr, Monat und Tag interpretiert, während 7 und 5 Ziffern als Jahr und Tag im Jahr interpretiert werden.
   b. Wenn das Token drei Ziffern hat und ein Jahr schon gelesen wurde, interpretiere sie als als Tagesnummer im Jahr.
   c. Wenn vier oder sechs Ziffern vorhanden sind und ein Jahr schon gelesen wurde, dann interpretiere sie als Zeitangabe.
   d. Wenn vier oder mehr Ziffern vorhanden sind, dann interpretiere sie als Jahr.
   e. Wenn im europäischen Datumsformat und das Tagesfeld noch nicht gelesen wurde und der Wert kleiner oder gleich 31 ist, interpretiere ihn als Tag.
   f. Wenn das Monatsfeld noch nicht gelesen wurde und der Wert kleiner oder gleich 12 ist, interpretiere ihn als Monat.
   g. Wenn das Tagesfeld noch nicht gelesen wurde und der Wert kleiner oder gleich 31 ist, interpretiere ihn als Tag.
   h. Wenn zwei Ziffern oder vier oder mehr Ziffern vorhanden sind, interpretiere sie als Jahr.
   i. Ansonsten erzeuge einen Fehler.
4. Wenn `BC` angegeben wurde, negiere das Jahr und addiere eins zur internen Speicherung. (Es gibt kein Jahr null im gregorianischen Kalender, also wird aus dem Jahr 1 v.u.Z. intern das Jahr null.)
5. Wenn `BC` nicht angegeben wurde und das Jahr zwei Ziffern lang ist, erweitere es auf vier Ziffern. Wenn das Feld kleiner als 70 ist, addiere 2000, ansonsten addiere 1900.

Tipp

Die Jahre 1-99 u.Z. im gregorianischen Kalender können vierziffrig mit führenden Nullen eingegeben werden (z.B. `0099` ist das Jahr 99). Frühere Versionen von PostgreSQL akzeptierten Jahreszahlen mit drei Ziffern und einer Ziffer, aber in Version 7.0 wurden die Regeln verschärft, um den Raum für Unklarheiten einzugrenzen.

A.2 Schlüsselwörter in Datums- und Zeitangaben

Tabelle A.1 zeigt die Tokens, die als Abkürzungen von (englischen) Monatsnamen erlaubt sind.

Monat	Abkürzung
April	Apr
August	Aug
December	Dec
February	Feb

Tabelle A.1: Abkürzungen von Monatsnamen

Monat	Abkürzung
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Tabelle A.1: Abkürzungen von Monatsnamen (Forts.)

Anmerkung

Für den Monat Mai (englisch *May*) ist aus offensichtlichen Gründen keine Abkürzung vorgesehen.

Tabelle A.2 zeigt die Tokens, die als Abkürzungen von (englischen) Wochentagsnamen erlaubt sind.

Wochentag	Abkürzung
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Tabelle A.2: Abkürzungen von Wochentagsnamen

Tabelle A.3 zeigt Tokens, die Datums- und Zeitangaben auf diverse Wiese modifizieren können.

Wort	Beschreibung
`ABSTIME`	Schlüsselwort ignoriert
`AM`	Zeit ist vor 12:00
`AT`	Schlüsselwort ignoriert
`JULIAN, JD, J`	Nächstes Feld ist julianisches Datum
`ON`	Schlüsselwort ignoriert
`PM`	Zeit ist um oder nach 12:00
`T`	Nächstes Feld ist die Zeit

Tabelle A.3: Modifikationswörter für Datums- und Zeitangaben

Das Schlüsselwort `ABSTIME` wird aus historischen Gründen ignoriert: In sehr alten Versionen von PostgreSQL wurden ungültige Werte des Typs abstime als `Invalid Abstime` ausgegeben. Das ist nicht mehr der Fall und das Schlüsselwort wird wahrscheinlich in einer zukünftigen Version entfernt werden.

Tabelle A.4 zeigt die von PostgreSQL erkannten Zeitzonenabkürzungen. PostgreSQL enthält interne Tabellen für die Entschlüsselung von Zeitzonen, da es keinen passenden Betriebssystemstandard für diese Aufgabe gibt. Die Zeitzoneninformationen im Betriebssystem werden jedoch bei der *Ausgabe* herangezogen.

Die Tabelle ist nach dem numerischen Unterschied der Zeitzone zu UTC sortiert, anstatt alphabetisch. Dadurch können Leser schnell eine erkannte Abkürzung für ihre lokale Zeitzone finden.

Zeitzone	Unterschied zu UTC	Beschreibung
NZDT	+13:00	New Zealand Daylight-Saving Time
IDLE	+12:00	Internationale Datumsgrenze, Ost
NZST	+12:00	New Zealand Standard Time
NZT	+12:00	New Zealand Time
AESST	+11:00	Australia Eastern Summer Standard Time
ACSST	+10:30	Central Australia Summer Standard Time
CADT	+10:30	Central Australia Daylight-Saving Time
SADT	+10:30	South Australian Daylight-Saving Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, Russland Zone 9
LIGT	+10:00	Melbourne, Australien
SAST	+09:30	South Australia Standard Time
CAST	+09:30	Central Australia Standard Time
AWSST	+09:00	Australia Western Summer Standard Time
JST	+09:00	Japan Standard Time, Russland Zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight-Saving Time
MT	+08:30	Moluccas Time
AWST	+08:00	Australia Western Standard Time
CCT	+08:00	China Coastal Time
WADT	+08:00	West Australian Daylight-Saving Time
WST	+08:00	West Australian Standard Time
JT	+07:30	Java Time
ALMST	+07:00	Almaty Summer Time
WAST	+07:00	West Australian Standard Time
CXT	+07:00	Christmas (Island) Time
MMT	+06:30	Myannar Time
ALMT	+06:00	Almaty Time
MAWT	+06:00	Mawson (Antarktis) Time

Tabelle A.4: Zeitzonenabkürzungen

Zeitzone	Unterschied zu UTC	Beschreibung
IOT	+05:00	Indian Chagos Time
MVT	+05:00	Maldives Island Time
TFT	+05:00	Kerguelen Time
AFT	+04:30	Afganistan Time
EAST	+04:00	Antananarivo Summer Time
MUT	+04:00	Mauritius Island Time
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IRT, IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight-Saving Time
HMT	+03:00	Hellas Mediterranean Time (?)
BDST	+02:00	British Double Standard Time
CEST	+02:00	Central European Summer Time
CETDST	+02:00	Central European Daylight-Saving Time
EET	+02:00	Eastern European Time, Russland Zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle European Summer Time
METDST	+02:00	Middle Europe Daylight-Saving Time
SST	+02:00	Swedish Summer Time
BST	+01:00	British Summer Time
CET	+01:00	Central European Time
DNT	+01:00	Dansk Normal Tid
FST	+01:00	French Summer Time
MET	+01:00	Middle European Time
MEWT	+01:00	Middle European Winter Time
MEZ	+01:00	Mitteleuropäische Zeit
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western European Daylight-Saving Time
GMT	00:00	Greenwich Mean Time
UT	00:00	Universal Time
UTC	00:00	Universal Coordinated Time
Z	00:00	gleich UTC

Tabelle A.4: Zeitzonenabkürzungen (Forts.)

Zeitzone	Unterschied zu UTC	Beschreibung
ZULU	00:00	gleich UTC
WET	00:00	Western European Time
WAT	-01:00	West Africa Time
NDT	-02:30	Newfoundland Daylight-Saving Time
ADT	-03:00	Atlantic Daylight-Saving Time
AWT	-03:00	(unbekannt)
NFT	-03:30	Newfoundland Standard Time
NST	-03:30	Newfoundland Standard Time
AST	-04:00	Atlantic Standard Time (Kanada)
ACST	-04:00	Atlantic/Porto Acre Summer Time
ACT	-05:00	Atlantic/Porto Acre Standard Time
EDT	-04:00	Eastern Daylight-Saving Time
CDT	-05:00	Central Daylight-Saving Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight-Saving Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight-Saving Time
AKDT	-08:00	Alaska Daylight-Saving Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight-Saving Time
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight-Saving Time
YST	-09:00	Yukon Standard Time
MART	-09:30	Marquesas Time
AHST	-10:00	Alaska/Hawaii Standard Time
HST	-10:00	Hawaii Standard Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	Internationale Datumsgrenze, West

Tabelle A.4: Zeitzonenabkürzungen (Forts.)

Australische Zeitzonen. Es gibt drei Namenskonflikte zwischen australischen Zeitzonennamen und Zeitzonennamen aus Nord- und Südamerika: `ACST`, `CST` und `EST`. Wenn der Konfigurationsparameter `australian_timezones` angestellt ist, werden `ACST`, `CST`, `EST` und `SAT` als australische Zeitzonennamen interpretiert, wie in Tabelle A.5 gezeigt. Wenn er aus ist (was die Standardeinstellung ist), werden

ACST, CST und EST als amerikanische Zeitzonen angenommen und SAT ist ein Füllwort, das für *Saturday* steht.

Zeitzone	Unterschied zu UTC	Beschreibung
ACST	+09:30	Central Australia Standard Time
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time
SAT	+09:30	South Australian Standard Time

Tabelle A.5: Australische Zeitzonenabkürzungen

A.3 Geschichte der Kalendersysteme

Das julianische Datum wurde vom französischen Gelehrten Joseph Justus Scaliger (1540–1609) erfunden und hat seinen Namen wahrscheinlich von Scaligers Vater, dem italienischen Gelehrten Julius Caesar Scaliger (1484–1558), erhalten. Astronomen verwenden ihn, um jedem Tag seit dem 1. Januar 4713 v.u.Z. eine eindeutige Nummer zuzuweisen. Das ist das so genannte julianische Datum (JD). JD 0 bezeichnet die 24 Stunden von Mittag UTC am 1. Januar 4713 v.u.Z bis Mittag UTC am 2. Januar 4713 v.u.Z.

Das "julianische Datum" ist etwas anderes als der "julianische Kalender". Der julianische Kalender wurde 45 v.u.Z. von Julius Cäsar eingeführt und war bis 1582 in allgemeiner Benutzung, als die Länder anfingen, zum gregorianischen Kalender zu wechseln. Im julianischen Kalender wird das tropische Jahr als 365 1/4 = 365,25 Tage angenommen. Daraus ergibt sich ein Fehler von etwa einem Tag in 128 Jahren.

Der sich anhäufende Fehler im Kalender veranlasste Papst Gregor XIII, den Kalender in Übereinstimmung mit den Anweisungen des Tridentiner Konzils zu reformieren. Im gregorianischen Kalender wird das tropische Jahr als 365 + 97 / 400 = 365,2424 Tage angenommen. Daher dauert es etwa 3300 Jahre, bis das tropische Jahr sich um einen Tag gegenüber dem gregorianischen Kalender verschiebt.

Die Annäherung 365 + 97 / 400 wird durch 97 Schaltjahre alle 400 Jahre erreicht, und zwar nach den folgenden Regeln:

Jedes durch 4 teilbare Jahr ist ein Schaltjahr.

Jedes durch 100 teilbare Jahr ist jedoch kein Schaltjahr.

Jedes durch 400 teilbare Jahr ist allerdings doch ein Schaltjahr.

Also sind 1700, 1800, 1900, 2100 und 2200 keine Schaltjahre. Aber 1600, 2000 und 2400 sind Schaltjahre. Im Gegensatz dazu waren im alten julianischen Kalender nur durch 4 teilbare Jahre Schaltjahre.

Die päpstliche Bulle vom Februar 1582 verfügte, dass 10 Tage vom Oktober 1582 ausgelassen werden sollten, sodass der 15. Oktober direkt auf den 4. Oktober folgen sollte. Befolgt wurde das in Italien, Polen, Portugal und Spanien. Andere katholische Länder folgten kurze Zeit später, aber protestantische Länder waren lange Zeit widerwillig, und griechisch-orthodoxe Länder änderten ihre Kalender erst am Anfang des 20. Jahrhunderts. In Großbritannien und den Dominions (einschließlich dem, was heute die USA ist) wurde die Reform 1752 umgesetzt. Auf den 2. September 1752 folgte also der 14. September 1752. Daher gibt auf Unix-Systemen das Programm `cal` Folgendes aus:

```
$ cal 9 1752
   September 1752
So Mo Di Mi Do Fr Sa
       1  2 14 15 16
```

```
   17 18 19 20 21 22 23
   24 25 26 27 28 29 30
```

Anmerkung

Der SQL-Standard bestimmt, dass innerhalb der Definition einer Datums-/Zeit-Konstante die Datums-/Zeit-Werte durch die natürlichen Regeln des gregorianischen Kalenders beschränkt werden. Daten zwischen dem 3.9.1752 und dem 13.9.1752 folgen den "natürlichen Regeln", trotzdem sie in einigen Ländern durch päpstliche Anordnung eliminiert wurden, und sind daher gültige Daten.

In verschiedenen Teilen der Welt wurden andere Kalender entwickelt, viele noch vor dem gregorianischen. Zum Beispiel können die Anfänge des chinesischen Kalenders bis in das 14. Jahrhundert v.u.Z. zurückverfolgt werden. Der Legende nach erfand der Kaiser Huandgi den Kalender 2637 v.u.Z. Die Volksrepublik China verwendet den gregorianischen Kalender für zivile Zwecke. Der chinesische Kalender wird zur Bestimmung von Festtagen verwendet.

SQL-Schlüsselwörter

Tabelle B.1 listet alle Tokens, die im SQL-Standard oder in PostgreSQL Schlüsselwörter sind. Hintergrundinformationen können in Abschnitt 4.1.1 gefunden werden.

SQL unterscheidet zwischen **reservierten** und **nicht reservierten** Schlüsselwörtern. Dem Standard nach sind die reservierten Schlüsselwörter die einzigen richtigen Schlüsselwörter; sie sind niemals als Namen zulässig. Nicht reservierte Schlüsselwörter haben ihre besondere Bedeutung nur in bestimmten Zusammenhängen und können in anderen Zusammenhängen als Namen verwendet werden. Die meisten nicht reservierten Schlüsselwörter sind eigentlich die Namen von eingebauten Tabellen oder Funktionen, die im SQL-Standard definiert sind. Der Begriff der nicht reservierten Schlüsselwörter besteht im Grunde genommen nur, um anzugeben, dass das Wort in manchen Zusammenhängen eine vordefinierte Bedeutung hat.

Im Parser von PostgreSQL ist das Leben etwas komplizierter. Es gibt mehrere verschiedene Klassen von Tokens, die sich zwischen denen bewegen, die niemals als Name zulässig sind, und denen, die im Parser verglichen mit normalen Namen überhaupt keine Sonderstellung haben. (Letzteres ist in der Regel bei vom SQL-Standard definierten Funktionen der Fall.) Sogar reservierte Schlüsselwörter sind in PostgreSQL nicht vollkommen reserviert, sondern können als Spaltenaliasnamen verwendet werden (zum Beispiel `SELECT 55 AS CHECK`, obwohl CHECK ein reserviertes Schlüsselwort ist).

In Tabelle B.1, in der Spalte für PostgreSQL, ordnen wir jene Schlüsselwörter, die dem Parser ausdrücklich bekannt sind, aber in den meisten oder allen Zusammenhängen, wo ein Name erwartet wird, zulässig sind, als “nicht reserviert” ein. Einige Schlüsselwörter, die ansonsten nicht reserviert sind, können nicht als Funktions- oder Datentypnamen verwendet werden und sind entsprechend gekennzeichnet. (Die meisten dieser Wörter stehen für eingebaute Funktionen oder Datentypen mit besonderer Syntax. Die Funktion oder der Typ ist nach wie vor verfügbar, aber kann vom Benutzer nicht umdefiniert werden.) Als “reserviert” sind jene Tokens bezeichnet, die nur als Spaltenaliasnamen mit “AS” (und womöglich in einigen wenigen anderen Zusammenhängen) erlaubt sind. Einige reservierte Schlüsselwörter sind als Namen von Funktionen zulässig; das wird auch in der Tabelle gezeigt.

Prinzipiell sollten Sie, wenn Sie seltsame Parsefehler in Befehlen sehen, die eines der gelisteten Schlüsselwörter enthält, versuchen, das Wort in Anführungszeichen einzuschließen, und sehen ,ob das Problem verschwindet.

Bevor Sie die Tabelle B.1 untersuchen, ist es ist wichtig, zu verstehen, dass die Tatsache, dass ein Schlüsselwort in PostgreSQL nicht reserviert ist, nicht bedeutet, dass das Feature, das mit dem Wort zusammen-

hängt, nicht vorhanden ist. Umgekehrt gilt auch, dass das Vorhandensein eines Schlüsselworts nicht das Vorhandensein eines Features anzeigt.

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
ABORT	nicht reserviert		
ABS		nicht reserviert	
ABSOLUTE	nicht reserviert	reserviert	reserviert
ACCESS	nicht reserviert		
ACTION	nicht reserviert	reserviert	reserviert
ADA		nicht reserviert	nicht reserviert
ADD	nicht reserviert	reserviert	reserviert
ADMIN		reserviert	
AFTER	nicht reserviert	reserviert	
AGGREGATE	nicht reserviert	reserviert	
ALIAS		reserviert	
ALL	reserviert	reserviert	reserviert
ALLOCATE		reserviert	reserviert
ALTER	nicht reserviert	reserviert	reserviert
ANALYSE	reserviert		
ANALYZE	reserviert		
AND	reserviert	reserviert	reserviert
ANY	reserviert	reserviert	reserviert
ARE		reserviert	reserviert
ARRAY		reserviert	
AS	reserviert	reserviert	reserviert
ASC	reserviert	reserviert	reserviert
ASENSITIVE		nicht reserviert	
ASSERTION	nicht reserviert	reserviert	reserviert
ASSIGNMENT	nicht reserviert	nicht reserviert	
ASYMMETRIC		nicht reserviert	
AT	nicht reserviert	reserviert	reserviert
ATOMIC		nicht reserviert	
AUTHORIZATION	reserviert (darf Funktion sein)	reserviert	reserviert
AVG		nicht reserviert	reserviert
BACKWARD	nicht reserviert		
BEFORE	nicht reserviert	reserviert	
BEGIN	nicht reserviert	reserviert	reserviert
BETWEEN	reserviert (darf Funktion sein)	nicht reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
BIGINT	nicht reserviert (darf keine Funktion oder Typ sein)		
BINARY	reserviert (darf Funktion sein)	reserviert	
BIT	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
BITVAR		nicht reserviert	
BIT_LENGTH		nicht reserviert	reserviert
BLOB		reserviert	
BOOLEAN	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	
BOTH	reserviert	reserviert	reserviert
BREADTH		reserviert	
BY	nicht reserviert	reserviert	reserviert
C		nicht reserviert	nicht reserviert
CACHE	nicht reserviert		
CALL		reserviert	
CALLED	nicht reserviert	nicht reserviert	
CARDINALITY		nicht reserviert	
CASCADE	nicht reserviert	reserviert	reserviert
CASCADED		reserviert	reserviert
CASE	reserviert	reserviert	reserviert
CAST	reserviert	reserviert	reserviert
CATALOG		reserviert	reserviert
CATALOG_NAME		nicht reserviert	nicht reserviert
CHAIN	nicht reserviert	nicht reserviert	
CHAR	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
CHARACTER	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
CHARACTERISTICS	nicht reserviert		
CHARACTER_LENGTH		nicht reserviert	reserviert
CHARACTER_SET_CATALOG		nicht reserviert	nicht reserviert
CHARACTER_SET_NAME		nicht reserviert	nicht reserviert
CHARACTER_SET_SCHEMA		nicht reserviert	nicht reserviert
CHAR_LENGTH		nicht reserviert	reserviert
CHECK	reserviert	reserviert	reserviert
CHECKED		nicht reserviert	
CHECKPOINT	nicht reserviert		
CLASS	nicht reserviert	reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
CLASS_ORIGIN		nicht reserviert	nicht reserviert
CLOB		reserviert	
CLOSE	nicht reserviert	reserviert	reserviert
CLUSTER	nicht reserviert		
COALESCE	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
COBOL		nicht reserviert	nicht reserviert
COLLATE	reserviert	reserviert	reserviert
COLLATION		reserviert	reserviert
COLLATION_CATALOG		nicht reserviert	nicht reserviert
COLLATION_NAME		nicht reserviert	nicht reserviert
COLLATION_SCHEMA		nicht reserviert	nicht reserviert
COLUMN	reserviert	reserviert	reserviert
COLUMN_NAME		nicht reserviert	nicht reserviert
COMMAND_FUNCTION		nicht reserviert	nicht reserviert
COMMAND_FUNCTION_CODE		nicht reserviert	
COMMENT	nicht reserviert		
COMMIT	nicht reserviert	reserviert	reserviert
COMMITTED	nicht reserviert	nicht reserviert	nicht reserviert
COMPLETION		reserviert	
CONDITION_NUMBER		nicht reserviert	nicht reserviert
CONNECT		reserviert	reserviert
CONNECTION		reserviert	reserviert
CONNECTION_NAME		nicht reserviert	nicht reserviert
CONSTRAINT	reserviert	reserviert	reserviert
CONSTRAINTS	nicht reserviert	reserviert	reserviert
CONSTRAINT_CATALOG		nicht reserviert	nicht reserviert
CONSTRAINT_NAME		nicht reserviert	nicht reserviert
CONSTRAINT_SCHEMA		nicht reserviert	nicht reserviert
CONSTRUCTOR		reserviert	
CONTAINS		nicht reserviert	
CONTINUE		reserviert	reserviert
CONVERSION	nicht reserviert		
CONVERT	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
COPY	nicht reserviert		
CORRESPONDING		reserviert	reserviert
COUNT		nicht reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
CREATE	reserviert	reserviert	reserviert
CREATEDB	nicht reserviert		
CREATEUSER	nicht reserviert		
CROSS	reserviert (darf Funktion sein)	reserviert	reserviert
CUBE		reserviert	
CURRENT		reserviert	reserviert
CURRENT_DATE	reserviert	reserviert	reserviert
CURRENT_PATH		reserviert	
CURRENT_ROLE		reserviert	
CURRENT_TIME	reserviert	reserviert	reserviert
CURRENT_TIMESTAMP	reserviert	reserviert	reserviert
CURRENT_USER	reserviert	reserviert	reserviert
CURSOR	nicht reserviert	reserviert	reserviert
CURSOR_NAME		nicht reserviert	nicht reserviert
CYCLE	nicht reserviert	reserviert	
DATA		reserviert	nicht reserviert
DATABASE	nicht reserviert		
DATE		reserviert	reserviert
DATETIME_INTERVAL_CODE		nicht reserviert	nicht reserviert
DATETIME_INTERVAL_PRECISION		nicht reserviert	nicht reserviert
DAY	nicht reserviert	reserviert	reserviert
DEALLOCATE	nicht reserviert	reserviert	reserviert
DEC	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
DECIMAL	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
DECLARE	nicht reserviert	reserviert	reserviert
DEFAULT	reserviert	reserviert	reserviert
DEFERRABLE	reserviert	reserviert	reserviert
DEFERRED	nicht reserviert	reserviert	reserviert
DEFINED		nicht reserviert	
DEFINER	nicht reserviert	nicht reserviert	
DELETE	nicht reserviert	reserviert	reserviert
DELIMITER	nicht reserviert		
DELIMITERS	nicht reserviert		
DEPTH		reserviert	
DEREF		reserviert	
DESC	reserviert	reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
DESCRIBE		reserviert	reserviert
DESCRIPTOR		reserviert	reserviert
DESTROY		reserviert	
DESTRUCTOR		reserviert	
DETERMINISTIC		reserviert	
DIAGNOSTICS		reserviert	reserviert
DICTIONARY		reserviert	
DISCONNECT		reserviert	reserviert
DISPATCH		nicht reserviert	
DISTINCT	reserviert	reserviert	reserviert
DO	reserviert		
DOMAIN	nicht reserviert	reserviert	reserviert
DOUBLE	nicht reserviert	reserviert	reserviert
DROP	nicht reserviert	reserviert	reserviert
DYNAMIC		reserviert	
DYNAMIC_FUNCTION		nicht reserviert	nicht reserviert
DYNAMIC_FUNCTION_CODE		nicht reserviert	
EACH	nicht reserviert	reserviert	
ELSE	reserviert	reserviert	reserviert
ENCODING	nicht reserviert		
ENCRYPTED	nicht reserviert		
END	reserviert	reserviert	reserviert
END-EXEC		reserviert	reserviert
EQUALS		reserviert	
ESCAPE	nicht reserviert	reserviert	reserviert
EVERY		reserviert	
EXCEPT	reserviert	reserviert	reserviert
EXCEPTION		reserviert	reserviert
EXCLUSIVE	nicht reserviert		
EXEC		reserviert	reserviert
EXECUTE	nicht reserviert	reserviert	reserviert
EXISTING		nicht reserviert	
EXISTS	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
EXPLAIN	nicht reserviert		
EXTERNAL	nicht reserviert	reserviert	reserviert
EXTRACT	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
FALSE	reserviert	reserviert	reserviert
FETCH	nicht reserviert	reserviert	reserviert
FINAL		nicht reserviert	
FIRST		reserviert	reserviert
FLOAT	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
FOR	reserviert	reserviert	reserviert
FORCE	nicht reserviert		
FOREIGN	reserviert	reserviert	reserviert
FORTRAN		nicht reserviert	nicht reserviert
FORWARD	nicht reserviert		
FOUND		reserviert	reserviert
FREE		reserviert	
FREEZE	reserviert (darf Funktion sein)		
FROM	reserviert	reserviert	reserviert
FULL	reserviert (darf Funktion sein)	reserviert	reserviert
FUNCTION	nicht reserviert	reserviert	
G		nicht reserviert	
GENERAL		reserviert	
GENERATED		nicht reserviert	
GET	nicht reserviert	reserviert	reserviert
GLOBAL	nicht reserviert	reserviert	reserviert
GO		reserviert	reserviert
GOTO		reserviert	reserviert
GRANT	reserviert	reserviert	reserviert
GRANTED		nicht reserviert	
GROUP	reserviert	reserviert	reserviert
GROUPING		reserviert	
HANDLER	nicht reserviert		
HAVING	reserviert	reserviert	reserviert
HIERARCHY		nicht reserviert	
HOLD		nicht reserviert	
HOST		reserviert	
HOUR	nicht reserviert	reserviert	reserviert
IDENTITY		reserviert	reserviert
IGNORE		reserviert	
ILIKE	reserviert (darf Funktion sein)		

