SelfLinux-0.10.0

PostgreSQL

Autor: Steffen Dettmer (*steffen@dett.de*)
Formatierung: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)

Inhaltsverzeichnis

1 Einleitung

Datenbanken Managementsysteme helfen, komplexe Aufgaben einfach zu lösen, da etliche Teilaufgaben von diesen erledigt werden.

PostgreSQL ist ein relationales Datenbank Managementsystem (RDBMS). Es ist OpenSource und verfügt über Leistungsmerkmale, die es für den Einsatz in Produktionsumgebungen qualifizieren. *PostgreSQL* gilt wohl als das zuverlässigste und fortschrittlichste OpenSource DBMS.

1.1 Über dieses Dokument

Dieses Dokument kann nur einen kleinen Überblick über *PostgreSQL* geben. Viele Themen werden nur angerissen oder überhaupt nicht erwähnt. Da *PostgreSQL* sehr komplex ist, und ständig weiterentwickelt wird, kann es ebenfalls sein, dass etliche der hier genannte Informationen nicht mehr aktuell sind. Trotz aller Sorgfalt kann es dennoch sein, dass einige der Informationen aus diesem Dokument fehlerhaft sind.

Es ist versucht worden, dieses Dokument so zu schreiben, dass man nur wenig Datenbankvorwissen benötigt. Grundlegende Linux-Administrationskenntnisse werden benötigt. So sollte bekannt sein, wie man Softwarepakete im Allgemeinen installiert.

Dieser Text beschäftigt sich hauptsächlich mit den Besonderheiten von *PostgreSQL*. Um effektiv mit einem Datenbanksystem arbeiten zu können, sollten allgemeine Kenntnisse von relationalen Datenbanken vorhanden sein (Relationen, Normalformen, Datenmodelle, SQL usw.).

Hier wird auch nicht SQL erklärt. Interessierte finden in einer *PostgreSQL* Referenz oder in einem SQL Buch Informationen. Es wird jedoch versucht, auf *PostgreSQL* Eigenheiten einzugehen. Das SQL ist nah am Standard SQL92 und realisiert SQL99 teilweise.

Dank gebührt *Mirko Zeibig* für die fachliche Kontrolle und Korrektur, insbesondere vielen Anpassungen und Aktualisierungen für die Version 7.3. Der Autor  *Steffen Dettmer* freut sich natürlich über Kommentare und Anregungen zu diesem Text.

1.2 Datenbanken allgemein

Datenbanken sind nicht einfach nur **Daten in Tabellen**. Vielmehr bestehen sie neben den eigentlichen Daten aus Zugriffsbedingungen und -berechtigungen, Benutzern, Regeln, Funktionen und Weiterem. Sie dienen dazu, umfangreiche Datenmengen zu speichern und wiederzugewinnen und dies mehreren Anwendungen gleichzeitig zu ermöglichen.

Es geht hier also nicht einfach nur um Datenspeicher, sondern auch um das Knüpfen von Regeln an Daten. So kann man beispielsweise sicherstellen, dass nur bestimmte Benutzer diese Daten lesen oder ändern können. Man kann sicherstellen, dass die Daten Konsistenzbedingungen genügen; beispielsweise, dass gespeicherte Adressen entweder gar keine oder eine fünfstellige Postleitzahl haben, die nur aus Ziffern besteht.

Datenbanken werden von sogenannten Datenbank Mangementsystemen (DBMS) verwaltet. Nur das DBMS selbst kann direkt auf die Datenbanken zugreifen. Anwendungen greifen immer über das DBMS auf diese zu. Das DBMS prüft und kontrolliert dabei und führt komplexe Operationen im Auftrag von Anwendungen aus.

Solche Systeme werden eingesetzt, um Datenredundanz zu verringern. In der Regel wird jedes Datum nur einmal in der Datenbank gespeichert. Verschiedene Repräsentationen werden vom DBMS bei Bedarf bereitgestellt, ohne diese Daten etwa zu kopieren.

