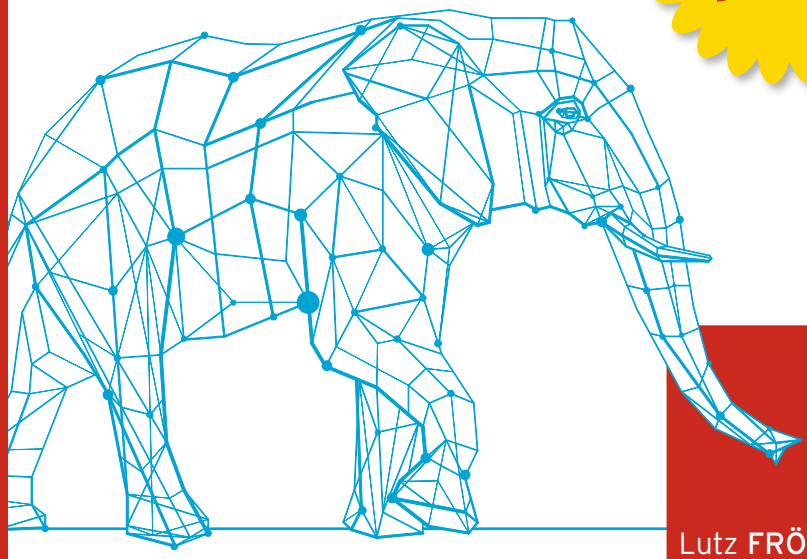
Update inside



Lutz FRÖHLICH

PostgreSQL

PRAXISBUCH FÜR ADMINISTRATOREN UND ENTWICKLER

Zu Version PostgreSQL 14

HANSER

Fröhlich

PostgreSQL

Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf **plus.hanser-fachbuch.de** einfach diesen Code ein:

plus-92Fgh-ci4R7

Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter

Update inside.

Mit unserem kostenlosen Update-Service zum Buch erhalten Sie aktuelle Infos zu den neuen Versionen von PostgreSQL.

Und so funktioniert es:

1. Registrieren Sie sich unter:
   www.hanser-fachbuch.de/postgresql-update
2. Geben Sie diesen Code ein:

2WHK-SeoTr-bF5x

Der Update-Service läuft bis Mai 2024.
Als registrierter Nutzer werden Sie in diesem Zeitraum persönlich per E-Mail informiert, sobald ein neues Buch-Update zum Download verfügbar ist.

Wenn Sie Fragen haben, wenden Sie sich gerne mit dem Betreff „PostgreSQL“ an:
update-inside@hanser.de

Lutz Fröhlich

PostgreSQL

Praxisbuch für Administratoren und Entwickler

HANSER

Der Autor:

Lutz Fröhlich
lutz@lutzfroehlich.de

Bibliografische Information der Deutschen Nationalbibliothek:

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

Lektorat: Sylvia Hasselbach
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © shutterstock.com/yosart
Gesamtherstellung: Eberl & Koesel, Altusried-Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN 978-3-446-46929-7
E-Book-ISBN 978-3-446-47315-7
E-Pub-ISBN: 978-3-446-47510-6

Inhalt

1 Einführung und Geschichte

Seit dem Erscheinen des Buchs „PostgreSQL 10 – Praxisbuch für Administratoren und Entwickler“ sind fast vier Jahre vergangen. In dieser Zeit hat sich bezüglich der Weiterentwicklung und des Einsatzes von PostgreSQL-Datenbanken viel getan. Ein guter Grund, die neue Version mit ihren neuen Features und Verbesserungen zu präsentieren.

So wie bei anderen Herstellern von Datenbanksystemen ist man dazu übergegangen, anstelle von großen Major-Release-Paketen neue Features kontinuierlicher zur Verfügung zu stellen. Das hat zur Folge, dass die Versionen schneller hochgezählt werden als in der Vergangenheit. Das Buch bezieht sich auf die Version 14. Es ist keine einfache Überarbeitung des Vorgängers, sondern wurde inhaltlich neu strukturiert und erweitert. Das Buch präsentiert sich weiterhin im bekannten Stil des Autors mit vielen Beispielen und Skripten.

Für die Themenwahl wurde auch die allgemeine Entwicklung im IT-Sektor berücksichtigt. Für die Installation gibt es nun auch eine Anleitung für macOS. Macbook-Besitzer können nun direkt einsteigen und wahlweise aus Paketen oder dem Quellcode installieren.

In Kapitel 4 finden Sie wiederum eine ausführliche Einführung in die Architektur, um das Verständnis für interne Prozesse und Abläufe zu fördern. Das Thema „Sicherheit und Überwachung“ wurde um interne Details zur Authentifizierung und Hacker-Abwehr erweitert.

PostgreSQL tummelt sich zunehmend im Bereich der großen Datenbanken. Die Themen „Performance- und SQL-Optimierung“ sowie der Umgang mit großen Datenbanken bilden deshalb einen Schwerpunkt. Dazu wurde das Kapitel 11 „Indexe effizient einsetzen“ neu aufgenommen. Es zeigt die Überlegenheit von PostgreSQL in diesem Umfeld gegenüber manch anderem Datenbanksystem. Für den Einsatz von großen Datenbanken zeigen die Möglichkeiten der Parallelisierung, wie PostgreSQL-Datenbanken und SQL-Anweisungen gegen rapides Wachstum skalieren können.

Ein weiterer Schwerpunkt ist der Bereich „Data Science und maschinelles Lernen“. Es werden die Möglichkeiten und Vorteile für den Einsatz von PostgreSQL als Datenbasis, aber auch der vorhandenen Features für Analyse und Auswertungen von Daten dargestellt. Hierbei spielt die Textverarbeitung, der ein eigenes Kapitel gewidmet wurde, eine wichtige Rolle. Die Datenbankprogrammierung mit Python, client- und serverseitig, wurde ebenfalls neu aufgenommen und ergänzt die Darstellung und Beispiele der anderen Programmiersprachen. Das Buch wird damit Administratoren und Entwicklern gleichermaßen gerecht.

Am Thema „Cloud“ führt längst kein Weg mehr vorbei. Die Cloud-Angebote haben sich vervielfacht. Aber auch die Einsatzmöglichkeiten und Features haben sich erweitert. Aktuelle Trends wie die Hybrid Cloud machen auch vor PostgreSQL nicht halt.

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

PostgreSQL hat in den vergangenen Jahren an Verbreitung und Popularität erheblich zugenommen. Dazu haben in erheblichem Maß die permanente Erweiterung mit neuen Features und die Anpassung an die Belange der Anwender beigetragen. PostgreSQL ist der lebende Beweis, dass Open-Source-Software nicht nur mit kommerziellen Produkten mithalten kann, sondern in einigen Bereichen sogar überlegen ist. Der kommerzielle Druck steht nicht im Vordergrund und lässt die Entwickler-Community frei arbeiten und Innovationen umsetzen.

Neben einem robusten Transaktionskern sowie einer hohen Zuverlässigkeit bietet PostgreSQL viele Features eines modernen Datenbankbetriebssystems und kann problemlos in eine vorhandene IT-Infrastruktur integriert werden. Durch den hohen Kompatibilitätsgrad zu Oracle ist der Migrationsaufwand überschaubar und ein Mischbetrieb gut umzusetzen.

PostgreSQL kann auf allen populären Plattformen wie Linux, macOS, Solaris oder Windows eingesetzt werden. Obwohl es sich um ein Open-Source-Produkt handelt, kann kommerzieller Support zu einem vernünftigen Preis hinzugekauft werden. Einem professionellen Einsatz steht damit nichts im Wege.

Freuen Sie sich auf einen PostgreSQL-Server 14 mit spannenden neuen Features!

1.1 Die Geschichte von PostgreSQL

PostgreSQL geht zurück auf das POSTGRES-Projekt, das an der University of California at Berkeley in den 1980er-Jahren angesiedelt war. Die erste vorzeigbare Version erschien im Jahre 1987 als Postgres-Version 1. Als Reaktion auf die ersten Kritiken wurde das noch heute in PostgreSQL vorhandene Rule-System entwickelt. Version 3 erschien im Jahre 1991 mit einer Weiterentwicklung der Abfrageeinheit. 1993 beendete die University of California das Projekt mit der Version 4.2, um die rasant wachsenden Supportanforderungen nicht mehr tragen zu müssen.

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

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

Heute präsentiert sich PostgreSQL als ausgereift und stabil und erfüllt alle Anforderungen an ein modernes relationales Datenbanksystem. Für viele überraschend: Die Performance ist vergleichbar mit so manchem kommerziellen Produkt.

1.2 Verwendete Versionen

Das Buch bezieht sich auf die während der Manuskripterstellung vorliegende aktuelle Version 14. Schauen Sie regelmäßig nach weiteren Veröffentlichungen, insbesondere für neuere Features und Versionen, auf der Webseite des Verlags und der Autoren-Webseite vorbei. Alles rund um die PostgreSQL-Community finden Sie auf der Webseite *http://www.postgresql.org*.

1.3 Konventionen

Begriffe in spitzen Klammern bezeichnen eine zu ersetzende Variable (so ist zum Beispiel der Ausdruck <VERSION> in der Regel durch die aktuelle Version 14.1 zu ersetzen). Die meisten Darstellungen beziehen sich gleichermaßen auf UNIX- und Windows-Betriebssysteme. Die Darstellung der Umgebungsvariablen erfolgt im Wesentlichen im UNIX-Format, das heißt z. B. $BIN statt %BIN% für Windows. Sie können das Format einfach nach Windows übertragen. Das Gleiche gilt für das Trennzeichen der Pfade: „/“ unter Unix sowie „\“ unter Windows.

1.4 Software und Skripte

Sie können die aktuelle Version von PostgreSQL aus dem Internet herunterladen und installieren. Es wird die Installation aus dem Quellcode empfohlen, um alle Beispiele nachvollziehen zu können. Ideal ist es, wenn Sie auf einem Linux-, macOS- oder Windows-Betriebssystem arbeiten.

Alle nummerierten Listings im Buch können als Datei von der Webseite des Verlags sowie von der Autoren-Webseite *https://www.lutzfroehlich.de* heruntergeladen werden:

Geben Sie auf

https://plus.hanser-fachbuch.de

diesen Code ein:

```
plus-92Fgh-ci4R7
```

1.5 Update inside

PostgreSQL stellt jährlich neue Major-Release-Versionen mit neuen Features und Erweiterungen zur Verfügung. Davon sind alle Bereiche vom Kernsystem bis hin zur Applikationsentwicklung betroffen.

Damit Sie möglichst lange mit diesem Buch arbeiten können, haben Sie die Möglichkeit, sich für den kostenlosen Update inside-Service zu registrieren: Geben Sie unter

www.hanser-fachbuch.de/postgresql-update

diesen Code ein:

`2WHK-SeoTr-bF5x`

Dann erhalten Sie bis Mai 2024 Aktualisierungen in Form zusätzlicher Kapitel als PDF. Darin stelle ich Ihnen wichtige Neuerungen vor und gehe auf Änderungen ein, die die Inhalte dieses Buches betreffen.

Putbus und Oasis del Sol, im April 2022

Lutz Fröhlich

lutz@lutzfroehlich.de

2 Installation mit Paketen und aus dem Quellcode

Die Installation kann sowohl von vorgefertigten Paketen als auch aus dem Quellcode erfolgen. Wenn Sie eine schnelle Installation bevorzugen, ist die einfachere Paketinstallation zu empfehlen. Allerdings müssen Sie dann mit dem Setup leben, das das Paket zur Verfügung stellt. Mit der Installation aus dem Quellcode sind Anpassungen und Erweiterungen möglich. Für den produktiven Einsatz ist zu beachten, dass Supportleistungen von Anbietern sich ausschließlich auf die entsprechenden Pakete beziehen. Diese Pakete sind bis zu einem gewissen Grad getestet. Beachten Sie, dass damit eine Abhängigkeit vom Release-Zyklus des Drittanbieters besteht. Ein wichtiger Vorteil der Paketinstallation ist, dass einheitliche Installationsstände ausgerollt werden können.

In diesem Kapitel werden beide Varianten vorgestellt. Um alle in diesem Buch vorgestellten Features und Optionen nachvollziehen zu können, wird eine Installation aus dem Quellcode empfohlen.

2.1 Paketinstallation

Alle erforderlichen Informationen für die Installation befinden sich auf der Seite *https://postgresql.org/download* unter dem Abschnitt *Binary Packages*. Wir führen zuerst Paketinstallationen für Linux und danach für Windows und macOS durch.

2.1.1 Paketinstallation unter Linux

Pakete für Linux stehen für die gebräuchlichsten Derivate zur Verfügung. Wir führen eine Installation unter Oracle Linux 7 durch. Das Vorgehen ist identisch mit einer Installation unter Red Hat Linux und CentOS. Danach erfolgt die Installation unter Ubuntu.

2.1.1.1 Paketinstallation unter Oracle Linux, Red Hat oder CentOS

Am einfachsten ist die Installation mit Hilfe des yum-Repository. Die folgenden Schritte beschreiben die Installation:

1. Installieren Sie das RPM-Paket im Repository.

```
# yum install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/
pgdg-redhat-repo-latest.noarch.rpm
```

2. Die Installation des Pakets erfolgt mit dem yum-Utility.

```
# yum install -y postgresql14-server
```

3. Zum Schluss wird die Datenbank initialisiert und für den automatischen Start konfiguriert.

```
# /usr/pgsql-13/bin/postgresql-14-setup initdb
# systemctl enable postgresql-14
# systemctl start postgresql-14
```

Falls keine Verbindung vom Datenbankserver zum Internet besteht, muss das Paket manuell heruntergeladen und übertragen werden. Gehen Sie dazu auf die Seite *https://yum.postgresql.org* und wählen Sie das passende Paket aus. Die Installation kann mit dem yum- oder dem rpm-Utility erfolgen.

2.1.1.2 Paketinstallation unter Ubuntu

Ubuntu ist ein weit verbreitetes Linux-Derivat und wird häufig als Betriebssystem für Desktops und Laptops verwendet. Mit den folgenden Schritten erfolgt die Installation der neuesten verfügbaren Version.

1. Erstellen Sie die Konfiguration für das Repository.

```
$ sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-
pgdg main" > /etc/apt/sources.list.d/pgdg.list'
```

2. Importieren Sie den Key für das Repository und führen Sie das Upgrade für die Paketliste durch.

```
$  wget --quiet -O – https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-
key add –
$ sudo apt-get update
```

3. Installieren Sie die letzte verfügbare Version von PostgreSQL.

```
$ sudo apt-get -y install postgresql
. . .
Success. You can now start the database server using:
    pg_ctlcluster 14 main start
```

Die Paketinstallation ist damit abgeschlossen.