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
IMMEDIATE	nicht reserviert	reserviert	reserviert
IMMUTABLE	nicht reserviert		
IMPLEMENTATION		nicht reserviert	
IMPLICIT	nicht reserviert		
IN	reserviert (darf Funktion sein)	reserviert	reserviert
INCREMENT	nicht reserviert		
INDEX	nicht reserviert		
INDICATOR		reserviert	reserviert
INFIX		nicht reserviert	
INHERITS	nicht reserviert		
INITIALIZE		reserviert	
INITIALLY	reserviert	reserviert	reserviert
INNER	reserviert (darf Funktion sein)	reserviert	reserviert
INOUT	nicht reserviert	reserviert	
INPUT	nicht reserviert	reserviert	reserviert
INSENSITIVE	nicht reserviert	nicht reserviert	reserviert
INSERT	nicht reserviert	reserviert	reserviert
INSTANCE		nicht reserviert	
INSTANTIABLE		nicht reserviert	
INSTEAD	nicht reserviert		
INT	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
INTEGER	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
INTERSECT	reserviert	reserviert	reserviert
INTERVAL	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
INTO	reserviert	reserviert	reserviert
INVOKER	nicht reserviert	nicht reserviert	
IS	reserviert (darf Funktion sein)	reserviert	reserviert
ISNULL	reserviert (darf Funktion sein)		
ISOLATION	nicht reserviert	reserviert	reserviert
ITERATE		reserviert	
JOIN	reserviert (darf Funktion sein)	reserviert	reserviert
K		nicht reserviert	
KEY	nicht reserviert	reserviert	reserviert
KEY_MEMBER		nicht reserviert	
KEY_TYPE		nicht reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
LANCOMPILER	nicht reserviert		
LANGUAGE	nicht reserviert	reserviert	reserviert
LARGE		reserviert	
LAST		reserviert	reserviert
LATERAL		reserviert	
LEADING	reserviert	reserviert	reserviert
LEFT	reserviert (darf Funktion sein)	reserviert	reserviert
LENGTH		nicht reserviert	nicht reserviert
LESS		reserviert	
LEVEL	nicht reserviert	reserviert	reserviert
LIKE	reserviert (darf Funktion sein)	reserviert	reserviert
LIMIT	reserviert	reserviert	
LISTEN	nicht reserviert		
LOAD	nicht reserviert		
LOCAL	nicht reserviert	reserviert	reserviert
LOCALTIME	reserviert	reserviert	
LOCALTIMESTAMP	reserviert	reserviert	
LOCATION	nicht reserviert		
LOCATOR		reserviert	
LOCK	nicht reserviert		
LOWER		nicht reserviert	reserviert
M		nicht reserviert	
MAP		reserviert	
MATCH	nicht reserviert	reserviert	reserviert
MAX		nicht reserviert	reserviert
MAXVALUE	nicht reserviert		
MESSAGE_LENGTH		nicht reserviert	nicht reserviert
MESSAGE_OCTET_LENGTH		nicht reserviert	nicht reserviert
MESSAGE_TEXT		nicht reserviert	nicht reserviert
METHOD		nicht reserviert	
MIN		nicht reserviert	reserviert
MINUTE	nicht reserviert	reserviert	reserviert
MINVALUE	nicht reserviert		
MOD		nicht reserviert	
MODE	nicht reserviert		
MODIFIES		reserviert	
MODIFY		reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
MODULE		reserviert	reserviert
MONTH	nicht reserviert	reserviert	reserviert
MORE		nicht reserviert	nicht reserviert
MOVE	nicht reserviert		
MUMPS		nicht reserviert	nicht reserviert
NAME		nicht reserviert	nicht reserviert
NAMES	nicht reserviert	reserviert	reserviert
NATIONAL	nicht reserviert	reserviert	reserviert
NATURAL	reserviert (darf Funktion sein)	reserviert	reserviert
NCHAR	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
NCLOB		reserviert	
NEW	reserviert	reserviert	
NEXT	nicht reserviert	reserviert	reserviert
NO	nicht reserviert	reserviert	reserviert
NOCREATEDB	nicht reserviert		
NOCREATEUSER	nicht reserviert		
NONE	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	
NOT	reserviert	reserviert	reserviert
NOTHING	nicht reserviert		
NOTIFY	nicht reserviert		
NOTNULL	reserviert (darf Funktion sein)		
NULL	reserviert	reserviert	reserviert
NULLABLE		nicht reserviert	nicht reserviert
NULLIF	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
NUMBER		nicht reserviert	nicht reserviert
NUMERIC	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
OBJECT		reserviert	
OCTET_LENGTH		nicht reserviert	reserviert
OF	nicht reserviert	reserviert	reserviert
OFF	reserviert	reserviert	
OFFSET	reserviert		
OIDS	nicht reserviert		
OLD	reserviert	reserviert	
ON	reserviert	reserviert	reserviert
ONLY	reserviert	reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
OPEN		reserviert	reserviert
OPERATION		reserviert	
OPERATOR	nicht reserviert		
OPTION	nicht reserviert	reserviert	reserviert
OPTIONS		nicht reserviert	
OR	reserviert	reserviert	reserviert
ORDER	reserviert	reserviert	reserviert
ORDINALITY		reserviert	
OUT	nicht reserviert	reserviert	
OUTER	reserviert (darf Funktion sein)	reserviert	reserviert
OUTPUT		reserviert	reserviert
OVERLAPS	reserviert (darf Funktion sein)	nicht reserviert	reserviert
OVERLAY	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	
OVERRIDING		nicht reserviert	
OWNER	nicht reserviert		
PAD		reserviert	reserviert
PARAMETER		reserviert	
PARAMETERS		reserviert	
PARAMETER_MODE		nicht reserviert	
PARAMETER_NAME		nicht reserviert	
PARAMETER_ORDINAL_POSITION		nicht reserviert	
PARAMETER_SPECIFIC_CATALOG		nicht reserviert	
PARAMETER_SPECIFIC_NAME		nicht reserviert	
PARAMETER_SPECIFIC_SCHEMA		nicht reserviert	
PARTIAL	nicht reserviert	reserviert	reserviert
PASCAL		nicht reserviert	nicht reserviert
PASSWORD	nicht reserviert		
PATH	nicht reserviert	reserviert	
PENDANT	nicht reserviert		
PLACING	reserviert		
PLI		nicht reserviert	nicht reserviert
POSITION	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
POSTFIX		reserviert	
PRECISION	nicht reserviert	reserviert	reserviert
PREFIX		reserviert	
PREORDER		reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
PREPARE	nicht reserviert	reserviert	reserviert
PRESERVE		reserviert	reserviert
PRIMARY	reserviert	reserviert	reserviert
PRIOR	nicht reserviert	reserviert	reserviert
PRIVILEGES	nicht reserviert	reserviert	reserviert
PROCEDURAL	nicht reserviert		
PROCEDURE	nicht reserviert	reserviert	reserviert
PUBLIC		reserviert	reserviert
READ	nicht reserviert	reserviert	reserviert
READS		reserviert	
REAL	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
RECHECK	nicht reserviert		
RECURSIVE		reserviert	
REF		reserviert	
REFERENCES	reserviert	reserviert	reserviert
REFERENCING		reserviert	
REINDEX	nicht reserviert		
RELATIVE	nicht reserviert	reserviert	reserviert
RENAME	nicht reserviert		
REPEATABLE		nicht reserviert	nicht reserviert
REPLACE	nicht reserviert		
RESET	nicht reserviert		
RESTRICT	nicht reserviert	reserviert	reserviert
RESULT		reserviert	
RETURN		reserviert	
RETURNED_LENGTH		nicht reserviert	nicht reserviert
RETURNED_OCTET_LENGTH		nicht reserviert	nicht reserviert
RETURNED_SQLSTATE		nicht reserviert	nicht reserviert
RETURNS	nicht reserviert	reserviert	
REVOKE	nicht reserviert	reserviert	reserviert
RIGHT	reserviert (darf Funktion sein)	reserviert	reserviert
ROLE		reserviert	
ROLLBACK	nicht reserviert	reserviert	reserviert
ROLLUP		reserviert	
ROUTINE		reserviert	
ROUTINE_CATALOG		nicht reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
ROUTINE_NAME		nicht reserviert	
ROUTINE_SCHEMA		nicht reserviert	
ROW	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	
ROWS		reserviert	reserviert
ROW_COUNT		nicht reserviert	nicht reserviert
RULE	nicht reserviert		
SAVEPOINT		reserviert	
SCALE		nicht reserviert	nicht reserviert
SCHEMA	nicht reserviert	reserviert	reserviert
SCHEMA_NAME		nicht reserviert	nicht reserviert
SCOPE		reserviert	
SCROLL	nicht reserviert	reserviert	reserviert
SEARCH		reserviert	
SECOND	nicht reserviert	reserviert	reserviert
SECTION		reserviert	reserviert
SECURITY	nicht reserviert	nicht reserviert	
SELECT	reserviert	reserviert	reserviert
SELF		nicht reserviert	
SENSITIVE		nicht reserviert	
SEQUENCE	nicht reserviert	reserviert	
SERIALIZABLE	nicht reserviert	nicht reserviert	nicht reserviert
SERVER_NAME		nicht reserviert	nicht reserviert
SESSION	nicht reserviert	reserviert	reserviert
SESSION_USER	reserviert	reserviert	reserviert
SET	nicht reserviert	reserviert	reserviert
SETOF	nicht reserviert (darf keine Funktion oder Typ sein)		
SETS		reserviert	
SHARE	nicht reserviert		
SHOW	nicht reserviert		
SIMILAR	reserviert (darf Funktion sein)	nicht reserviert	
SIMPLE	nicht reserviert	nicht reserviert	
SIZE		reserviert	reserviert
SMALLINT	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
SOME	reserviert	reserviert	reserviert
SOURCE		nicht reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
SPACE		reserviert	reserviert
SPECIFIC		reserviert	
SPECIFICTYPE		reserviert	
SPECIFIC_NAME		nicht reserviert	
SQL		reserviert	reserviert
SQLCODE			reserviert
SQLERROR			reserviert
SQLEXCEPTION		reserviert	
SQLSTATE		reserviert	reserviert
SQLWARNING		reserviert	
STABLE	nicht reserviert		
START	nicht reserviert	reserviert	
STATE		reserviert	
STATEMENT	nicht reserviert	reserviert	
STATIC		reserviert	
STATISTICS	nicht reserviert		
STDIN	nicht reserviert		
STDOUT	nicht reserviert		
STORAGE	nicht reserviert		
STRICT	nicht reserviert		
STRUCTURE		reserviert	
STYLE		nicht reserviert	
SUBCLASS_ORIGIN		nicht reserviert	nicht reserviert
SUBLIST		nicht reserviert	
SUBSTRING	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
SUM		nicht reserviert	reserviert
SYMMETRIC		nicht reserviert	
SYSID	nicht reserviert		
SYSTEM		nicht reserviert	
SYSTEM_USER		reserviert	reserviert
TABLE	reserviert	reserviert	reserviert
TABLE_NAME		nicht reserviert	nicht reserviert
TEMP	nicht reserviert		
TEMPLATE	nicht reserviert		
TEMPORARY	nicht reserviert	reserviert	reserviert
TERMINATE		reserviert	

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
THAN		reserviert	
THEN	reserviert	reserviert	reserviert
TIME	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
TIMESTAMP	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
TIMEZONE_HOUR		reserviert	reserviert
TIMEZONE_MINUTE		reserviert	reserviert
TO	reserviert	reserviert	reserviert
TOAST	nicht reserviert		
TRAILING	reserviert	reserviert	reserviert
TRANSACTION	nicht reserviert	reserviert	reserviert
TRANSACTIONS_COMMITTED		nicht reserviert	
TRANSACTIONS_ROLLED_BACK		nicht reserviert	
TRANSACTION_ACTIVE		nicht reserviert	
TRANSFORM		nicht reserviert	
TRANSFORMS		nicht reserviert	
TRANSLATE		nicht reserviert	reserviert
TRANSLATION		reserviert	reserviert
TREAT	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	
TRIGGER	nicht reserviert	reserviert	
TRIGGER_CATALOG		nicht reserviert	
TRIGGER_NAME		nicht reserviert	
TRIGGER_SCHEMA		nicht reserviert	
TRIM	nicht reserviert (darf keine Funktion oder Typ sein)	nicht reserviert	reserviert
TRUE	reserviert	reserviert	reserviert
TRUNCATE	nicht reserviert		
TRUSTED	nicht reserviert		
TYPE	nicht reserviert	nicht reserviert	nicht reserviert
UNCOMMITTED		nicht reserviert	nicht reserviert
UNDER		reserviert	
UNENCRYPTED	nicht reserviert		
UNION	reserviert	reserviert	reserviert
UNIQUE	reserviert	reserviert	reserviert
UNKNOWN	nicht reserviert	reserviert	reserviert
UNLISTEN	nicht reserviert		
UNNAMED		nicht reserviert	nicht reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

Schlüsselwort	PostgreSQL	SQL 99	SQL 92
UNNEST		reserviert	
UNTIL	nicht reserviert		
UPDATE	nicht reserviert	reserviert	reserviert
UPPER		nicht reserviert	reserviert
USAGE	nicht reserviert	reserviert	reserviert
USER	reserviert	reserviert	reserviert
USER_DEFINED_TYPE_CATALOG		nicht reserviert	
USER_DEFINED_TYPE_NAME		nicht reserviert	
USER_DEFINED_TYPE_SCHEMA		nicht reserviert	
USING	reserviert	reserviert	reserviert
VACUUM	nicht reserviert		
VALID	nicht reserviert		
VALIDATOR	nicht reserviert		
VALUE		reserviert	reserviert
VALUES	nicht reserviert	reserviert	reserviert
VARCHAR	nicht reserviert (darf keine Funktion oder Typ sein)	reserviert	reserviert
VARIABLE		reserviert	
VARYING	nicht reserviert	reserviert	reserviert
VERBOSE	reserviert (darf Funktion sein)		
VERSION	nicht reserviert		
VIEW	nicht reserviert	reserviert	reserviert
VOLATILE	nicht reserviert		
WHEN	reserviert	reserviert	reserviert
WHENEVER		reserviert	reserviert
WHERE	reserviert	reserviert	reserviert
WITH	nicht reserviert	reserviert	reserviert
WITHOUT	nicht reserviert	reserviert	
WORK	nicht reserviert	reserviert	reserviert
WRITE	nicht reserviert	reserviert	reserviert
YEAR	nicht reserviert	reserviert	reserviert
ZONE	nicht reserviert	reserviert	reserviert

Tabelle B.1: SQL-Schlüsselwörter (Forts.)

SQL-Konformität

Dieser Abschnitt wird versuchen zu umreißen, inwiefern PostgreSQL mit dem SQL-Standard konform ist. Volle Konformität mit dem Standard oder ein vollständige Aussage über die Konformität mit dem Standard ist kompliziert und nicht besonders nützlich. Daher kann dieser Abschnitt nur einen Überblick liefern.

Der formelle Name des SQL-Standards ist ISO/IEC 9075 *Database Language SQL*. Von Zeit zu Zeit wird eine überarbeitete Version des Standards herausgegeben; die jüngste kam 1999 heraus. Diese Version wird als ISO/IEC 9075:1999 bezeichnet, oder umgangssprachlich als SQL99. Die Version davor war SQL92. Die Entwicklung von PostgreSQL hat in der Regel die Konformität mit der neuesten offiziellen Version des Standards als Ziel, soweit diese Konformität keinen traditionellen Funktionsmerkmalen oder dem gesunden Menschenverstand widerspricht. Zum Zeitpunkt, als dies hier geschrieben wird, ist die Abstimmung über eine neue Version des Standards im Gange, welche, falls angenommen, schließlich das Konformitätsziel zukünftiger PostgreSQL-Entwicklungen werden wird.

SQL92 definierte für die Konformität drei Mengen von Leistungsmerkmalen: *Entry* (einfach), *Intermediate* (fortgeschritten) und *Full* (voll). Die meisten Datenbankprodukte, die behaupteten, mit dem SQL-Standard konform zu sein, waren nur auf dem einfachen Niveau konform, da die Gesamtmenge von Leistungsmerkmalen auf dem fortgeschrittenen und vollen Niveau entweder zu umfangreich war oder im Widerspruch zum bisherigen Verhalten stand.

SQL99 definiert eine große Zahl individueller Leistungsmerkmale anstelle der ineffektiven, zu breit gefassten drei Niveaus in SQL92. Eine große Teilmenge dieser Leistungsmerkmale stellen die Kernmerkmale (*core features*) dar, welche jede konforme SQL-Implementierung anbieten muss. Der Rest der Leitungsmerkmale ist vollkommen wahlfrei. Einige wahlfreie Leistungsmerkmale sind in Gruppen zusammengefasst, die “Paket” (*package*) heißen. Eine SQL-Implementierung kann Konformität mit bestimmten Paketen angeben und gibt damit die Konformität mit bestimmten Gruppen von Leistungsmerkmalen an.

Der SQL99-Standard wurde außerdem in fünf Teile aufgespalten: *Framework*, *Foundation*, *Call Level Interface*, *Persistent Stored Modules* und *Host Language Bindings*. PostgreSQL deckt nur die Teile 1, 2 und 5 ab. Teil 3 ist vergleichbar mit der ODBC-Schnittstelle und Teil 4 ist vergleichbar mit der Programmiersprache PL/pgSQL, aber genaue Konformität ist in beiden Fällen nicht beabsichtigt.

In den folgenden beiden Unterabschnitten zeigen wir die Liste der Leistungsmerkmale, die PostgreSQL unterstützt, gefolgt von der Liste von Leistungsmerkmalen, die in SQL99 definiert sind, aber von PostgreSQL noch nicht unterstützt werden. Beide Listen sind Annäherungen: Es mag kleinere Einzelheiten geben, die bei einem als unterstützt gelisteten Leistungsmerkmal nicht konform sind, und große Teile eines nicht unterstützten Leistungsmerkmals könnten tatsächlich schon implementiert sein. Der Hauptteil der Dokumentation enthält immer die genauesten Informationen über was funktioniert und was nicht.

Anmerkung

Die Codes von Leistungsmerkmalen, die Bindestriche enthalten, sind untergeordnete Merkmale. Wenn ein bestimmtes untergeordnetes Merkmal nicht unterstützt wird, dann wird daher das übergeordnete Merkmal auch als nicht unterstützt aufgelistet, selbst wenn andere untergeordnete Merkmale unterstützt werden.

C.1 Unterstützte Leistungsmerkmale

Kennung	Paket	Beschreibung	Kommentar
B012	Core	Eingebettetes C	
B021		Direktes SQL	
E011	Core	Numerische Datentypen	
E011-01	Core	INTEGER und SMALLINT Datentypen	
E011-02	Core	REAL, DOUBLE PRECISON und FLOAT Datentypen	
E011-03	Core	DECIMAL und NUMERIC Datentypen	
E011-04	Core	Arithmetische Operatoren	
E011-05	Core	Numerische Vergleiche	
E011-06	Core	Implizite Umwandlung zwischen den numerischen Datentypen	
E021	Core	Zeichendatentypen	
E021-01	Core	CHARACTER Datentyp	
E021-02	Core	CHARACTER VARYING Datentyp	
E021-03	Core	Zeichenkonstanten	
E021-04	Core	CHARACTER_LENGTH Funktion	
E021-05	Core	OCTET_LENGTH Funktion	
E021-06	Core	SUBSTRING Funktion	
E021-07	Core	Zeichenverknüpfung	
E021-08	Core	UPPER und LOWER Funktionen	
E021-09	Core	TRIM Funktion	
E021-10	Core	Implizite Umwandlung zwischen den Zeichendatentypen	
E021-11	Core	POSITION Funktion	
E011-12	Core	Zeichenvergleiche	
E031	Core	Bezeichner	
E031-01	Core	Bezeichner in Anführungszeichen	
E031-02	Core	Bezeichner in Kleinbuchstaben	
E031-03	Core	Abschließender Unterstrich	
E051	Core	Grundlegende Anfrage	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY-Klausel	

Kennung	Paket	Beschreibung	Kommentar
E051-04	Core	GROUP BY kann Spalten enthalten, die nicht in der Select-Liste sind	
E051-05	Core	Select-Listen-Elemente können umbenannt werden	AS ist erforderlich
E051-06	Core	HAVING-Klausel	
E051-07	Core	Qualifizierter * in Select-Liste	
E051-08	Core	Korrelationsnamen in der FROM-Klausel	
E051-09	Core	Spalten in der FROM-Klausel umbenennen	
E061	Core	Grundlegende Prädikate und Suchbedingungen	
E061-01	Core	Vergleichsprädikat	
E061-02	Core	BETWEEN-Prädikat	
E061-03	Core	IN-Prädikat mit Wertliste	
E061-04	Core	LIKE-Prädikat	
E061-05	Core	LIKE-Prädikat ESCAPE-Klausel	
E061-06	Core	NULL-Prädikat	
E061-07	Core	Quantifiziertes Vergleichsprädikat	
E061-08	Core	EXISTS-Prädikat	
E061-09	Core	Unteranfragen in Vergleichsprädikat	
E061-11	Core	Unteranfragen in IN-Prädikat	
E061-12	Core	Unteranfragen in quantifiziertem Vergleichsprädikat	
E061-13	Core	Korrelativierte Unteranfragen	
E061-14	Core	Suchbedingung	
E071	Core	Grundlegende Anfrageausdrücke	
E071-01	Core	UNION-DISTINCT-Tabellenoperator	
E071-02	Core	UNION-ALL-Tabellenoperator	
E071-03	Core	EXCEPT-DISTINCT-Tabellenoperator	
E071-05	Core	Mit Tabellenoperatoren verbundene Spalten müssen nicht genau den gleichen Datentyp haben	
E071-06	Core	Tabellenoperatoren in Unteranfragen	
E081-01	Core	SELECT-Privileg	
E081-02	Core	DELETE-Privileg	
E081-03	Core	INSERT-Privileg auf Tabellenebene	
E081-04	Core	UPDATE-Privileg auf Tabellenebene	
E081-06	Core	REFERENCES-Privileg auf Tabellenebene	
E091	Core	Mengenfunktionen	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	

Kennung	Paket	Beschreibung	Kommentar
E091-05	Core	SUM	
E091-06	Core	ALL-Quantifizierung	
E091-07	Core	DISTINCT-Quantifizierung	
E101	Core	Grundlegende Datenmanipulation	
E101-01	Core	INSERT-Befehl	
E101-03	Core	Suchender UPDATE-Befehl	
E101-04	Core	Suchender DELETE-Befehl	
E111	Core	Einzelzeilen-SELECT-Befehl	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER-BY-Spalten müssen nicht in Select-Liste sein	
E121-03	Core	Wertausdrücke in ORDER-BY-Klausel	
E121-08	Core	CLOSE-Befehl	
E121-10	Core	FETCH-Befehl implizitem NEXT	
E131	Core	NULL-Wert-Unterstützung (NULL-Werte anstelle normaler Werte)	
E141	Core	Grundlegende Integritäts-Constraints	
E141-01	Core	NOT-NULL-Constraints	
E141-02	Core	UNIQUE-Constraints von NOT-NULL-Spalten	
E141-03	Core	PRIMARY KEY Constraints	
E141-04	Core	Grundlegender FOREIGN KEY Constraint mit NO ACTION als Vorgabe für die Lösch- und Aktualisierungsaktionen	
E141-06	Core	CHECK-Constraints	
E141-07	Core	Spaltenvorgabewerte	
E141-08	Core	NOT-NULL in PRIMARY-KEY enthalten	
E141-10	Core	Namen im Fremdschlüssel können in beliebiger Reihenfolge angegeben werden	
E151	Core	Transaktionsunterstützung	
E151-01	Core	COMMIT-Befehl	
E151-02	Core	ROLLBACK-Befehl	
E152-01	Core	SET-TRANSACTION-Befehl: ISOLATION-LEVEL-SERIALIZABLE-Klausel	
E161	Core	SQL-Kommentare mit doppeltem Minuszeichen	
F031	Core	Grundlegende Schemamanipulation	
F031-01	Core	CREATE-TABLE-Befehl zur Erzeugung von persistenten Basistabellen	
F031-02	Core	CREATE-VIEW-Befehl	
F031-03	Core	GRANT-Befehl	
F031-04	Core	ALTER-TABLE-Befehl: ADD COLUMN Klausel	
F031-13	Core	DROP-TABLE-Befehl: RESTRICT-Klausel	
F031-16	Core	DROP-VIEW-Befehl: RESTRICT-Klausel	