Wie bereits angedeutet, werden sie eingesetzt, um die Datenintegrität (Konsistenz) zu gewährleisten. Bei Veränderungen werden beispielsweise notwendige Folgeänderungen automatisch durchgeführt, das Eintragen von inkonsistenten Daten wird verhindert. Des weiteren werden die Daten vor unberechtigtem Zugriff geschützt. Auch sind die Daten unabhängiger von den Anwendungsprogrammen; es spielt keine Rolle, wie diese tatsächlich gespeichert werden. Wird hier etwas geändert, so müssen die Anwendungen nicht alle angepasst werden.

1.3 Schnittstellen und Clients

Schnittstellen zur Datenbank sind immer Schnittstellen zum Datenbank Managementsystem. Da dies immer so ist, spricht man auch einfach von **Datenbankschnittstellen**.

Anwendungen kommunizieren über Schnittstellen mit dem DBMS. Diese Schnittstellen sind fast immer in mehrere Ebenen zu unterteilen. Die meisten Schnittstellen sind netzwerkfähig, üblicherweise verwenden DBMS TCP/IP. Darüber setzen Anwendungen Befehle oder Kommandos in Datenbanksprachen ab. Die sicherlich bekannteste Datenbanksprache ist *SQL* (structured query language). Natürlich verfügt jede Datenbank über eine eigene Schnittstelle, dass heißt, der genaue Aufbau der Kommandos und vor allem die Übertragung über das Netzwerk unterscheiden sich erheblich - selbst bei der Verwendung von *SQL*.

DBMS verfügen über Dienstprogramme, die über solche Schnittstellen mit der Datenbank kommunizieren. Diese Dienstprogramme sind für bestimmte Aufgaben unumgänglich, beispielsweise für die erste Einrichtung. Oft bieten diese Zusatzfunktionen, die man für die Administration benötigt, beispielsweise das Anlegen von Benutzern. Zusätzlich bieten die meisten DBMS solche Funktionen auch über *SQL* an: die Kommandos sind so definiert, dass sie den *SQL* Standards genügen bzw. diesen nicht widersprechen. Man spricht hier von *SQL*-Erweiterungen. Die meisten Datenbanken verfügen über etliche Erweiterungen, die man nach Möglichkeit jedoch sparsam einsetzen sollte, um sich spätere Migrationsprobleme zu ersparen.

Damit Anwendungen nicht so DBMS abhängig sind, gibt es eine weitere Ebene der Schnittstellen, die Programmier- oder Applikationsebene. Das ist in der Regel eine Reihe von Funktionen, die aus einer Programmiersprache aufgerufen werden können. Die wohl bekanntesten Schnittstellen sind hier ODBC und JDBC. ODBC, OpenDataBaseConnectivity, beschreibt, wie Programme mit DBMS kommunizieren können. ODBC selbst ist in etwa eine Funktionsbibliothek, die *SQL* im Prinzip voraussetzt. ODBC regelt aber auch den Verbindungsaufbau zu einem DBMS, Benutzerauthentifizierung und anderes. JDBC ist das Standard-Verfahren, Java-Anwendungen mit DBMS Unterstützung zu versehen.

Auch wenn in der Theorie die Schnittstellen ODBC und JDBC suggerieren, dass man das DBMS problemlos transparent wechseln kann, ist das in der Praxis selten so. In der Praxis sind diese Standards nämlich selten vollständig implementiert, und es gibt etliche Punkte, wo sie nicht eindeutig sind. Gerade über ODBC ist es zudem äußerst umständlich, wirklich datenbankunabhängig zu arbeiten, so dass sich meistens einige ärgerliche Abhängigkeiten einschleichen.

Beispiele für Anwendungen beziehungsweise Clients sind Datenbank Frontends. Die wohl beliebtesten für *PostgreSQL* sind `psql` (eine textbasierter, interaktiver *SQL* Interpreter), `pgaccess` (ein graphisches Frontend, mit dem man gut Tabellen, Views, Reports und vieles mehr anlegen und bearbeiten kann) und `phpPgAdmin`, einem sehr flexiblen Webfrontend, dass man sich unbedingt anschauen sollte, wenn man Webfrontends mag.