2.1.2 Paketinstallation unter Windows

Für die Paketinstallation unter Windows erfolgt eine Weiterleitung von der Seite *https://postgresql.org/download* auf die Webseite von „EnterpriseDB“. Dort finden Sie das Paket für Windows in den letzten verfügbaren Versionen als Windows-Installer. EnterpriseDB ist ein Anbieter von vorgefertigten Paketen und Supportleistungen für diese Distributionen. Führen Sie die folgenden Schritte für die Paketinstallation aus:

1. Starten Sie den Windows-Installer.

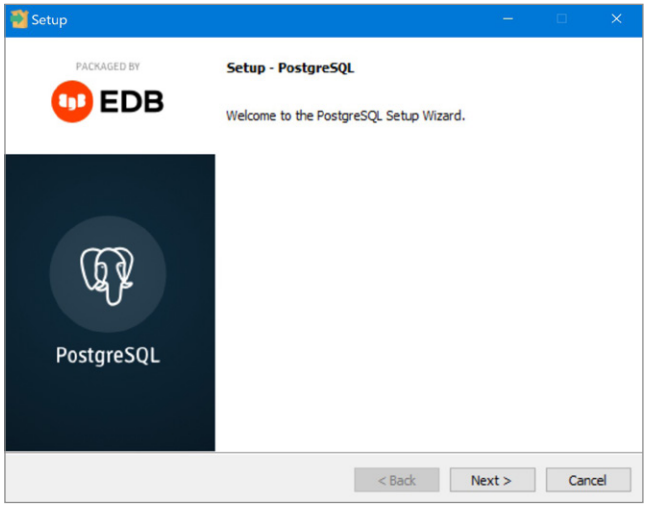


Bild 2.1 Die PostgreSQL-Paketinstallation für Windows starten

2. Wählen Sie das Installationsverzeichnis aus.

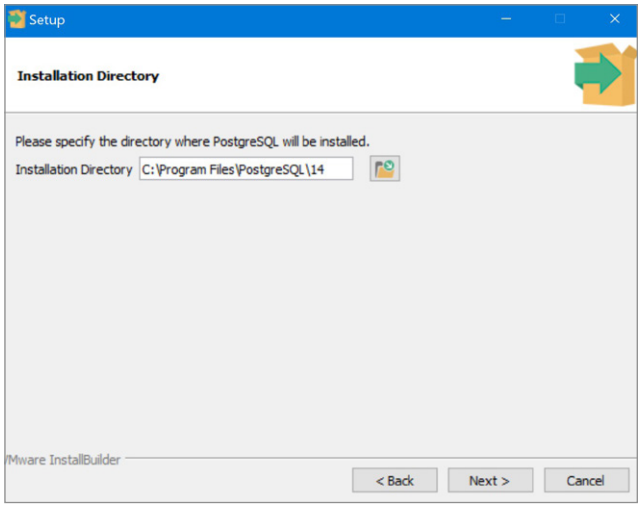


Bild 2.2 Das Installationsverzeichnis festlegen

3. Im nächsten Schritt können die Komponenten für die Installation ausgewählt werden.

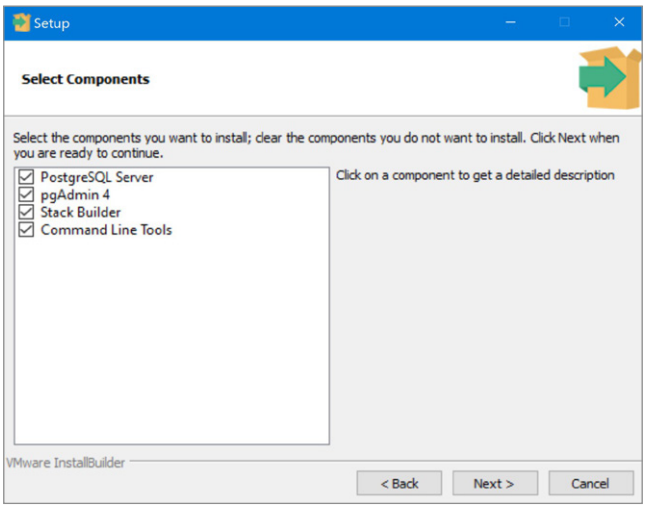


Bild 2.3 Auswahl der Komponenten für die Installation

4. Es erfolgt die Festlegung des Verzeichnisses für die Daten.
5. In den folgenden Schritten werden das Passwort für den Benutzer „postgres“, der Port sowie die Locale festgelegt.
6. Starten Sie die Installation.

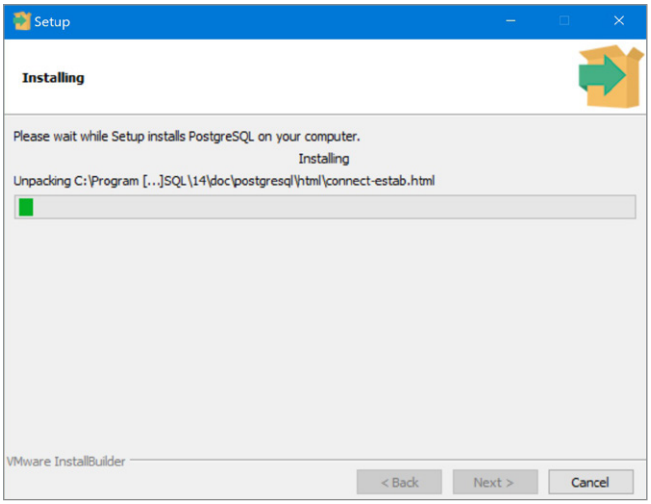


Bild 2.4 Die Paketinstallation ausführen

Der Installer legt einen Windows-Dienst zur Verwaltung des PostgreSQL-Servers an. Damit ist die Installation abgeschlossen.

2.1.3 Paketinstallation unter macOS

Vorgefertigte Pakete für macOS gibt es von verschiedenen Anbietern. An dieser Stelle verwenden wir ein Paket der Firma EnterpriseDB, analog zur Windows-Installation. Der Link zum Download lautet:

https://www.enterprisedb.com/downloads/postgres-postgresql-downloads

1. Öffnen Sie das Paket. Es erscheint das Startfenster des Setup Wizzard.

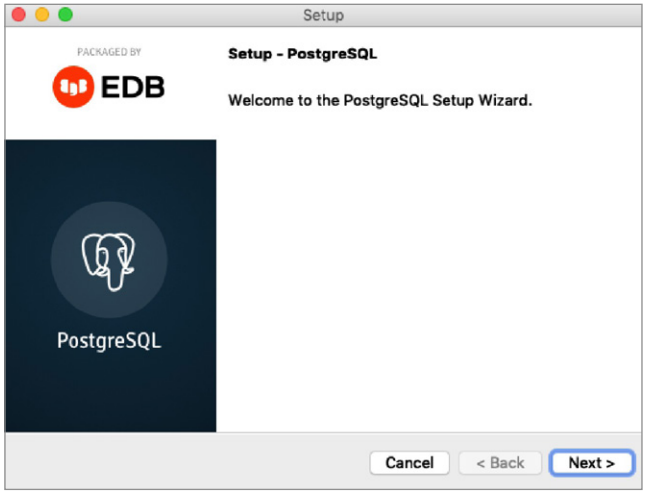


Bild 2.5 Der Setup Wizzard für macOS

2. Im nächsten Schritt kann das Installationsverzeichnis ausgewählt werden.

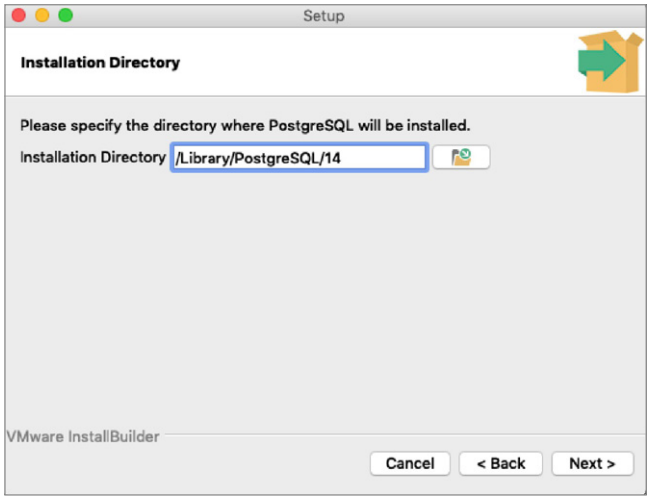


Bild 2.6 Das Installationsverzeichnis unter macOS auswählen

3. Es folgt die Auswahl der zu installierenden Komponenten. Neben dem PostgreSQL-Server können „pgAdmin“, der Stack Builder sowie die Kommandozeilen-Werkzeuge mit installiert werden.

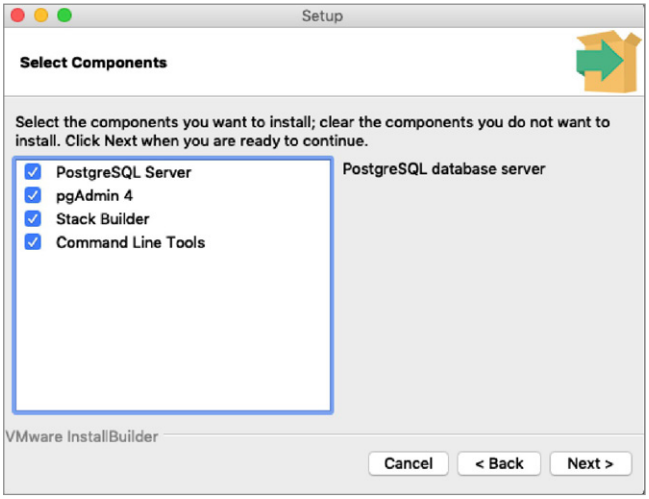


Bild 2.7 Auswahl der Komponenten für die Paket-Installation

4. Vergeben Sie Benutzername und Passwort für den Superuser (zum Beispiel „postgres“).
5. In den nächsten Schritten werden der Port sowie die Locale-Einstellungen abgefragt.
6. Verifizieren Sie die Eingaben in der Zusammenfassung und starten Sie die Installation.

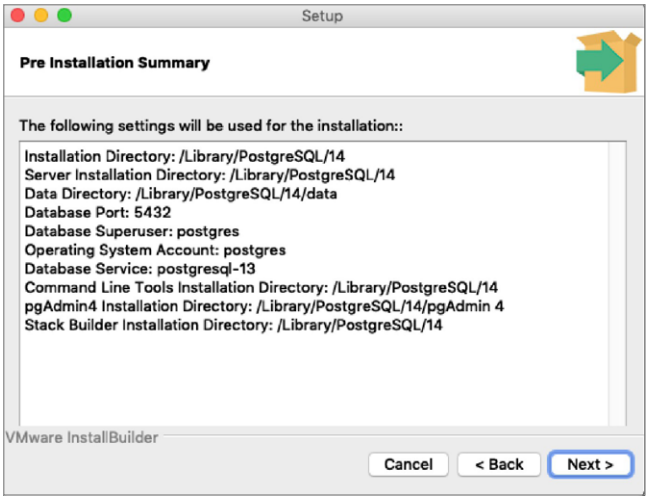


Bild 2.8 Zusammenfassung der Installation

Nach der Erfolgsmeldung ist die Installation abgeschlossen und der PostgreSQL-Server kann verwendet werden.

2.2 Installation aus dem Quellcode

Das Vorgehen bei der Installation aus dem Quellcode ist für alle Betriebssysteme ähnlich. Die Quellprogramme werden kompiliert, gelinkt und anschließend im Installationsverzeichnis bereitgestellt. In diesem Abschnitt werden die Schritte für Linux, Windows und macOS vorgestellt. Mit einer Installation aus dem Quellcode kann besser individualisiert und erweitert werden. Sie können sogar eigene Programmkomponenten einbinden.

2.2.1 Installation aus dem Quellcode unter Linux

Wir installieren an dieser Stelle zunächst den Standardumfang. Es wird im weiteren Verlauf des Buchs darauf hingewiesen, wenn zusätzliche Optionen eingebunden werden. Für PostgreSQL 14 bieten sich die Linux-Versionen 7 oder 8, zum Beispiel Red Hat, CentOS oder Oracle Linux an. Da die Programme komplett aus dem Quellcode übersetzt und mit den im Betriebssystem installierten Bibliotheken verbunden werden, sind wir an dieser Stelle sehr flexibel, was die Versionen und Derivate des Betriebssystems betrifft. Das folgende Vorgehen beschreibt die Installation:

1. Laden Sie den Quellcode von der Webseite *https://www.postgresql.org/ftp/source* herunter. Der Dateiname hat das Format *postgresql-<version>.tar.gz*.
2. Entpacken Sie den Quellcode und wechseln Sie in das Verzeichnis *postgresql-<version>*.

```
# tar -xvzf postgresql-14.1.tar.gz
# cd postgresql-14.1
```

3. Im ersten Schritt der Installation werden die Quellen für das Betriebssystem konfiguriert. Das Skript *configure* führt einige Tests aus, um die Werte einiger systemabhängiger Variablen zu bestimmen.

```
# ./configure
```

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

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

5. Im nächsten Schritt erfolgt die Installation der Binaries in die Zielverzeichnisse. Dieser Schritt muss ebenfalls unter dem Benutzer *root* ausgeführt werden, um die erforderlichen Berechtigungen zu erstellen. Standardmäßig erfolgt die Installation in das Verzeichnis */usr/local/pgsql*.