Kennung	Paket	Beschreibung	Kommentar
F032		CASCADE-Verhalten bei DROP	
F033		ALTER-TABLE Befehl: DROP-COLUMN-Klausel	
F041	Core	Grundlegende verbundene Tabelle	
F041-01	Core	Innerer Verbund (aber nicht unbedingt das INNER-Schlüsselwort)	
F041-02	Core	INNER-Schlüsselwort	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Äußere Verbunde können geschachtelt werden	
F041-07	Core	Die innere Tabelle in einem linken oder rechten äußeren Verbund kann auch in einem inneren Verbund verwendet werden	
F041-08	Core	Alle Vergleichsoperationen werden unterstützt (anstatt nur =)	
F051	Core	Grundlegende Datum/Zeit-Unterstützung	
F051-01	Core	DATE-Datentyp (einschließlich Unterstützung für DATE-Konstante)	
F051-02	Core	TIME-Datentyp (einschließlich Unterstützung für TIME-Konstante) mit Bruchsekundengenauigkeit von mindestens 0	
F051-03	Core	TIMESTAMP-Datentyp (einschließlich Unterstützung für TIMESTAMP-Konstante) mit Bruchsekundengenauigkeit von mindestens 0 und 6	
F051-04	Core	Vergleichsprädikat für DATE-, TIME- und TIMESTAMP-Datentypen	
F051-05	Core	Ausdrückliches CAST zwischen Datum/Zeit-Typen und Zeichentypen	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Verbesserte Datum/Zeit-Fähigkeiten	Arithmetik mit Zeitspannen, Datum, Zeit	
F081	Core	UNION und EXCEPT in Sichten	
F111-02		READ-COMMITTED -Isolationsgrad	
F131	Core	Gruppierte Operationen	
F131-01	Core	WHERE-, GROUP-BY- und HAVING-Klauseln in Anfragen mit gruppierten Sichten unterstützt	
F131-02	Core	Mehrere Tabellen in Anfragen mit gruppierten Sichten unterstützt	
F131-03	Core	Mengenfunktionen in Anfragen mit gruppierten Sichten unterstützt	
F131-04	Core	Unteranfragen mit GROUP-BY und HAVING-Klauseln und gruppierten Sichten	
F131-05	Core	Einzelzeiliges SELECT mit GROUP-BY und HAVING-Klauseln und gruppierten Sichten	

Kennung	Paket	Beschreibung	Kommentar
F171		Mehrere Schemas pro Benutzer	
F191	Verbesserte Integritätsverwaltung	Fremdschlüssel-Löschaktionen	
F201	Core	CAST-Funktion	
F221	Core	Ausdrückliche Vorgabewerte	
F222		INSERT-Befehl: DEFAULT-VALUES-Klausel	
F251		Domain-Unterstützung	
F261	Core	CASE-Ausdruck	
F261-01	Core	Einfaches CASE	
F261-02	Core	Suchendes CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F271		Zusammengesetzte Zeichenkonstanten	
F281		LIKE-Verbesserungen	
F302	OLAP-Fähigkeiten	INTERSECT Tabellenoperator	
F302-01	OLAP-Fähigkeiten	INTERSECT-DISTINCT-Tabellenoperator	
F302-02	OLAP-Fähigkeiten	INTERSECT-ALL-Tabellenoperator	
F304	OLAP-Fähigkeiten	EXCEPT-ALL-Tabellenoperator	
F311	Core	Schemadefinitionsbefehl	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE für persistente Basistabellen	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT-Befehl	
F321		Benutzerauthorisierung	
F361		Unterprogrammunterstützung	
F381		Erweiterte Schemamanipulation	
F381-01		ALTER-TABLE-Befehl: ALTER-COLUMN-Klausel	
F381-02		ALTER-TABLE-Befehl: ADD-CONSTRAINT-Klausel	
F381-03		ALTER-TABLE-Befehl: DROP-CONSTRAINT-Klausel	
F391		Lange Namen	
F401	OLAP-Fähigkeiten	Erweiterte verbundene Tabelle	
F401-01	OLAP-Fähigkeiten	NATURAL JOIN	
F401-02	OLAP-Fähigkeiten	FULL OUTER JOIN	
F401-03	OLAP-Fähigkeiten	UNION JOIN	
F401-04	OLAP-Fähigkeiten	CROSS JOIN	
F411	Verbesserte Datum/ Zeit-Fähigkeiten	Zeitzonenangabe	
F421		NATIONAL CHARACTER	
F431-01		FETCH mit explizitem NEXT	

Kennung	Paket	Beschreibung	Kommentar
F431-04		FETCH PRIOR	
F431-06		FETCH RELATIVE	
F441		Erweiterte Mengenfunktionsunterstützung	
F471	Core	Skalare Unteranfragewerte	
F481	Core	Erweitertes NULL-Prädikat	
F491	Verbesserte Integritätsverwaltung	Constraint-Verwaltung	
F511		BIT-Datentyp	
F531		Temporäre Tabellen	
F555	Verbesserte Datum/ Zeit-Fähigkeiten	Verbesserte Sekundengenauigkeit	
F561		Volle Wertausdrücke	
F571		Wahrheitswerttests	
F591	OLAP-Fähigkeiten	Abgeleitete Tabellen	
F611		Indikator-Datentypen	
F651		Katalognamenqualifizierung	
F701	Verbesserte Integritätsverwaltung	Fremdschlüssel-Aktualisierungsaktionen	
F761		Sitzungsverwaltung	
F791		Insensitive Cursor	
F801		Volle Mengenfunktion	
S071	Erweiterte Objektunterstützung	SQL-Pfade in Funktions- und Typnamenauflösung	
S111	Erweiterte Objektunterstützung	ONLY in Anfrageausdrücken	
S211	Erweiterte Objektunterstützung, SQL/MM-Unterstützung	Benutzerdefinierte Cast-Funktionen	
T031		BOOLEAN-Datentyp	
T141		SIMILAR-Prädikat	
T151		DISTINCT-Prädikat	
T191	Verbesserte Integritätsverwaltung	Fremdschlüsselaktion RESTRICT	
T201	Verbesserte Integritätsverwaltung	Vergleichbare Datentypen für Fremdschlüssel-Constraints	
T211-01	Verbesserte Integritätsverwaltung	Triggers durch UPDATE, INSERT oder DELETE in einer Basistabelle aktiviert	
T211-02	Verbesserte Integritätsverwaltung	BEFORE-Trigger	
T211-03	Verbesserte Integritätsverwaltung	AFTER-Trigger	

Kennung	Paket	Beschreibung	Kommentar
T211-04	Verbesserte Integritätsverwaltung	FOR-EACH-ROW-Trigger	
T211-07	Verbesserte Integritätsverwaltung	TRIGGER-Privileg	
T231		SENSITIVE-Cursor	
T241		START-TRANSACTION-Befehl	
T312		OVERLAY-Funktion	
T321-01	Core	Benutzerdefinierte Funktionen ohne Überladen	
T321-03	Core	Funktionsaufruf	
T322	PSM, SQL/MM-Unterstützung	Überladen von Funktionen und Prozeduren mit SQL-Aufruf	
T323		Explizite Sicherheit für externe Routinen	
T351		Erweiterte SQL-Kommentare (/*...*/ Kommentare)	
T441		ABS- und MOD-Funktionen	
T501		Verbessertes EXISTS-Prädikat	
T551		Optionale Schlüsselwörter für Standardsyntax	
T581		SUBSTRING-Funktion mit regulären Ausdrücken	
T591		UNIQUE-Constraints für Spalten, die NULL sein können	

C.2 Nicht unterstützte Leistungsmerkmale

Die folgenden in SQL99 definierten Leistungsmerkmale sind in dieser Version von PostgreSQL nicht implementiert. In einigen Fällen ist gleichwertige Funktionalität vorhanden.

Kennung	Paket	Beschreibung	Kommentar
B011	Core	Eingebettetes Ada	
B013	Core	Eingebettetes COBOL	
B014	Core	Eingebettetes Fortran	
B015	Core	Eingebettetes MUMPS	
B016	Core	Eingebettetes Pascal	
B017	Core	Eingebettetes PL/I	
B031		Grundlegendes dynamisches SQL	
B032		Erweitertes dynamisches SQL	
B032-1		<describe input> Befehl	
B041		Erweiterungen zu Exception-Deklarationen in eingebettetem SQL	
B051		Verbesserte Ausführungsrechte	
E081	Core	Grundlegende Privilegien	
E081-05	Core	UPDATE-Privileg auf Spaltenebene	

Kennung	Paket	Beschreibung	Kommentar
E081-07	Core	REFERENCES-Privileg auf Spaltenebene	
E081-08	Core	WITH GRANT OPTION	
E121	Core	Grundlegende Cursor-Unterstützung	
E121-04	Core	OPEN-Befehl	
E121-06	Core	Positionierter UPDATE-Befehl	
E121-07	Core	Positionierter DELETE-Befehl	
E121-17	Core	WITH-HOLD-Cursor	
E152	Core	Grundlegender SET-TRANSACTION-Befehl	
E152-02	Core	SET-TRANSACTION-Befehl: READ-ONLY- und READ-WRITE Klauseln	Syntax akzeptiert; READ ONLY nicht unterstützt
E153	Core	Aktualisierbare Anfragen mit Unteranfragen	
E171	Core	SQLSTATE-Unterstützung	
F181		Unterstützung für mehrere Module	
E182	Core	Modulsprache	
F021	Core	Grundlegendes Informationsschema	
F021-01	Core	COLUMNS-Sicht	
F021-02	Core	TABLES-Sicht	
F021-03	Core	VIEWS-Sicht	
F021-04	Core	TABLE_CONSTRAINTS-Sicht	
F021-05	Core	REFERENTIAL_CONSTRAINTS-Sicht	
F021-06	Core	CHECK_CONSTRAINTS-Sicht	
F031-19	Core	REVOKE-Befehl: RESTRICT-Klausel	
F034		Erweiterter REVOKE-Befehl	
F034-01		REVOKE-Befehl von jemand anderem als dem Eigentümer des Schemaobjekts durchgeführt	
F034-02		REVOKE-Befehl: GRANT-OPTION-FOR-Klausel	
F034-03		REVOKE-Befehl, um Privilegien, die der Empfänger mit GRANT-OPTION hat, zu entziehen	
F111		Isolationsgrade außer SERIALIZABLE	
F111-01		READ-UNCOMMITTED Isolationsgrad	
F111-03		REPEATABLE-READ Isolationsgrad	
F121		Grundlegende Diagnosenverwaltung	
F121-01		GET-DIAGNOSTICS-Befehl	
F121-02		SET-TRANSACTION-Befehl: DIAGNOSTICS-SIZE-Klausel	
F231		Privilegientabellen	
F231-01		TABLE_PRIVILEGES-Sicht	
F231-02		COLUMN_PRIVILEGES-Sicht	

Kennung	Paket	Beschreibung	Kommentar
F231-03		USAGE_PRIVILEGES-Sicht	
F291		UNIQUE-Prädikat	
F301		CORRESPONDING in Anfrageausdrücken	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F341		Verwendungstabellen	
F431		Nur lesbare scrollbare Cursor	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-05		FETCH ABSOLUTE	
F451		Zeichensatzdefinition	
F461		Benannte Zeichensätze	
F501	Core	Sichten über Leistungsmerkmale und Konformität	
F501-01	Core	SQL_FEATURES-Sicht	
F501-02	Core	SQL_SIZING-Sicht	
F501-03	Core	SQL_LANGUAGES-Sicht	
F502		Verbesserte Dokumentationstabellen	
F502-01		SQL_SIZING_PROFILES-Sicht	
F502-02		SQL_IMPLEMENTATION_INFO-Sicht	
F502-03		SQL_PACKAGES-Sicht	
F521	Verbesserte Integritätsverwaltung	Assertions	
F641	OLAP-Fähigkeiten	Zeilen- und Tabellenkonstruktionsklauseln	
F661		Einfache Tabellen	
F671	Verbesserte Integritätsverwaltung	Unteranfragen in CHECK	absichtlich weggelassen
F691		Kollation und Translation	
F711		ALTER DOMAIN	
F721		Verschiebbare Constraints	nur Fremdschlüssel
F731		INSERT-Spaltenprivilegien	
F741		Fremdschlüssel MATCH-Typen	noch kein MATCH PARTIAL
F751		Sicht CHECK Verbesserungen	
F771		Verbindungsverwaltung	
F781		Selbstbezogene Operationen	
F811		Erweitertes Flagging	
F812	Core	Grundlegendes Flagging	
F813		Erweitertes Flagging nur für Core SQL und Nachsehen im Katalog	

Kennung	Paket	Beschreibung	Kommentar
F821		Lokale Tabellenverweise	
F831		Volle Cursor-Aktualisierung	
F831-01		Aktualisierbare scrollbare Cursor	
F831-02		Aktualisierbare geordnete Cursor	
S011	Core	Distinkte Datentypen	
S011-01	Core	USER_DEFINED_TYPES-Sicht	
S023	Grundlegende Objektunterstützung, SQL/MM-Unterstützung	Grundlegende strukturierte Typen	
S024, SQL/MM-Unterstützung	Erweiterte Objektunterstützung	Verbesserte strukturierte Typen	
S041	Grundlegende Objektunterstützung	Grundlegende Referenztypen	
S043	Erweiterte Objektunterstützung	Verbesserte Referenztypen	
S051	Grundlegende Objektunterstützung	Tabelle eines Typs erzeugen	
S081	Erweiterte Objektunterstützung	Untertabellen	
S091	SQL/MM-Unterstützung	Grundlegende Arrayunterstützung	Arrays in PostgreSQL sind anders
S091-01	SQL/MM-Unterstützung	Arrays aus eingebauten Datentypen	
S091-02	SQL/MM-Unterstützung	Arrays aus distinkten Typen	
S091-03	SQL/MM-Unterstützung	Arrayausdrücke	
S092	SQL/MM-Unterstützung	Arrays aus benutzerdefinierten Typen	
S094		Arrays aus Referenztypen	
S151	Grundlegende Objektunterstützung	Typprädikat	
S161	Erweiterte Objektunterstützung	Untertypbehandlung	
S201		SQL-Routinen mit Arrays	
S201-01		Arrayparameter	
S201-02		Array als Ergebnistyp von Funktionen	
S231	Erweiterte Objektunterstützung	Locators für strukturierte Typen	
S232		Locators für Arrays	

Kennung	Paket	Beschreibung	Kommentar
S241	Erweiterte Objektunterstützung	Transformfunktionen	
S251		Benutzerdefinierte Ordnungen	
S261		SPECIFICTYPE-Methode	
T011		Zeiten im Informationsschema	
T041	Grundlegende Objektunterstützung	Grundlegende LOB-Datentyp-Unterstützung	
T041-01	Grundlegende Objektunterstützung	BLOB-Datentyp	
T041-02	Grundlegende Objektunterstützung	CLOB-Datentyp	
T041-03	Grundlegende Objektunterstützung	POSITION-, LENGTH-, LOWER-, TRIM-, UPPER- und SUBSTRING-Funktionen für LOB-Datentypen	
T041-04	Grundlegende Objektunterstützung	Verknüpfung von LOB-Datentypen	
T041-05	Grundlegende Objektunterstützung	LOB-Locator: nicht haltbar	
T042		Erweiterte LOB-Datentyp-Unterstützung	
T051		Zeilentypen	
T111		Aktualisierbare Verbunde, Vereinigungsmengen und Spalten	
T121		WITH (ohne RECURSIVE) in Anfrageausdrücken	
T131		Rekursive Anfrage	
T171		LIKE-Klausel in Tabellendefinition	
T211	Verbesserte Integritätsverwaltung, Aktive Datenbank	Grundlegende Triggerfähigkeit	
T211-05	Verbesserte Integritätsverwaltung	Möglichkeit, eine Suchbedingung anzugeben, die wahr sein muss, bevor ein Trigger aufgerufen wird	
T211-06	Verbesserte Integritätsverwaltung	Unterstützung für Laufzeitregeln für die Wechselwirkung von Triggern und Constraints	
T211-08	Verbesserte Integritätsverwaltung	Mehrere Trigger für das gleiche Ereignis werden in der Reihenfolge ausgeführt, in der sie erzeugt wurden	
T212	Verbesserte Integritätsverwaltung	Erweiterte Triggerfähigkeit	
T251		SET-TRANSACTION-Befehl: LOCAL-Option	
T261		Verkettete Transaktionen	
T271		Sicherungspunkte	
T281		SELECT-Privileg auf Spaltenebene	
T301		Funktionelle Abhängigkeiten	
T321	Core	Grundlegende Routinen mit SQL-Aufruf	
T321-02	Core	Benutzerdefinierte gespeicherte Prozeduren ohne Überladen	
T321-04	Core	CALL-Befehl	
T321-05	Core	RETURN-Befehl	

Kennung	Paket	Beschreibung	Kommentar
T321-06	Core	ROUTINES-Sicht	
T321-07	Core	PARAMETERS-Sicht	
T331		Grundlegende Rollen	
T332		Erweiterte Rollen	
T401		INSERT in einen Cursor	
T411		UPDATE-Befehl: SET-ROW-Option	
T431	OLAP-Fähigkeiten	CUBE- und ROLLUP-Operationen	
T461		Symmetrisches BETWEEN-Prädikat	
T471		Ergebnismengen als Rückgabewert	
T491		LATERAL abgeleitete Tabelle	
T511		Transaktionszählungen	
T541		Aktualisierbare Tabellenreferenzen	
T561		Haltbare Locators	
T571		Externe Funktionen mit SQL-Aufruf die Arrays zurückgeben	
T601		Lokale Cursor-Referenzen	

Versionsgeschichte

D.1 Version 7.3.3

Veröffentlichungsdatum: 22.05.2003

Diese Version enthält verschiedene Berichtigungen gegenüber Version 7.3.2.

D.1.1 Umstieg auf Version 7.3.3

Datensicherung und -wiederherstellung ist für Benutzer von Version 7.3.* nicht notwendig.

D.1.2 Änderungen

- Gelegentlich falsche Berechnung von StartUpID nach Absturz repariert
- Langsamkeit bei vielen verschobenen Triggern in einer Transaktion vermieden (Stephan)
- Sperrt die Primärschlüsselzeile nicht, wenn UPDATE den Wert des Fremdschlüssels nicht ändert (Jan)
- Sparc verwendet `-fPIC` anstatt `-fpic` (Tom Callaway)
- Fehlende Schemaunterstützung in `contrib/reindexdb` repariert
- Fehler in `contrib/intarray` bei Ergebnisarray mit null Elementen repariert (Teodor)
- createuser kann immer mit Strg+C beendet werden (Oliver)
- Fehler, wenn der Typ einer gelöschten Spalte selbst gelöscht worden ist, berichtigt
- CHECKPOINT verursacht keine Datenbankpanik bei Fehlern in nicht entscheidenden Schritten
- 60 in den Sekundenfeldern von Eingabewerten für timestamp, time und interval erlaubt
- Wenn Präzision für timestamp, time oder interval zu groß ist, wird nur ein Hinweis, kein Fehler erzeugt
- Typumwandlung von `abstime` in time berichtigt (tritt erst nach `initdb` in Effekt)
- `pg_proc`-Eintrag von `timestamptz_izone` berichtigt (tritt erst nach `initdb` in Effekt)
- EXTRACT(EPOCH FROM timestamp without time zone) behandelt Eingabe jetzt als Ortszeit
- 'now'::`timestamptz` lieferte falsches Ergebnis, wenn die Zeitzone vorher in der Transaktion geändert wurde
- HAVE_INT64_TIMESTAMP-Code für time with timezone überschrieb die Eingabe

- GLOBAL TEMP/TEMPORARY wird als Synonym für TEMPORARY akzeptiert
- Falsche Schemaprivilegienprüfung bei Fremdschlüsseltriggern berichtigt
- Fehler in Fremdschlüsseltriggern für Aktion SET DEFAULT berichtigt
- Falsche Zeitprüfung beim Zeilen Lesen in UPDATE- und DELETE-Triggern berichtigt
- Fremdschlüsselklauseln wurden von ALTER TABLE ADD COLUMN geparst aber ignoriert
- createlang-Fehler, wenn die Handlerfunktion bereits existiert, berichtigt
- Fehlverhalten bei Tabellen mit null Spalten in `pg_dump`, COPY, ANALYZE und anderswo berichtigt
- Fehlverhalten von `func_error`() bei Typnamen mit '%' berichtigt
- Fehlverhalten von `replace`() bei Zeichenketten mit '%' berichtigt
- Reguläre Ausdrucke mit bestimmten mehrbytigen Zeichen funktionierten nicht
- NULL-Werte werden bei der Abschätzung der Verbundgröße an mehr Stellen mit einbezogen
- Konflikt mit Systemdefinition von `isblank`() verhindert
- Fehler bei Umwandlung von großen Codepoint-Werten in `EUC_TW`-Konversionen berichtigt (Tatsuo)
- Fehlerbehandlung in `SSL_read`/`SSL_write` berichtigt
- Frühe Konstantenumwandlung in Typumwandlungsausdrücken geschieht nicht mehr
- Seitenkopffelder werden sofort nach dem Lesen der Seite auf Gültigkeit überprüft
- Falsche Prüfung auf ungruppierte Variablen in unbenannten Verbunden repariert
- Pufferüberlauf in `to_ascii` berichtigt (Guido Notari)
- Berichtigungen in `contrib`/`ltree` (Teodor)
- `Core-Dump` in Verklemmungserkennung auf Maschinen mit nicht vorzeichenbehafteten char repariert
- Verbrauch aller Puffer durch Indexscan in mehrere Richtungen vermieden (Fehler in 7.3 eingeschlichen)
- Auswahlselektivitätsschätzfunktionen des Planers für Domänen berichtigt
- Fehler in der Speicherzuteilung in `dbmirror` berichtigt (Steven Singer)
- Endlosschleife in ln(numeric) wegen Rundungsfehler vermieden
- GROUP BY wurde von mehreren gleichen GROUP-BY-Elementen verwirrt
- Schlechter Plan, wenn vererbtes UPDATE/DELETE eine weitere vererbte Tabelle verwendet, berichtigt
- Clustern mit unvollständigen Indexen (partiell oder ohne NULL-Werte) verhindert
- Anforderung zum Herunterfahren wird zur richtigen Zeit bearbeitet, wenn sie während des Startvorgangs eintrifft
- Linke Verbindungen in temporären Indexen repariert (Rückwärts-Scans könnten dadurch Einträge verpassen)
- Falsche Behandlung von `client_encoding`-Einstellung in `postgresql.conf` berichtigt (Tatsuo)
- Nach Ausführung von `Async_NotifyHandler` wurde auf '`pg_ctl stop` -m fast' nicht mehr reagiert
- SPI im Fall, wenn Regel mehrere Befehle des gleichen Typs enthält, berichtigt
- Prüfung der falschen Privilegientypen in Anfrage mit Regel berichtigt
- Problem mit EXCEPT in CREATE RULE berichtigt
- Problem beim Löschen von temporären Tabellen mit serial-Spalten verhindert
- Versagen von `replace_vars_with_subplan_refs` in komplexen Sichten repariert
- Langsame reguläre Ausdrücke bei mehrbytigen Kodierungen berichtigt (Tatsuo)
- Qualifizierte Typnamen in CREATE CAST und DROP CAST erlaubt
- `SETOF typ`[] wird akzeptiert, musste früher als `SETOF _typ` geschrieben werden
- `Core-Dump` in `pg_dump` in manchen Fällen mit prozeduralen Sprachen repariert
- `pg_dump` verwendet Datumsstil ISO, zur besseren Portierbarkeit (Oliver)
- `pg_dump` konnte Fehlerergebnis von `lo_read` nicht verarbeiten (Oleg Drokin)
- `pg_dumpall` konnte Gruppen ohne Mitglieder nicht verarbeiten (Nick Eskelinen)

- `pg_dumpall` hat Option --`globals`-only nicht erkannt
- `pg_restore` konnte Large Objects nicht wiederherstellen, wenn -`X disable-triggers` verwendet wurde
- Intrafunktions-Speicherleck in PL/pgSQL repariert
- `PL/Tcls elog`-Befehl erzeugte Core-Dump bei falschen Parametern (Ian Harding)
- `PL/Python` hatte falschen Wert von atttypmod verwendet (Brad McLean)
- Unzureichendes Quoting von boolean-Werten in PyGreSQL berichtigt (D'Arcy)
- Methode addDataType() zum Interface PGConnection in JDBC hinzugefügt
- Diverse Probleme mit aktualisierbaren ResultSets in JDBC berichtigt (Shawn Green)
- Diverse Probleme mit DatabaseMetaData in JDBC berichtigt (Kris Jurka, Peter Royal)
- Probleme beim Parsen von ACLs in JDBC berichtigt
- Bessere Fehlermeldung bei Problemen mit der Zeichensatzumwandlung in JDBC

D.2 Version 7.3.2

Veröffentlichungsdatum: 04.02.2003

Diese Version enthält verschiedene Berichtigungen gegenüber Version 7.3.1.