1.4 PostgreSQL

PostgreSQL ist ein relationales DBMS (RDBMS). Als Hauptsprache findet *SQL* Verwendung (natürlich ebenfalls mit etlichen Erweiterungen). Wie viele andere Datenbanken unterstützt auch *PostgreSQL* nicht den vollständigen *SQL* Standard. Die Funktionen von *PostgreSQL* sind aber sehr umfangreich, so dass man in der Praxis selten Beispiele findet, wo sich eine Anforderung nicht realisieren läßt.

PostgreSQL ist ein Nachfolger von ***INGRES*** und ***POSTGRES***, ist jedoch in vielen Punkten stark erweitert und verbessert worden. Die Entwickler unternehmen große Anstrengungen, um möglichst standardkonform zu sein. Das DBMS ist in weiten Teilen *SQL92* kompatibel und unterstützt einiges aus *SQL99*. Es ist unter einer BSD-Style Lizenz verfügbar.

Auch wenn bekannt ist, dass es mit großen Datenmengen umgehen kann (es sind Installationen mit mehr als 60 GB Datenbasis bekannt), eignet es sich wohl weniger für Enterprise-Class Anwendungen. Hier sollten professionelle Datenbanken wie IBM DB2 erwogen werden.

Bei einfachen Datenbankanwendungen, beispielsweise CGI basierten Webanwendungen, die ein paar Adressen speichern, wird von vielen ein einfacheres, schnelleres DBMS vorgezogen: `mySQL`. `mySQL` gilt als schnell installiert und einfach bedienbar.

Natürlich ist *PostgreSQL* auch in kleinen Systemen eine sehr gute Wahl, und da kleine Systeme zum Wachsen neigen, ergibt sich hier schnell ein weiterer Vorteil. *PostgreSQL* wartet dafür auch mit etlichen **high-end** Datenbankfunktionen auf.

1.5 Verfügbarkeit

PostgreSQL, JDBC und ODBC sind für verschiedene Plattformen verfügbar, beispielsweise für Linux, BSD und Windows. *PostgreSQL* kann über unixODBC und Perl::DBI verwendet werden, als Programmierplattformen sind C/C++, Java, PHP, Perl, TCL und viele andere bekannt.

Auch Microsoft Windows Benutzer können Vorteile ziehen, so kann MS Access beispielsweise problemlos über ODBC auf das DBMS zugreifen. Damit kann man die Datenbank gut in die MS Office Anwendungen einbetten.

1.6 Vorteile und Funktionen

Die ISO Transaktionsmodelle **read committed** und **serializable** werden unterstützt. Umfangreiche Möglichkeiten für Benutzerberechtigungen und Datenbankregeln stehen zur Verfügung. JDBC und ODBC Treiber sind für verschiedene Plattformen verfügbar, beispielsweise für Linux, BSD und Windows. Die Software wird von einer stabilen, erfahrenen und weltweit arbeitenden Gruppe gepflegt und weiterentwickelt.

PostgreSQL gilt nach 16 jähriger Entwicklungszeit als sehr stabil und zuverlässig. "Advocacy" schreibt hierzu:

> Im Gegensatz zu Benutzern vieler kommerzieller Datenbanksysteme ist es bei Unternehmen, die *PostgreSQL* einsetzen der Normalfall, dass das Datenbanksystem noch kein einziges Mal abgestürzt ist. Auch nicht bei jahrelangem Einsatz und großem Datenaufkommen. Es läuft einfach.

Wie bereits erwähnt, versuchen die Entwickler, nah an den Standards zu arbeiten. Fast alle von *SQL92* und *SQL99* spezifizierten Datentypen werden unterstützt, eigene Datentypen können erzeugt werden. Fremdschlüssel, Trigger und Views sind verfügbar. Alle von *SQL99* spezifizierten **joins** sind implementiert. Internationale Zeichentabellen, Unicode und locale gehören ebenso dazu, wie Unterstützung von Unterabfragen (sub queries), GROUP BY, UNION, INTERSECT, LIMIT, LIKE und vollständige POSIX konforme reguläre Ausdrücke, verschiedene Indexverfahren. Das DBMS ist an weiten Teilen erweiterbar. Transaktionen werden unterstützt (ISO **read commited** und **serializable**), umfangreiche Sicherheitskonzepte sind realisierbar (Benutzer, SSL/TLS, Algorithmen). Mehrere CPUs können verwendet werden, eben so **virtuelle hosts**.