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

6. Damit ist die Installation abgeschlossen. In der Nachbereitung sind folgende Schritte erforderlich:

   Einen Benutzer *postgres* als Eigentümer der Software und des Servers erstellen. Der Benutzername ist frei wählbar:

```
# adduser postgres
```

   Ein Verzeichnis zum Speichern der Datenbankdateien erstellen. Das Verzeichnis kann beliebig gewählt werden:

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

   Den Datenbankserver initialisieren:

```
# su – postgres
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
. . .
initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.
Success. You can now start the database server using:
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

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

```
$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
waiting for server to start.... done
server started
[postgres@serv1 ~]$ ps -ef|grep postg
postgres 14543     1  0 10:40 ? 00:00:00 /usr/local/pgsql/bin/postgres -D /usr/local/
pgsql/data
postgres 14545 14543  0 10:40 ? 00:00:00 postgres: checkpointer
postgres 14546 14543  0 10:40 ? 00:00:00 postgres: background writer
postgres 14547 14543  0 10:40 ? 00:00:00 postgres: walwriter
postgres 14548 14543  0 10:40 ? 00:00:00 postgres: autovacuum launcher
postgres 14549 14543  0 10:40 ? 00:00:00 postgres: stats collector
postgres 14550 14543  0 10:40 ? 00:00:00 postgres: logical replication launcher
```

   Eine Testdatenbank mit dem Namen *hanser* anlegen:

```
$ /usr/local/pgsql/bin/createdb hanser
$ /usr/local/pgsql/bin/psql hanser
psql (14.1)
Type "help" for help.
hanser=# SELECT version();
                                version
------------------------------------------------------------------------
 PostgreSQL 14.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red
Hat 4.8.5-44.0.3), 64-bit
(1 row)
```

Damit ist die Installation aus dem Quellcode abgeschlossen und der PostgreSQL-Server ist einsatzbereit.

2.2.2 Installation aus dem Quellcode unter Windows

Die Installation aus dem Quellcode auf einem Windows-Betriebssystem umfasst dieselben Schritte wie unter Linux. Der Quellcode kann mit Hilfe des Visual C++-Compilers von Microsoft übersetzt und verbunden werden.

HINWEIS: Für das Erstellen der Software aus dem Quellcode wird mindestens das Microsoft Windows SDK benötigt. Alternativ kann Microsoft Visual Studio installiert werden, das ebenfalls die erforderlichen Programme für das Kompilieren und Linken der Quellen enthält.

In den folgenden Schritten erfolgt die Installation unter Windows 10 auf einer 64-Bit-Plattform.

1. Installieren Sie Microsoft Windows SDK oder Visual Studio.
2. Für die Installation ist ein Perl-Interpreter erforderlich. Installieren Sie im Bedarfsfall *ActiveState Perl*. Das Paket sowie die Installationsanleitung finden Sie auf der Webseite *https://www.activestate.com*.
3. Entpacken Sie den heruntergeladenen PostgreSQL-Quellcode. Auch wenn die Datei die Endung *.tar.gz* besitzt, kann sie mit den meisten Kompressionswerkzeugen direkt unter Windows ausgepackt werden. Im vorliegenden Beispiel werden die Quellen in das Verzeichnis *C:\PostgreSQL* gelegt. Beim Entpacken wird ein Unterverzeichnis im Format *postgresql-<version>* angelegt.
4. Öffnen Sie eine Windows-Eingabeaufforderung (DOS-Fenster) mit Administratorrechten und wechseln Sie in das Verzeichnis *C:\PostgreSQL\postgresql-<version>*.
5. Gegebenenfalls muss noch die Umgebung für das Windows SDK oder Visual Studio gesetzt werden. Compiler und Linker müssen sich im Pfad befinden und auf 64-Bit-Programme gestellt werden. Für das Visual Studio lautet der Aufruf:

```
C:\PostgreSQL\postgresql-14.1>"C:\Program Files (x86)\Microsoft Visual Studio\2019\
Community\VC\Auxiliary\Build\vcvars64.bat"
**********************************************************************
** Visual Studio 2019 Developer Command Prompt v16.8.4
** Copyright (c) 2020 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'
```

6. Wechseln Sie in das Unterverzeichnis *...\src\tools\msvc* und starten Sie Kompilierung und Verbinden des Quellcodes. Achten Sie auf die Erfolgsmeldung am Ende.

```
C:\PostgreSQL\postgresql-14.1>cd src\tools\msvc
C:\PostgreSQL\postgresql-14.1\src\tools\msvc>build
. . .
Der Buildvorgang wurde erfolgreich ausgeführt.
    0 Warnung(en)
    0 Fehler
Verstrichene Zeit 00:03:38.04
```

7. Im letzten Schritt erfolgt die Installation. Geben Sie das gewünschte Verzeichnis an, unter dem die Software installiert werden soll.

```
C:\PostgreSQL\postgresql-14.1\src\tools\msvc>install C:\PostgreSQL
Installing version 14 for release in C:\PostgreSQL
. . .
Installation complete.
```

8. Analog zu den Installationsschritten unter Linux lässt sich der Datenbankserver jetzt initialisieren. Das Verzeichnis für den Cluster kann frei gewählt werden. Legen Sie das Verzeichnis vorher an.

```
C:\PostgreSQL\bin>initdb -D C:\PostgreSQL\data
. . .
Success. You can now start the database server using:
    pg_ctl -D ^"C^:^\PostgreSQL^\data^" -l logfile start
```

9. Zum Schluss kann der Server gestartet und eine Datenbank mit dem Namen *hanser* angelegt werden.

```
C:\PostgreSQL\bin>pg_ctl -D C:\PostgreSQL\data -l logfile start
waiting for server to start.... done
server started
C:\PostgreSQL\bin>createdb hanser
```

10. Mit dem Kommandozeilen-Utility *psql* lässt sich die Verbindung zur Datenbank herstellen und SQL-Anweisungen können verarbeitet werden.

```
C:\PostgreSQL\bin>psql hanser
psql (14.1)
Type "help" for help.
hanser=# SELECT version();
                           version
-------------------------------------------------------------
 PostgreSQL 14.1, compiled by Visual C++ build 1928, 64-bit
(1 row)
```

Damit ist die Installation abgeschlossen und der PostgreSQL-Server kann verwendet werden.

HINWEIS: Die Syntax sowie die meisten Beispiele im Buch beziehen sich auf ein Linux-System. Wenn Sie vorzugsweise unter Windows arbeiten, dann können Sie dennoch alles nachvollziehen, wenn Sie die betriebssystemspezifischen Abweichungen beachten. Das Verhalten von PostgreSQL ist plattformübergreifend analog.

Optional kann ein Windows-Dienst eingerichtet werden, der das automatische Starten und Stoppen bei Start und Shutdown des Betriebssystems übernimmt.

```
C:\PostgreSQL\bin>pg_ctl register -D C:\PostgreSQL\data -N PostgreSQL14 -U Lutz -P
xxxxxxx
```

Der Dienst ist nun in der Dienste-Verwaltung von Windows sichtbar.

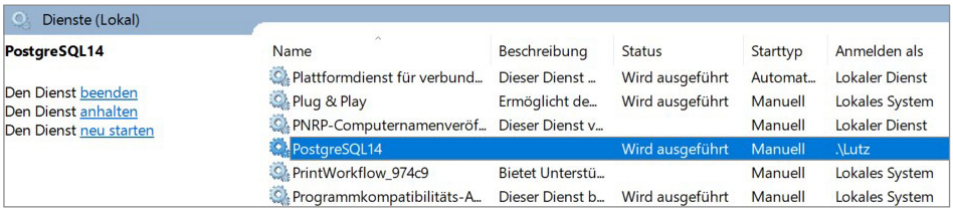


Bild 2.9 Einen Windows-Dienst für PostgreSQL anlegen

2.2.3 Installation aus dem Quellcode unter macOS

Die Installation verläuft ähnlich wie die Linux-Installation. Das folgende Beispiel beschreibt die Installation:

1. Laden Sie den Quellcode von der Webseite *https://www.postgresql.org/ftp/source* herunter. Der Dateiname hat das Format *postgresql-<version>.tar.gz*.
2. Entpacken Sie den Quellcode und wechseln Sie in das Verzeichnis *postgresql-<version>*.

```
$ tar -xvzf postgresql-14.1.tar.gz
$ cd postgresql-14.1
```

3. Für das Kompilieren und Linken des Quellcodes werden die Entwicklungstools für die Kommandozeile von Apple benötigt. Installieren Sie diese falls erforderlich.

```
$ xcode-select --install
```

4. Im ersten Schritt der Installation werden die Quellen für das Betriebssystem konfiguriert. Das Skript *configure* führt einige Tests aus, um die Werte einiger systemabhängiger Variablen zu bestimmen. Dafür wird der Pfad für „sysroot“ benötigt. Der Pfad variiert je nach installierter Version des Betriebssystems. Sie finden den Pfad mit dem Kommando „xrun“.

```
$ xrun –show-sdk-path
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
$ ./configure PG_SYSROOT=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
```

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

```
$ make PG_SYSROOT=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk all
. . .
All of PostgreSQL successfully made. Ready to install.
```

6. Im nächsten Schritt erfolgt die Installation der Binaries in die Zielverzeichnisse. Dieser Schritt muss ebenfalls unter dem Benutzer *root* ausgeführt werden, um die erforderlichen Berechtigungen zu erstellen. Standardmäßig erfolgt die Installation in das Verzeichnis */usr/local/pgsql*.

```
$ make install
. . .
PostgreSQL installation complete.
```

7. Damit ist die Installation abgeschlossen. In der Nachbereitung wird das Verzeichnis für den Server erzeugt.

```
$ mkdir /usr/local/pgsql/data
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
. . .
initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.
Success. You can now start the database server using:
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

8. Der Datenbankserver kann nun gestartet werden. Verifizieren Sie, ob alle Prozesse laufen.

```
$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
waiting for server to start.... done
server started
$ ps -ef|grep postg
501 14543     1  0 10:40 ? 00:00:00 /usr/local/pgsql/bin/postgres -D /usr/local/
pgsql/data
501 14545 14543  0 10:40 ? 00:00:00 postgres: checkpointer
501 14546 14543  0 10:40 ? 00:00:00 postgres: background writer
501 14547 14543  0 10:40 ? 00:00:00 postgres: walwriter
501 14548 14543  0 10:40 ? 00:00:00 postgres: autovacuum launcher
501 14549 14543  0 10:40 ? 00:00:00 postgres: stats collector
501 14550 14543  0 10:40 ? 00:00:00 postgres: logical replication launcher
```

9. Schließlich haben wir noch eine Testdatenbank mit dem Namen „hanser“ angelegt:

```
$ /usr/local/pgsql/bin/createdb hanser
$ /usr/local/pgsql/bin/psql hanser
psql (14.1)
Type "help" for help.
hanser=# SELECT version();
                                  version
-------------------------------------------------------------------------
 PostgreSQL 14.1 on x86_64-apple-darwin18.2.0, compiled by Apple LLVM version 10.0.1
(clang-1001.0.46.4), 64-bit
(1 row)
```

Damit ist die Installation aus dem Quellcode unter macOS abgeschlossen und der PostgreSQL-Server ist einsatzbereit.

2.3 Erste Schritte

Für ein bequemes Handling sollten mindestens die Umgebungsvariablen *PATH* und *PGDATA* gesetzt werden.

Listing 2.1 Den Ausführungspfad sowie die Variable PGDATA setzen

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

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

Listing 2.2 Den PostgreSQL-Server starten und stoppen

```
$ pg_ctl start -l logfile
waiting for server to start.... done
server started
$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

Sobald der Server gestartet ist, können Verbindungen zur Datenbank hergestellt werden. Das Kommandozeilen-Utility *psql* ist Bestandteil der Installation und kann direkt auf dem Server aufgerufen werden.

Listing 2.3 Eine Verbindung zur Datenbank mit psql herstellen

```
$ psql hanser
psql (14.1)
Type "help" for help.
hanser=# SELECT current_database();
 current_database
------------------
 hanser
(1 row)
hanser=# SELECT version();
                                version
-----------------------------------------------------------------------
PostgreSQL 14.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red Hat
4.8.5-44.0.3),
64-bit
(1 row)
hanser=# \q
```

Die lokale Verbindung zum PostgreSQL-Server ist nach der Installation direkt freigeschaltet. Der Versuch, eine Verbindung von einem entfernten Client herzustellen, scheitert zunächst mit der folgenden Fehlermeldung:

```
C:\PostgreSQL\bin>psql -h 192.168.56.100 -p 5432 -U postgres -W
Password:
psql: error: connection to server at "192.168.56.100", port 5432 failed: Connection
refused (0x0000274D/10061)
        Is the server running on that host and accepting TCP/IP connections?
```

Nach der Installation hört der Listener des Servers standardmäßig nur auf *localhost*. Für Verbindungen von außen muss er jedoch auch auf das Netzwerk-Interface hören. Dies wird

in der Konfigurationsdatei *postgresql.conf* konfiguriert. Sie befindet sich im Verzeichnis *$PGDATA*. Der Parameter lautet: *listen_addresses*. Ein anschließender Neustart des PostgreSQL-Servers ist erforderlich.

Listing 2.4 Den Listener für Verbindungen von außen aktivieren

```
listen_addresses = ,localhost,192.168.56.100'
```

HINWEIS: Wenn eine Firewall auf dem Linux-Server aktiviert ist, dann muss der Port 5432 freigeschaltet oder die Firewall deaktiviert werden, um den Zugriff auf den PostgreSQL-Server zu erlauben.

Ein erneuter Versuch, eine Verbindung vom Client zum PostgreSQL-Server aufzubauen, führt zu folgender Fehlermeldung:

```
C:\PostgreSQL\bin>psql -h 192.168.56.100 -p 5432 -U postgres -W
Password:
psql: error: connection to server at "192.168.56.100", port 5432 failed: FATAL:  no
pg_hba.conf entry for host "192.168.56.1", user "postgres", database "postgres", no
encryption
```

Der PostgreSQL-Listener verfügt selbst über eine Art Firewall. Diese wird über die Authentifizierungsdatei *pg_hba.conf* gesteuert, die sich im Verzeichnis *$PGDATA* befindet. Aus Sicherheitsgründen ist der Zugriff von einem entfernten Server oder Client nach der Installation abgeschaltet. Fügen Sie an dieser Stelle den Servernamen oder die IP-Adresse des Clients hinzu, von dem aus Sie zugreifen möchten. Nach der Änderung ist ein Neustart des PostgreSQL-Servers erforderlich. Weitere Informationen zu diesem Thema finden Sie in Kapitel 8 „Sicherheit und Überwachung“.

Listing 2.5 Einen Client in der Datei pg_hba.conf freischalten

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.56.1/32         trust
```

Nach der Änderung ist eine Verbindung vom Client zum PostgreSQL-Server möglich.

Umgebungsvariablen

Wie Sie sicherlich bemerkt haben, müssen dem psql-Utility für die Verbindung zu einem entfernten Server die Verbindungsinformationen mitgegeben werden. Alternativ können diese in folgenden Umgebungsvariablen des Betriebssystems hinterlegt werden:

PGHOST: Name oder IP-Adresse des Servers

PGPORT: TCP/IP-Port des PostgreSQL-Servers (Standard: 5432)

PGDATABASE: Name der Datenbank

PGUSER: Benutzername für die Verbindung

Für die Verwaltung von Server und Datenbanken gibt es eine Anwendung mit grafischer Oberfläche mit dem Name *pgAdmin*. Es ist ebenfalls ein Open-Source-Programm und kann von der Webseite *https://www.pgadmin.org/download* heruntergeladen werden. Erstellen Sie eine Serververbindung durch Eingabe der Verbindungsparameter. Da wir den Client bereits freigeschaltet haben, wird die Verbindung direkt funktionieren.