D.2.1 Umstieg auf Version 7.3.2

Datensicherung und -wiederherstellung ist für Benutzer von Version 7.3.* nicht notwendig.

D.2.2 Änderungen

- CREATE TABLE AS / SELECT INTO erzeugt wieder eine OID-Spalte
- `Core-Dump` in `pg_dump` bei Sichten mit Kommentaren repariert
- `pg_dump` gibt Constraints mit DEFERRABLE/INITIALLY DEFERRED richtig aus
- UPDATE, wenn sich die Spaltennummerierung in der Kindtabelle von der Elterntabelle unterscheidet, repariert
- Vorgabewert von `max_fsm_relations` erhöht
- Problem beim rückwärts Lesen in einem Cursor mit einzeiliger Anfrage repariert
- Rückwärts Lesen bei einem Cursor mit SELECT-DISTINCT-Anfrage funktioniert jetzt richtig
- Probleme beim Laden von `pg_dump`-Dateien, die `contrib/lo` verwenden, berichtigt
- Probleme mit Benutzernamen, die nur aus Ziffern bestehen, berichtigt
- Mögliches Speicherleck und Core-Dump beim Trennen der Verbindung in libpgtcl repariert
- PL/Pythons spi_execute verarbeitet NULL-Werte richtig (Andrew Bosma)
- Fehlermeldungen in PL/Python angepasst, damit sein Regressionstest wieder erfolgreich ist
- Arbeitet jetzt mit Bison 1.875 zusammen
- Namen mit gemischter Groß-/Kleinschreibung werden in PL/pgSQLs %type richtig verarbeitet (Neil)
- Core-Dump in `PL/Tcl` bei der Ausführung einer von Regeln umgeschriebenen Anfrage repariert
- Arrayindexüberschreitung repariert (nach Bericht von Yichen Xie)
- MAX_TIME_PRECISION im Fließkommafall von 13 auf 10 verringert
- Groß-/Kleinschreibung in datenbank- und benutzerspezifischen Einstellungen wird richtig verarbeitet

- ❏ Core-Dump in PL/pgSQLs RETURN NEXT repariert, wenn SELECT in einen Record keine Zeilen ergibt
- ❏ Überholte Verwendung von `pg_type.typprtlen` in PyGreSQL berichtigt
- ❏ Bruchteile von Sekunden in timestamp-Werten werden vom JDBC-Treiber richtig verarbeitet
- ❏ Leistung von `getImportedKeys`() in JDBC verbessert
- ❏ Symlinks für dynamische Bibliotheken funktionieren auf HP-UX jetzt normal (Giles)
- ❏ Uneinheitliches Runden in timestamp, time und interval berichtigt
- ❏ Berichtigungen in der SSL-Verhandlung (Nathan Mueller)
- ❏ `libpqs ~/.pgpass` funktioniert jetzt richtig, wenn man mit PQconnectDB verbindet
- ❏ `my2pg, ora2pg` aktualisiert
- ❏ Aktualisierte Übersetzungen
- ❏ Typumwandlungen zwischen lo und oid in `contrib/lo` hinzugefügt
- ❏ Fastpath prüft jetzt die Privilegien der aufgerufenen Funktion

D.3 Version 7.3.1

Veröffentlichungsdatum: 18.12.2002

Diese Version enthält verschiedene Berichtigungen gegenüber Version 7.3.

D.3.1 Umstieg auf Version 7.3.1

Datensicherung und -wiederherstellung ist für Benutzer von Version 7.3 nicht notwendig. Allerdings wurde die Hauptversion der PostgreSQL-Schnittstellenbibliothek `libpq` geändert, sodass Clientcode, der `libpq` verwendet, neu gelinkt werden muss.

D.3.2 Änderungen

- ❏ Core-Dump bei COPY TO, wenn Client- und Serverkodierung nicht übereinstimmen, repariert (Tom)
- ❏ `pg_dump` mit Servern vor Version 7.2 ermöglicht (Philip)
- ❏ Berichtigungen in contrib/adddepend (Tom)
- ❏ Problem beim Löschen von benutzer- und datenbankspezifischen Konfigurationseinstellungen repariert (Tom)
- ❏ Berichtigung in contrib/vacuumlo (Tom)
- ❏ “password”-Authentifizierung, auch wenn `pg_shadow` MD5-Passwörter enthält, erlaubt (Bruce)
- ❏ Berichtigung in contrib/dbmirror (Steven Singer)
- ❏ Optimiererberichtigung (Tom)
- ❏ Berichtigungen in `contrib/tsearch` (Teodor Sigaev, Magnus)
- ❏ Kodierungsnamenauflösung in türkischer Locale repariert (Nicolai Tufar)
- ❏ Hauptversionsnummer von `libpq` erhöht (Bruce)
- ❏ Fehlermeldungen über `pg_hba.conf` berichtigt (Bruce, Neil)
- ❏ SCO OpenServer 5.0.4 als unterstützte Plattform hinzugefügt (Bruce)
- ❏ Absturz des Servers bei EXPLAIN verhindert (Tom)
- ❏ SSL-Berichtigungen (Nathan Mueller)
- ❏ Erzeugung von Spalten mit zusammengesetzten Typen über ALTER TABLE verhindert (Tom)

- ❏ Umwandlung zwischen LATIN9 und UNICODE repariert (Peter)
- ❏ Aktualisierte Übersetzungen

D.4 Version 7.3

Veröffentlichungsdatum: 27.11.2002

D.4.1 Überblick

Die wichtigsten Änderungen in dieser Version:

Schemas

Mit Schemas können Benutzer Objekte (z.B. Tabellen) in getrennten Namensräumen erzeugen, damit zwei Benutzer oder Anwendungen Objekte mit dem gleichen Namen haben können. Es gibt ein öffentliches Schema für von allen verwendete Objekte. Die Erzeugung von Objekten kann unterbunden werden, indem die Privilegien für das öffentliche Schema entzogen werden.

Spalten löschen

PostgreSQL unterstützt jetzt das Löschen von Spalten mit `ALTER TABLE ... DROP COLUMN`.

Tabellenfunktionen

Funktionen, die mehrere Zeilen und/oder mehrere Spalten zurückgeben, sind jetzt viel einfacher als vorher anzuwenden. Sie können eine solche "Tabellenfunktion" in der `FROM`-Klausel des Befehls `SELECT` aufrufen und das Ergebnis wie eine Tabelle weiterverarbeiten. Des Weiteren können `PL/pgSQL`-Funktionen jetzt Ergebnismengen zurückgeben.

Vorbereitete Befehle

PostgreSQL unterstützt jetzt vorbereitete Befehl für bessere Leistung.

Verfolgung von Abhängigkeiten

PostgreSQL zeichnet jetzt die Abhängigkeiten zwischen Objekten auf, wodurch sich Verbesserungen in vielen Bereichen ergeben. `DROP`-Befehle können jetzt entweder `CASCADE` oder `RESTRICT` angeben, um zu kontrollieren, ob abhängige Objekte mit gelöscht werden sollen.

Privilegien

Funktionen und prozedurale Sprachen haben jetzt Privilegien, und Funktionen können jetzt so definiert werden, dass sie mit den Privilegien des Erzeugers ausgeführt werden.

Internationalisierung

Unterstützung für Mehrbyte-Zeichensätze und Locales ist jetzt immer angeschaltet.

Loggen

Verschiedene Log-Optionen wurden verbessert.

Schnittstellen

Einige Client-Schnittstellen wurden auf http://gborg.postgresql.org verlegt, wo sie unabhängig entwickelt und veröffentlicht werden können.

Funktionen, Bezeichner

In der Voreinstellung können Funktionen jetzt bis zu 32 Argumente haben und Bezeichner bis zu 63 Bytes lang sein. Außerdem sollte die Typbezeichnung opaque nicht mehr verwendet werden: Es

gibt jetzt einzelne “Pseudotypen”, die in Funktionsargumenten und rückgabetypen die früheren Aufgaben von opaque übernehmen.

D.4.2 Umstieg auf Version 7.3

Wenn Sie Daten von einer früheren Version übernehmen wollen, dann müssen Sie die Daten mit `pg_dump` sichern und wiederherstellen. Wenn Ihre Anwendung die Systemkataloge liest, sind weitere Änderungen nötig, aufgrund der Einführung von Schemas; weitere Informationen dazu finden Sie unter .

Beachten Sie die folgenden Inkompatibilitäten:

- Clients von vor Version 6.3 werden nicht mehr unterstützt.
- `pg_hba.conf` hat jetzt eine Spalte für den Benutzernamen und weitere Merkmale. Bestehende Dateien müssen angepasst werden.
- Mehrere Log-Parameter in `postgresql.conf` wurden umbenannt.
- `LIMIT` *`zahl`*`, zahl` wurde entfernt; verwenden Sie `LIMIT zahl OFFSET zahl`.
- `INSERT`-Befehle mit Spaltenlisten müssen für jede angegebene Spalte einen Wert enthalten. Zum Beispiele ist `INSERT INTO tab (sp1, sp2) VALUES ('wert1')` jetzt nicht mehr gültig. Es ist immer noch erlaubt, weniger Spalten als erwartet anzugeben, wenn im `INSERT` keine Spaltenliste angegeben ist.
- Spalten des Typs serial haben nicht mehr automatisch einen Unique Constraint; daher wird auch kein Index mehr automatisch erzeugt.
- `SET`-Befehle in Transaktionsblöcken werden jetzt zurückgerollt, wenn die Transaktion abgebrochen wird.
- `COPY` interpretiert fehlende Werte am Zeilenende nicht mehr als NULL-Werte. Alle Spalten müssen angegeben werden. (Eine ähnliche Wirkung wie früher kann man jedoch erreichen, indem man bei `COPY` eine Spaltenliste angibt.)
- Der Datentyp timestamp entspricht jetzt timestamp without time zone anstatt timestamp with time zone.
- Datenbanken von früheren Versionen, die in 7.3 geladen werden, werden nicht die neuen Objektabhängigkeiten für serial-Spalten, Unique Constraints und Fremdschlüssel haben. Eine detaillierte Beschreibung dazu und ein Skript, das diese Abhängigkeiten erzeugt, finden Sie im Verzeichnis `contrib/adddepend/`.
- Eine leere Zeichenkette (`''`) ist keine gültige Eingabe in ein Zahlenfeld mehr. Früher wurde sie stillschweigend als 0 verstanden.

D.4.3 Änderungen

Serveroperation

- Neue Sicht `pg_locks`, um Sperren anzuzeigen (Neil)
- Sicherheitslücken in der Speicherzuweisung bei der Passwortauthentifizierung berichtigt (Neil)
- Unterstützung für Client/Server-Protokoll Version 0 (PostgreSQL 6.2 und davor) entfernt (Tom)
- Einige Verbindungen für Superuser reserviert, dafür Parameter `superuser_reserved_connections` (Nigel J. Andrews)

Leistung

- Startzeit verbessert, indem `localtime`() nur einmal aufgerufen wird (Tom)
- Systemkataloginformationen in normalen Dateien für bessere Startzeit gecacht (Tom)
- Cachen von Indexinformationen verbessert (Tom)

- Optimiererverbesserungen (Tom, Fernando Nasser)
- Katalogcaches speichern jetzt auch fehlgeschlagene Suchen (Tom)
- Verbesserungen bei Hashfunktionen (Neil)
- Leistung im Scanner und bei der Netzwerkkommunikation verbessert (Peter)
- Geschwindigkeitsverbesserungen bei der Wiederherstellung von Large Objects (Mario Weilguni)
- Abgelaufene Indexeinträge werden beim ersten Fund markiert, um spätere Heap-Fetches zu vermeiden (Tom)
- Zu viel NULL-Bitmap-Padding wird vermieden (Manfred Koizar)
- qsort() mit BSD-Lizenz wird für Solaris für bessere Leistung verwendet (Bruce)
- Zeilengröße um vier Bytes verringert (Manfred Koizar)
- Fehler im GEQO-Optimierer repariert (Neil Conway)
- WITHOUT OIDS spart wirklich vier Bytes (Manfred Koizar)
- Neuer Parameter `default_statistics_target` für ANALYZE (Neil Conway)
- Lokaler Puffer-Cache wird für temporäre Tabellen verwendet, um WAL zu umgehen (Tom)
- Freespace-Map-Leistung bei großen Tabellen verbessert (Stephen Marshall, Tom)
- Verbesserter WAL-Schreibdurchsatz im Mehrbenutzerbetrieb (Tom)

Privilegien

- Privilegien für Funktionen und prozedurale Sprachen (Peter)
- Option OWNER in CREATE DATABASE, damit Superuser Datenbanken für andere erzeugen können (Gavin Sherry, Tom)
- Neue Privilegienbits für EXECUTE und USAGE (Tom)
- SET SESSION AUTHORIZATION DEFAULT und RESET SESSION AUTHORIZATION (Tom)
- Ausführung von Funktionen mit den Privilegien des Eigentümers ermöglicht (Peter)

Serverkonfiguration

- Serverlogmeldungen erscheinen jetzt als LOG, nicht DEBUG (Bruce)
- Spalte für Benutzername in pg_hba.conf (Bruce)
- `log_connections` gibt zwei Zeilen in Logdatei aus (Tom)
- `debug_level` aus postgresql.conf entfernt, jetzt `server_min_messages` (Bruce)
- Neue Befehle ALTER DATABASE/USER ... SET für benutzer- und datenbankspezifische Konfiguration (Peter)
- Neue Parameter `server_min_messages` und `client_min_messages` kontrollieren, welche Meldungen an Serverlog und Clientanwendung gehen (Bruce)
- Listen von Benutzern/Datenbanken in `pg_hba.conf` erlaubt, Gruppen mit +, Dateinamen mit @ (Bruce)
- SSL-Verbesserungen (Bear Giles)
- Verschlüsselt gespeicherte Passwörter sind jetzt die Voreinstellung (Bruce)
- `pg_statistics` kann mit `pg_stat_reset`() zurückgesetzt werden (Christopher)
- Neuer Parameter `log_duration` (Bruce)
- `debug_print_query` umbenannt in `log_statement` (Bruce)
- `show_query_stats` umbenannt in `show_statement_stats` (Bruce)
- Neuer Parameter `log_min_error_statement`, um Befehle bei Fehler zu loggen (Gavin)

Anfragen

- Cursor insensitiv gemacht, d.h., ihr Inhalt ändert sich nicht (Tom)
- Syntax LIMIT x,y entfernt; jetzt nur noch LIMIT x OFFSET y unterstützt (Bruce)

- ❏ Bezeichnerlänge auf 63 erhöht (Neil, Bruce)
- ❏ Berichtigung bei UNION mit >= 3 Spalten unterschiedlicher Länge (Tom)
- ❏ Schlüsselwort DEFAULT in INSERT, z.B. INSERT ... (..., DEFAULT, ...) (Rod)
- ❏ Vorgabewerte für Sichten mit ALTER COLUMN ... SET DEFAULT erlaubt (Neil)
- ❏ INSERT mit Spaltenlisten, die nicht alle Spalten angeben, verboten, z.B. INSERT INTO tab (sp1, sp2) VALUES ('wert1'); (Rod)
- ❏ Berichtigung für Aliasnamen in Verbunden (Tom)
- ❏ Berichtigung für FULL OUTER JOIN (Tom)
- ❏ Meldungen über Fehlerstellen verbessert (Tom, Gavin)
- ❏ Berichtigung für OPEN cursor (argument) (Tom)
- ❏ Verwendung von '`ctid`' in Sichten und currtid (sichtname) erlaubt (Hiroshi)
- ❏ Berichtigung von CREATE TABLE AS mit UNION (Tom)
- ❏ Parameter `statement_timeout` um Anfragen abzubrechen (Bruce)
- ❏ Vorbereitete Anfragen mit PREPARE/EXECUTE (Neil)
- ❏ FOR UPDATE nach LIMIT/OFFSET erlaubt (Bruce)
- ❏ Parameter autocommit (Tom, David Van Wie)

Objektmanipulation

- ❏ Ist-gleich-Zeichen in CREATE DATABASE optional gemacht (Gavin Sherry)
- ❏ ALTER TABLE OWNER ändert auch Indexeigentümer (Neil)
- ❏ Neuer Befehl ALTER TABLE tabname ALTER COLUMN spname SET STORAGE kontrolliert TOAST-Speicherung und Komprimierung (John Gray)
- ❏ Schemaunterstützung, CREATE/DROP SCHEMA (Tom)
- ❏ Schema für temporäre Tabellen (Tom)
- ❏ Parameter `search_path` für Schemasuche (Tom)
- ❏ Neuer Befehl ALTER TABLE SET/DROP NOT NULL (Christopher)
- ❏ Neue Volatilitätsniveaus für CREATE FUNCTION (Tom)
- ❏ Regelnamen tabellengebunden gemacht (Tom)
- ❏ Klausel ON tabellenname in DROP RULE und COMMENT ON RULE (Tom)
- ❏ Neuer Befehl ALTER TRIGGER RENAME (Joe)
- ❏ Neue Funktionen `current_schema`() und `current_schemas`() (Tom)
- ❏ Funktionen, die mehrere Zeilen zurückgeben (Joe)
- ❏ WITH in CREATE DATABASE optional gemacht (Bruce)
- ❏ Verfolgung von Objektabhängigkeiten (Rod, Tom)
- ❏ Klauseln RESTRICT/CASCADE in DROP-Befehlen (Rod)
- ❏ ALTER TABLE DROP für nicht-Check-Constraints (Rod)
- ❏ Sequenz für serial-Spalte wird automatisch mit der Tabelle gelöscht (Rod)
- ❏ Spalten Löschen verhindert, wenn Spalte von Fremdschlüssel verwendet (Rod)
- ❏ Constraints werden automatisch gelöscht, wenn die Tabelle gelöscht wird (Rod)
- ❏ Neuer Befehl CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- ❏ Neuer Befehl ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- ❏ Umbenennen oder Löschen von vererbten Spalten verhindert (Alvaro Herrera)
- ❏ Fremdschlüsselfehler bei inkonsistenten Datenbankzuständen verhindert (Stephan)
- ❏ Umbenennen von Spalten wird auch in Fremdschlüsseln ausgeführt
- ❏ Neuer Befehl CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- ❏ Neuer Befehl CREATE OR REPLACE RULE (Gavin, Neil, Tom)

- ❏ Regeln werden in alphabetischer Reihenfolge ausgeführt (Tom)
- ❏ Trigger werden in alphabetischer Reihenfolge ausgeführt (Tom)
- ❏ contrib/adddepend um Objektabhängigkeiten für Datenbanken vor 7.3 herzustellen (Rod)
- ❏ Bessere Typumwandlungen beim Einfügen oder Aktualisieren (Tom)

Hilfsbefehle

- ❏ COPY TO gibt Carriage-Returns und Newlines als \r und \n aus (Tom)
- ❏ DELIMITER in COPY FROM ist 8-Bit-sicher (Tatsuo)
- ❏ `pg_dump` verwendet ALTER TABLE ADD PRIMARY KE für bessere Leistung (Neil)
- ❏ Eckige Klammern in Regeldefinition nicht mehr erlaubt (Bruce)
- ❏ Aufruf von VACUUM in einer Funktion verhindert (Bruce)
- ❏ dropdb und andere Skripts können mit Bezeichnern mit Leerzeichen umgehen (Bruce)
- ❏ Änderungen von Datenbankkommentaren auf aktuelle Datenbank beschränkt
- ❏ Kommentare für Operatoren unabhängig von der internen Funktion erlaubt (Rod)
- ❏ SET-Befehle werden in abgebrochenen Transaktionen zurückgerollt (Tom)
- ❏ EXPLAIN-Ausgabe jetzt wie eine Anfrage (Tom)
- ❏ EXPLAIN gibt Bedingungsausdrücke und Sortierschlüssel aus (Tom)
- ❏ SET LOCAL var = wert setzt Konfigurationsparameter in einzelnen Transaktionen (Tom)
- ❏ ANALYZE kann in einer Transaktion ausgeführt werden (Bruce)
- ❏ COPY-Syntax verbessert, mit Rückwärtskompatibilität (Bruce)
- ❏ pg_dump gibt konsistent Tags in nicht-ASCII-Formaten aus (Bruce)
- ❏ Fremdschlüssel-Constraints in Dump-Datei klarer gemacht (Rod)
- ❏ Neu COMMENT ON CONSTRAINT (Rod)
- ❏ In COPY TO/FROM können Spaltennamen angegeben werden (Brent Verner)
- ❏ `pg_dump` gibt Unique und Primärschlüssel-Constraints als ALTER TABLE aus (Rod)
- ❏ SHOW-Ausgabe als Anfrageergebnis (Joe)
- ❏ Kurze COPY-Zeilen erzeugen Fehler, keine NULL-Werte (Neil)
- ❏ CLUSTER repariert, damit es alle Tabellenattribute erhält (Alvaro Herrera)
- ❏ Neue Tabelle `pg_settings`, um Konfigurationsparameter einzusehen und zu ändern (Joe)
- ❏ Schlauere Verwendung von Anführungszeichen, bessere Portierbarkeit in `pg_dump`-Ausgabe (Peter)
- ❏ `pg_dump` gibt serial-Spalten als serial aus (Tom)
- ❏ Unterstützung für große Dateien (>2 GB) für `pg_dump` (Peter, Philip Warner, Bruce)
- ❏ TRUNCATE in mit Fremdschlüsseln verknüpften Tabellen verhindert (Rod)
- ❏ TRUNCATE löscht auch automatisch die TOAST-Tabelle einer Relation (Tom)
- ❏ Hilfsprogramm clusterdb, das die ganze Datenbank automatisch anhand früherer CLUSTER-Operatoren clustert (Alvaro Herrera)
- ❏ `pg_dumpall` überholt (Peter)
- ❏ REINDEX mit TOAST-Tabellen erlaubt
- ❏ START TRANSACTION nach SQL99 implementiert (Neil)
- ❏ Seltene Indexverfälschung, wenn Seiten-Split Massenlöschoperation beinträchtigt, berichtigt (Tom)
- ❏ ALTER TABLE ... ADD COLUMN für Vererbung berichtigt (Alvaro Herrera)

Datentypen und Funktionen

- ❏ Fakultät von 0 ergibt 1 (Bruce)
- ❏ Verbesserung von Datum/Zeit/Zeitzone (Thomas)
- ❏ Lesen von Arraystücken berichtigt (Tom)

- ❏ `Extract/date_part` mit timestamp ergibt richtige Anzahl Mikrosekunden (Tatsuo)
- ❏ `text_substr`() und `bytea_substr`() können TOAST-Werte effektiver lesen (John Gray)
- ❏ Unterstützung für Domänen (Rod)
- ❏ WITHOUT TIME ZONE zur Voreinstellung für Datentypen TIMESTAMP und TIME gemacht (Thomas)
- ❏ Alternatives Speicherschema für Datum-/Zeittypen mit 64-Bit-Ganzzahlen mit `--enable-integer-datetimes in configure` (Thomas)
- ❏ timezone(timestamptz) gibt timestamp anstatt Zeichenkette zurück (Thomas)
- ❏ Bruchsekunden in Datum-/Zeittypen für Daten vor 1 v.u.Z. erlaubt (Thomas)
- ❏ timestamp auf sechs Dezimalstellen Präzision begrenzt (Thomas)
- ❏ Zeitzonenumwandlungsfunktion von `timetz`() in `timezone`() geändert (Thomas)
- ❏ Konfigurationsparameter datestyle und timezone hinzugefügt (Tom)
- ❏ Funktion `overlay`(), erlaubt Ersetzen von Teilzeichenketten in einer Zeichenkette (Thomas)
- ❏ Neu SIMILAR TO (Thomas, Tom)
- ❏ Neu SUBSTRING (zeichenkette FROM muster FOR escape) mit regulären Ausdrücken (Thomas)
- ❏ Funktionen LOCALTIME und LOCALTIMESTAMP (Thomas)
- ❏ Benannte zusammengesetzte Typen mit CREATE TYPE typname AS (attribut) (Joe)
- ❏ Definition zusammengesetzter Typen in der Tabellenaliasklausel ermöglicht (Joe)
- ❏ Neue API, um Erzeugung von Tabellenfunktionen in C zu vereinfachen (Joe)
- ❏ ODBC-kompatible leere Klammern von SQL99-Funktionen entfernt, wo die Klammern nicht dem Standard entsprechen (Thomas)
- ❏ Typ macaddr akzeptiert zwölf hexadezimale Ziffern ohne Trennzeichen (Mike Wyer)
- ❏ CREATE/DROP CAST (Peter)
- ❏ Operator IS DISTINCT FROM (Thomas)
- ❏ SQL99-Funktion TREAT(), Synonym für CAST() (Thomas)
- ❏ `Funktion pg_backend_pid`() (Bruce)
- ❏ Typprädikat IS OF / IS NOT OF (Thomas)
- ❏ Bitkettenkonstanten ohne vollständig angegebene Länge ermöglicht (Thomas)
- ❏ Umwandlung zwischen 8-Byte-Ganzzahlen und Bitketten ermöglicht (Thomas)
- ❏ Hexadezimalkonstanten werden in Bitketten umgewandelt (Thomas)
- ❏ Tabellenfunktionen können in der FROM-Klausel verwendet werden (Joe)
- ❏ Maximale Anzahl Funktionsargumente auf 32 erhöht (Bruce)
- ❏ Kein automatischer Index mehr für serial-Spalte (Tom)
- ❏ Funktion `current_database`() (Rod)
- ❏ Pufferüberlauf in `cash_words`() repariert (Tom)
- ❏ Funktionen `replace`(), `split_part`(), `to_hex`() (Joe)
- ❏ LIKE für bytea als rechtes Argument berichtigt (Joe)
- ❏ Abstürze durch SELECT `cash_out`(2) repariert (Tom)
- ❏ `to_char`(1,'FM999.99') berichtigt, gibt Punkt zurück (Karel)
- ❏ Funktionen für Trigger/Typ/Sprachen geben jetzt richtigen Typ anstatt opaque zurück (Tom)