Es gibt viele Erfolgsgeschichten über *PostgreSQL*; im Internet finden sich viele Projekte und Firmen, die erfolgreich aufwendige Systeme mit dieser Datenbank realisiert haben.

1.7 Grenzen von PostgreSQL

PostgreSQL verfügt wohl über keine praxisrelevanten Grenzen mehr. Plant man eine Anwendung, die möglicherweise in die Nähe der im Folgenden angegebenen Werte kommt, sollte man ernsthaft den Einsatz von IBM DB/2 oder anderen Enterprise Class Systemen erwägen.

Maximale Datenbankgröße	unbegrenzt (60 GB Datenbanken existieren)
Maximale Tabellengröße	64 TB (65536 GB) auf allen Plattformen
Maximale Größe einer Zeile	unbegrenzt
Maximale Größe eines Feldes	1 GB
Maximale Anzahl von Zeilen in einer Tabelle	unbegrenzt
Maximale Anzahl von Spalten in einer Tabelle	1600
Maximale Anzahl von Indizes einer Tabelle	unbegrenzt

2 Installation

Die Installation gliedert sich im Wesentlichen in zwei Komplexe. Zum Einen muss zunächst natürlich die Software selbst installiert werden. Zum Anderen müssen einige Dinge eingestellt und eine erste Datenbank muss erzeugt werden.

Die meisten Linuxdistributionen sollten *PostgreSQL* als Softwarepakete anbieten. Auf Grund der Größe sind es oft sogar mehrere. Mit der Installation solcher Pakete wird in der Regel auch die Grundkonfiguration durchgeführt, so dass *PostgreSQL* sofort nach dem Installieren gestartet und benutzt werden kann.

2.1 Installation der Software

Verfügt die verwendete Distribution über Softwarepakete, so sollten diesem im Allgemeinen vorgezogen werden und mit den DistributionsProgramme installiert werden. Dies spart mindestens viel Arbeit und Zeit. Installiert man beispielsweise die Pakete von SuSE, so kann man die Datenbank sofort nach der Installation starten.

Es ist natürlich auch möglich, *PostgreSQL* als Quellpaket über http://www.postgresql.org/ downzuladen und es selbst zu kompilieren. Dies ist insbesondere dann notwendig, wenn man ganz bestimmte Einstellungen benötigt, beispielsweise Unterstützung für bestimmte Zeichensätze. Der Rest dieses Kapitels beschäftigt sich mit diesem Verfahren und kann ausgelassen werden, wenn ein Distributionspaket verfügbar ist.

PostgreSQL verwendet `./configure` und `make` zum Übersetzen und verhält sich damit sehr ähnlich zu GNU Software - jedoch sind einige zusätzliche Schritte nach dem Installieren notwendig. Die Installation unter Windows/Cygwin ist nicht Thema dieses Dokumentes, hier wird ausschließlich auf Linux eingegangen.

2.1.1 Vorbedingungen

Um die Datenbanksoftware selbst übersetzen zu können, müssen etliche Programme verfügbar sein. Eine halbwegs aktuelle Linuxdistribution vorausgesetzt, sind diese aber entweder bereits installiert oder als Softwarepakete verfügbar.

Neben `GNU-make` ist natürlich ein C Compiler erforderlich. Der `GCC` ist hier gut geeignet. *PostgreSQL* stellt also keine hohen oder speziellen Anforderungen an das System.

2.1.2 Auspacken und Vorbereiten

Über http://www.postgresql.org/ besorgt man sich ein Paket der Software. Im Beispiel wird die Version 7.2.1 verwendet. Die Schritten sollten bei neueren (und älteren) Versionen analog sein.