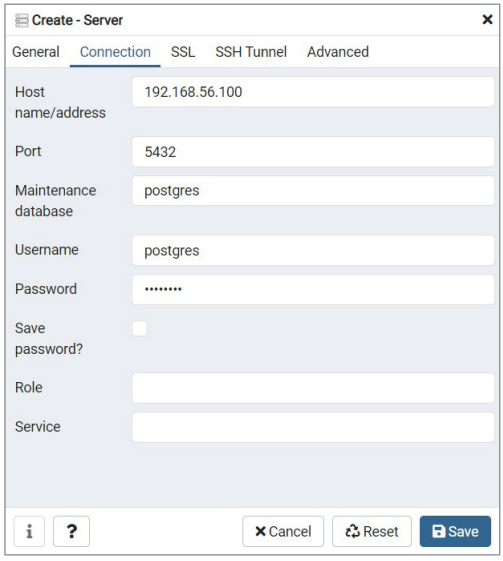


Bild 2.10
Einen Server im pgAdmin-Tool registrieren

Das Tool vereinfacht die Navigation durch Server und Datenbanken und verfügt über zusätzliche Funktionen im Vergleich zur Kommandozeile.

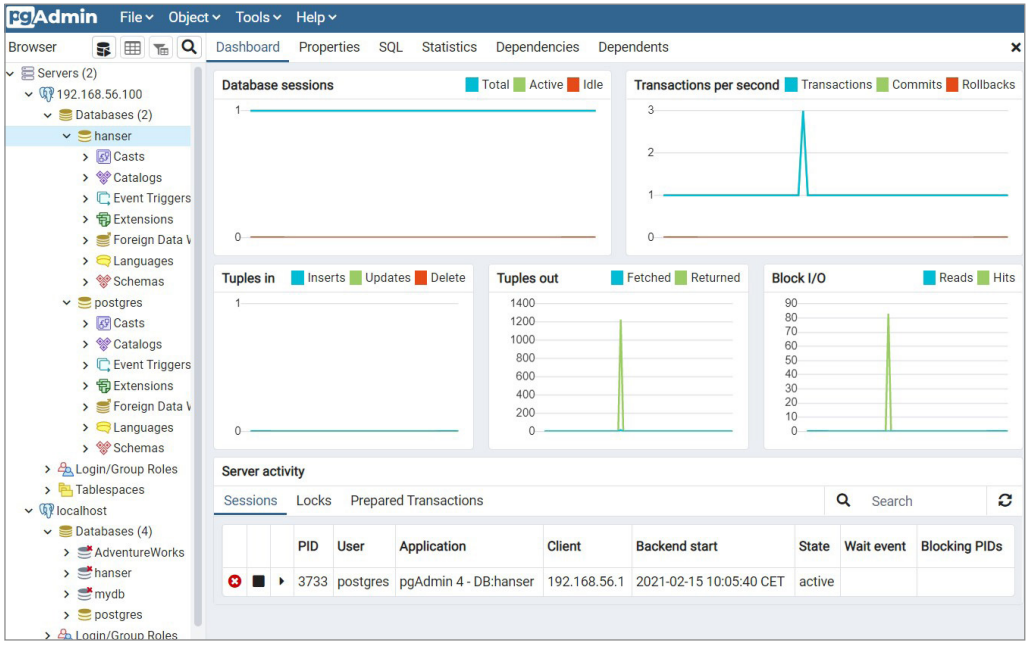


Bild 2.11 Das pgAdmin-Tool

Wie Sie vielleicht bemerkt haben, sendet der Server seine Nachrichten nach stdout, wenn er ohne die Logfile-Option gestartet wird. In der Standardkonfiguration ist das Schreiben von Nachrichten in Logdateien ausgeschaltet. Mit der folgenden Einstellung in der Konfigurationsdatei *postgresql.conf* wird ein minimales Schreiben von Fehlermeldungen und Servernachrichten eingeschaltet:

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

Mit dem Neustart erscheint die Meldung:

```
$ pg_ctl start
waiting for server to start....2021-02-15 17:23:17.925 CET [2494]
LOG:  redirecting log output to logging collector process
2021-02-15 17:23:17.925 CET [2494] HINT:  Future log output will appear in directory
"log".
 done
server started
```

Unter *$PGDATA* wurde das Verzeichnis *log* angelegt. Darin befindet sich die Logdatei des PostgreSQL-Servers:

```
$ cd $PGDATA/log
$ ls
postgresql-2021-02-15_172317.log
```

Wenn Sie den Server mit *pg_ctl stop* heruntergefahren haben, dann konnten Sie feststellen, dass der Server gestoppt wurde, obwohl noch Clients verbunden waren. Mit der Version 9.5 wurde der Standard der Shutdown-Optionen von *smart* auf *fast* geändert. Auch in der Version 14 ist der Standard *fast,* d. h., der Befehl ist identisch mit *pg_ctl stop -m fast*. Tabelle 2.1 beschreibt die Optionen.

Tabelle 2.1 Optionen zum Stoppen des PostgreSQL-Servers

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

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

Tabelle 2.2 Vergleich der Shutdown-Optionen zwischen PostgreSQL und Oracle

PostgreSQL	Oracle
smart	normal
nicht vorhanden	transactional
fast	immediate
immediate	abort

3 Upgrades und Versionen

Im Vergleich zu anderen Datenbanksystemen ist das Upgrade-Management von PostgreSQL einfach und gut zu beherrschen. Im Fall einer Paketinstallation besteht die Abhängigkeit von den durch den Support-Provider zur Verfügung gestellten Paketen und Release-Zyklen. Eine Installation aus dem Quellcode bietet die Möglichkeit, den Release-Zyklus selbst zu bestimmen.

Mit der Version 10 hat sich die Nummerierung der Releases geändert. Ein Major-Release-Wechsel bedeutet seitdem, dass sich die erste Nummer ändert, also zum Beispiel von 13 auf 14. Vorher wurde ein Major-Release-Wechsel durch die zweite Stelle gekennzeichnet. Demnach ist ein Upgrade von Version 9.5 auf 9.6 ebenfalls ein Major-Release-Wechsel.

Ein Minor-Release-Wechsel liegt vor, wenn sich maximal der zweite Teil der Versionsnummer ändert, zum Beispiel bei einem Upgrade von 13.2 auf 13.3. Grundsätzlich werden in diesem Fall nur Bug Fixes implementiert.

TIPP: Obwohl Upgrades immer ein Risiko mit sich bringen, sollten Sie regelmäßig auf das letzte Minor Release gehen. Zusätzlich zu allgemeinen Bug Fixes werden auch Sicherheitslücken geschlossen. Das Risiko, wenn Upgrades ausgelassen werden, wird als höher eingestuft.

Ein Upgrade Version 13 auf Version 14 ist ein Major-Release-Wechsel. Mit einem neuen Major Release ändert sich unter anderem das interne Format von Systemtabellen und Datenfiles. Diese Änderungen können komplexe Auswirkungen haben, sodass eine Rückwärtskompatibilität nicht gegeben ist.

Es gibt drei empfohlene Methoden für ein Major-Release-Upgrade:

- Entladen der Datenbanken mit *pg_dumpall* und Laden in die neue Version
- Upgrade mit *pg_upgrade*
- Upgrade mit logischer Replikation

Wir führen ein Upgrade von der Version 13.3 auf die Version 14.1, also ein Major-Release-Upgrade durch.

3.1 Upgrade mit pg_dumpall

Mit der Hilfe von *pg_dumpall* kann ein logisches Backup vom PostgreSQL-Cluster mit der niedrigeren Major-Release-Nummer erstellt und in das Cluster mit der höheren Release-Nummer geladen werden. Diese Vorgehensweise funktioniert rückwärts betrachtet bis zur Version 7.0.

Da es sich um ein Out-of-place-Upgrade handelt, müssen beide Versionen parallel installiert sein. Dies kann sowohl auf derselben als auch auf getrennter Hardware der Fall sein. Befinden sich beide Cluster auf demselben Server, dann können Sie zwischen beiden Umgebungen hin- und herschalten.

Es wird empfohlen, auch für den Export das Utility *pg_dumpall* der höheren Version zu verwenden. Es kann allerdings nicht einfach in die Binary-Installation der alten Version kopiert werden. Das heißt, wenn Sie auf einen anderen Server migrieren, benötigen Sie eine zweite PostgreSQL-Installation mit der höheren Version. Die folgenden Schritte beschreiben, wie eine zweite Version installiert werden kann:

1. Wir nehmen an, es existiert bereits eine Installation mit der Version 13.3 auf dem Server. Standardmäßig befindet sich die Distribution im Verzeichnis */usr/local/pgsql*.
2. Führen Sie die Installation aus dem Quellcode, so wie in Kapitel 2 beschrieben, durch. Im Standardverzeichnis befindet sich bereits die Installation der Version 13.3. Verwenden Sie für den Installationsbefehl die Option „prefix“, um die Distribution in ein anderes Verzeichnis zu lenken:

```
make prefix=/usr/local/pgsql.new install
```

3. Damit befinden sich zwei Binär-Installationen auf dem Server und Sie können hin- und herschalten.

Die folgenden Schritte zeigen eine logische Migration von der Version 13.3 auf die Migration 14.1:

1. Stellen Sie sicher, dass während des Exports keine Transaktionen auf der Datenbank stattfinden. Eine einfache, aber wirksame Methode ist die Anpassung der Datei *pg_hba.conf*, um neue Verbindungen zu vermeiden, und ein Neustart des Clusters.
2. Setzen Sie die Umgebung für die Daten auf die alte und die für die Binaries auf die neue Version.

```
$ export PGDATA=/usr/local/pgsql/data
$ export PATH=/usr/local/pgsql.new/bin:$PATH
$ which pg_dumpall
/usr/local/pgsql.new/bin/pg_dumpall
```

3. Führen Sie das Backup des Clusters durch.

```
$ pg_dumpall > backup_v133.sql
```

4. Das Backup ist eine Textdatei bestehend aus SQL-Befehlen zum Wiederherstellen der Komponenten des Clusters. Aus diesem Grund wird die Methode auch logische Migration genannt. Da keine Abhängigkeiten zur Binär-Installation bestehen, kann das Verfah-

ren auch zur Migration in ein anderes Betriebssystem, zum Beispiel Windows, eingesetzt werden.

5. Stoppen Sie das alte PostgreSQL-Cluster.

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

6. Kopieren Sie, falls erforderlich, die Backup-Datei auf den neuen Server.
7. Starten Sie das Cluster mit der neuen Version und dem neuen Datenverzeichnis.

```
[postgres@serv2 ~]$ export PGDATA=/usr/local/pgsql.new/data
[postgres@serv2 ~]$ export PATH=/usr/local/pgsql.new/bin:$PATH
$ pg_ctl start
waiting for server to start.... done
server started
```

8. Laden Sie die Daten aus dem Backup.

```
$ psql -d postgres -f backup_v133.sql -o upgrade.log > errors.log 2>&1
```

9. Prüfen Sie die Logdatei und die Error-Datei auf Fehler.

Damit ist das Upgrade abgeschlossen.

TIPP: Die mit *pg_dumpall* erzeugte Datei ist eine Textdatei, die mit psql verarbeitet werden kann. Das Entladen und das Laden können mit Hilfe eines Pipes in einem Schritt ausgeführt werden. Dazu müssen beide Cluster parallel auf verschiedenen Ports laufen:

```
$ pg_dumpall -p 5432 | psql -d postgres -p 5433
```

Mit dieser Methode kann die Downtime reduziert werden. Es wird kein zusätzlicher Speicherplatz für das Backup benötigt.

3.2 Upgrade mit pg_upgrade

Mit *pg_upgrade* ist es möglich, ein vorhandenes Cluster im Falle eines Major-Version-Upgrades zu konvertieren. Dies ist nicht erforderlich, wenn es sich um ein Minor-Version-Upgrade handelt, also zum Beispiel von 13.2 auf 13.3. In diesem Fall ist es ausreichend, die neuen Binaries zu installieren.

Die minimale Version, von der mit *pg_upgrade* auf 14.1 migriert werden kann, ist 8.4. Auch bei diesem Vorgehen muss ein Cluster der neuen Version installiert werden. Alter und neuer Cluster müssen sich auf demselben Server befinden.

Ein Major-Release-Wechsel schließt ein, dass sich das Layout der Systemtabellen, also des Cluster- und Datenbankkatalogs, ändert. Diese Änderungen werden durch das Utility vorgenommen. Es wird jedoch angenommen, dass sich das Speicherformat der Dateien für die Tablespaces und Tabellen nicht ändert. Solche Änderungen werden durch *pg_upgrade* nicht erfasst. Im Fall eines Upgrades von der Version 13 auf die Version 14 ist das sichergestellt. Die Aussage vom Development-Team für künftige Release-Wechsel zu diesem Thema ist jedoch vage:

Falls ein zukünftiger Major-Release-Wechsel eine Veränderung des Speicherformats einschließt und die alten Daten nicht mehr lesbar sind, dann kann pg_upgrade nicht verwendet werden. Die Community versucht, solche Situationen zu vermeiden.

TIPP: Vergewissern Sie sich im Zweifelsfall durch Lesen der Release Notes oder Fragen an die Community, dass mit dem Major-Release-Wechsel keine Änderung des Speicherformats verbunden ist. In so einem Fall kann *pg_upgrade* nicht als Migrationswerkzeug eingesetzt werden. Für die Version 14 besteht diese Gefahr nicht.

Das Utility *pg_upgrade* führt eine Prüfung auf Binär-Kompatibilität durch, allerdings nicht für externe Module. Es führt weiterhin die Änderungen im Datenbankkatalog durch und kopiert die Dateien aus dem alten $PGDATA-Verzeichnis in das neue.