Internationalisierung

- ❏ Neue Kodierungen: Koreanisch (JOHAB), Thai (WIN874), Vietnamesisch (TCVN), Arabisch (WIN1256), Vereinfachtes Chinesisch (GBK), Koreanisch (UHC) (Eiji Tokuya)
- ❏ Locale-Unterstützung ist immer an (Peter)
- ❏ Parameter für Locale-Einstellungen (Peter)
- ❏ Bytes >= `0x7f` werden in `PQescapeBytea/PQunescapeBytea` durch Fluchtfolgen ersetzt (Tatsuo)

- Locale-Unterstützung in Zeichenklassen in regulären Ausdrücken
- Unterstützung für mehrbytige Zeichensätze immer an (Tatso)
- Unterstützung für GB18030 (Bill Huang)
- CREATE/DROP CONVERSION, ladbare Konversionen (Tatsuo, Kaori)
- Tabelle `pg_conversion` (Tatsuo)
- SQL99-Funktion CONVERT() (Tatsuo)
- `pg_dumpall`, `pg_controldata` und `pg_resetxlog` jetzt mit übersetzten Meldungen (Peter)
- Neue und aktualisierte Übersetzungen

Serverseitige Sprachen

- Rekursive SQL-Funktionen ermöglicht (Peter)
- PL/Tcl-Build verwendet konfigurierten Compiler und Makefile.shlib (Peter)
- PL/pgSQL-Variable FOUND überholt, um besser mit Oracle kompatibel zu sein (Neil, Tom)
- PL/pgSQL kann mit Bezeichnern in Anführungszeichen umgehen (Tom)
- PL/pgSQL-Funktionen, die Ergebnismengen zurückgeben (Neil)
- Schemaunterstützung in PL/pgSQL (Joe)
- Einige Speicherlecks entfernt (Nigel J. Andrews, Tom)

Psql

- `\connect` wandelt den Datenbanknamen nicht in Kleinbuchstaben um, Kompatibilität mit 7.2 (Tom)
- `\timing` stoppt die Zeit von Befehlen (Greg Sabino Mullane)
- \d zeigt Indexinformation (Greg Sabino Mullane)
- `\dD` zeigt Domänen (Jonathan Eisler)
- `psql` zeigt Regeln für Sichten (Paul ?)
- `psql`-Variableneinsetzung berichtigt (Tom)
- `\d` zeigt Strukturen von temporären Tabellen (Tom)
- `\d` zeigt Fremdschlüssel (Rod)
- \? berücksichtigt \pset pager (Bruce)
- `psql` berichtet beim Start seine Versionsnummer (Tom)
- Bei `\copy` können Spaltennamen angegeben werden (Tom)

libpq

- `$HOME/.pgpass` speichert Host/Benutzer/Passwort-Kombinationen (Alvaro Herrera)
- Funktion PQunescapeBytea() (Patrick Welche)
- Probleme beim Senden von großen Anfragen über nicht blockierende Verbindungen berichtigt (Bernhard Herzog)
- Zeitnahme in `libpq` auf Windows berichtigt (David Ford)
- `libpq` kann mit Servern mit anderer Bezeichnerlänge Benachrichtigungen austauschen (Tom)
- Funktionen `PQescapeString`() und `PQescapeBytea`() für Windows (Bruce)
- SSL mit nicht blockierenden Verbindungen repariert (Jack Bates)
- Parameter für Zeitüberschreitung bei Verbindung (Denis A. Ustimenko)

JDBC

- Kompilierung von JDBC mit JDK 1.4 ermöglicht (Dave)
- Unterstützung für JDBC 3 hinzugefügt (Barry)
- JDBC kann Logniveau mit ?`loglevel`=X in der Verbindungs-URL setzen (Barry)

- ❏ Neue Meldung in Driver.info() gibt die Versionsnummer aus (Barry)
- ❏ Aktualisierbare Ergebnismengen (Raghu Nidagal, Dave)
- ❏ Unterstützung für CallableStatement (Paul Bethe)
- ❏ Möglichkeit, einen Befehl abzubrechen
- ❏ Möglichkeit, eine Zeile aufzufrischen (Dave)
- ❏ MD5-Verschlüsselung mit Multibyte-Servern berichtigt (Jun Kawai)
- ❏ Unterstützung für vorbereitete Befehle (Barry)

Diverse Schnittstellen

- ❏ ECPG-Fehler mit Oktalzahlen in Apostrophen berichtigt (Michael)
- ❏ `src/interfaces/libpgeasy` nach http://gborg.postgresql.org verlegt (Marc, Bruce)
- ❏ Python-Schnittstelle verbessert (Elliot Lee, Andrew Johnson, Greg Copeland)
- ❏ Neues Verbindungsende-Ereignis in `libpgtcl` (Gerhard Hintermayer)
- ❏ `src/interfaces/libpq++` nach http://gborg.postgresql.org verlegt (Marc, Bruce)
- ❏ `src/interfaces/odbc` nach http://gborg.postgresql.org verlegt (Marc)
- ❏ `src/interfaces/libpgeasy` nach http://gborg.postgresql.org verlegt (Marc, Bruce)
- ❏ `src/interfaces/perl5` nach http://gborg.postgresql.org verlegt (Marc, Bruce)
- ❏ `src/bin/pgaccess` aus Quellcodebaum entfernt, jetzt unter http://www.pgaccess.org (Bruce)
- ❏ Neuer Befehl `pg_on_connection_loss` in `libpgtcl` (Gerhard Hintermayer, Tom)

Quellcode

- ❏ Paralleles Make repariert (Peter)
- ❏ Berichtigungen für Linken von Tcl auf AIX (Andreas Zeugswetter)
- ❏ Kompilierung von PL/Perl auf Cygwin ermöglicht (Jason Tishler)
- ❏ MIPS-Compilierung verbessert (Peter, Oliver Elphick)
- ❏ Autoconf Version 2.53 benötigt (Peter)
- ❏ Readline und Zlib werden von configure erfordert (Peter)
- ❏ Solaris verwendet Intimate Shared Memory (ISM) für bessere Leistung (Scott Brunza, P.J. Josh Rovero)
- ❏ Kompiliert immer mit Syslog, Option `--enable-syslog` entfernt (Tatsuo)
- ❏ Kompiliert immer mit Multibyte, Option `--enable-multibyte` entfernt (Tatsuo)
- ❏ Kompiliert immer mit Locale, Option `--enable-locale` entfernt (Peter)
- ❏ DLL-Erzeugung auf Windows berichtigt (Magnus Naeslund)
- ❏ Verwendung von link() im WAL-Code auf Windows, BeOS berichtigt (Jason Tishler)
- ❏ sys/types.h in c.h eingefügt, in anderen Dateien entfernt (Peter, Bruce)
- ❏ Aufhängen von AIX auf SMP-Maschinen berichtigt (Tomoyuki Niijima)
- ❏ Behandlung von Daten vor 1970 in neueren `glibc`-Bibliotheken berichtigt (Tom)
- ❏ SMP-Sperren auf PowerPC berichtigt (Tom)
- ❏ Verwendung von `gcc -ffast-math` verhindert (Peter, Tom)
- ❏ Bison >= 1.50 wird von Entwicklern benötigt
- ❏ Kerberos-5-Unterstützung funktioniert jetzt mit Heimdal (Peter)
- ❏ Anhang im User's Guide, der SQL-Features auflistet (Thomas)
- ❏ Laden von Modulen verwendet RTLD_NOW (Tom)
- ❏ Neue Fehlerniveaus WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- ❏ Neues Verzeichnis `src/port` enthält ersetzte `libc`-Funktionen (Peter, Bruce)
- ❏ Systemkatalog `pg_namespace` für Schemas (Tom)

- ❏ pg_class.relnamespace für Schemas (Tom)
- ❏ pg_type.typnamespace für Schemas (Tom)
- ❏ pg_proc.pronamespace für Schemas (Tom)
- ❏ Aggregate umstrukturiert, haben jetzt pg_proc-Einträge (Tom)
- ❏ Systemrelationen haben jetzt eigenen Namensraum, Test auf pg_* nicht mehr nötig (Fernando Nasser)
- ❏ TOAST-Indexe heißen jetzt *_index anstatt *_idx (Neil)
- ❏ Namensräume für Operatoren, Operatorklassen (Tom)
- ❏ Zusätzliche Prüfungen für Serverkontrolldatei (Thomas)
- ❏ Neue polnische FAQ (Marcin Mazurek)
- ❏ Unterstützung für POSIX-Semaphore (Tom)
- ❏ Notwendigkeit von Reindizierung dokumentiert (Bruce)
- ❏ Einige interne Bezeichner umbenannt um Win32-Kompilierung zu erleichtern (Jan, Katherine Ward)
- ❏ Dokumentation zur Berechnung von Festplattenplatz (Bruce)
- ❏ KSQO aus GUC entfernt (Bruce)
- ❏ Speicherleck in R-Tree berichtigt (Kenneth Been)
- ❏ Einige Fehlermeldungen einheitlicher gemacht (Bruce)
- ❏ Unbenutzte Systemtabellenspalten entfernt (Peter)
- ❏ Systemspalten wo angebracht NOT NULL gemacht (Tom)
- ❏ snprintf() bevorzugt vor sprintf() verwendet (Neil, Jukka Holappa)
- ❏ Opaque entfernt, durch spezifische Typen ersetzt (Tom)
- ❏ Interne Behandlung von Arrays aufgeräumt (Joe, Tom)
- ❏ pg_atoi('') verboten (Bruce)
- ❏ Parameter wal_files entfernt, weil WAL-Dateien jetzt wiederverwendet werden (Bruce)
- ❏ Versionsnummern für Heap-Seiten (Tom)

Contrib

- ❏ inet-Arrays in contrib/array erlaubt (Neil)
- ❏ GiST-Berichtigungen (Teodor Sigaev, Neil)
- ❏ contrib/mysql aktualisiert
- ❏ Neu contrib/dbsize, zeigt Tabellengröße ohne VACUUM (Peter)
- ❏ Neue Ganzzahlaggregatroutinen in contrib/intagg (mlw)
- ❏ contrib/oid2name verbessert (Neil, Bruce)
- ❏ contrib/tsearch verbessert (Oleg, Teodor Sigaev)
- ❏ In contrib/rserver aufgeräumt (Alexey V. Borzov)
- ❏ Umwandlungshilfsmittel contrib/oracle aktualisiert (Gilles Darold)
- ❏ contrib/dblink aktualisiert (Joe)
- ❏ Optionen von contrib/vacuumlo verbessert (Mario Weilguni)
- ❏ Verbesserungen in contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- ❏ Neu contrib/reindexdb (Shaun Thomas)
- ❏ contrib/isbn_issn um Indizierung ergänzt (Dan Weston)
- ❏ Neu contrib/dbmirror (Steven Singer)
- ❏ contrib/pgbench verbessert (Neil)
- ❏ Tabellenfunktionsbeispiele in contrib/tablefunc (Joe)
- ❏ Neuer Datentyp für Baumstrukturen in contrib/ltree (Teodor Sigaev, Oleg Bartunov)
- ❏ contrib/pg_controldata, pg_resetxlog in Hauptbereich verschoben (Bruce)
- ❏ Berichtigungen in contrib/cube (Bruno Wolff)

- Verbesserungen in `contrib/fulltextindex` (Christopher)

D.5 Version 7.2.3

Veröffentlichungsdatum: 01.10.2002

Diese Version enthält verschiedene Berichtigungen gegenüber 7.2.2, unter anderem, um möglichem Datenverlust vorzubeugen. Datensicherung und -wiederherstellung ist für Benutzer von Version 7.2.* nicht notwendig.

D.6 Version 7.2.2

Veröffentlichungsdatum: 23.08.2002

Diese Version enthält verschiedene Berichtigungen gegenüber 7.2.1. Datensicherung und wiederherstellung ist für Benutzer von Version 7.2.* nicht notwendig.

D.7 Version 7.2.1

Veröffentlichungsdatum: 21.03.2002

Diese Version enthält verschiedene Berichtigungen gegenüber 7.2.1. Datensicherung und wiederherstellung ist für Benutzer von Version 7.2 nicht notwendig.

D.8 Version 7.2

Veröffentlichungsdatum: 04.02.2002

D.8.1 Überblick

Diese Version verbessert PostgreSQL für Anwendungen mit großen Datenmengen.

Wichtige Änderungen in dieser Version:

`VACUUM`

Durch `VACUUM` werden keine Tabellen mehr gesperrt, die Tabellen können normal weiterverwendet werden. Ein neuer Befehl `VACUUM FULL` bietet das alte Verhalten, bei dem die Tabelle gesperrt wird, und die Größe der Tabelle auf der Festplatte verkleinert wird.

Transaktionen

Es gibt keine Probleme mehr, wenn die Installation vier Milliarden Transaktionen überschreitet.

OIDs

OIDs sind jetzt optional. Benutzer können Tabellen ohne OIDs erzeugen, wenn der OID-Verbrauch übermäßig ist.

Optimierer

Das System berechnet jetzt mit `ANALYZE` Histogrammstatistiken, wodurch der Optimierer viel bessere Entscheidungen treffen kann.

Sicherheit

Die neue MD5-Verschlüsselungsoption erlaubt sicherere Speicherung und Übertragung von Passwörtern. Auf Linux- und BSD-Systemen gibt es einen neue Authentifizierungsmöglichkeit über Unix-Domain-Sockets.

Statistiken

Administratoren können das neue Statistikmodul verwenden, um detaillierte Informationen über die Verwendung von Tabellen und Indexen zu erhalten.

Internationalisierung

Die Meldungen von Programmen und Bibliotheken können jetzt in verschiedenen Sprachen ausgegeben werden.

D.8.2 Umstieg auf Version 7.2

Alle Benutzer, die von einer früheren Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen.

Beachten Sie die folgenden Inkompatibilitäten:

- Die Bedeutung des Befehls `VACUUM` hat sich in dieser Version geändert. Sie sollten eventuell Ihre Wartungsprozeduren anpassen.
- In dieser Version ergeben Vergleiche mit `= NULL` immer falsch (genauer gesagt den NULL-Wert). Frühere Versionen wandelten diese Syntax automatisch in `IS NULL` um. Das alte Verhalten kann mit einer Einstellung in `postgresql.conf` wieder angeschaltet werden.
- Die Konfiguration in `pg_hba.conf` und `pg_ident.conf` wird jetzt nur nach einem `SIGHUP` neu geladen, nicht bei jeder Verbindung.
- Die Funktion `octet_length()` gibt jetzt die unkomprimierte Datenlänge zurück.
- Die Datums- und Zeitangabe `'current'` ist nicht mehr verfügbar. Sie werden Ihre Anwendungen umschreiben müssen.
- Die Funktionen `timestamp()`, `time()` und `interval()` sind nicht mehr verfügbar. Statt `timestamp()` verwenden Sie `timestamp 'wert'` oder `CAST`.

Die Syntax `SELECT ... LIMIT x,y` wird in der nächsten Version entfernt werden. Sie sollten Ihre Anfragen mit getrennten `LIMIT`- und `OFFSET`-Klauseln schreiben, z.B. `LIMIT 10 OFFSET 20`.

D.9 Version 7.1.3

Veröffentlichungsdatum: 15.08.2001

Diese Version enthält verschiedene Berichtigungen gegenüber 7.1.2. Datensicherung und wiederherstellung ist für Benutzer von Version 7.1.* nicht notwendig.

D.10 Version 7.1.2

Veröffentlichungsdatum: 11.05.2001

Diese Version enthält verschiedene Berichtigungen gegenüber 7.1.1. Datensicherung und wiederherstellung ist für Benutzer von Version 7.1.* nicht notwendig.

D.11 Version 7.1.1

Veröffentlichungsdatum: 05.05.2001

Diese Version enthält verschiedene Berichtigungen gegenüber 7.1. Datensicherung und wiederherstellung ist für Benutzer von Version 7.1 nicht notwendig.

D.12 Version 7.1

Veröffentlichungsdatum: 13.04.2001

Diese Version beseitigt Einschränkungen, die seit langer Zeit im PostgreSQL-Code bestanden haben.

Wichtige Änderungen in dieser Version:

Write-Ahead Log (WAL)

> Um die Beständigkeit der Datenbank im Falle eines Betriebssystemabsturzes zu erhalten, haben frühere Versionen von PostgreSQL all Datenänderungen am Ende jeder Transaktion auf die Festplatte zurückgeschrieben. Mit WAL muss nur eine Logdatei zurückgeschrieben werden, wodurch die Leistung erheblich verbessert wird. Wenn Sie die Option `-F` verwendet haben, um dieses Zurückschreiben zu verhindern, sollten Sie dies jetzt überdenken.

TOAST

> Frühere Versionen hatten eine eingebaute Grenze der Zeilengröße, meistens 8 kB bis 32 kB. Diese Grenze machte lange Textfelder schwierig. Mit TOAST können lange Zeilen beliebiger Größe mit guter Leistung gespeichert werden.

Äußere Verbunde

> Äußere Verbunde werden jetzt unterstützt. Die Ausweichlösung mit `UNION` und `NOT IN` ist nicht mehr nötig. Wir verwenden die Verbundsyntax aus dem SQL-Standard.

Funktionsmanager

> Der vorherige C-Funktionsmanager konnte nicht richtig mit NULL-Werten umgehen und unterstützt keine 64-Bit-CPUs (Alpha). Sie können Ihre alten eigenen Funktionen weiter verwenden, sollten Sie aber in der Zukunft für die neue Funktions-Schnittstelle umschreiben.

Komplexe Anfragen

> Eine große Zahl komplexer Anfragen, die in früheren Versionen nicht unterstützt wurden, funktionieren jetzt. Viele Kombinationen aus Sichten, Aggregatfunktionen, `UNION`, `LIMIT`, Cursorn, Unteranfragen und vererbten Tabellen funktionieren jetzt richtig. Geerbte Tabellen werden jetzt automatisch mit einbezogen. Unteranfragen in `FROM` werden jetzt unterstützt.

Alle Benutzer, die von einer früheren Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen.

D.13 Version 7.0.3

Veröffentlichungsdatum: 11.11.2000

Diese Version enthält verschiedene Berichtigungen gegenüber 7.0.2. Datensicherung und wiederherstellung ist für Benutzer von Version 7.0.* nicht notwendig.

D.14 Version 7.0.2

Veröffentlichungsdatum: 05.06.2000

Dies ist Version 7.0.1 nochmal mit der Dokumentation eingepackt. Datensicherung und wiederherstellung ist für Benutzer von Version 7.0.* nicht notwendig.

D.15 Version 7.0.1

Veröffentlichungsdatum: 01.06.2000

Diese Version enthält verschiedene Berichtigungen gegenüber 7.0. Datensicherung und wiederherstellung ist für Benutzer von Version 7.0 nicht notwendig.

D.16 Version 7.0

Veröffentlichungsdatum: 08.05.2000

D.16.1 Überblick

Diese Version enthält Verbesserungen in vielen Bereichen und zeigt die anhaltende Entwicklung von PostgreSQL. In Version 7.0 gibt es mehr Verbesserungen und Berichtigungen als in irgendeiner früheren Version. Die Entwickler sind sich sicher, dass dies bisher die beste Version ist; wir tun unser Bestes, um nur solide Versionen zu veröffentlichen, und diese ist da keine Ausnahme.

Wichtige Änderungen in dieser Version:

Fremdschlüssel

> Fremdschlüssel sind jetzt implementiert, außer die Variante `PARTIAL MATCH`. Viele Anwender haben danach gefragt, und jetzt sind wir froh, dass wir diese Funktionalität anbieten können.

Überholung des Optimierers

> Aufbauend auf der vor einem Jahr begonnenen Arbeit wurde der Optimierer verbessert. Daraus ergibt sich bessere Anfrageplanauswahl, bessere Leistung und weniger Speicherverbrauch.

Aktualisiertes `psql`

> `psql`, unser interaktives Terminalprogramm, wurde um viele Funktionen verbessert. Einzelheiten finden Sie auf der Referenzseite von `psql`.

Verbundsyntax

Die Verbundsyntax aus dem SQL-Standard wird jetzt unterstützt, allerdings in dieser Version nur als `INNER JOIN`. `JOIN`, `NATURAL JOIN`, `JOIN/USING` und `JOIN/ON` stehen zur Verfügung, ebenso Spaltenkorrelationsnamen.

D.16.2 Umstieg auf Version 7.0

Alle Benutzer, die von einer früheren Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen. Wenn Sie von Version 6.5.* umsteigen, können Sie stattdessen `pg_upgrade` verwenden, um das Datenverzeichnis zu aktualisieren; eine vollständige Datensicherung und -wiederherstellung ist jedoch die robusteste Methode des Umstiegs.

Einige mögliche Kompatibilitätprobleme beim Umstieg auf die neue Version sind:

- Die Typen datetime und timespan wurden durch die im SQL-Standard definierten Typen timestamp und interval ersetzt. Obwohl versucht wurde, den Umstieg zu vereinfachen, indem PostgreSQL die alten Typnamen erkennt und in die neuen Typnamen umwandelt, ist dieser Mechanismus für Ihre bestehende Anwendung vielleicht nicht vollständig transparent.
- Der Optimierer wurde im Bereich der Kostenschätzung von Anfragen erheblich verbessert. In einigen Fällen führt das zu verringerten Laufzeiten von Anfragen, weil der Optimierer eine bessere Wahl des Plans tätigen kann. In wenigen Fällen, meistens mit pathologischer Datenverteilung, kann die Laufzeit von Anfragen jedoch steigen. Wenn Sie mit großen Datenmengen umgehen, sollten Sie die Leistung Ihrer Anfragen überprüfen.
- Die Zeichenkettenfunktion `char_length` ist jetzt eine richtige Funktion. In früheren Versionen wurden die Aufrufe in die Funktion `length` umgewandelt, was zu Unklarheiten führen konnte, wenn andere Typen auch eine Funktion `length` hatten, zum Beispiel die geometrischen Typen.

D.17 Version 6.5.3

Veröffentlichungsdatum: 13.10.1999

Diese Version enthält verschiedene Berichtigungen gegenüber 6.5.2. Ein neues `PgAccess` ist enthalten und ein Windows-spezifischer Fehler wurde behoben. Datensicherung und wiederherstellung ist für Benutzer von Version 6.5.* nicht notwendig.

D.18 Version 6.5.2

Veröffentlichungsdatum: 15.09.1999

Diese Version enthält verschiedene Berichtigungen gegenüber 6.5.1. Datensicherung und wiederherstellung ist für Benutzer von Version 6.5.1 nicht notwendig.

D.19 Version 6.5.1

Veröffentlichungsdatum: 15.07.1999

Diese Version enthält verschiedene Berichtigungen gegenüber 6.5. Datensicherung und wiederherstellung ist für Benutzer von Version 6.5 nicht notwendig.

D.20 Version 6.5

Veröffentlichungsdatum: 09.06.1999

Diese Version stellt einen großen Schritt in der Beherrschung des von Berkeley übernommenen Quellcodes durch das Entwicklungsteam dar. Sie werden sehen, dass wir jetzt mit Leichtigkeit größere Funktionalität hinzufügen, da sich die Größe und Erfahrung des Entwicklungsteams ständig verbessern.

Hier ist eine kurze Zusammenfassung der bemerkenswerteren Änderungen:

Multiversion concurrency control (MVCC)

Das alte Tabellensperrsystem wurde entfernt und durch ein Sperrensystem ersetzt, das den meisten kommerziellen Datenbanksystemen überlegen ist. In einem traditionellen System wird jede geänderte Zeile gesperrt, bis die Transaktion abgeschlossen wird. MVCC bedient sich des natürlichen Multiversionsverhaltens von PostgreSQL, um Lesevorgänge zu ermöglichen, konsistente Daten zu lesen, während gleichzeitig Schreibvorgänge stattfinden. Schreibvorgänge verwenden weiterhin das kompakte Transaktionssystem mit `pg_log`. All das kann geschehen, ohne, wie bei traditionellen Datenbanksystemen, jede Zeile sperren zu müssen. Im Prinzip haben wir also nicht mehr einfache Sperren auf Tabellenebene, sondern sogar noch etwas Besseres als Sperren auf Zeilenebene.

“Hot Backups” mit `pg_dump`

`pg_dump` nutzt das neue MVCC-System und kann eine konsistente Datenbanksicherung erstellen, während die Datenbank für Anfragen verfügbar bleibt.