Die erhaltenen Quellen packt man zunächst aus:

```
root@linux / # tar xzf postgresql-7.2.1.tar.gz
```

Führt man ein `update` durch, so sollte man unbedingt spätestens jetzt die Datenbank in eine Datei sichern. Das kann man mit dem *PostgreSQL* Programm `pg_dump` oder `pg_dumpall` erledigen:

```
root@linux / # pg_dumpall > backup.sql
```

Genaueres findet sich im Abschnitt **Backup**. Anschließend vergewissert man sich, dass das Backup erfolgreich

war und stoppt die (alte) Datenbank. Das Datenverzeichnis der alten Datenbank sollte aus Sicherheitsgründen umbenannt werden:

```
root@linux / # mv /usr/local/pgsql /usr/local/pgsql.old
```

Dieser Pfad ist bei Distributionen in der Regel anders; SuSE und RedHat verwenden beispielsweise `/var/lib/pgsql/data`.

Nun führt man in dem Verzeichnis, das durch das tar Kommando entstanden ist, `configure` aus. Dabei kann man etliche Optionen angeben. Neben die üblichen GNU Optionen wie beispielsweise `--prefix`, gibt es auch viele *PostgreSQL* spezifische Optionen.

Ist ein produktiver Einsatz geplant, so sollte die Dokumentation zu Rate gezogen werden, und ausführliche Tests gefahren werden.

Einige wichtige Optionen:

`--enable-locale`	Aktiviert locale-Unterstützung. Dies kostet etwas Performanz, ist aber im nicht-englischsprachigem Raum sehr sinnvoll
`--enable-multibyte`	Aktiviert Multibyte Unterstüzung (unter anderem Unicode). Java und TCL erwarten beispielsweise Multibyte. Diese Option sollte daher nach Möglichkeit gesetzt werden.
`--enable-nls`	Aktiviert Sprachunterstützungen, um Meldungen in Landessprache geben zu können.
`--with-CXX`	C++, Perl, Python, TCL beziehungsweise Java (JDBC) Bibliotheken erzeugen.
`--with-perl`	Für Java wird das Programm "ant" benötigt.
`--with-python`	
`--with-tcl`	
`--with-java`	
`--enable-odbc`	Erzeugt ODBC Treiber. Es kann unabhängig vom DriverManger erzeugt werden
`--with-iodbc`	(weder --with-iodbc noch --with-unixodbc), für die Verwendung mit iODBC oder
`--with-unixodbc`	unixODBC, nicht jedoch für beide.
`--with-openssl`	OpenSSL SSL/TLS Unterstützung aktivieren
`--with-pam`	PAM Unterstützung aktivieren
`--enable-syslog`	Syslog Unterstützung aktivieren (kann dann bei Bedarf konfiguriert werden)

Ein Aufruf könnte also wie folgt aussehen:

```
root@linux / # ./configure --enable-unicode-conversion
--enable-multibyte=UNICODE \ --with-CXX --with-perl --with-python
--with-tcl --with-java \ --enable-odbc --with-unixodbc \ --with-pam
--enable-syslog \ --enable-locale
```

Die Auswahl der Parameter ist kompliziert und hängt von vielen Faktoren ab, daher sollte auf Distributionspakete zurückgegriffen werden, soweit möglich.

2.1.3 Übersetzen und Installieren

Nach dem `configure` übersetzt man wie gewohnt mit:

```
root@linux / # make
```

Optional kann man Regressionstests durchführen:

```
root@linux / # make check
```

Die eigentliche Installation wird mit

```
root@linux / # make install
```

durchgeführt. Dies macht man in der Regel als **root**.

Je nach Configure-Optionen installiert *PostgreSQL* Bibliotheken beispielsweise in `/usr/local/pgsql/lib`. Dieser Pfad sollte dann in `/etc/ld.so.conf` eingetragen werden (das Ausführen von `ldconfig` ist danach notwendig). Gegebenenfalls fügt man den Pfad zu den Binärprogrammen zum Pfad hinzu, beispielsweise in dem man `/usr/local/pgsql/bin` zum `PATH` in `/etc/profile` hinzufügt.