Die folgenden Schritte beschreiben die Verwendung von *pg_upgrade*:

1. Verschieben Sie den alten Cluster, wenn sich die Installation im Standardverzeichnis befindet.

```
# mv /usr/local/pgsql /usr/local/pgsql.old
```

2. Kompilieren Sie den Quellcode, installieren Sie die Binaries für die Version 14 und installieren Sie das Cluster Es ist nicht erforderlich, den neuen Server zu starten. Passen Sie die Konfigurationsdateien *pg_hba.conf* und *postgresql.conf* an.

```
# make
# make install
# su – postgres
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data14
```

3. Installieren Sie an dieser Stelle die Shared Libraries von Extensions, die vom alten Cluster übernommen werden sollen. Eine Ausführung des Befehls „CREATE EXTENSION“ ist nicht erforderlich. Das erledigt das pg_upgrade-Utility.
4. Passen Sie gegebenenfalls im neuen Cluster die Datei *pg_hba.conf* an, so dass *pg_upgrade* in der Lage ist, sich zum Server zu verbinden.
5. Stoppen Sie den alten Server. Beide Server müssen vor Ausführung des Utility gestoppt sein.
6. Führen Sie das Upgrade durch. Es erfolgt ein Upgrade des Datenbankkatalogs und die Dateien werden in das neue $PGDATA-Verzeichnis kopiert. Unter Windows muss die Shell mit Administratorrechten ausgeführt werden.

```
$ pg_upgrade --old-bindir /usr/local/pgsql.old/bin --new-bindir /usr/local/pgsql/bin
--old-datadir /usr/local/pgsql/data --new-datadir /usr/local/pgsql/data14
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for user-defined encoding conversions              ok
Checking for user-defined postfix operators                 ok
Creating dump of global objects                             ok
Creating dump of database schemas                           ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
------------------
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster               ok
Copying user relation files                                 ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok

Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/local/pgsql/bin/vacuumdb --all --analyze-in-stages

Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
```

7. Starten Sie das neue Cluster und führen Sie den VACUUM-Befehl aus.

```
$ pg_ctl start
$ /usr/local/pgsql/bin/vacuumdb --all --analyze-in-stages
```

Beide Methoden führen also ein sogenanntes Out-of-place-Upgrade durch. Es werden eine neue Version von Binaries sowie ein neues Datenverzeichnis installiert, während die alten erhalten bleiben. Ein Nachteil liegt darin, dass der doppelte Speicherbedarf für die Migration benötigt wird. Ein Vorteil ist, dass ein Rollback sehr einfach durchgeführt werden kann, indem man die alte Version wieder aktiviert.

Das Kopieren der Dateien in das neue $PGDATA-Verzeichnis kostet Zeit, die insbesondere bei großen Datenbanken einzuplanen ist. Das Kopieren der Dateien mit *pg_upgrade* bietet aber immer noch einen Zeitvorteil gegenüber der Methode mit *pg_dumpall*. Bei Verwendung von *pg_upgrade* müssen sich altes und neues Cluster auf derselben Hardware befinden. Die Methode mit *pg_dumpall* bietet darüber hinaus die Möglichkeit, im Rahmen der Migration zusätzlich einen Wechsel des Betriebssystems vorzunehmen.

3.3 Upgrade mit logischer Replikation

Für Datenbanken, die nur eine geringe Downtime erlauben, empfiehlt es sich, die logische Replikation als Methode für das Upgrade einzusetzen. Mit der logischen Replikation können Cluster unterschiedlicher Versionen und Betriebssysteme synchronisiert werden. Auch wenn der Upgrade-Prozess mit der Migration in ein anderes Betriebssystem oder die Cloud verbunden werden soll, ist die logische Replikation eine sehr gute Option. Das Umschalten vom alten Cluster in das neue, und damit die Downtime für die Migration, nimmt nur wenige Sekunden in Anspruch. Einzelheiten zum Thema „Logische Replikation“ finden Sie in Kapitel 10.

Erstellen Sie ein PostgreSQL-Cluster mit der neuen Version und setzen Sie die zugehörigen Datenbanken als Standby-Datenbanken auf. Der Cluster kann sich auf demselben oder einem entfernten Server befinden. Ist die Synchronisation zwischen der alten und der neuen Version hergestellt, kann der Rollentausch vollzogen werden. Die neue Version wird zum Master und die alte zur Standby-Konfiguration. Die Standby-Datenbanken können noch für die Fallback-Strategie aktiviert bleiben, bis die „GO-Entscheidung“ für die Migration gefallen ist.

HINWEIS: Nach dem Rollentausch kommt es vor, dass Clients versuchen, sich zum alten Cluster zu verbinden. Passen Sie die Datei *pg_hba.conf* so an, dass Zugriffe von außen verhindert und nur der Replikation eine Verbindung gestattet wird.

3.4 Migration nach Native Partitioning

Mit der Version 10 wurde das Native Partitioning eingeführt, das mit vielen Vorzügen und signifikant besserer Performance überzeugt. Es ist abzusehen, dass die Partitioning-Technologie mit der Table-Inheritance-Methode nicht weiterentwickelt wird. Sollten sich in der Datenbank noch Tabellen befinden, die mit Inheritance-Partitioning definiert sind, ist zu empfehlen, diese zu migrieren. Da für die Migration nach Native Partitioning die Daten neu geladen werden müssen, sollten sie am besten in die Migration der anderen Tabellen eingebunden werden. Das spart ein doppeltes Laden der Daten.

Die folgenden Schritte beschreiben die Migration:

1. Entladen der Tabellen mit Inheritance Partitioning aus der Quelldatenbank
2. Erstellen der Tabellen in der Zieldatenbank mit Native-Partitioning-Syntax
3. Laden der Daten in die Zieldatenbank mit der Option „Data Only“

Unabhängig davon, welche Werkzeuge Sie für die Migration benutzen, sind die Schritte dieselben. Im folgenden Beispiel wird die Tabelle „order_entry“ verwendet (siehe Listing 3.1).

Listing 3.1 Tabellenstruktur mit Inheritance-Partitioning

```
psql (13.3)
(postgres@[local]:5432)[postgres]> \d+ order_entry
                      Table "public.order_entry"
   Column   |  Type   | Modifiers | Storage | Stats target | Description
------------+---------+-----------+---------+--------------+-------------
 order_id   | integer |           | plain   |              |
 customer   | integer |           | plain   |              |
 article    | integer |           | plain   |              |
 amount     | integer |           | plain   |              |
 entry_date | date    | not null  | plain   |              |
Child tables: order_entry_201901,
              order_entry_201902,
              order_entry_201903,
              order_entry_201904,
              order_entry_201905,
              order_entry_201906
```

Die Tabelle wird mit dem Utility *pg_dump* entladen:

```
pg_dump -d postgres -p 5432 -t order_entry > export_13.3.dmp
```

Jetzt wird die Tabelle in der Zieldatenbank nach der neuen Syntax für das Native Partitioning angelegt (siehe Listing 3.2). In diesem Fall kommt das Range Partitioning zum Einsatz.

Listing 3.2 Tabelle mit Native Partitioning anlegen

```
psql(14.1)
(postgres@[local]:5432)[postgres]> CREATE TABLE order_entry(
> order_id    INT,
> customer    INT,
> article     INT,
> amount      INT,
```

```
> entry_date  DATE NOT NULL)
> PARTITION BY RANGE(entry_date);
CREATE TABLE
postgres@[local]:5432)[postgres]> CREATE TABLE order_entry_201201
> PARTITION OF order_entry
> FOR VALUES FROM ('2019-01-01') TO ('2019-01-31');
CREATE TABLE
. . .
(postgres@[local]:5432)[postgres]> CREATE TABLE order_entry(
> order_id    INT,
> customer    INT,
> article     INT,
> amount      INT,
> entry_date  DATE NOT NULL)
> PARTITION BY RANGE(entry_date);
CREATE TABLE
(postgres@[local]:5432)[postgres]> CREATE TABLE order_entry_201201
> PARTITION OF order_entry
> FOR VALUES FROM ('2019-01-01') TO ('2019-01-31');
CREATE TABLE
. . .
postgres@[local]:5432)[postgres]> \d+ order_entry
                                  Table "public.order_entry"
   Column   |  Type   | Collation | Nullable | Default | Storage | Stats target |
------------+---------+-----------+----------+---------+---------+--------------+
 order_id   | integer |           |          |         | plain   |              |
 customer   | integer |           |          |         | plain   |              |
 article    | integer |           |          |         | plain   |              |
 amount     | integer |           |          |         | plain   |              |
 entry_date | date    |           | not null |         | plain   |              |
Partition key: RANGE (entry_date)
Partitions: order_entry_201201 FOR VALUES FROM ('2019-01-01') TO ('2019-01-31'),
            order_entry_201202 FOR VALUES FROM ('2019-02-01') TO ('2019-02-28'),
            order_entry_201203 FOR VALUES FROM ('2019-03-01') TO ('2019-03-31'),
            order_entry_201204 FOR VALUES FROM ('2019-04-01') TO ('2019-04-30'),
            order_entry_201205 FOR VALUES FROM ('2019-05-01') TO ('2019-05-31'),
            order_entry_201206 FOR VALUES FROM ('2019-06-01') TO ('2019-06-30')
```

Nachdem die leere Tabellenstruktur existiert, können die Daten geladen werden. Verwenden Sie die Option „Data only“ oder entfernen Sie den CREATE TABLE-Befehl, falls die Daten im Textformat entladen wurden:

```
psql (14.1)
(postgres@[local]:5432[postgres]> \i order_entry_13.3.dmp
```

Prüfen Sie, ob es zu Fehlern beim Laden gekommen ist. Es könnten Partitionen fehlen oder sie wurden möglicherweise falsch angelegt.

3.5 Regressionstests

Im Zusammenhang mit einem Major-Release-Wechsel sollten Regressionstests geplant werden. Der Hintergrund ist, kritische Fehler aufzudecken, die nach dem Upgrade auf eine neue Version zum Vorschein treten. Solche Fehler können sowohl funktionaler als auch technischer Natur sein. Deshalb führt man in der Regel einen funktionalen und einen technischen Regressionstest durch. Da diese Tests vor dem eigentlichen Upgrade-Termin stattfinden müssen, sollte genügend Zeit für eine eventuell erforderliche Problembeseitigung bereitgestellt werden.

Der funktionale Test setzt die Kenntnis der Applikation voraus. Es werden zumindest die Basisfunktionen getestet und die Ergebnisse analysiert. Mit dem technischen Test gilt es sicherzustellen, dass es nicht zu Fehlern im Applikationsablauf kommt, alle SQL-Anweisung sauber und ohne Performance-Einbußen durchlaufen sowie neue Features wie geplant funktionieren. Regressionstests sind besonders wichtig, wenn Sie einen Plattformwechsel oder Wechsel des Betriebssystems planen. Obwohl PostgreSQL verspricht, plattformunabhängig zu sein, kann es dennoch zu unterschiedlichem Verhalten kommen. Darüber hinaus gibt es Bugs, die nur auf einer bestimmten Plattform auftreten.

PostgreSQL unterstützt die Regressionstests mit dem Utility *pg_regress*. Obwohl es nicht Ihre speziellen Datenbanken untersucht, führt es Prüfungen auf einer separaten Instanz an. Dabei werden verschiedene SQL-Anweisungen ausgeführt und die Ergebnisse mit Erwartungen verglichen. Damit kann herausgefunden werden, ob das von Ihnen installierte Cluster möglicherweise Probleme in bestimmten Bereichen hat. Sie können die gefundenen Differenzen im Detail überprüfen. Möglicherweise sind bestimmte Fehler oder Abweichungen von der Erwartung für Ihre Datenbanken auch nicht relevant.

TIPP: Führen Sie bei einem Upgrade auf PostgreSQL 14 einen Regressionstest mit *pg_regress* durch. Bei einem Major-Release-Wechsel wird von den Programmierern der PostgreSQL-Software ein relativ großer Anteil des Quellcodes angefasst und geändert. Das hat natürlich zur Folge, dass die Anzahl von Problemen und Bugs in der neuen Version größer ist und erst mit dem Ausrollen der folgenden Minor-Release-Upgrades abnimmt.

Sie können die Tests gegen ein bestehendes Cluster laufen oder eine temporäre Instanz anlegen lassen. Die folgenden Schritte beschreiben, wie ein solcher applikationsunabhängiger Regressionstest mit einer temporären Instanz durchgeführt werden kann.

1. Installieren Sie die Software und initialisieren Sie das Cluster so wie in Kapitel 2 beschrieben.
2. Wechseln Sie in das Installationsverzeichnis als Benutzer *postgres*.
3. Führen Sie den Befehl *make check* aus.

```
$ make check
. . .
============== creating temporary instance            ==============
============== initializing database system           ==============
```

```
============== starting postmaster                    ==============
running on port 50848 with PID 24761
============== creating database "regression"         ==============
. . .
CREATE DATABASE
ALTER DATABASE
============== running regression test queries        ==============
. . .
============== shutting down postmaster               ==============
============== removing temporary instance            ==============
=======================
 All 209 tests passed.
=======================
make[1]: Leaving directory '/tmp/postgresql-10/src/test/regress'
```

4. Die temporäre Instanz wurde durch das Utility am Ende gelöscht. Erhalten bleibt das Verzeichnis *.../srvTest/regress*. Darin finden Sie die Ergebnisse des Tests. Mit dem *diff*-Befehl können die Unterschiede herausgestellt werden.

Listing 3.3 Ergebnis des Regressionstests aufzeigen

```
$ cd src/test/regress
$ diff expected results
```

HINWEIS: Mit der Option „check“ werden nur die Kernkomponenten des PostgreSQL-Clusters geprüft. In der Distribution des Quellcodes befinden sich weitere Komponenten, wie zum Beispiel prozedurale Sprachen. Um alle verfügbaren Optionen zu testen, muss die Anweisung „make“ mit der Option „check-world“ ausgeführt werden.

4 Die Architektur von PostgreSQL

Die Kenntnis der Architektur ist nicht nur für den Administrator sehr wichtig. Egal, ob Sie als Architekt, Systemberater oder Entwickler arbeiten, das Eindringen in die Architektur ist der Schlüssel für eine erfolgreiche Arbeit mit dem System. Nur wer den Aufbau, die Zusammenhänge sowie die internen Prozessabläufe kennt, ist in der Lage, Architekturen zu planen und zu implementieren, die täglichen Supportanforderungen zu meistern und Troubleshooting zu betreiben.

Die Architektur des PostgreSQL-Clusters ist komplex und es ist längere praktische Erfahrung erforderlich, um sie in ihrer Gesamtheit kennenzulernen und zu beherrschen. Je länger und intensiver Sie sich mit diesem Thema beschäftigen, desto umfangreicher wird Ihr Wissen und desto plausibler werden die Zusammenhänge.