Datentyp numeric

Wir haben jetzt einen richtigen numeric-Datentyp mit benutzerdefinierter Präzision.

Temporäre Tabellen

Temporäre Tabellen haben garantiert einmalige Namen innerhalb einer Datenbanksitzung und werden am Ende der Sitzung zerstört.

Neue SQL-Funktionalität

Wir haben jetzt Unterstützung für `CASE`, *`INTERSECT`* und `EXCEPT`. Neu sind auch `LIMIT/OFFSET`, `SET TRANSACTION ISOLATION LEVEL`, `SELECT ... FOR UPDATE` ein verbesserter `LOCK TABLE`-Befehl.

Geschwindigkeit

Die Geschwindigkeit von PostgreSQL wurde weiter verbessert, insbesondere in den Bereichen Speicherzuweisung, Tabellenverbunden und Zeilenübertragungsroutinen.

Ports

Wir erweitern unsere Portliste ständig, diesmal unter anderem um `Windows NT/ix86` und `NetBSD/arm32`.

Schnittstellen

Die meisten Schnittstellen haben neue Versionen und die bestehende Funktionalität wurde verbessert.

Dokumentation

Neues und aktualisiertes Material findet sich überall in der Dokumentation. Neue FAQs gibt es für die Plattformen SGI und AIX. Das *Tutorial* hat einführende Informationen über SQL von Stefan Simkovics. Im *User's Guide* gibt es neue Referenzseiten über postmaster und weitere Hilfsprogramme und einen neuen Anhang über die Interna der Datums- und Zeitunterstützung. Der *Administrator's Guide* hat ein neues Kapitel über Problemlösung von Tom Lane. Und der *Programmer's Guide* hat eine Beschreibung der Anfrageverarbeitung, ebenfalls von Stefan, und Einzelheiten, wie man den Quellcodebaum von PostgreSQL über CVS oder CVSup erhalten kann.

D.20.1 Umstieg auf Version 6.5

Alle Benutzer, die von einer früheren Versionen umsteigen wollen, müssen ihre Daten mit pg_dump sichern und wiederherstellen. pg_upgrade kann für den Umstieg auf diese Version nicht verwendet werden, weil sich die Struktur der Tabellendaten auf der Festplatte gegenüber früheren Versionen geändert hat.

Das neue MVCC-System ergibt in Mehrbenutzerumgebungen ein etwas anderes Verhalten. Lesen und verstehen Sie den Rest dieses Abschnitts, um sich zu versichern, dass ihre bestehenden Anwendungen das von Ihnen gewünschte Verhalten haben.

Da Lesevorgänge in 6.5 keine Daten sperren, unabhängig von Transaktionsisolationsgrad, können die vom einer Transaktion gelesenen Daten von einer anderen überschrieben werden. Anders ausgedrückt, wenn eine Zeile von SELECT zurückgegeben wird, heißt das weder, dass die Zeile zu dem Zeitpunkt, als sie zurückübergeben wurde (also irgendwann, nachdem der Befehl oder die Transaktion begonnen hatte), noch existiert, noch, dass die Zeile vor Lösch- oder Aktualisierungsversuchen von gleichzeitigen Transaktionen geschützt ist, solange die eigenen Transaktion noch nicht zu Ende ist.

Um die tatsächliche Existenz einer Zeile zu sichern und sie gegen gleichzeitige Aktualisierungen zu schützen, müssen Sie SELECT FOR UPDATE oder einen passenden LOCK TABLE-Befehl verwenden. Das sollten Sie bedenken, wenn Sie Anwendungen von früheren PostgreSQL-Versionen und anderen Datenbanksystemen portieren.

Bedenken Sie des eben Gesagte auch, wenn Sie die Trigger in contrib/refint.* für die referenzielle Integrität verwenden. Jetzt sind zusätzliche Techniken erforderlich. Ein Möglichkeit ist LOCK elterntabelle IN SHARE ROW EXCLUSIVE MODE zu verwenden, wenn eine Transaktion einen Primärschlüssel aktualisieren oder löschen möchte, und LOCK elterntabelle IN SHARE MODE, wenn eine Transaktion eine Primärschlüssel aktualisieren oder einfügen möchte.

Beachten Sie, dass, wenn Sie eine Transaktion im Modus SERIALIZABLE ausführen, Sie die LOCK-Befehle vor dem ersten Datenmodifikationsbefehl ausführen müssen (SELECT, INSERT, UPDATE, DELETE, FETCH, COPY TO).

Die Probleme werden in der Zukunft verschwinden, wenn es möglich sein wird, Daten aus nicht abgeschlossenen Transaktionen zu lesen (unabhängig vom Transaktionsisolationsgrad), und richtige referenzielle Integrität implementiert wird.

D.21 Version 6.4.2

Veröffentlichungsdatum: 20.12.1998

Version 6.4.1 war nicht richtig eingepackt. Diese Version enthält außerdem eine Berichtigung. Datensicherung und wiederherstellung ist für Benutzer von Version 6.4.* nicht notwendig.

D.22 Version 6.4.1

Veröffentlichungsdatum: 18.12.1998

Diese Version enthält verschiedene Berichtigungen gegenüber 6.4. Datensicherung und wiederherstellung ist für Benutzer von Version 6.4 nicht notwendig.

D.23 Version 6.4

Veröffentlichungsdatum: 30.10.1998

In dieser Version gibt es viele neue Features und Verbesserungen. Dank unserer Entwickler hat fast jeder Teil des Systems seit der letzten Version Aufmerksamkeit erhalten. Hier ist eine kurze, unvollständige Zusammenfassung:

- Sichten und Regeln funktionieren jetzt, dank umfangreichem neuen Code im Umschreiberegelsystem von Jan Wieck. Er hat auch ein Kapitel im *Programmer's Guide* darüber geschrieben.
- Jan hat auch eine zweite prozedurale Sprache, `PL/pgSQL`, geschrieben, die die ursprüngliche prozedurale Sprache `PL/Tcl`, die er in der letzten Version beigesteuert hat, ergänzt.
- Wir haben jetzt optionale Unterstützung für Mehrbyte-Zeichensätze von Tatsuo Ishii, die die vorhandene Locale-Unterstützung ergänzt.
- Die Client/Server-Kommunikation wurde aufgeräumt, mit besserer Unterstützung für asynchrone Benachrichtigungen und Interrupts, von Tom Lane.
- Der Parser führt jetzt automatische Typumwandlungen durch, um die Argumente an die verfügbaren Operatoren und Funktionen anzupassen und um Spalten und Ausdrücke an die Zielspalten anzupassen. Das ist ein allgemein gefasster Mechanismus, der die Typenerweiterbarkeit in PostgreSQL unterstützt. Ein neues Kapitel im *User's Guide* behandelt dieses Thema.
- Drei neue Datentypen wurden hinzugefügt. Zwei Typen, inet und cidr, unterstützen verschiedene Formen von IP-Netzwerken und Hostadressen. Auf manchen Plattformen gibt es einen Typ für 8-Byte-Ganzzahlen. Einzelheiten finden Sie im Datentypen-Kapitel im *User's Guide*. Ein vierter Typ, serial, wird jetzt vom Parser als Kombination aus dem Typ int4, einer Sequenz und einem Unique Index unterstützt.
- Verschiedene SQL92-kompatible Syntaxelemente wurde hinzugefügt, zum Beispiel `INSERT DEFAULT VALUES`
- Das automatische Konfigurations- und Installationssystem hat einige Aufmerksamkeit erfahren und sollte jetzt für mehr Plattformen als je zuvor robuster sein.

Alle Benutzer, die von einer früheren PostgreSQL-Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen.

D.24 Version 6.3.2

Veröffentlichungsdatum: 07.04.1998

Diese Version enthält verschiedene Verbesserungen gegenüber 6.3.1. Datensicherung und wiederherstellung ist für Benutzer von Version 6.3.* nicht notwendig.

D.25 Version 6.3.1

Veröffentlichungsdatum: 23.03.1998

Diese Version enthält verschiedene Verbesserungen gegenüber 6.3. Datensicherung und wiederherstellung ist für Benutzer von Version 6.3 nicht notwendig.

D.26 Version 6.3

Veröffentlichungsdatum: 01.03.1998

In dieser Version gibt es viele neue Features und Verbesserungen. Hier ist eine kurze, unvollständige Zusammenfassung:

- Viele neue SQL-Features, einschließliche Unteranfragen (außer in der Select-Liste).
- Unterstützung für Umgebungsvariablen für Zeitzone und Datumsformat auf der Clientseite.
- Unix-Domain-Sockets für Client/Server-Kommunikation. Das ist jetzt die Voreinstellung, also müssen Sie `postmaster` jetzt eventuell mit der Option `-i` starten.
- Bessere Passwortauthentifizierung. Die Standardprivilegien für Tabellen wurden geändert.
- Die alte "Zeitreise"-Funktionalität wurde entfernt. Die Leistung wurde verbessert.

Alle Benutzer, die von einer früheren PostgreSQL-Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen.

D.27 Version 6.2.1

Veröffentlichungsdatum: 17.10.1997

6.2.1 ist eine Version mit Fehlerberichtigungen und Verbesserungen gegenüber 6.2. Datensicherung und -wiederherstellung ist für Benutzer von Version 6.2 nicht notwendig.

D.28 Version 6.2

Veröffentlichungsdatum: 02.10.1997

Alle Benutzer, die von einer früheren PostgreSQL-Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen. Beachten Sie, dass `pg_dump` und `pg_dumpall` aus Version 6.2 verwendet werden sollten, um die Version-6.1-Datenbank zu sichern.

Die Benutzer, die von Postgres95-Versionen 1.* umsteigen wollen, sollten erst auf Version 1.09 umsteigen, weil das Ausgabeformat von `COPY` in Version 1.02 verbessert wurde.

D.29 Version 6.1.1

Veröffentlichungsdatum: 22.07.1997

Dies ist eine kleine Version mit Fehlerberichtigungen. Datensicherung und -wiederherstellung ist nicht nötig, wenn Sie von Version 6.1 umsteigen, aber sie ist für alle Versionen vor 6.1 nötig. Weitere Einzelheiten finden Sie in den Informationen zu Version 6.1.

D.30 Version 6.1

Veröffentlichungsdatum: 08.06.1997

Die Regressionstests wurden umfassend verändert und an die Version 6.1 von PostgreSQL angepasst.

Drei neue Datentypen (datetime, timespan und circle) wurden hinzugefügt. Punkte, Rechtecke, Pfade und Polygone haben jetzt in allen Datentypen konsistente Ausgabeformate. Die Polygon-Ausgabe in `misc.out` wurde nur kurz auf Richtigkeit gegenüber der ursprünglichen Ausgabe der Regressionstests überprüft.

PostgreSQL 6.1 bietet einen neuen, alternativen Optimierer mit genetischen Algorithmen. Diese Algorithmen erzeugen zufällige Reihenfolgen in Anfrageergebnissen, wenn die Anfrage mehrere Bedingungen oder mehrere Tabellen enthält (wodurch der Optimierer die Auswahl bei der Reihenfolge der Auswertung hat). Mehrere Regressionstests wurden verändert, um ihre Ergebnisse ausdrücklich zu sortieren und damit unempfindlich gegen Optimiererentscheidungen. Einige Regressionstests für Datentypen, die von Natur her nicht sortiert werden können (z.B. Punkte und Zeitspannen), und Tests mit diesen Typen wurden ausdrücklich von `set geqo to 'off'` und `reset geqo` eingeklammert.

Die Interpretation von Arrayangaben (geschweifte Klammern um atomare Werte) hat sich scheinbar irgendwann geändert, seit die ursprünglichen Regressionstests erzeugt worden waren. Die aktuellen Dateien `./expected/*.out` spiegeln die neue Interpretation wieder, welche vielleicht nicht richtig ist.

Der Regressionstest `float8` scheitert zumindest auf einigen Plattformen. Das liegt an unterschiedlichen Implementierungen von `pow()` und `exp()` und am Mechanismus, mit dem Überlauf und Unterlauf signalisiert werden.

Die zufälligen Ergebnisse im Test `random` sollten dafür sorgen, dass der Test zufällig durchfällt, da die Regressionstests mit einem einfachen `diff` ausgewertet werden. Auf der Testmaschine (Linux/`GCC`/i686) scheint das aber nicht der Fall zu sein.

Alle Benutzer, die von Versionen 6.0 umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen. Die Benutzer, die von Postgres95-Versionen 1.* umsteigen wollen, sollten erst auf Version 1.09 umsteigen, weil das Ausgabeformat von `COPY` in Version 1.02 verbessert wurde.

D.31 Version 6.0

Veröffentlichungsdatum: 29.01.1997

Alle Benutzer, die von einer früheren PostgreSQL-Versionen umsteigen wollen, müssen ihre Daten mit `pg_dump` sichern und wiederherstellen.

Die Benutzer, die von Postgres95-Versionen vor 1.09 umsteigen wollen, sollten erst auf Version 1.09 umsteigen, weil das Ausgabeformat von `COPY` in Version 1.02 verbessert wurde.

Systemkataloge

In den Systemkatalogen speichern relationale Datenbanksysteme ihre Schemadaten, wie Informationen über Tabellen und Spalten, sowie interne Verwaltungsinformationen. In PostgreSQL sind die Systemkataloge normale Tabellen. Sie können die Tabellen löschen und neu erzeugen, Werte einfügen und ändern und auf diese Weise ziemlich einfach das Datenbanksystem komplett durcheinander bringen. Normalerweise sollte man die Systemkataloge nicht von Hand ändern, denn dafür gibt es immer SQL-Befehle. (Zum Beispiel fügt `CREATE DATABASE` eine Spalte in den Katalog `pg_database` ein – und erzeugt die Datenbank auch tatsächlich auf der Festplatte.) Es gibt einige Ausnahmen für besonders esoterische Operationen, wie das Erzeugen einer neuen Indexmethode.

E.1 Überblick

Tabelle E.1 listet die Systemkataloge. Einzelheiten zu jedem Systemkatalog folgen weiter unten.

Die meisten Systemkataloge werden bei der Erzeugung einer Datenbank aus der Template-Datenbank kopiert und sind danach nur für diese eine Datenbank zuständig. Ein paar Kataloge werden aber von allen Datenbanken in einem Cluster gemeinsam verwendet; das ist in der Beschreibung des Katalogs angegeben.

Katalogname	Zweck
`pg_aggregate`	Aggregatfunktionen
`pg_am`	Indexmethoden
`pg_amop`	Operatoren für Operatorklassen
`pg_amproc`	Unterstützungsprozeduren für Operatorklassen
`pg_attrdef`	Spaltenvorgabewerte
`pg_attribute`	Tabellenspalten (“Attribute”)
`pg_cast`	Datentypumwandlungen
`pg_class`	Tabellen, Sichten, Indexe, Sequenzen (“Relationen”)

Tabelle E.1: Systemkataloge

Katalogname	Zweck
pg_constraint	Check-Constraints, Unique-Constraints, Primärschlüssel-Constraints, Fremdschlüssel-Constraints
pg_conversion	Zeichensatzkonversionsinformationen
pg_database	Datenbanken in diesem Cluster
pg_depend	Abhängigkeiten zwischen Datenbankobjekten
pg_description	Beschreibungen oder Kommentare über Datenbankobjekte
pg_group	Gruppen von Datenbankbenutzern
pg_index	zusätzliche Indexinformationen
pg_inherits	Tabellenvererbungshierarchie
pg_language	Sprachen für Funktionen
pg_largeobject	Large Objects
pg_listener	Unterstützung für asynchrone Benachrichtigungen
pg_namespace	Schemas
pg_opclass	Operatorklassen für Indexzugriffsmethoden
pg_operator	Operatoren
pg_proc	Funktionen und Prozeduren
pg_rewrite	Anfrageumschreiberegeln
pg_shadow	Datenbankbenutzer
pg_statistic	Planerstatistiken
pg_trigger	Trigger
pg_type	Datentypen

Tabelle E.1: Systemkataloge (Forts.)

E.2 pg_aggregate

Der Katalog pg_aggregate speichert Informationen über Aggregatfunktionen. Eine Aggregatfunktion ist eine Funktion, die einen Wert aus einer Menge von Werten berechnet (meistens aus einer Spalte jeder Zeile, die eine bestimmte Bedingung erfüllt). Typische Aggregatfunktionen sind sum, count und max. Jeder Eintrag in pg_aggregate ist eine Erweiterung eines Eintrags in pg_proc. Der Eintrag in pg_proc enthält den Namen der Aggregatfunktion, die Eingabe- und Ausgabedatentypen und andere Informationen, die einer normalen Funktion ähnlich sind.

Name	Typ	Verweist auf	Beschreibung
aggfnoid	regproc	pg_proc.oid	OID der Aggregatfunktion in pg_proc
aggtransfn	regproc	pg_proc.oid	Übergangsfunktion
aggfinalfn	regproc	pg_proc.oid	Abschlussfunktion (null, wenn keine)

Tabelle E.2: Spalten von pg_aggregate

Name	Typ	Verweist auf	Beschreibung
aggtranstype	oid	pg_type.oid	Der Typ des internen Zustandswerts der Aggregatfunktion
agginitval	text		Der Anfangswert des Zustandswertes. Dies ist ein Textfeld, das den Anfangswert in der Zeichenkettendarstellung enthält. Wenn der Wert der NULL-Wert ist, dann fängt der Zustandswert mit dem NULL-Wert an.

Tabelle E.2: Spalten von pg_aggregate (Forts.)

Neue Aggregatfunktionen werden mit dem Befehl CREATE AGGREGATE registriert. Weitere Informationen über das Schreiben von neuen Aggregatfunktionen, die Bedeutung der Übergangsfunktion usw. finden Sie in Abschnitt 33.13.

E.3 pg_am

Der Katalog pg_am speichert Informationen über Indexzugriffsmethoden. Es gibt eine Zeile für jede vom System unterstützte Indexmethode.

Name	Typ	Verweist auf	Beschreibung
amname	name		Name der Zugriffsmethode
amowner	int4	pg_shadow.usesysid	Benutzernummer des Eigentümers (gegenwärtig nicht verwendet)
amstrategies	int2		Anzahl der Operatorstrategien für diese Zugriffsmethode
amsupport	int2		Anzahl der Unterstützungsroutinen für diese Zugriffsmethode
amorderstrategy	int2		Null, wenn der Index keine Sortierreihenfolge bietet, ansonsten die Strategienummer des Sortieroperators
amcanunique	bool		Unterstützt die Zugriffsmethode Unique Indexe?
amcanmulticol	bool		Unterstützt die Zugriffsmethode mehrspaltige Indexe?
amindexnulls	bool		Unterstützt die Zugriffsmethode NULL-Werte im Index?
amconcurrent	bool		Unterstützt die Zugriffsmethode gleichzeitige Aktualisierungen?
amgettuple	regproc	pg_proc.oid	Funktion “Nächstes gültiges Tupel”
aminsert	regproc	pg_proc.oid	Funktion “Füge dieses Tupel ein”
ambeginscan	regproc	pg_proc.oid	Funktion “Starte einen neuen Scan”
amrescan	regproc	pg_proc.oid	Funktion “Starte diesen Scan neu”
amendscan	regproc	pg_proc.oid	Funktion “Beende diesen Scan”
ammarkpos	regproc	pg_proc.oid	Funktion “Markiere aktuelle Scan-Position”
amrestrpos	regproc	pg_proc.oid	Funktion “Stelle markierte Scan-Position wieder her”
ambuild	regproc	pg_proc.oid	Funktion “Baue neuen Index”

Tabelle E.3: Spalten von pg_am

Name	Typ	Verweist auf	Beschreibung
ambulkdelete	regproc	pg_proc.oid	Massenlöschfunktion
amcostestimate	regproc	pg_proc.oid	Funktion zum Schätzen des Aufwands für einen Indexscan

Tabelle E.3: Spalten von pg_am (Forts.)

Eine Indexzugriffsmethode, die mehrspaltige Indexe unterstützt (amcanmulticol ist wahr), *muss* NULL-Werte unterstützen, weil der Planer davon ausgeht, dass der Index für Anfragen verwendet werden kann, die nur einige der ersten Spalten verwenden. Betrachten Sie zum Beispiel einen Index für (a, b) und eine Anfrage mit WHERE a = 4. Das System geht davon aus, dass es den Index verwenden kann, um Zeilen mit a = 4 zu finden, was aber falsch wäre, wenn der Index Zeilen, wo b den NULL-Wert hat, auslässt. Es ist jedoch zulässig, Zeilen, bei denen die erste Spalte den NULL-Wert hat, auszulassen. (GiST macht das gegenwärtig.) amindexnulls sollte nur auf wahr gesetzt werden, wenn die Indexzugriffsmethode alle Zeilen indiziert, einschließlich aller Kombinationen mit NULL-Werten.

E.4 pg_amop

Der Katalog pg_amop speichert Informationen über die zu einer Operatorklasse gehörenden Operatoren. Es gibt eine Zeile für jeden Operator, der ein Mitglied einer Operatorklasse ist.

Name	Typ	Verweist auf	Beschreibung
amopclaid	oid	pg_opclass.oid	Die Operatorklasse, zu der dieser Eintrag gehört
amopstrategy	int2		Strategienummer des Operators
amopreqcheck	bool		Indextreffer müssen erneut geprüft werden.
amopopr	oid	pg_operator.oid	Die OID des Operators

Tabelle E.4: Spalten von pg_amop

E.5 pg_amproc

Der Katalog pg_amproc speichert Informationen über die zu einer Operatorklasse gehörenden Unterstützungsprozeduren. Es gibt eine Zeile für jede Unterstützungsprozedur, die zu einer Operatorklasse gehört.

Name	Typ	Verweist auf	Beschreibung
amopclaid	oid	pg_opclass.oid	Die Operatorklasse, zu der dieser Eintrag gehört
amprocnum	int2		Nummer der Unterstützungsprozedur
amproc	regproc	pg_proc.oid	Die OID der Prozedur

Tabelle E.5: Spalten von pg_amproc

E.6 pg_attrdef

Der Katalog pg_attrdef speichert Spaltenvorgabewerte. Die restlichen Informationen über Spalten sind in pg_attribute. Nur Spalten mit einem ausdrücklichen Vorgabewert (der bei der Erzeugung der Tabelle oder der Spalte angegeben wurde) haben hier einen Eintrag.

Name	Typ	Verweist auf	Beschreibung
adrelid	oid	pg_class.oid	Die Tabelle, zu der diese Spalte gehört
adnum	int2	pg_attribute.attnum	Die Nummer der Spalte
adbin	text		Die interne Darstellung des Spaltenvorgabewerts
adsrc	text		Eine für Anwender lesbare Darstellung des Vorgabewerts

Tabelle E.6: Spalten von pg_attrdef

E.7 pg_attribute

Der Katalog pg_attribute speichert Informationen über Tabellenspalten. Es gibt genau eine Zeile in pg_attribute für jede Spalte in jeder Tabelle in der Datenbank. (Es gibt auch Einträge für die Attribute von Indexen und anderen Objekten. Siehe pg_class.)

Der Begriff Attribut bedeutet dasselbe wie Spalte und wird aus historischen Gründen verwendet.

Name	Typ	Verweist auf	Beschreibung
attrelid	oid	pg_class.oid	Die Tabelle, zu der diese Spalte gehört
attname	name		Der Spaltenname
atttypid	oid	pg_type.oid	Der Datentyp dieser Spalte
attstattarget	int4		attstattarget kontrolliert, wie detailliert die von ANALYZE für diese Spalte gesammelten Statistiken sein sollen. Null gibt an, dass keine Statistiken gesammelt werden sollen. Ein negativer Wert gibt an, dass das Standardstatistikziel verwendet werden soll. Die genaue Bedeutung von positiven Werten hängt vom Datentyp ab. Bei skalaren Datentypen ist attstattarget der Zielwert für die "häufigsten Werte" und für die Histogrammpartitionierung.
attlen	int2		Eine Kopie von pg_type.typlen für den Datentyp dieser Spalte
attnum	int2		Die Nummer der Spalte. Normale Spalten zählen von 1 an. Systemspalten wie oid haben (willkürliche) negative Nummern.
attndims	int4		Anzahl der Dimensionen, wenn die Spalte einen Arraytyp hat; ansonsten 0. (Die Anzahl der Dimensionen eines Arrays wird gegenwärtig nicht durchgesetzt, daher bedeutet ein Wert ungleich null einfach "Es ist ein Array".)
attcacheoff	int4		Abgespeichert immer -1, aber wenn die Zeile in einen Tupeldeskriptor im Hauptspeicher geladen wird, wird dieser Wert möglicherweise auf den Cache-Offset des Attributs im Tupel gesetzt.