2.2 Grundkonfiguration

Bevor *PostgreSQL* gestartet werden kann, müssen noch einige Einstellungen durchgeführt werden. Verwendet man Distributionspakete, so können diese Schritte in der Regel entfallen.

2.2.1 Datenbank Systembenutzer

Das DBMS benötigt einen Systembenutzer. Darüber hinaus können natürlich viele Datenbankbenutzer existieren. Beide Benutzerarten dürfen keinesfalls verwechselt werden. Der Systembenutzer ist der Benutzer, dem später die Datenbankdateien gehören. Diesem Benutzer ist wohl nie eine Person assoziiert (im Gegensatz zu den Datenbankbenutzern).

Ein gutes Beispiel für einen Systembenutzernamen ist **postgres**, wie er beispielsweise von *SuSE* und *RedHat* verwendet wird (**root** ist in keinem Fall geeignet). Den Benutzer kann man einfach erzeugen:

```
root@linux / # useradd postgres
```

und so sperren, dass man sich nicht einloggen kann:

```
root@linux / # passwd -l postgres
```

Nun kann nur **root** über das `su` Kommando auf den **postgres** Benutzer zugreifen.

2.2.2 Erzeugen einer initialen Datenbank

Um *PostgreSQL* starten zu können, muss eine Datenbankgrundstruktur vorhanden sein. Hierbei handelt es sich im Wesentlichen um eine komplizierte Verzeichnisstruktur, die mit dem Programm `initdb` erzeugt werden sollte. Dieses Programm sollte auf jeden Fall mit dem oben genannten Benutzer durchgeführt werden. Ist das Datenverzeichnis beispielsweise `/usr/local/pgsql/data`, bietet sich folgende Kommandokette (begonnen als **root**!) an:

```
root@linux / # mkdir /usr/local/pgsql/data
```

```
root@linux / # chown postgres /usr/local/pgsql/data
root@linux / # su - postgres
root@linux postgres/ # initdb --pwprompt -D /usr/local/pgsql/data
```

Dieses Kommando verwendet die gerade eingestellten `locale` als Sortierfolge in Indizes. Diese kann später nicht mehr einfach geändert werden. Ein Nachteil bei der Verwendung von `locale` ist, dass der `LIKE` Operator und reguläre Ausdrücke diese Indizes nicht verwenden können - und dadurch langsam werden. Möchte man lieber keine (oder andere) `locale` für diesen Fall, setzt man vor `initdb` die Variable `LC_COLLATE`. Beispielsweise kann man die locale auf den Standard `C` setzen:

```
root@linux postgres/ # LC_COLLATE="C" initdb --pwprompt -D
/usr/local/pgsql/data
```

Möchte man dies später ändern, so muss ein Komplettbackup gemacht werden, die Datenbank heruntergefahren, `initdb` erneut ausgeführt und nach dem Start muss das Backup wieder eingespielt werden. Dies ist ja nichts überraschendes; die `sort order` kann man eben nicht nachträglich einstellen, dass ist wohl bei allen DBMS so.

Durch `initdb` wird eine erste Datenbank erzeugt. Diese heißt `template1`. Wie der Name schon andeutet, wird diese als Vorlage beim Erzeugen neuer Datenbanken verwendet; daher sollte man mit dieser Datenbank nicht arbeiten (da dies zukünftige neue Datenbanken beeinflussen würde).

Im Beispielaufruf wird auch gleich ein Passwort für den Datenbankadministrator gesetzt. Setzt man kein Passwort, so gibt es kein gültiges (das heißt, man kann sich nicht als Administrator verbinden, wenn ein Passwort gefordert ist). Auf dieses Verhalten sollte man sich jedoch nicht verlassen, und das System lieber korrekt konfigurieren.

3 Administration

Der Abschnitt Administration wendet sich an Datenbank Administratoren und beschreibt Aufgaben wie Einrichtung und Backup. Benutzer, die eine von Anderen administrierte Datenbank verwenden, können diesen Abschnitt daher auslassen.