4.1 Überblick

Eine Sammlung von Datenbanken, die von einem PostgreSQL-Server verwaltet werden, wird als PostgreSQL-Cluster bezeichnet. Ein Server läuft auf einem Computer und verwaltet ein Datenbank-Cluster. In der Sprache von PostgreSQL bedeutet das Wort „Cluster“ also nicht eine Gruppe von Datenbankservern.

Die Instanz eines Clusters besteht aus einer Reihe von Prozessen, die mehr oder weniger unabhängig voneinander arbeiten, sowie aus Memory-Strukturen. Auf der Disk befinden sich die Strukturen für Verwaltung und Speicherung von Datenbanken, Tablespaces sowie Tabellen und Indexe.

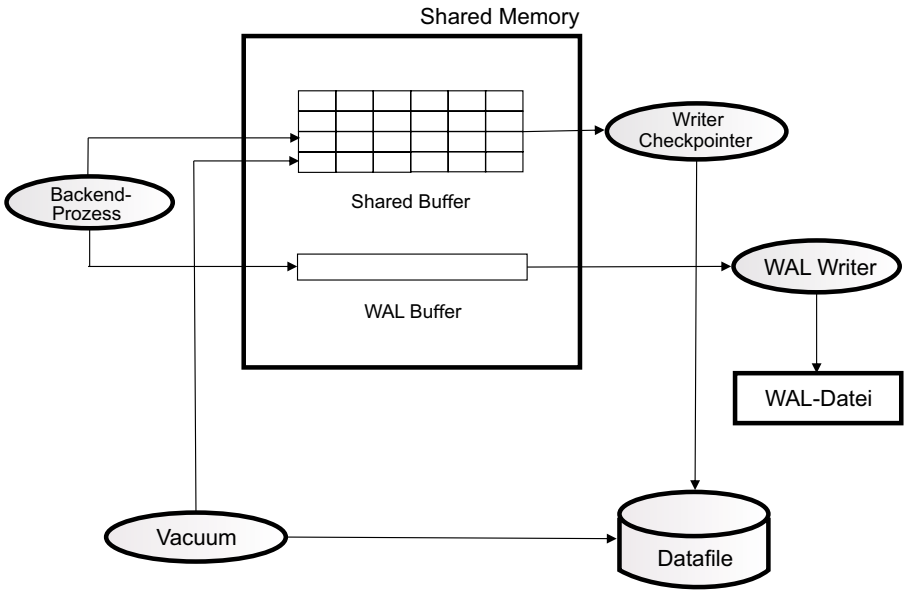


Bild 4.1 Die Architektur-Übersicht

Die wichtigsten Bereiche im Shared Memory sind der Shared Buffer und der WAL Buffer. Die Aufgabe des Shared Buffer ist es, die I/O-Operationen mit der Disk zu minimieren und möglichst viele Operationen im Memory durchzuführen. Der Writer-Prozess schreibt Datenblöcke aus dem Shared Buffer auf die Disk in die Tablespaces. Im Fall eines Checkpoints werden alle geänderten Blöcke auf die Disk geschrieben. Die Zustände des Shared Buffer und der Tablespaces werden damit synchronisiert.

Der WAL Buffer speichert temporär WAL-Sätze, bevor sie auf die Disk geschrieben werden. Er dient der Optimierung der Schreibvorgänge. Bei einem hohen Transaktionsaufkommen wird der WAL Buffer stark frequentiert.

4.2 Memory und Prozesse

Das Memory dient vorwiegend als Puffer, um einen Performancegewinn zu erzielen und die Zugriffe auf die Disk zu minimieren. Die Prozesse übernehmen die Verarbeitung von Daten und Aufgaben des Servers und der Datenbanken. Sie arbeiten unabhängig voneinander, kommunizieren jedoch miteinander und senden gegenseitig Anforderungen.

4.2.1 Hintergrundprozesse

Die Hintergrundprozesse eines PostgreSQL-Clusters arbeiten weitgehend unabhängig voneinander. Sie kommunizieren miteinander und triggern gegenseitig Abläufe oder Prozesse. Der Hauptprozess wird als Postmaster-Prozess bezeichnet.

Listing 4.1 Die Hintergrundprozesse des PostgreSQL-Servers

```
$ ps -ef|grep postgres
postgres 31755     1  0 16:56 pts/0  00:00:00 /usr/local/pgsql/bin/postgres -D /usr/
local/pgsql/data
postgres 31757 31755  0 16:56 ?      00:00:00 postgres: checkpointer process
postgres 31758 31755  0 16:56 ?      00:00:00 postgres: writer process
postgres 31759 31755  0 16:56 ?      00:00:00 postgres: wal writer process
postgres 31760 31755  0 16:56 ?      00:00:00 postgres: autovacuum launcher process
postgres 31761 31755  0 16:56 ?      00:00:00 postgres: stats collector process
postgres 31762 31755  0 16:56 ?      00:00:00 postgres: bgworker: logical replication
launcher
$ pstree -p 31755
postgres(31755)  ┬─postgres(31757)
                 ├─postgres(31758)
                 ├─postgres(31759)
                 ├─postgres(31760)
                 ├─postgres(31761)
                 └─postgres(31762)
```

Die Hintergrundprozesse haben bestimmte Aufgaben. Eine Zusammenfassung finden Sie in Tabelle 4.1.

Tabelle 4.1 Aufgaben der Hintergrundprozesse

Prozess	Aufgabe
checkpointer	Schreibt bei einem Checkpoint die „Dirty Buffer“ auf die Disk.
writer	Schreibt periodisch „Dirty Buffer“ auf die Disk.
wal writer	Schreibt Sätze aus dem WAL Buffer in die WAL-Datei.
autovacuum launcher	Startet Autovacuum-Arbeiterprozesse, wenn ein VACUUM im laufenden Betrieb notwendig ist.
archiver	Wird gestartet, wenn das Cluster im Archivelog-Modus arbeitet. Kopiert die WAL-Datei ins Archiv-Verzeichnis.
stats collector	Sammelt Statistiken, unter anderem über Sessions und Tabellenbenutzung.
bgworker	Verschiedene Arbeiterprozesse, die von Hintergrundprozessen gestartet werden.

HINWEIS: Unter Windows laufen die Hintergrundprozesse bedingt durch die Architektur des Betriebssystems als Threads unter dem Hauptprozess „postgres.exe“. Die Threads können mit geeigneten Werkzeugen, zum Beispiel mit dem Windows Process Explorer, sichtbar gemacht werden.

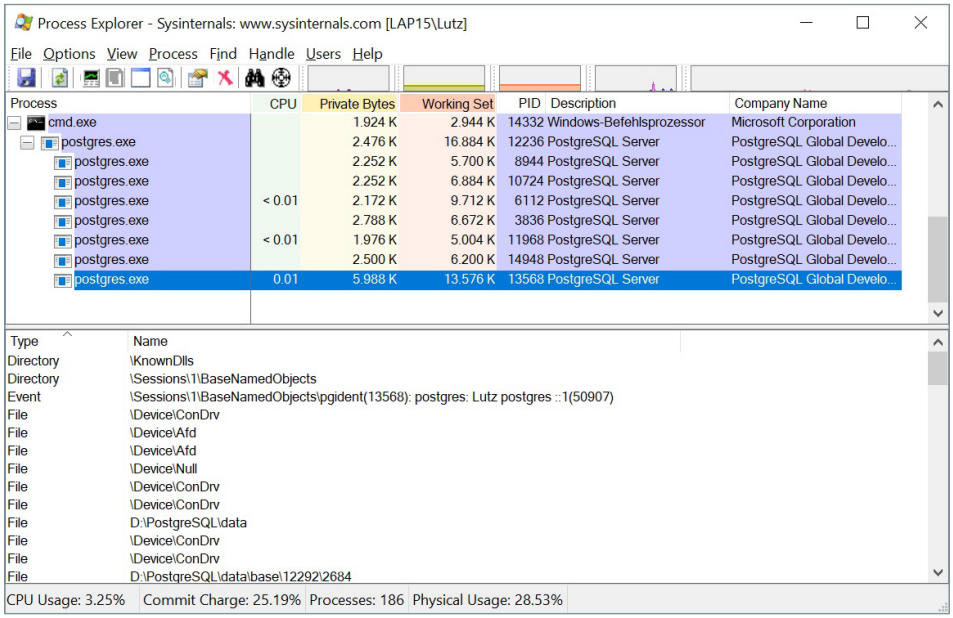


Bild 4.2 Hintergrundprozesse im Windows Process Explorer

Für jeden Client wird ein Backend-Prozess gestartet. Er führt die SQL-Anweisungen des Clients aus und gibt die Ergebnisse zurück. Der Client sendet eine Verbindungsanfrage an den Server, zum Beispiel über TCP/IP und Post 5432 an den Postmaster-Prozess des Servers. Der Postmaster nimmt die Verbindungsanfrage an und überprüft, ob der Client autorisiert ist, sich zum Cluster zu verbinden. Nach einer erfolgreichen Authentifizierung startet er mittels fork-Kommando den Backend-Prozess. Der Backend-Prozess stellt die Verbindung mit dem Client her.

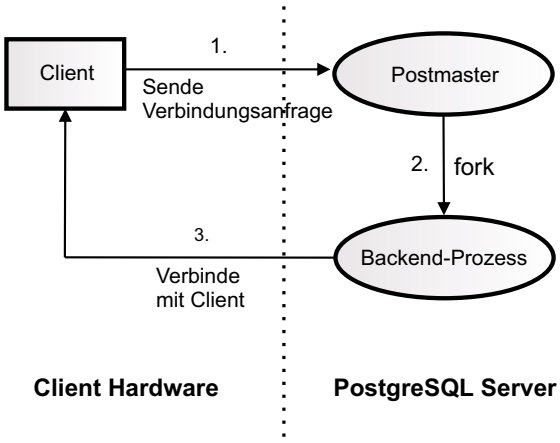


Bild 4.3 Verbindungsprozess des Clients

Die maximale Anzahl von Backend-Prozessen wird durch den Parameter *max_connections* begrenzt. Der Standardwert ist 100. Für die Ausführung von SQL-Anweisungen werden Memory-Strukturen benötigt. Sie werden *Local Memory* genannt und können mit folgenden Parametern eingestellt werden:

- *work_mem:* Wird für Sortiervorgänge, Hash Joins und Bitmap-Operationen benutzt. Der Standardwert ist 4 MB.
- *maintenance_work_mem:* Speicher für Vacuum und Indexerstellung. Standardgröße ist 64 MB.
- *temp_buffers:* Memory für temporäre Tabellen. Der Standardwert ist 8 MB.

4.2.2 Shared Memory

Im Normalfall passen nicht alle Daten und Indexe in den Shared Buffer. Es gibt solche Produkte, die als In-Memory-Datenbanken bezeichnet werden. Die Shared-Buffer-Architektur von PostgreSQL ist so designt, dass sich ein Teil der Datenbank im Memory befindet und bei Bedarf von der Disk gelesen werden kann. Die Verwaltung der Inhalte ist darauf ausgerichtet, die Zugriffe auf die Disk zu minimieren und möglichst vielen Clients einen optimalen Datenzugriff zur Verfügung zu stellen.

Während der Shared Buffer sowohl für lesende als auch für schreibende Operationen Vorteile bringt, ist der WAL Buffer eine Einbahnstraße. WAL steht für „Write Ahead Log“ und ist das Transaktionslog des Clusters. Er puffert das Schreiben von WAL-Sätzen in die WAL-Dateien. WAL-Dateien sind die Grundlage für eine erfolgreiche Wiederherstellung des Clusters.

4.2.2.1 Shared Buffer

Die Hauptaufgabe des Shared Buffer ist, I/O-bezogene Datenbankaktionen zu optimieren und möglichst wenig I/O-Operationen zu erzeugen. I/O-Operationen sind sehr zeitintensiv. Dies gilt insbesondere für Single-Block-Operationen. Muss ein bestimmter Datensatz gelesen werden, dann muss der Schreib-/Lesekopf der Disk positioniert und die Operation durchgeführt werden. Auch wenn moderne I/O-Subsysteme schneller arbeiten, liegen die Antwortzeiten auch da im Bereich von 5 bis 20 Millisekunden oder darüber.

Antwortzeiten von Disk-Operationen sind darüber hinaus im Normalbetrieb selten optimal. Je mehr Operationen gleichzeitig angefordert werden, desto mehr wird die Disk beschäftigt und die Antwortzeiten steigen. Eine Leseoperation aus dem Shared Buffer benötigt dagegen deutlich weniger Zeit. Die Antwortzeit verlängert sich darüber hinaus kaum, wenn viele Anfragen parallel gestellt werden.

Für Schreiboperationen gelten ähnliche Voraussetzungen. Die Änderung eines Datenblocks im Memory geht wesentlich schneller vonstatten und reduziert den Auslastungsgrad der Disks.

Hinter der Verwaltung des Shared Buffer liegt ein intelligenter Prozess, der sogenannte *Clock-Sweep-Algorithmus*. Schließlich sollen Datenblöcke, die häufig angefragt werden, möglichst lange im Memory bleiben.

Der Shared Buffer besteht aus den folgenden vier Hauptkomponenten:

1. Hash-Tabelle
2. Hash-Elemente
3. Buffer Descriptor
4. Buffer Pool

Eine Hash-Tabelle verwaltet Buffer im Memory sowohl für lesende als auch schreibende Operationen sehr effektiv. Die Zugriffszeiten erhöhen sich allerdings, wenn sehr viele Hash-Elemente einen identischen Hash-Wert generieren und auf diese gleichzeitig zugegriffen wird. Dieses Problem wird Hash-Kollision genannt. PostgreSQL versucht, dieses Problem zu umgehen, indem die Hash-Tabelle in logische Segmente unterteilt wird. Ein Segment kann aus maximal 256 Buckets bestehen. Für die Verwaltung der Segmente wird ein sogenanntes Directory erstellt. Darin wird die Startposition eines jeden Segments hinterlegt.