Tabelle E.7: Spalten von pg_attribute

Name	Typ	Verweist auf	Beschreibung
atttypmod	int4		atttypmod zeichnet bei der Erzeugung der Tabelle datentypspezifische Werte auf (zum Beispiel die Höchstlänge bei einer varchar-Spalte). Er wird der Eingabefunktion des Typs und der Funktion zur Größeneinstellung übergeben. Der Wert ist in der Regel 1 bei Typen, die atttypmod nicht benötigen.
attbyval	bool		Eine Kopie von pg_type.typbyval für den Datentyp dieser Spalte
attstorage	char		Normalerweise ein Kopie von pg_type.typstorage für den Datentyp dieser Spalte. Bei Datentypen mit variabler Länge kann dieser Wert nach der Erzeugung der Spalte geändert werden, um die Speicherungsstrategie zu kontrollieren.
attisset	bool		Wenn wahr, dann stellt diese Spalte eine Menge dar. In diesem Fall wird in der Spalte eigentlich die OID einer Zeile im Katalog pg_proc gespeichert. Die pg_proc-Zeile enthält den Anfragetext, der diese Menge definiert, d.h., die Anfrage wird ausgeführt, um die Menge zu ermitteln. Die Spalte atttypid (siehe oben) verweist auf den Typ, den die Anfrage zurückgibt, aber die tatsächliche Größe dieser Spalte entspricht einer oid. - Das ist zumindest die Theorie. All das funktioniert heutzutage wohl nicht mehr richtig.
attalign	char		Eine Kopie von pg_type.typalign für den Datentyp dieser Spalte
attnotnull	bool		Hierdurch wird ein NOT-NULL-Constraint dargestellt. Man kann diese Spalte ändern, um den Constraint ein- oder auszuschalten.
atthasdef	bool		Diese Spalte hat einen Vorgabewert. In dem Fall gibt es einen entsprechenden Eintrag im Katalog pg_attrdef, der den eigentlichen Vorgabewert definiert.
attisdropped	bool		Diese Spalte wurde gelöscht und ist nicht mehr gültig. Eine gelöschte Spalte ist weiterhin physikalisch in der Tabelle vorhanden, wird aber vom Parser ignoriert und kann daher nicht in SQL-Befehlen verwendet werden.
attislocal	bool		Diese Spalte ist lokal in der Relation definiert. Beachten Sie, dass eine Spalte gleichzeitig lokal definiert und geerbt sein kann.
attinhcount	int4		Die Anzahl der direkten Vorfahren, die diese Spalte in der Veerbungshierarchie hat. Eine Spalte mit Vorfahren kann nicht gelöscht oder umbenannt werden.

Tabelle E.7: Spalten von pg_attribute (Forts.)

E.8 pg_cast

Der Katalog pg_cast speichert Umwandlungspfade zwischen Datentypen, sowohl eingebaute als auch die mit CREATE CAST erzeugten.

Name	Typ	Verweist auf	Beschreibung
castsource	oid	pg_type.oid	OID des Quelldatentyps
casttarget	oid	pg_type.oid	OID des Zieldatentyps

Tabelle E.8: Spalten von pg_cast

Name	Typ	Verweist auf	Beschreibung
castfunc	oid	pg_proc.oid	Die OID der Funktion, die die Umwandlung durchführt. Null wird gespeichert, wenn die Datentypen binärkompatibel sind (das heißt, dass keine Laufzeitoperation zur Umwandlung nötig ist).
castcontext	char		Gibt an, unter welchen Umständen die Umwandlung ausgeführt werden kann. e bedeutet nur bei ausdrücklichem Aufruf (mit CAST, :: oder der Funktionssyntax). a bedeutet implizit bei der Speicherung in eine Zielspalte oder bei ausdrücklichem Aufruf. i bedeutet implizit in Ausdrücken und auch in den anderen Situationen.

Tabelle E.8: Spalten von pg_cast (Forts.)

E.9 pg_class

Der Katalog pg_class speichert Tabellen und die meisten anderen Dinge, die Spalten haben oder anderweitig mit Tabellen ähnlich sind. Das enthält Indexe (siehe aber auch unter pg_index), Sequenzen, Sichten und einige besondere Relationen; siehe relkind. Diese Objekte nennen wir unten alle "Relationen". Nicht alle Spalten sind für alle Relationstypen von Bedeutung.

Name	Typ	Verweist auf	Beschreibung
relname	name		Name der Tabelle, des Index, der Sicht usw.
relnamespace	oid	pg_namespace.oid	Das Schema, das diese Relation enthält
reltype	oid	pg_type.oid	OID des Datentyps, der dieser Tabelle entspricht, wenn vorhanden (Null für Indexe, welche keinen Eintrag in pg_type haben)
relowner	int4	pg_shadow.usesysid	Eigentümer der Relation
relam	oid	pg_am.oid	Wenn dies ein Index ist, dann die Zugriffsmethode (B-Tree, Hash usw.)
relfilenode	oid		Name der Datei für diese Relation; 0, wenn keine
relpages	int4		Größe der Tabelle auf der Festplatte in Seiten (Größe *BLCKSZ*). Dies ist nur eine vom Planer verwendete Schätzung. Sie wird von VACUUM, ANALYZE und CREATE INDEX aktualisiert.
reltuples	float4		Anzahl der Zeilen in der Tabelle. Dies ist nur eine vom Planer verwendete Schätzung. Sie wird von VACUUM, ANALYZE und CREATE INDEX aktualisiert.
reltoastrelid	oid	pg_class.oid	OID der zu dieser Tabelle gehörenden TOAST-Tabelle, 0, wenn keine. Die TOAST-Tabelle speichert große Spaltenwerte in einer Nebentabelle.
reltoastidxid	oid	pg_class.oid	Für eine TOAST-Tabelle, die OID ihres Index. 0 ,wenn keine TOAST-Tabelle.
relhasindex	bool		Wahr, wenn dies eine Tabelle ist und sie Indexe hat (oder kürzlich noch hatte). Dies wird von CREATE INDEX gesetzt, aber von DROP INDEX nicht sofort gelöscht. VACUUM löscht relhasindex, wenn es feststellt, dass die Tabelle keine Indexe hat.

Tabelle E.9: Spalten von pg_class

Name	Typ	Verweist auf	Beschreibung
relisshared	bool		Wahr, wenn dies eine Tabelle ist und sie von allen Datenbanken im Cluster gemeinsam verwendet wird. Nur bestimmte Systemkataloge (zum Beispiel pg_database) werden gemeinsam verwendet.
relkind	char		r = normale Tabelle, i = Index, S = Sequenz, v = Sicht, c = zusammengesetzter Typ, s = spezielle Relation, t = TOAST-Tabelle
relnatts	int2		Anzahl der Benutzerspalten in der Relation (Systemspalten nicht mitgezählt). So viele entsprechende Einträge muss es in pg_attribute geben. Siehe auch pg_attribute.attnum.
relchecks	int2		Anzahl der Check-Constraints für diese Tabelle; siehe Katalog pg_constraint
reltriggers	int2		Anzahl der Trigger für diese Tabelle; siehe Katalog pg_trigger
relukeys	int2		unbenutzt (*nicht* die Anzahl von Unique-Constraint-Schlüsseln)
relfkeys	int2		unbenutzt (*nicht* die Anzahl von Fremdschlüsseln für die Tabelle)
relrefs	int2		unbenutzt
relhasoids	bool		Wahr, wenn wir für jede Spalte der Relation eine OID erzeugen.
relhaspkey	bool		Wahr, wenn die Tabelle einen Primärschlüssel hat (oder einmal hatte).
relhasrules	bool		Die Tabelle hat Regeln; siehe Katalog. pg_rewrite
relhassubclass	bool		Mindestens eine Tabelle erbt von dieser.
relacl	aclitem[]		Zugriffsprivilegien; Einzelheiten finden Sie in der Beschreibung der Befehle GRANT und REVOKE.

Tabelle E.9: Spalten von pg_class (Forts.)

E.10 pg_constraint

Der Katalog pg_constraint speichert Check-Constraints, Primärschlüssel, Unique Constraints und Fremdschlüssel. (Tabellen-Constraints und Spalten-Constraints werden identisch gespeichert, da jeder Spalten-Constraint als Tabellen-Constraint dargestellt werden kann.) NOT-NULL-Constraints werden im Katalog pg_attribute gespeichert.

Check-Constraints für Domänen werden ebenfalls hier gespeichert.

Name	Typ	Verweist auf	Beschreibung
conname	name		Constraint-Name (nicht unbedingt einmalig!)
connamespace	oid	pg_namespace.oid	Das Schema, das diesen Constraint enthält

Tabelle E.10: Spalten von pg_constraint

Name	Typ	Verweist auf	Beschreibung
contype	char		c = Check-Constraint, f = Fremdschlüssel-Constraint, p = Primärschlüssel-Constraint, u = Unique Constraint
condeferrable	bool		Kann die Prüfung des Constraints bis an das Transaktionsende verschoben werden?
condeferred	bool		Wird der Constraint in der Voreinstellung erst am Transaktionsende geprüft?
conrelid	oid	pg_class.oid	Die Tabelle, für die dieser Constraint gilt; 0, wenn kein Tabellen-Constraint
contypid	oid	pg_type.oid	Die Domäne, für die dieser Constraint gilt; 0, wenn kein Domänen-Constraint
confrelid	oid	pg_class.oid	Wenn Fremdschlüssel, dann die Tabelle, auf die er verweist, ansonsten 0
confupdtype	char		Für Fremdschlüssel, der Code der Aktion für UPDATE
confdeltype	char		Für Fremdschlüssel, der Code der Aktion für DELETE
confmatchtype	char		Für Fremdschlüssel, der MATCH-Typ
conkey	int2[]	pg_attribute.attnum	Wenn Tabellen-Constraint, dann die Liste der Spalten, die der Constraint beschränkt
confkey	int2[]	pg_attribute.attnum	Wenn Fremdschlüssel, dann die Liste der Spalten, auf die er verweist
conbin	text		Wenn Check-Constraint, dann die interne Darstellung des Ausdrucks
consrc	text		Wenn Check-Constraint, dann eine für Anwender lesbare Darstellung des Ausdrucks

Tabelle E.10: Spalten von pg_constraint (Forts.)

Anmerkung

pg_class.relchecks muss mit der Anzahl der für eine bestimmte Relation in dieser Tabelle enthaltenen Check-Constraints übereinstimmen.

E.11 pg_conversion

Der Katalog pg_conversion speichert Informationen über Zeichensatzkonversionen (umwandlungen). Weitere Informationen finden Sie unter dem Befehl CREATE CONVERSION.

Name	Typ	Verweist auf	Beschreibung
conname	name		Konversionsname
connamespace	oid	pg_namespace.oid	Das Schema, das diese Konversion enthält
conowner	int4	pg_shadow.usesysid	Eigentümer der Konversion
conforencoding	int4		Nummer der Quellkodierung
contoencoding	int4		Nummer der Zielkodierung

Tabelle E.11: Spalten von pg_conversion

Name	Typ	Verweist auf	Beschreibung
conproc	regproc	pg_proc.oid	Funktion, die die Konversion durchführt
condefault	bool		Wahr, wenn dies die Standardkonversion ist

Tabelle E.11: Spalten von `pg_conversion` (Forts.)

E.12 pg_database

Der Katalog `pg_database` speichert Informationen über die verfügbaren Datenbanken. Datenbanken werden mit dem Befehl `CREATE DATABASE` erzeugt. Weitere Informationen über die Bedeutung der Parameter finden Sie in Kapitel 18.

Im Gegensatz zu den meisten Systemkatalogen wird `pg_database` von allen Datenbanken eines Clusters gemeinsam genutzt. Es gibt nur eine Instanz von `pg_database` pro Cluster, nicht eine pro Datenbank.

Name	Typ	Verweist auf	Beschreibung
datname	name		Datenbankname
datdba	int4	pg_shadow.usesysid	Eigentümer der Datenbank
encoding	int4		Zeichensatzkodierung für diese Datenbank
datistemplate	bool		Wenn wahr, kann diese Datenbank in der `TEMPLATE`-Klausel von `CREATE DATABASE` verwendet werden, um eine neue Datenbank als Kopie von dieser zu erzeugen.
datallowconn	bool		Wenn falsch, kann niemand mit dieser Datenbank verbinden. Damit wird die Datenbank `template0` vor Änderungen geschützt.
datlastsysoid	oid		Die letzte System-OID in der Datenbank; insbesondere für `pg_dump` nützlich.
datvacuumxid	xid		Alle Zeilenversionen, die von Transaktionsnummern vor dieser eingefügt oder gelöscht wurden, wurden in dieser Datenbank als garantiert abgeschlossen oder garantiert abgebrochen markiert. Das wird verwendet, um festzustellen, wann Commit-Log-Platz wiederverwendet werden kann.
datfrozenxid	xid		Alle Zeilenversionen, die von Transaktionsnummern vor dieser eingefügt wurden, wurden in dieser Datenbank als permanent (“eingefroren”) markiert. Das ist nützlich, um zu prüfen, ob die Datenbank bald gevacuumt werden muss, um Transaktionsnummernüberlauf zu verhindern.
datpath	text		Wenn die Datenbank an einem alternativen Speicherplatz abgelegt ist, wird hier der Ort aufgezeichnet. Es ist entweder eine Umgebungsvariable oder ein absoluter Pfad, je nachdem , wie es eingegeben wurde.
datconfig	text[]		Sitzungsvorgabewerte für Laufzeitkonfigurationsparameter
datacl	aclitem[]		Zugriffsprivilegien

Tabelle E.12: Spalten von `pg_database`

E.13 pg_depend

Der Katalog pg_depend speichert Abhängigkeiten zwischen Datenbankobjekten. Mit diesen Informationen können DROP-Befehle die Objekte finden, die bei DROP CASCADE mit gelöscht werden müssen, oder die bei DROP RESTRICT den Löschvorgang verhindern.

Name	Typ	Verweist auf	Beschreibung
classid	oid	pg_class.oid	Die OID des Systemkatalogs, in dem das abhängige Objekt gespeichert ist.
objid	oid	irgendeine OID-Spalte	Die OID des abhängigen Objekts.
objsubid	int4		Bei einer Tabellenspalte, die Spaltennummer (objid und classid verweisen auf die Tabelle selbst). Bei allen anderen Objekttypen ist diese Spalte null.
refclassid	oid	pg_class.oid	Die OID des Systemkatalogs, in dem das Objekt gespeichert ist, von dem das andere abhängt.
refobjid	oid	irgendeine OID-Spalte	Die OID des Objekts, von dem das andere abhängt.
refobjsubid	int4		Bei einer Tabellenspalte, die Spaltennummer (refobjid und refclassid verweisen auf die Tabelle selbst). Bei allen anderen Objekttypen ist diese Spalte null.
deptype	char		Ein Code, der die Bedeutung der Abhängigkeitsbeziehung näher definiert; siehe Text.

Tabelle E.13: Spalten von pg_depend

In jedem Fall gibt ein Eintrag in pg_depend an, dass ein Objekt nicht gelöscht werden darf, ohne dass alle abhängigen Objekte ebenfalls gelöscht werden. Es gibt jedoch einige unterschiedliche Arten von Abhängigkeiten, zwischen denen die Spalte deptype auswählt. Diese sind:

DEPENDENCY_NORMAL (n)

Ein normales Verhältnis zwischen getrennt erzeugten Objekten. Das abhängige Objekt kann gelöscht werden, ohne das andere Objekt zu beeinträchtigen. Das andere Objekt kann nur mit CASCADE gelöscht werden, wodurch das abhängige Objekt mit gelöscht wird. Beispiel: Eine Tabellenspalte hat eine normale Abhängigkeit von ihrem Datentyp.

DEPENDENCY_AUTO (a)

Das abhängige Objekt kann getrennt vom anderen Objekt gelöscht und sollte automatisch gelöscht werden (unabhängig von RESTRICT und CASCADE), wenn das andere Objekt gelöscht wird. Beispiel: Ein benannter Constraint hat eine automatische Abhängigkeit von seiner Tabelle, also sollte er gelöscht werden, wenn die Tabelle gelöscht wird.

DEPENDENCY_INTERNAL (i)

Das abhängige Objekt wurde als Teil der Erzeugung des anderen Objekts erzeugt und ist eigentlich nur Teil seiner internen Implementierung. Ein DROP des abhängigen Objekts wird auf jeden Fall abgelehnt. (Stattdessen sagen wir dem Benutzer, dass der das andere Objekt mit DROP löschen soll.) Ein DROP des anderen Objekts löscht das abhängige Objekt mit, egal ob CASCADE angegeben wurde oder nicht. Beispiel: Ein Trigger, der einen Fremdschlüssel-Constraint implementiert, hat eine interne Abhängigkeit vom pg_constraint-Eintrag des Constraints.

DEPENDENCY_PIN (p)

Es gibt kein abhängiges Objekt; dieser Eintragstyp gibt an, dass das System selbst das Objekt benötigt und es deswegen nie gelöscht werden darf. Einträge dieses Typs werden nur von initdb erzeugt. Die Spalten für das abhängige Objekt enthalten null.

In der Zukunft werden vielleicht weitere Abhängigkeitstypen benötigt werden.

E.14 pg_description

Der Katalog `pg_description` kann für jedes Datenbankobjekt eine optionale Beschreibung oder einen Kommentar speichern. Beschreibungen werden mit dem Befehl `COMMENT` erzeugt und verändert und können mit den `\d`-Befehlen in `psql` angesehen werden. Der anfängliche Inhalt von `pg_description` enthält die Beschreibungen von vielen eingebauten Systemobjekten.

Name	Typ	Verweist auf	Beschreibung
`objoid`	`oid`	irgendeine OID-Spalte	Die OID des Objekts, auf das sich diese Beschreibung bezieht
`classoid`	`oid`	`pg_class.oid`	Die OID des Systemkatalogs, in dem das Objekt eingetragen ist
`objsubid`	`int4`		Bei einer Beschreibung einer Tabellenspalte ist dies die Spaltennummer (`objoid` und `classoid` verweisen auf die Tabelle selbst). Bei allen anderen Objekttypen ist diese Spalte null.
`description`	`text`		Beliebiger Text, der die Beschreibung des Objekts darstellt.

Tabelle E.14: Spalten von `pg_description`

E.15 pg_group

Der Katalog `pg_group` definiert Gruppen und speichert, welche Benutzer zu welchen Gruppen gehören. Gruppen werden mit dem Befehl `CREATE GROUP` erzeugt. Weitere Informationen über die Verwaltung von Benutzerprivilegien erhalten Sie in Kapitel 17.

Da Benutzer- und Gruppenidentitäten im gesamten Cluster gelten, wird `pg_group` von allen Datenbanken eines Clusters gemeinsam genutzt. Es gibt nur eine Instanz von `pg_group` pro Cluster, nicht eine pro Datenbank.

Name	Typ	Verweist auf	Beschreibung
`groname`	`name`		Name der Gruppe
`grosysid`	`int4`		Eine willkürliche Zahl, die diese Gruppe identifiziert
`grolist`	`int4[]`	`pg_shadow.usesysid`	Ein Array mit den Nummern der Benutzer in dieser Gruppe

Tabelle E.15: Spalten von `pg_group`

E.16 pg_index

Der Katalog `pg_index` enthält einen Teil der Informationen über Indexe. Der Rest ist hauptsächlich in `pg_class`.

Name	Typ	Verweist auf	Beschreibung
indexrelid	oid	pg_class.oid	Die OID des pg_class-Eintrags für diesen Index
indrelid	oid	pg_class.oid	Die OID des pg_class-Eintrags der Tabelle, für die dieser Index ist
indproc	regproc	pg_proc.oid	Die OID der Funktion, wenn dies ein Funktionsindex ist, ansonsten null
indkey	int2vector	pg_attribute.attnum	Dies ist ein Array aus bis zu *INDEX_MAX_KEYS* Werten, die anzeigen, welche Tabellenspalten den Indexschlüssel bilden. Der Wert `1 3` würde zum Beispiel bedeuten, dass die erste und die dritte Spalte den Schlüssel bilden. Bei einem Funktionsindex sind diese Spalten die Argumente für die Funktion und das Ergebnis der Funktion ist der Indexschlüssel.
indclass	oidvector	pg_opclass.oid	Für jede Spalte im Indexschlüssel enthält dieser Wert einen Verweis auf die verwendete Operatorklasse.
indisclustered	bool		Wenn wahr, wurde die Tabelle zuletzt mit diesem Index geclustert.
indisunique	bool		Wenn wahr, ist dies ein Unique Index.
indisprimary	bool		Wenn wahr, stellt dieser Index den Primärschlüssel der Tabelle dar. (Wenn dies wahr ist, sollte immer auch indisunique wahr sein.)
indreference	oid		unbenutzt
indpred	text		Ausdrucksbaum (in der Darstellungsform von nodeToString()) des Prädikats eines partiellen Index. Wenn kein partieller Index, dann eine leere Zeichenkette.

Tabelle E.16: Spalten von pg_index

E.17 pg_inherits

Der Katalog pg_inherits speichert Informationen über die Tabellenvererbungshierarchien.

Name	Typ	Verweist auf	Beschreibung
inhrelid	oid	pg_class.oid	Die OID der Kindtabelle
inhparent	oid	pg_class.oid	Die OID der Elterntabelle
inhseqno	int4		Wenn es mehrere Elterntabellen für die Kindtabelle gibt (Mehrfachvererbung), dann gibt diese Zahl an, wie die geerbten Spalten angeordnet werden sollen. Die Zählung fängt bei 1 an.

Tabelle E.17: Spalten von pg_inherits

E.18 pg_language

Der Katalog pg_language registriert Sprachen oder Aufrufsschnittstellen, mit denen man Funktionen oder Prozeduren schreiben kann. Weitere Informationen über Sprachhandler und andere Parameter finden Sie unter CREATE LANGUAGE und in Kapitel 37.

Name	Typ	Verweist auf	Beschreibung
lanname	name		Name der Sprache (der bei der Erzeugung einer Funktion angegeben wird)
lanispl	bool		Dies ist falsch bei eingebauten Sprachen (wie SQL) und wahr bei benutzerdefinierten Sprachen. Gegenwärtig verwendet pg_dump diese Spalte, um festzustellen, welche Sprachen gesichert werden müssen, aber das könnte in der Zukunft durch einen anderen Mechanismus ersetzt werden.
lanpltrusted	bool		Gibt an, dass dies eine sichere/vertrauenswürdige Sprache ist.
lanplcallfoid	oid	pg_proc.oid	Bei nicht internen Sprachen ist dies ein Verweis auf den Sprachhandler, welcher eine besondere Funktion ist, die für die Ausführung aller in dieser Sprache geschriebenen Funktionen verantwortlich ist.
lanvalidator	oid	pg_proc.oid	Dies ist ein Verweis auf eine Prüffunktion, die dafür verantwortlich ist, die Syntax und die Gültigkeit von neuen Funktionen zu prüfen, wenn sie erzeugt werden.
lanacl	aclitem[]		Zugriffsprivilegien

Tabelle E.18: Spalten von pg_language

E.19 pg_largeobject

Der Katalog pg_largeobject enthält die Daten, aus denen Large Objects bestehen. Jedes Large Object wird durch eine OID identifiziert, die ihm zugewiesen wird, wenn es erzeugt wird. Jedes Large Object wird in Segmente oder "Seiten" aufgeteilt, die klein genug sind, dass sie in Zeilen in pg_largeobject gespeichert werden können. Die Größe der Daten pro Seite ist als LOBLKSIZE definiert (gegenwärtig BLCKSZ/4, typischerweise 2 kB).

Name	Typ	Verweist auf	Beschreibung
loid	oid		Die Identifikation des Large Objects, zu dem diese Seite gehört
pageno	int4		Nummer der Seite innerhalb des Large Objects (zählt von null an)
data	bytea		Die eigentlichen Daten, die in dem Large Object gespeichert sind. Dies sind nie mehr als *LOBLKSIZE* Bytes, aber möglicherweise weniger.

Tabelle E.19: Spalten von pg_largeobject

Jede Zeile in pg_largeobject enthält die Daten einer Seite eines Large Object welche bei pageno * LOBLKSIZE im Large Object anfängt. Die Implementierung erlaubt Speicherung mit Lücken: Seiten können ganz fehlen und können kürzer als LOBLKSIZE Bytes sein, selbst wenn Sie nicht die letzte Seite des Objekts sind. Fehlende Regionen ergeben beim Auslesen Nullen.

E.20 pg_listener

Der Katalog pg_listener unterstützt die Befehle LISTEN und NOTIFY. LISTEN erzeugt einen Eintrag in pg_listener mit dem Namen der Benachrichtigung, auf die es wartet. NOTIFY durchsucht pg_listener und markiert jeden passenden Eintrag, um anzuzeigen, dass die Benachrichtigung stattgefunden hat. NOTIFY sendet außerdem an jeden Prozess, der auf eine Benachrichtigung wartet, ein Signal (anhand der in der Tabelle gespeicherten PID), um ihn eventuell aufzuwecken.

Name	Typ	Verweist auf	Beschreibung
relname	name		Der Name der Benachrichtungsbedingung. (Der Wert muss nichts mit einer Relation in der Datenbank zu tun haben; der Spaltenname relname ist historisch bedingt.)
listenerpid	int4		Die PID des Serverprozesses, der diesen Eintrag erzeugt hat
notification	int4		Null, wenn kein Ereignis für diese Benachrichtigung anhängig ist. Wenn ein Ereignis anhängig ist, dann die PID des Serverprozesses, der die Benachrichtigung gesendet hat.

Tabelle E.20: Spalten von pg_listener

E.21 pg_namespace

Der Katalog pg_namespace speichert Namensräume. Ein Namensraum ist eine Struktur, die intern ein SQL-Schema darstellt: jeder Namensraum enthält Relationen, Typen usw., deren Namen keine Konflikte mit den Namen in anderen Namensräumen auslösen.