Dieses Kapitel setzt voraus, dass *PostgreSQL* bereits installiert ist. Hat man ein Distributionspaket verwendet, so ist vermutlich wenig bis gar nichts an Konfiguration notwendig, wenn man keine besonderen Einstellungen benötigt.

Es gibt zwei Hauptkonfigurationsdateien: `postgresql.conf` und `pg_hba.conf`. Die erstere ist die eigentliche Konfigurationsdatei, in der zweiten konfiguriert man Zugriffsbeschränkungen.

Etliche Aktionen kann man wahlweise über externe Programme oder über SQL-Kommandos durchführen, beispielsweise das Anlegen neuer Datenbankbenutzer.

3.1 Konfiguration

Dieser Abschnitt richtet sich an fortgeschrittene *PostgreSQL* Administratoren. Für kleinere Systeme (weniger als 100.000 Datensätze) sind die Voreinstellungen sicherlich ausreichend. In solchen Fällen diesen Abschnitt einfach auslassen.

Die hier genannten Optionen können auch über Kommandozeilenparameter gesetzt werden. Man sollte natürlich darauf achten, keine widersprüchlichen Optionswerte einzustellen. Einige Optionen kann man auf zur Laufzeit über das SQL Kommando `SET` einstellen.

Die Konfigurationsdatei heißt `postgresql.conf`. Hier können viele Optionen auf bestimmte Werte gesetzt werden. In jeder Zeile der Datei kann eine Option stehen, die das Format

```
option = wert
```

hat (genau genommen kann das `=` weggelassen werden). Zeilen, die mit `#` beginnen, sind Kommentare.

Zunächst gibt es eine Reihe von Optionen, die das Verhalten des Planers beeinflussen. Hier kann man die relativen Kosten für bestimmte Operationen einstellen. Diese Optionen enden mit `_cost`.

Mit den Optionen `debug_level`, `log_connections` und `log_timestamp` kann die Protokollierung beeinflusst werden. Soll diese durch `syslog` erfolgen, kann man dies mit den Optionen `syslog`, `syslog_facility` und `syslog_ident` einstellen.

Die Optionen `deadlock_timeout` und `default_transaction_isolation` beeinflussen das Transaktionslocking. Die Option `password_encryption` gibt an, ob Passwörter im Klartext oder verschlüsselt gespeichert werden sollen. Mit `fsync` kann gefordert werden, dass die Daten wirklich auf Festplatte geschrieben werden, wenn sie geändert wurden. Dies kostet zwar Performanz, sollte aber aus Sicherheitsgründen aktiviert werden, sonst kann es bei Abstürzen zu Problemen und Datenverlusten kommen.