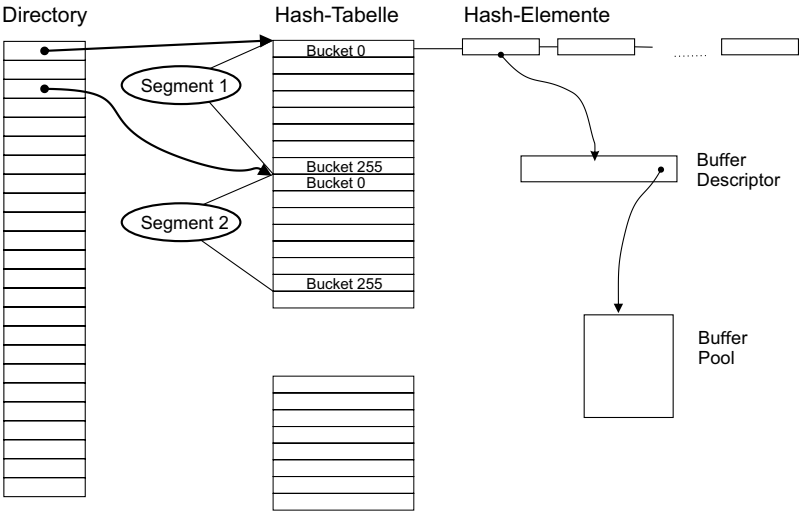


Bild 4.4 Struktur von Hash-Tabellen und Hash-Elementen

Das Hash-Element enthält drei Werte: den Hash-Wert für den aktuellen Eintrag, einen Link auf das nächste Element im selben Bucket sowie den Element-Schlüssel. Mithilfe des Schlüssels kann der zugehörige *Buffer Descriptor* gefunden werden.

Der Buffer Descriptor enthält den Index, um den Buffer im Buffer Pool zu finden. Will ein Prozess auf den Block zugreifen, dann muss er zuerst einen Sperrmechanismus bedienen, um zu verhindern, dass gleichzeitige Zugriffe auf den Buffer erfolgen. PostgreSQL regelt das über zwei Arten von Locks: *Spin-Lock* und *LW-Lock*.

Ein Spin-Lock wird für Operationen verwendet, die in sehr kurzer Zeit ausgeführt werden können. Damit ist die Wahrscheinlichkeit für wartende Prozesse, nach wenigen Sleeps den Lock zu erhalten, sehr groß. Spin-Locks verfügen ausschließlich über einen exklusiven Modus. LW-Locks werden verwendet, wenn auf eine Struktur zugegriffen werden soll. Sie können im exklusiven und im geteilten Modus vergeben werden.

HINWEIS: An dieser Stelle könnte die Frage aufkommen, ob sich dieser relativ komplexe Algorithmus, der dazu dient, einen Block im Memory zu finden und zu bearbeiten, auszahlt. Fakt ist, ein Block muss vor gleichzeitigen und unberechtigten Zugriffen geschützt werden. Dateisysteme verfügen über ähnliche Mechanismen. Alle Operationen, einschließlich des Findens eines Blocks sowie das Bilden des Hash-Werts laufen sehr schnell ab und sind wesentlich schneller als die Bearbeitung auf der Disk. Allerdings kann es auch im Memory zu Hotspots kommen, wenn einige Blöcke von mehreren Prozessen gleichzeitig stark frequentiert werden. In großen Shared-Memory-Einheiten können sich die Suchzeiten vergrößern, wenn ein Bucket zu viele Hash-Elemente enthält, da hier eine sequenzielle Suche stattfindet. Beides ist aber eine Aufgabe für das Performance-Tuning und stellt nicht die Algorithmen selbst in Frage.

Die folgenden Schritte beschreiben das Vorgehen, wenn ein Block im Shared Buffer gelesen wird:

1. Berechnung des Hash-Werts
2. Die Bucket-Nummer aus dem Hash-Wert ermitteln
3. LW-Lock im Modus „Shared“ holen
4. Den Block „pinnen“ und den „Usage Count“ erhöhen
5. Den Datenblock lesen
6. Den LW-Lock freigeben

Wie Sie sehen, wird selbst beim Lesen eines Blocks aus dem Cache ein Lock gesetzt, wenngleich im Shared-Modus. Bei einer Vielzahl von Prozessen, die gleichzeitig versuchen, auf eine geringe Anzahl von Blöcken zuzugreifen, kann es zu einer sogenannten Shared Buffer Contention kommen.

Im Shared Buffer Pool kann in der Regel nur ein Teil der Datenblöcke untergebracht werden. Es werden leere Blöcke im Shared Buffer Pool benötigt, wenn neue Daten von der Disk gelesen werden. Sind keine leeren Blöcke verfügbar, dann müssen Daten auf die Disk geschrieben werden.

Populäre Blöcke, auf die häufig zugegriffen wird, sollen dabei möglichst lange im Memory verweilen. PostgreSQL benutzt dafür den *Clock-Sweep-Algorithmus*. Dabei werden die Buffer zuerst ausgelagert, die den geringsten „Usage Count“ haben. Zur Erinnerung: Der Usage Count befindet sich im Buffer Descriptor.

HINWEIS: Einer der Vorzüge einer Open-Source-Datenbank ist, dass man in den Quellcode schauen kann. Er wurde von den Programmierern sehr gut kommentiert, so dass man viele technische Details daraus ableiten kann.

Schauen wir uns die Struktur des Buffer Descriptor an. Sie befindet sich in der Datei *buf_internals.h*.

Listing 4.2 Die Struktur des Buffer Descriptor im Quellcode

```
typedef struct BufferDesc
{
  BufferTag         tag;                /* ID of page contained in buffer */
  int               buf_id;             /* buffer's index number (from 0) */
  pg_atomic_uint32  state;              /* state of the tag, containing flags,
                                                refcount and usagecount */
  int               wait_backend_pid;   /* backend PID of pin-count waiter */
  int               freeNext;           /* link in freelist chain */
  LWLock            content_lock;       /* to lock access to buffer contents */
} BufferDesc;
```

Der Usage Count ist also in der Variable „state“ gespeichert. Weiterhin findet man den folgenden Kommentar zum Aufbau der Variable „state“:

```
/*
 * Buffer state is a single 32-bit variable where following data is combined.
 * - 18 bits refcount
 * - 4 bits usage count
 * - 10 bits of flags
 * Combining these values allows to perform some operations without locking
 * the buffer header, by modifying them together with a CAS loop.
```

Es handelt sich um eine 32-Bit-Variable, bei der 4 Bit für den Usage Count vorgesehen sind. Veränderungen und Abfragen werden durch Bit-Operationen im C-Sprachcode realisiert und sind daher sehr schnell.

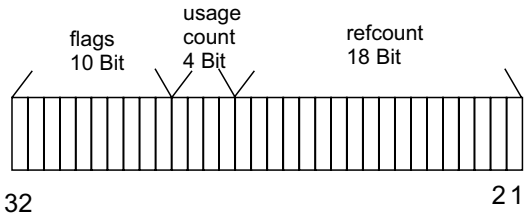


Bild 4.5 Aufbau der Variable „state“

Bei jedem Zugriff auf den Buffer erhöht sich der Usage Count um eins. Der Maximalwert wird durch BM_MAX_USAGE_COUNT begrenzt mit einem Standardwert von 5. Dies scheint auf den ersten Blick sehr niedrig. Der Wert ist ein Kompromiss zwischen Genauigkeit und Geschwindigkeit des Algorithmus. Schließlich sollte es nicht zu lange dauern, um die auszulagernden Blöcke festzulegen und neue freie Blöcke zur Verfügung zu stellen.

TIPP: Dies ist ein weiterer Vorteil von Open-Source-Datenbanken. Wenn Sie erfahren genug sind, dann können Sie auch solche Parameter oder Schwellwerte ändern und eine eigene Kompilation erstellen, die Ihren individuellen Bedürfnissen angepasst ist.

Die Konstante befindet sich ebenfalls in der Datei „buf_internals.h“:

```
#define BM_MAX_USAGE_COUNT 5
```

Der Clock Sweep-Algorithmus durchsucht die Buffer im Uhrzeigersinn nach Opfern. Dabei geht er so vor, dass der erste Buffer, der die Kriterien erfüllt, sofort zurückgemeldet wird. Danach steigt die Suche beim folgenden wieder ein und sucht den nächsten. Es muss also nicht gewartet werden, bis alle Buffer durchlaufen sind, und es können sofort und laufend freie Buffer zur Verfügung gestellt werden. Im Detail läuft der Algorithmus wie folgt ab:

1. Den Buffer Descriptor lesen.
2. Den Buffer Header sperren.
3. Falls „refcount“ einen Wert größer als null besitzt, wird die Sperre aufgehoben.
4. Falls „refcount“ den Wert null besitzt, wird der Usage Count gelesen.
5. Besitzt der Usage Count den Wert null, dann wird der Buffer zurückgemeldet, gespeichert und in die Freelist eingetragen.
6. Besitzt der Usage Count einen Wert größer als null, dann wird dieser um eins reduziert.

Es handelt sich um einen sehr einfachen, aber effektiven Algorithmus, um populäre Buffer im Memory zu behalten. Schließlich geht es auch darum, nicht zu viel Overhead zu erzeugen und schnell freie Buffer zur Verfügung zu stellen. Jetzt wird auch klarer, weshalb eine Erhöhung des Maximalwerts zu längeren Suchzeiten führt, auch wenn damit genauere Ergebnisse geliefert werden.

4.2.2.2 Der WAL Buffer

Ähnlich wie die Shared Buffer für die Datenblöcke dient der WAL Buffer zur Performancesteigerung für die WAL-Sätze. Das Write Ahead Log, allgemein auch Transaktionslog genannt, ist ein sehr wichtiger Bestandteil des Datenbank-Clusters. Es dient der Wiederherstellung der Datenbanken im Fall eines Crashs oder Systemabsturzes und der Lesekonsistenz und es spielt eine grundlegende Rolle für die Implementierung von Standby-Datenbanken.

Die Funktionsweise des WAL Buffer ist allerdings sehr unterschiedlich zum Shared Buffer. Hier geht es darum, die WAL-Sätze möglichst schnell auf die Disk zu schreiben, es ist eine Einbahnstraße. Erst wenn die Sätze auf der Disk angekommen sind, ist die Integrität der Datenbank gesichert und die Transaktion erfolgreich abgeschlossen.

Die WAL-Datei, auch WAL-Segment genannt, hat in der Version 14 eine Standardgröße von 16 MByte mit einer Standardblockgröße von 8 KByte.

HINWEIS: Eine Änderung der Blockgröße ist selten sinnvoll, da der WAL Writer sehr kleine WAL-Sätze sehr häufig schreibt. Hier ist eher eine kleine Latenz der Schreibprozesse auf das I/O-Subsystem von Vorteil. Eine Erhöhung der Segmentgröße ist sinnvoll für Datenbanken mit einer hohen Transaktionslast. Die Standardgrößen können beim Kompilieren der Software geändert werden. Geben Sie dazu im `configure`-Befehl die Optionen `--with-wal-blocksize` und `--with-wal-segsize` an.

Die WAL-Datei befindet sich im Verzeichnis *$PGDATA/pg_wal*. Der WAL-Dateiname ist eine 24-stellige Hexadezimalzahl, die nach der folgenden Formel gebildet wird:

Listing 4.3 Formel für den WAL-Dateinamen

```
timelineID + (uint32)((LSN - 1)/16M*256)) + (uint32)((LSN - 1)/16M))%256
```

Die *timelineID* ist eine Zahl (4 Byte unsigned integer), die den Zeitstrahl widerspiegelt, und Teil des Point-in-Time-Recovery-Konzepts. *LSN* ist die Log Sequence Number, eine laufende Nummer, die bei jedem WAL-Wechsel erhöht wird.

Der erste Name der WAL-Datei ist somit *000000010000000000000001*. Wenn die erste Datei mit WAL-Sätzen vollständig gefüllt ist, dann wird eine neue mit dem Dateinamen *000000010000000000000002* geschrieben und es erfolgt ein sogenannter WAL-Segment-Switch. Die LSN wird immer weitergezählt und der Dateiname entsprechend der Formel in Listing 4.3 gebildet.

Umfassende Werte der Konfiguration liefert das Utility *pg_controldata*. Ein Beispiel finden Sie in Listing 4.4. Hier wurden die Standardgrößen angepasst. Die Segmentgröße beträgt hier 32 MByte und die Blockgröße 16 KByte.

Listing 4.4 WAL-Informationen mit `pg_controldata` abfragen

```
$ pg_controldata
pg_control version number:            1400
Catalog version number:               202107181
Database system identifier:           7006289689799659249
Database cluster state:               in production
pg_control last modified:             Sat 09 Oct 2021 11:20:38 AM CEST
Latest checkpoint location:           0/5111F700
Latest checkpoint's REDO location:    0/5111F700
Latest checkpoint's REDO WAL file:    000000010000000000000051
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
Latest checkpoint's NextXID:          0:7186
Latest checkpoint's NextOID:          16408
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        726
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  0
Latest checkpoint's oldestMultiXid:   1
Latest checkpoint's oldestMulti's DB: 1
Latest checkpoint's oldestCommitTsXid:0
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:            Fri 08 Oct 2021 05:33:57 PM CEST
Fake LSN counter for unlogged rels:   0/3E8
Minimum recovery ending location:     0/0
Min recovery ending loc's timeline:   0
Backup start location:                0/0
Backup end location:                  0/0
End-of-backup record required:        no
wal_level setting:                    replica
wal_log_hints setting:                off
max_connections setting:              100
max_worker_processes setting:         8
```

```
max_wal_senders setting:             10
max_prepared_xacts setting:          0
max_locks_per_xact setting:          64
track_commit_timestamp setting:      off
Maximum data alignment:              8
Database block size:                 8192
Blocks per segment of large relation: 131072
WAL block size:                      8192
Bytes per WAL segment:               16777216
Maximum length of identifiers:       64
Maximum columns in an index:         32
Maximum size of a TOAST chunk:       1996
Size of a large-object chunk:        2048
Date/time type storage:              64-bit integers
Float8 argument passing:             by value
Data page checksum version:          0
Mock authentication nonce:
c421e56d68ab813c48d921bff45eb2ecd375a129a46784c88ffb651be58387ea
```

In Bild 4.6 ist die Struktur des WAL-Segments dargestellt. Ein WAL-Block, auch WAL Page genannt, besitzt eine feste Größe, die im Standardfall 8 KByte beträgt. In diesem Block befinden sich ein oder mehrere WAL Records, die eine variable Länge besitzen. Ein WAL Record besteht aus Kopf- und Datenteil.