Name	Typ	Verweist auf	Beschreibung
nspname	name		Name des Namensraums
nspowner	int4	pg_shadow.usesysid	Eigentümer des Namensraums
nspacl	aclitem[]		Zugriffsprivilegien

Tabelle E.21: Spalten von pg_namespace

E.22 pg_opclass

Der Katalog pg_opclass definiert Operatorklassen für Indexzugriffsmethoden. Jede Operatorklasse definiert die Bedeutung von Indexspalten eines bestimmten Datentyps und einer bestimmten Indexmethode. Beachten Sie, dass es mehrere Operatorklassen für jede Kombination Datentyp/Indexmethode geben kann, wodurch mehrere Indizierungsverhalten unterstützt werden können.

Operatorklassen werden in Abschnitt 33.14 ausführlich beschrieben.

Name	Typ	Verweist auf	Beschreibung
opcamid	oid	pg_am.oid	Die Indexmethode, für die diese Operatorklasse gilt
opcname	name		Name dieser Operatorklasse

Tabelle E.22: Spalten von pg_opclass

Name	Typ	Verweist auf	Beschreibung
opcnamespace	oid	pg_namespace.oid	Schema dieser Operatorklasse
opcowner	int4	pg_shadow.usesysid	Eigentümer dieser Operatorklasse
opcintype	oid	pg_type.oid	Eingabedatentyp dieser Operatorklasse
opcdefault	bool		Wahr, wenn diese Operatorklasse die Standardklasse für opcintype ist
opckeytype	oid	pg_type.oid	Datentyp der Indexdaten, oder null, wenn gleicher Typ wie opcintype

Tabelle E.22: Spalten von pg_opclass (Forts.)

Die meisten Informationen, die eine Operatorklasse definieren, sind eigentlich nicht in ihrer pg_opclass-Zeile, sondern in den zugehörigen Zeilen in pg_amop und pg_amproc. Diese Zeilen sind Teil der Operatorklassendefinition. Das ist in etwa vergleichbar mit Tabellen, die durch eine Zeile in pg_class und zugehörige Zeilen in pg_attribute und anderen Tabelle definiert werden.

E.23 pg_operator

Der Katalog pg_operator speichert Informationen über Operatoren. Einzelheiten über diese Operatorenparameter finden Sie unter CREATE OPERATOR und Abschnitt 33.11.

Name	Typ	Verweist auf	Beschreibung
oprname	name		Name des Operators
oprnamespace	oid	pg_namespace.oid	Das Schema, das diesen Operator enthält
oprowner	int4	pg_shadow.usesysid	Eigentümer des Operators
oprkind	char		b = infix (“beide”), l = präfix (“links”), r = postfix (“rechts”)
oprcanhash	bool		Dieser Operator unterstützt Hash-Verbunde.
oprleft	oid	pg_type.oid	Datentyp des linken Operanden
oprright	oid	pg_type.oid	Datentyp des rechten Operanden
oprresult	oid	pg_type.oid	Datentyp des Ergebnisses
oprcom	oid	pg_operator.oid	Kommutator dieses Operators, falls vorhanden
oprnegate	oid	pg_operator.oid	Umkehrung dieses Operators, falls vorhanden
oprlsortop	oid	pg_operator.oid	Wenn dieser Operator Merge-Verbunde unterstützt, dann der Operator, der den Datentyp des linken Operanden sortiert (L<L)
oprrsortop	oid	pg_operator.oid	Wenn dieser Operator Merge-Verbunde unterstützt, dann der Operator, der den Datentyp des rechten Operanden sortiert (R<R)
oprltcmpop	oid	pg_operator.oid	Wenn dieser Operator Merge-Verbunde unterstützt, dann der Kleiner-als-Operator, der die linken und rechten Operandentypen vergleicht (L<R)
oprgtcmpop	oid	pg_operator.oid	Wenn dieser Operator Merge-Verbunde unterstützt, dann der Größer-als-Operator, der die linken und rechten Operandentypen vergleicht (L>R)
oprcode	regproc	pg_proc.oid	Die Funktion, die diesen Operator implementiert

Tabelle E.23: Spalten von pg_operator

Name	Typ	Verweist auf	Beschreibung
oprrest	regproc	pg_proc.oid	Auswahlselektivitätsschätzfunktion für diesen Operator
oprjoin	regproc	pg_proc.oid	Verbundselektivitätsschätzfunktion für diesen Operator

Tabelle E.23: Spalten von `pg_operator` (Forts.)

Unbenutzte Spalten enthalten null; zum Beispiel ist `oprleft` bei einem Präfixoperator null.

E.24 pg_proc

Der Katalog `pg_proc` speichert Informationen über Funktionen (oder Prozeduren). Die Beschreibung von `CREATE FUNCTION` sowie Abschnitt 33.3 enthalten weitere Informationen über die Bedeutung einiger Spalten.

Die Tabelle enthält neben normalen Funktionen auch Informationen über Aggregatfunktionen. Wenn `proisagg` wahr ist, dann sollte es eine passende Zeile in `pg_aggregate` geben.

Name	Typ	Verweist auf	Beschreibung
proname	name		Name der Funktion
pronamespace	oid	pg_namespace.oid	Das Schema, das diese Funktion enthält
proowner	int4	pg_shadow.usesysid	Eigentümer der Funktion
prolang	oid	pg_language.oid	Implementierungssprache oder -schnittstelle dieser Funktion
proisagg	bool		Die Funktion ist eine Aggregatfunktion.
prosecdef	bool		Die Funktion wird mit den Privilegien des Eigentümers ausgeführt (sog. “setuid”-Funktion).
proisstrict	bool		Die Funktion gibt den NULL-Wert zurück, wenn irgendein Argument den NULL-Wert hat. In diesem Fall wird die Funktion überhaupt nicht aufgerufen. Funktionen, die nicht “strikt” sind, müssen mit NULL-Werten umgehen können.
proretset	bool		Die Funktion gibt eine Ergebnismenge zurück (d.h. mehrere Werte des angegebenen Datentyps).
provolatile	char		`provolatile` gibt an, ob das Ergebnis der Funktion nur von den Eingabeargumenten abhängt oder von Faktoren außerhalb der Funktion beeinflusst wird. `i` steht für `IMMUTABLE`, `s` steht für `STABLE` und `v` steht für `VOLATILE`. (Eine Beschreibung finden Sie unter `CREATE FUNCTION`.)
pronargs	int2		Anzahl der Argumente
prorettype	oid	pg_type.oid	Datentyp des Rückgabewerts
proargtypes	oidvector	pg_type.oid	Ein Array mit den Datentypen der Funktionsargumente
prosrc	text		Diese Spalte zeigt dem Handler an, wie die Funktion aufgerufen werden soll. Sie könnte den eigentlichen Quelltext der Funktion, ein Link-Symbol, einen Dateinamen oder irgendetwas anderes enthalten, je nach Implementierungssprache.

Tabelle E.24: Spalten von `pg_proc`

Name	Typ	Verweist auf	Beschreibung
probin	bytea		Weitere Informationen darüber, wie die Funktion aufgerufen werden soll. Dies ist ebenfalls abhängig von der Sprache.
proacl	aclitem[]		Zugriffsprivilegien

Tabelle E.24: Spalten von pg_proc (Forts.)

Bei C-Funktionen (intern und dynamisch geladen) enthält prosrc den C-Namen (das Link-Symbol) der Funktion. Bei allen anderen Sprachen enthält prosrc den Quellcode der Funktion. probin wird nur bei dynamisch ladbaren C-Funktionen verwendet und enthält den Namen der dynamischen Bibliothek, in der die Funktion enthalten ist.

E.25 pg_rewrite

Der Katalog pg_rewrite speichert Umschreiberegeln für Tabellen und Sichten.

Name	Typ	Verweist auf	Beschreibung
rulename	name		Regelname
ev_class	oid	pg_class.oid	Die Tabelle, für die diese Regel gilt
ev_attr	int2		Die Spalte, für die diese Regel gilt (gegenwärtig immer null, gilt also für die ganze Tabelle)
ev_type	char		Ereignistyp, für den die Regel gilt: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
is_instead	bool		Wahr, wenn die Regel eine INSTEAD-Regel ist
ev_qual	text		Ausdrucksbaum (in der Darstellungsform von nodeToString()) für die Regelbedingung
ev_action	text		Anfragebaum (in der Darstellungsform von nodeToString()) der Regelaktion

Tabelle E.25: Spalten von pg_rewrite

pg_class.relhasrules muss wahr sein, wenn eine Tabelle in diesem Katalog irgendwelche Regeln hat.

E.26 pg_shadow

Der Katalog pg_shadow enthält Informationen über Datenbankbenutzer. Der Name stammt daher, dass diese Tabelle nicht öffentlich lesbar sein sollte, weil sie Passwörter enthält. pg_user ist eine von allen lesbare Sicht über pg_shadow, die das Passwortfeld ausblendet.

Kapitel 17 enthält detaillierte Informationen über die Verwaltung von Benutzern und Privilegien.

Da Benutzeridentitäten im gesamten Cluster gelten, wird pg_shadow von allen Datenbanken eines Clusters gemeinsam genutzt. Es gibt nur eine Instanz von pg_shadow pro Cluster, nicht eine pro Datenbank.

Name	Typ	Verweist auf	Beschreibung
usename	name		Benutzername
usesysid	int4		Benutzernummer (eine willkürliche Zahl, die diesen Benutzer identifiziert)
usecreatedb	bool		Der Benutzer darf Datenbanken erzeugen.
usesuper	bool		Der Benutzer ist ein Superuser.
usecatupd	bool		Der Benutzer darf die Daten in den Systemkatalogen verändern. (Selbst ein Superuser darf das nicht, wenn diese Spalte falsch ist.)
passwd	text		Passwort
valuntil	abstime		Ablaufzeit des Passworts
useconfig	text[]		Sitzungsvorgabewerte für Laufzeitkonfigurationsparameter

Tabelle E.26: Spalten von pg_shadow

E.27 pg_statistic

Der Katalog pg_statistic speichert statistische Informationen über den Inhalt der Datenbank. Einträge werden von ANALYZE erzeugt und danach vom Anfrageplaner verwendet. Es gibt einen Eintrag für jede Tabellenspalte, die analysiert wurde. Beachten Sie, dass alle statistischen Daten Annäherungen sind, selbst wenn sie gerade aktualisiert worden sind.

Da für verschiedene Arten von Daten unterschiedliche Arten von Statistiken von Nutzen sein können, wurde pg_statistic so entworfen, dass es wenig über die zu speichernden Statistiken voraussetzt. Nur sehr allgemein gültige Statistiken (zum Beispiel das Vorhandensein von NULL-Werten) haben eigene Spalten in pg_statistic. Alles andere wird in "Slots" gespeichert; das sind Gruppen von Spalten, deren Bedeutung durch einen Code in einer der Spalten des Slots identifiziert wird. Weitere Informationen dazu finden Sie in src/include/catalog/pg_statistic.h.

pg_statistic sollte nicht öffentlich lesbar sein, da selbst statistische Informationen über den Inhalt einer Tabelle vertrauliche Informationen enthalten könnten. (Zum Beispiel könnten der niedrigste und höchste Wert einer Spalte mit Gehaltsinformationen interessant sein.) pg_stats ist eine von allen lesbare Sicht über pg_statistic, die nur Informationen über die Tabellen preisgibt, die dem aktuellen Benutzer gehören. pg_stats wurde außerdem so entworfen, dass es die Informationen in einem einfacher zu lesenden Format darstellt als die Tabelle pg_statistic – allerdings mit dem Nachteil, dass ihre Definition jedes Mal erweitert werden muss, wenn neue Slottypen hinzugefügt werden.

Weitere Informationen zu den Planerstatistiken finden Sie in Abschnitt 13.2.

Name	Typ	Verweist auf	Beschreibung
starelid	oid	pg_class.oid	Die Tabelle, zu der die beschriebene Spalte gehört
staattnum	int2	pg_attribute.attnum	Die Nummer der beschriebenen Spalte
stanullfrac	float4		Der Anteil der Spaltenwerte, die den NULL-Wert haben
stawidth	int4		Die durchschnittliche Speichergröße der Spaltenwerte in Bytes, NULL-Werte nicht mitgerechnet

Tabelle E.27: Spalten von pg_statistic

Name	Typ	Verweist auf	Beschreibung
stadistinct	float4		Die Anzahl von verschiedenen Werten (außer NULL-Werten) in der Spalte. Ein Wert größer als null gibt die Anzahl der verschiedenen Werte an. Ein Wert kleiner als null ist der negative Wert eines Bruchteils der Zeilen in der Tabelle. (Zum Beispiel könnte bei einer Spalte, in der jeder Wert im Durchschnitt zweimal auftritt, stadistinct = -0,5 sein.) Null bedeutet, dass die Anzahl der verschiedenen Werte unbekannt ist.
stakindN	int2		Eine Codenummer, die angibt, welche Art von Statistik im *N*-ten "Slot" der pg_statistic-Zeile gespeichert ist.
staopN	oid	pg_operator.oid	Ein Operator, der für die Ermittlung der Statistiken im *N*-ten "Slot" verwendet wird. Ein Histogrammbalken würde zum Beispiel den Operator < angeben, der die Sortierreihenfolge der Daten definiert.
stanumbersN	float4[]		Numerische Statistiken der passenden Art für den *N*-ten "Slot", oder der NULL-Wert, wenn der Slot keine numerischen Werte speichert.
stavaluesN	text[]		Spaltendatenwerte der passenden Art für den *N*-ten "Slot" oder der NULL-Wert, wenn der Slot keine Datenwerte speichert. Um unabhängig vom Datentyp zu sein, werden alle Datenwerte in die externe Zeichenkettenform umgewandelt und als Werte vom Typ text gespeichert.

Tabelle E.27: Spalten von pg_statistic (Forts.)

E.28 pg_trigger

Der Katalog pg_trigger speichert Informationen über Trigger. Weitere Informationen erhalten Sie unter CREATE TRIGGER.

Name	Typ	Verweist auf	Beschreibung
tgrelid	oid	pg_class.oid	Die Tabelle, für die der Trigger gilt
tgname	name		Triggername (muss unter den Triggernamen für eine Tabelle einmalig sein)
tgfoid	oid	pg_proc.oid	Die aufzurufende Funktion
tgtype	int2		Bitmaske, die die Triggerbedingungen identifiziert
tgenabled	bool		Wahr, wenn der Trigger angeschaltet ist. (Wird gegenwärtig nicht überall geprüft, wo es sollte. Daher funktioniert das Ausschalten des Triggers hiermit nicht verlässlich.)
tgisconstraint	bool		Wahr, wenn dieser Trigger einen Fremdschlüssel-Constraint implementiert
tgconstrname	name		Name des Fremdschlüssel-Constraints
tgconstrrelid	oid	pg_class.oid	Die Tabelle, auf die der Fremdschlüssel-Constraint verweist
tgdeferrable	bool		Wahr, wenn die Prüfung des Constraints an das Transaktionsende verschoben werden kann

Tabelle E.28: Spalten von pg_trigger

Name	Typ	Verweist auf	Beschreibung
tginitdeferred	bool		Wahr, wenn der Constraint in der Voreinstellung am Transaktionsende geprüft wird
tgnargs	int2		Anzahl der Argumente für die Triggerfunktion
tgattr	int2vector		gegenwärtig unbenutzt
tgargs	bytea		Argumente für die Triggerfunktion, jedes mit einem Null-Byte abgeschlossen

Tabelle E.28: Spalten von pg_trigger (Forts.)

pg_class.reltriggers muss mit den Einträgen in dieser Tabelle übereinstimmen.

E.29 pg_type

Der Katalog pg_type speichert Informationen über Datentypen. Basistypen (skalare Typen) werden mit CREATE TYPE erzeugt. Ein zusammengesetzter Typ wird automatisch für jede Tabelle in der Datenbank erzeugt; er stellt die Zeilenstruktur der Tabelle dar. Außerdem kann man zusammengesetzte Typen mit CREATE TYPE AS und Domänen mit CREATE DOMAIN erzeugen.

Name	Typ	Verweist auf	Beschreibung
typname	name		Datentypname
typnamespace	oid	pg_namespace.oid	Das Schema, das diesen Typ enthält
typowner	int4	pg_shadow.usesysid	Eigentümer des Typs
typlen	int2		Bei Typen mit fester Länge ist typlen die Anzahl der Bytes in der internen Darstellung des Typs. Aber bei Typen mit variabler Länge ist typlen negativ. -1 steht für einen Typ mit variabler Länge (mit Längenwort am Anfang), -2 steht für eine C-Zeichenkette mit abschließendem Null-Byte.
typbyval	bool		typbyval bestimmt, ob die Werte dieses Typs an die internen Routinen mit Wertübergabe (wahr) oder Referenzübergabe (falsch) übergeben werden. Nur Typen, die äquivalent mit char, short oder int sind, können Wertübergabe verwenden. Wenn ein Typ also nicht 1, 2 oder 4 Bytes groß ist, sollte typbyval falsch sein. Typen mit variabler Länge verwenden immer Referenzübergabe. Beachten Sie, dass typbyval auch falsch sein kann, wenn der Typ eigentlich Wertübergabe erlauben würde; das ist zum Beispiel gegenwärtig bei float4 der Fall.
typtype	char		typtype ist b bei einem Basistyp, c bei einem zusammengesetzten Typ, d bei einer Domäne und p bei einem Pseudotyp. Siehe auch unter typrelid und typbasetype.
typisdefined	bool		Wahr, wenn der Typ definiert ist, falsch, wenn es ein Platzhalter für einen noch nicht definierten Typ ist. Wenn typisdefined falsch ist, dann kann man sich nur auf den Typnamen, das Schema und die OID verlassen.

Tabelle E.29: Spalten von pg_type

Name	Typ	Verweist auf	Beschreibung
typdelim	char		Das Zeichen, durch das zwei Werte dieses Typs in Arrayeingabewerten getrennt werden. Beachten Sie, dass das Trennzeichen für den Elementtyp des Arrays festgelegt ist, nicht für den Arraytyp.
typrelid	oid	pg_class.oid	Wenn dieser ein zusammengesetzter Typ ist (siehe typtype), zeigt diese Spalte auf den pg_class-Eintrag, der die entsprechende Tabelle definiert. (Bei freistehenden zusammengesetzten Typen stellt der pg_class-Eintrag keine richtige Tabelle dar, aber er wird trotzdem benötigt, damit die pg_attribute-Einträge des Typs darauf verweisen können.) Bei Basistypen ist diese Spalte null.
typelem	oid	pg_type.oid	Wenn typelem nicht 0 ist, dann kann der aktuelle Typ wie ein Array mit Elementen des Typs typelem verwendet werden. Ein "wahrer" Arraytyp hat variable Länge (typlen = -1), aber einige Typen mit fester Länge (typlen > 0) haben auch typelem ungleich 0, zum Beispiel name und oidvector. Wenn ein Typ mit fester Länge typelem ungleich 0 hat, dann muss seine interne Darstellung aus einer Folge von Werten des Typs typelem, ohne weitere Daten, bestehen. Arraytypen mit variabler Länge haben eine Kopfstruktur, die von den Arrayroutinen festgelegt wird.
typinput	regproc	pg_proc.oid	Eingabeumwandlungsfunktion
typoutput	regproc	pg_proc.oid	Ausgabeumwandlungsfunktion
typalign	char		typalign ist die Ausrichtung, die beim Speichern eines Werts dieses Typs benötigt wird. Der Wert gilt beim Speichern auf der Festplatte sowie für die meisten Darstellungen des Werts innerhalb von PostgreSQL. Wenn mehrere Werte hintereinander gespeichert werden, wie zum Beispiel bei einer kompletten Zeile auf der Festplatte, dann werden vor einem Wert Füllbytes eingefügt, damit der Wert auf der angegebenen Bytegrenze anfängt. Mögliche Werte sind: c char-Ausrichtung, d.h. keine Ausrichtung nötig s short-Ausrichtung (2 Bytes auf den meisten Maschinen) i int-Ausrichtung (4 Bytes auf den meisten Maschinen) d double-Ausrichtung (8 Bytes auf vielen Maschinen, aber auf keinen Fall auf allen) Bei Typen, die in Systemtabellen verwendet werden, ist es wichtig, dass die in pg_type gespeicherte Größe und Ausrichtung so angegeben sind, wie der Compiler eine Spalte dieses Typs in einem struct, das eine Tabellenzeile darstellt, anordnet.

Tabelle E.29: Spalten von pg_type (Forts.)

Name	Typ	Verweist auf	Beschreibung
typstorage	char		typstorage gibt bei Typen mit variabler Länge (die mit typlen = -1) die Standardspeicherungsstrategie an. Mögliche Werte sind p Der Wert wird immer unkomprimiert in der Haupttabelle gespeichert. e Der Wert kann in einer Nebentabelle gespeichert werden (wenn die Tabelle eine hat, siehe pg_class.reltoastrelid). m Der Wert wird in der Haupttabelle gespeichert, aber kann komprimiert werden. x Der Wert kann in der Haupt- oder Nebentabelle gespeichert und komprimiert werden. Spalten, bei denen m gesetzt ist, können ebenfalls in eine Nebentabelle verschoben werden, aber erst als letzten Ausweg (nachdem Spalten mit e oder x verschoben worden sind).
typnotnull	bool		typnotnull stellt einen NOT-NULL-Constraint für den Typ dar. Dies ist nur bei Domänen von Bedeutung.
typbasetype	oid	pg_type.oid	Wenn dies eine Domäne ist (siehe typtype), dann identifiziert typbasetype deren Basistyp. Ansonsten ist diese Spalte null.
typtypmod	int4		Bei Domänen wird in typtypmod der auf den Basistyp angewendete "typmod" gespeichert. Die Spalte ist -1, wenn der Basistyp keinen typmod-Wert benötigt oder dies keine Domäne ist.
typndims	int4		typndims enthält die Anzahl der Arraydimensionen, wenn dies eine Domäne ist, die ein Array ist. (Das heißt, dass typbasetype ein Arraytyp ist. typelem stimmt zwischen Domäne und Basistyp überein.) Bei allen Typen außer Arraydomänen ist diese Spalte null.
typdefaultbin	text		Wenn typdefaultbin nicht den NULL-Wert hat, dann ist es der Vorgabewertausdruck des Typs (in der Darstellungsform von nodeToString()). Diese Spalte wird nur für Domänen verwendet.
typdefault	text		typdefault ist der NULL-Wert, wenn der Typ keinen Vorgabewert hat. Wenn typdefaultbin nicht der NULL-Wert ist, dann enthält typdefault eine für Anwender lesbare Darstellung des Vorgabewertausdrucks in typdefaultbin. Wenn typdefaultbin der NULL-Wert ist, aber typdefault nicht, dann ist typdefault die externe Darstellung des Vorgabewerts des Typs, welche der Eingabefunktion des Typs übergeben werden kann, um eine Konstante zu erzeugen.

Tabelle E.29: Spalten von pg_type (Forts.)

Literaturverzeichnis

Einige Artikel und Berichte des ursprünglichen POSTGRES-Entwicklungsteams sind auf der Website der Informatikfakultät der Universität von Kalifornien in Berkeley verfügbar.

[Fong] Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*, University of California Computer Science Department.

[Olson 1993] Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

[Ong & Goh 1990] L. Ong und J. Goh, „A Unified Framework for Version Modeling Using Production Rules in a Database System“, *ERL Technical Memorandum M90/33*, University of California, April 1990.

[RFC 1413] M. St. Johns, *Identification Protocol*, Feb. 1993.

[Rowe & Stonebraker 1987] L. Rowe und M. Stonebraker, „*The POSTGRES data model*“, Proc. VLDB Conference, Sept. 1987.

[Seshardri & Swami 1995] P. Seshadri und A. Swami, „*Generalized Partial Indexes* “, Proc. Eleventh International Conference on Data Engineering, 6.-10. März 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.

[Stonebraker 1987] M. Stonebraker, „*The design of the POSTGRES storage system*“, Proc. VLDB Conference, Sept. 1987.

[Stonebraker 1989] M. Stonebraker, „*The case for partial indexes*“, *SIGMOD Record 18(4)*, Dez. 1989, 4-11.

[Stonebraker, Hanson, Hong 1987] M. Stonebraker, E. Hanson und C. H. Hong, „The design of the POSTGRES rules system“, Proc. IEEE Conference on Data Engineering, Feb. 1987.

[Stonebraker, Hearst, Potamianos 1989] M. Stonebraker, M. Hearst und S. Potamianos, „*A commentary on the POSTGRES rules system*“, *SIGMOD Record 18(3)*, Sept. 1989.

[Stonebraker, Jhingran, Goh 1990] M. Stonebraker, A. Jhingran, J. Goh und S. Potamianos, „*On Rules, Procedures, Caching and Views in Database Systems*“, Proc. ACM-SIGMOD Conference on Management of Data, Juni 1990.

[Stonebraker & Rowe 1986] M. Stonebraker und L. Rowe, „*The design of POSTGRES*“, Proc. ACM-SIGMOD Conference on Management of Data, Mai 1986.

[Stonebraker, Rowe, Hirohama 1990] M. Stonebraker, L. A. Rowe und M. Hirohama, „*The implementation of POSTGRES*“, *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, März 1990.

Stichwortverzeichnis

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z