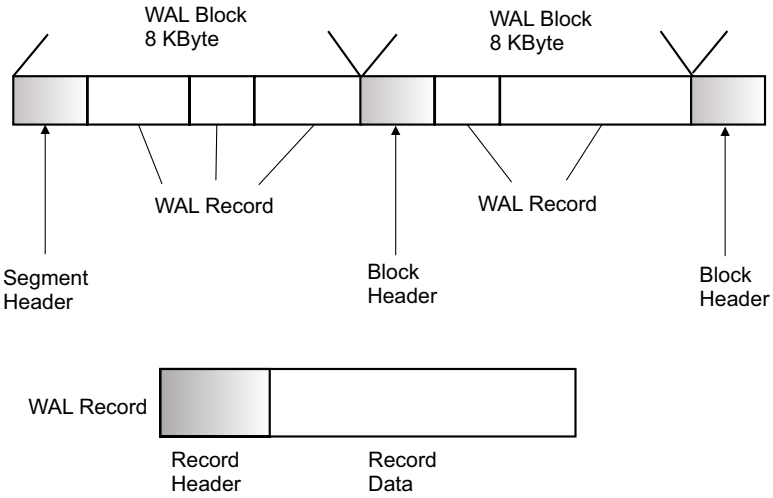


Bild 4.6 Struktur eines WAL-Segments

HINWEIS: WAL-Segmente lassen sich auslesen. Die internen Strukturen können aus dem Quellcode ermittelt werden. Sie finden diese in den Dateien *xlog_internal.h* und *xlogrecord.h*. Das Auslesen kann zum Beispiel für Replikationsmechanismen oder Transaktionsanalysen verwendet werden. Ein einfaches Auslesen ist mit dem Utility *pg_waldump* möglich.

WAL-Sätze, also INSERT-, UPDATE- oder DELETE-Anweisungen, werden zunächst in den WAL Buffer im Memory geschrieben. Sie werden permanent, aber spätestens bei einer COMMIT-Anweisung, in das WAL-Segment auf die Disk geschrieben. Das permanente Schreiben erfolgt durch den WAL-Writer-Prozess, der standardmäßig alle 200 Millisekunden prüft, ob WAL-Sätze geschrieben werden müssen. Die Frequenz wird durch den Parameter *wal_writer_delay* vorgegeben.

Die Log Sequence Number (LSN) repräsentiert den Speicherort in WAL-Segment. Sie ist die eindeutige ID für den WAL-Satz. Die eigentlichen Datenblöcke befinden sich nicht zwangsläufig auf der Disk, sondern teilweise im Shared Buffer Cache. Mit dem Schreiben der WAL-Sätze ist jedoch die Wiederherstellbarkeit der Transaktion gewährleistet.

Wie funktioniert der Recovery-Algorithmus? Wird zum Beispiel eine INSERT-Anweisung erzeugt, dann stellt PostgreSQL eine Page im Shared Buffer Pool bereit und die Daten werden in die Page im Memory geschrieben. Ein entsprechender Eintrag erfolgt als WAL-Satz im WAL Buffer. Mit dem Abschluss der Transaktion in Form einer COMMIT-Anweisung werden alle zur Transaktion gehörenden Sätze in das WAL-Segment geschrieben.

Für den Recovery-Prozess liest PostgreSQL den WAL-Satz aus dem WAL-Segment und stellt die Page der Tabelle im Shared Buffer Pool bereit. Es wird die LSN der Page mit der LSN des WAL-Satzes verglichen. Ist die LSN im WAL-Satz größer, dann wird die Änderung auf die Page im Buffer Pool angewandt. Dieser Vorgang wird so lange wiederholt, bis das Ende des WAL-Segments erreicht ist.

Der Checkpoint-Prozess schreibt alle geänderten Buffer aus dem Shared Buffer Pool auf die Disk und stellt damit einen konsistenten Systemzustand her. Ein Checkpoint wird in den folgenden Fällen ausgelöst:

1. Das Intervallende des Parameters *checkpoint_timeout* ist erreicht. Der Standardwert ist 5 Minuten.
2. Der Wert des Parameters *max_wal_size* wurde überschritten. Standard ist 1 GByte, also 64 WAL-Dateien.
3. Das PostgreSQL-Cluster wird mit der Option „smart“ oder „fast“ gestoppt.
4. Es wird ein Checkpoint-Kommando eingegeben.

Die folgenden Schritte werden bei einem Checkpoint ausgeführt:

1. Ein REDO-Punkt wird im Memory gespeichert. Er ist der Startpunkt für einen Recovery-Prozess.
2. Ein Checkpoint-Eintrag wird in das WAL-Segment geschrieben. Er enthält unter anderem Informationen des REDO-Punkts.
3. Alle geänderten Blöcke aus dem Shared Buffer Pool werden auf die Disk geschrieben.
4. Die Datei *pg_control* wird aktualisiert. In ihr befinden sich Basisinformationen, wie zum Beispiel der Speicherort des Checkpoint-Satzes.

Ein WAL-Segment-Switch wird ausgelöst, wenn eines der folgenden Ereignisse stattfindet:

1. Das aktuelle WAL-Segment ist voll.
2. Der Archive-Modus ist aktiviert und das Intervall „archive_timeout“ wird erreicht.
3. Ein manueller Switch wird ausgelöst durch die Funktion „pg_switch_wal“.

Ist der Archive-Modus aktiviert, dann können die WAL-Dateien archiviert werden. Produktive Datenbanken sollten grundsätzlich im Archive-Modus laufen. Die Archivierung liefert eine zusätzliche Kopie und bringt damit mehr Sicherheit in die Wiederherstellbarkeit des Clusters. Außerdem kann eine längere Historie behalten werden.

4.3 VACUUM

Jedes Datenbanksystem muss in der Lage sein, verschiedene Versionen von Sätzen zu verwalten. So wird ein geänderter Satz erst für andere Sessions sichtbar, wenn die Transaktion beendet, also die Änderung mit COMMIT abgeschlossen wurde. Andererseits muss die Möglichkeit bestehen, offene Transaktionen mit einem Rollback-Befehl zurückrollen zu können.

Die Verwaltung mehrerer Versionen wird auch als Multiversion Concurrency Control-Modell (MVCC) bezeichnet. Während einige Datenbanksysteme den Weg über spezielle UNDO-Strukturen gehen, verfolgt PostgreSQL einen anderen Ansatz. Verschiedene Versionen von Datensätzen werden in der Tabelle gespeichert und ältere Versionen gelöscht, wenn sie nicht mehr benötigt werden.

Damit entstehen natürlich permanent Lücken in den Tabellen. Aus diesem Grund wurde in PostgreSQL ein spezieller Prozess eingerichtet, der sich um diese Lücken kümmert. Der VACUUM-Prozess ist sehr wichtig, um eine starke Fragmentierung zu verhindern. Standardmäßig läuft ein Prozess mit dem Namen *autovacuum launcher process*, der sich im laufenden Betrieb um die Defragmentierung kümmert. VACUUM-Operationen können aber auch manuell angestoßen werden.

Der Launcher-Prozess startet für jede Datenbank einen VACUUM Worker-Prozess in dem Intervall, das der Parameter *autovacuum_naptime* vorgibt. Wenn Sie zum Beispiel fünf Datenbanken in einem Cluster betreiben und der Parameter auf 60 Sekunden eingestellt ist, dann startet der Launcher-Prozess alle zwölf Sekunden einen Worker-Prozess für genau eine Datenbank. Mit dem Parameter *autovacuum_max_workers* lässt sich begrenzen, wie viele Worker-Prozesse parallel laufen dürfen.

Ein Standard-VACUUM, so wie es durch das Auto-VACUUM durchgeführt wird, markiert nicht mehr benötigte Sätze (Dead Rows) als wiederverwendbar. Dieser Platz kann für zukünftige INSERT-Operationen verwendet werden. Der Speicherplatz wird allerdings nicht an das Betriebssystem zurückgegeben. Ist dies erforderlich, muss ein VACUUM FULL-Kommando ausgeführt werden. Ein VACUUM FULL ist sehr I/O-intensiv und sollte nur dann ausgeführt werden, wenn es unbedingt erforderlich ist. Neben dem Schrumpfen von Tabellen gibt es einen weiteren wichtigen Grund, weshalb ein VACUUM FULL ausgeführt werden sollte: das *XID Wraparound*.

Kommen wir noch einmal zum Thema Multiversion Concurrency Control zurück. Die Pflege mehrerer Versionen von Datensätzen ist nicht nur für das Transaktionsverhalten, also für COMMIT- und ROLLBACK-Operationen, erforderlich. PostgreSQL garantiert Lesekonsistenz, das bedeutet, die Daten werden für den Zeitpunkt angezeigt, zu dem die SQL-Abfrage gestartet wurde. Dafür wird das MVCC-Modell benötigt.

Nehmen wir an, eine SQL-Abfrage wird um 11:11 Uhr gestartet und benötigt aufgrund des großen Datenvolumens 20 Minuten für die Ausführung (siehe Bild 4.7).

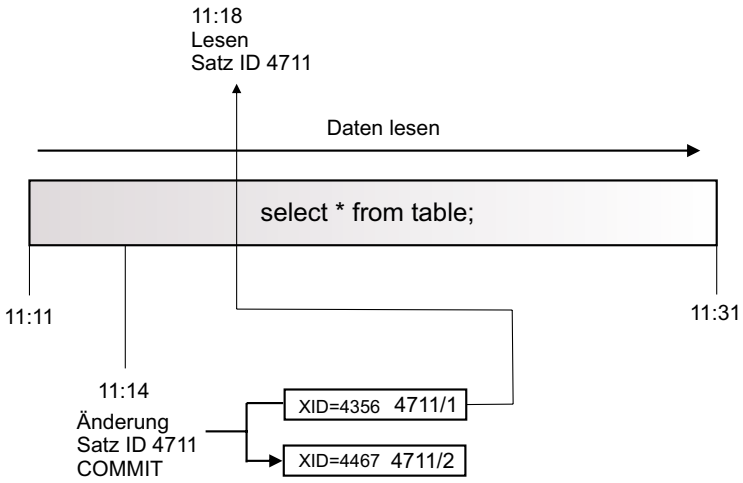


Bild 4.7 Lesekonsistenz mit dem MVCC-Modell

Um 11:14 Uhr wird der Satz mit der ID 4711 durch eine andere Session geändert und die Transaktion mit der XID 4467 mit COMMIT abgeschlossen. Es wird eine zweite Version des Satzes gespeichert. Er enthält die geänderten Daten und ist damit für alle anderen Sessions sichtbar. Um 11:18 kommt der Leseprozess an den Punkt, wo er den Satz mit der ID 4711 liest. Er benötigt allerdings nicht den geänderten Satz, sondern den Zustand von 11:14 Uhr und liest deshalb die ältere Version 4711/1.

PostgreSQL löst die Versionierung von Sätzen über die Transaktions-ID (XID). Diese ist in jedem Satz vermerkt und hat eine Größe von 32 Bit. Leider konnten sich die Programmierer auch in der Version 14 nicht dazu durchringen, eine Transaktions-ID von der Größe 64 Bit einzuführen. Allerdings sind die Auswirkungen nicht zu unterschätzen. Schließlich befindet sich die Transaktions-ID in den Spalten „xmin“ und „xmax“ in jedem Datensatz. Dies würde die Datenbanken signifikant aufblähen.

Listing 4.5 Definition der Transaktions-ID als 32-Bit-Wert

```
typedef uint32 TransactionId;
```

Mit dem 32-Bit-Wert ist die Anzahl von Transaktionen auf 4,3 Milliarden begrenzt und es kann auf großen Systemen zum gefürchteten *XID Wraparound* kommen. Wird die Grenze von 4,3 Milliarden erreicht, dann beginnt der Zähler wieder bei „null“. Damit würden alle Transaktionen, die eigentlich der Vergangenheit angehören, plötzlich in der Zukunft liegen.

Das Problem wurde mithilfe der sogenannten „Frozen Transaction ID“ adressiert. Sie wird bei der Initialisierung der Datenbank erstmalig festgelegt:

- Bootstrap XID: Wird von *initdb()* gesetzt, *Wert = 1*.
- Frozen XID: Wird auf *Wert = 2* gesetzt.
- Normale XID: Wird auf *Wert = 3* gesetzt.

Wie wir wissen, arbeitet das MVCC-Modell so, dass die Version eines Satzes abgefragt wird, in dem die XID kleiner oder gleich der XID ist, die dem Startzeitpunkt der SQL-Abfrage entspricht. Weiterhin wird angenommen, dass die Frozen XID immer kleiner ist als die aktuelle XID. Der VACUUM-Prozess markiert solche Sätze als „frozen“. Für solche Sätze wird nicht der normale Vergleich der XID angesetzt. Dagegen wird zum aktuellen Wert der XID der Wert 2^{31} addiert und mit der Frozen XID verglichen. Die Werte lassen sich abfragen und nachvollziehen.

Listing 4.6 XID und Frozen XID abfragen

```
(postgres@[local]:5432)[postgres]> SELECT datname,datfrozenxid FROM pg_database;
  datname  | datfrozenxid
-----------+--------------
 postgres  |          547
 hanser    |          547
 template1 |          547
 template0 |          547
(postgres@[local]:5432)[postgres]> SELECT txid_current();
 txid_current
--------------
          563
(postgres@[local]:5432)[postgres]> select xmin,* from test;
 xmin | id |           text
------+----+--------------------------
  556 |  1 | XXXXXXXXXXXXAAAAAAAAAAAA
  557 |  2 | XXXXXXXXXXXXAAAAAAAAAAAA
```

Bis zur Version 9.3 wurde die Frozen ID in die Spalte „xmin“ geschrieben und damit die normale XID ersetzt. In neueren Versionen einschließlich Version 14 wird nur ein „Frozen Flag“ gesetzt und die Original-XID bleibt erhalten. Damit ist es möglich, forensische Untersuchungen zuverlässiger durchführen zu können.

Der Vergleich von normalen XID-Werten erfolgt mit einer 2^{32}-Modulo-Arithmetik. Damit erscheinen immer 2 Milliarden Werte in der Vergangenheit.

TIPP: Stellen Sie sicher, dass jede Tabelle nach höchstens 2 Milliarden Transaktionen im Cluster einen VACUUM-Prozess komplett durchlaufen hat. Damit werden ältere Zeilen als „frozen“ markiert, das Wraparound-Problem kann mithilfe der Frozen ID gelöst werden und es kommt nicht zu fehlerhaften SQL-Abfragen oder Datenverlust.

In Listing 4.7 sehen Sie, wie der Zeitpunkt des letzten VACCUM-Laufs abgefragt werden kann.

Listing 4.7 Den Zeitpunkt des letzten VACUUM-Laufs abfragen

```
(postgres@[local]:5432)[postgres]> SELECT relname,last_vacuum, last_autovacuum
> FROM pg_stat_user_tables;
 relname |          last_vacuum          | last_autovacuum
---------+-------------------------------+-----------------
 test    | 2020-12-12 20:23:25.477454+00 |
```