postgresql.conf

```
#Beispieldatei postgresql.conf fuer SelfLinux [c] <steffen@dett.de>
#
#Diese Datei zeigt, wie man PostgreSQL fuer groessere Server
#  einstellen koennte.


#       Verbindungsoptionen

#Verbindungsparameter: TCP akzeptieren
tcpip_socket = true
# kein SSL verwenden
ssl = false

#Anzahl gleichzeitiger Verbindungen
max_connections = 64

#TCP Port
#port = 5432
#hostname_lookup = false
#show_source_port = false

#Parameter fuer Unix Domain Sockets (alternativ oder zusäztlich zu TCP)
#unix_socket_directory = ''
#unix_socket_group = ''
#unix_socket_permissions = 0777

#virtual_host = ''



#       Groesse des Shared Memories.
#
#2.2er Kernel erlauben erstmal nur 32 MB, jedoch kann das
#   ohne Reboot erhoeht werden, beispielsweise auf 128 MB:
#
#$ echo 134217728 >/proc/sys/kernel/shmall
#$ echo 134217728 >/proc/sys/kernel/shmmax

shared_buffers = 128           # 2*max_connections, min 16
max_fsm_relations = 500        # min 10, Voreinstellung 100, in pages
max_fsm_pages = 50000          # min 1000, Voreinstellung 10000, in pages
max_locks_per_transaction = 64 # min 10, Voreinstellung 64
wal_buffers = 8                # min 4

#Weitere Speichergroessen in KB:
sort_mem = 1024                # min 32, Voreinstellung 512
vacuum_mem = 8192              # min 1024, Voreinstellung 8192


#
#       Write-ahead log (WAL)
#
#wal_files = 0 # range 0-64
#wal_sync_method = fsync
#wal_debug = 0                 # range 0-16
#commit_delay = 0              # range 0-100000
#commit_siblings = 5           # range 1-1000
#checkpoint_segments = 3       # in logfile segments (16MB each), min 1
#checkpoint_timeout = 300      # in seconds, range 30-3600
fsync = true


#
#       Optimizer Optionen
#
#enable_seqscan = true
#enable_indexscan = true
#enable_tidscan = true
#enable_sort = true
```

```
#enable_nestloop = true
#enable_mergejoin = true
#enable_hashjoin = true

#Key Set Query Optimizer: viele AND, ORs duerfen
#  in UNIONs optimiert werden. Achtung, Resultat kann abweichen
#  (wegen DISTINCT).
#  Diese Option macht eventuell Sinn, wenn hauptsaechlich ueber
#  MS Access gearbeitet wird. Handoptimierung sollte natuerlich
#  immer vorgezogen werden!
ksqo = false

#effective_cache_size = 1000   # Voreinstellung in 8k pages
#random_page_cost = 4
#cpu_tuple_cost = 0.01
#cpu_index_tuple_cost = 0.001
#cpu_operator_cost = 0.0025


#
#       Genetic Query Optimizer Optionen
#
#geqo = true
#geqo_selection_bias = 2.0     # range 1.5-2.0
#geqo_threshold = 11
#geqo_pool_size = 0            # Voreinstellung basiert auf Anzahl der
                        # Tabellen der Abfrage; 128-1024
#geqo_effort = 1
#geqo_generations = 0
#geqo_random_seed = -1         # -1 --> auto


#
#       Logging und Debuganzeigen
#
#silent_mode = false

#log_connections = false
#log_timestamp = false
#log_pid = false

#debug_level = 0               # 0-16

#debug_print_query = false
#debug_print_parse = false
#debug_print_rewritten = false
#debug_print_plan = false
#debug_pretty_print = false

# requires USE_ASSERT_CHECKING
#debug_assertions = true


#
#       Syslog
#
#(nur, wenn entsprechend uebersetzt!)
#syslog = 0 # range 0-2
#syslog_facility = 'LOCAL0'
#syslog_ident = 'postgres'


#
#       Statistiken
#
#show_parser_stats = false
#show_planner_stats = false
#show_executor_stats = false
#show_query_stats = false

#show_btree_build_stats = false
```

```
#
#       Zugriffsstatistiken
#
#stats_start_collector = true
#stats_reset_on_server_start = true
#stats_command_string = false
#stats_row_level = false
#stats_block_level = false


#
#       Lock Behandlung
#
#trace_notify = false

#(nur, wenn mit LOCK_DEBUG uebersetzt)
#trace_locks = false
#trace_userlocks = false
#trace_lwlocks = false
#debug_deadlocks = false
#trace_lock_oidmin = 16384
#trace_lock_table = 0


#
#       Allgemeines
#
#dynamic_library_path = '$libdir'
#australian_timezones = false
#authentication_timeout = 60    # min 1, max 600
#deadlock_timeout = 1000
#default_transaction_isolation = 'read committed'
#max_expr_depth = 10000         # min 10
#max_files_per_process = 1000   # min 25
#password_encryption = false
#sql_inheritance = true
#transform_null_equals = false
```

Der Bedarf an Shared Memory wird durch die Kombination der Anzahl von *PostgreSQL* Instanzen (`max_connections`) und der geteilten Speicherpuffer (`shared_buffers`) bestimmt. Erhöht man diese Parameter, kann es sein, dass sich *PostgreSQL* beschwert, es sei zu wenig Shared Memory vorhanden.

Wie auch im Kommentar zu lesen, helfen folgende Kommandos, die maximale Größe des Shared Memory in Linux 2.2.x auf beispielsweise 128 MB zu erhöhen:

```
root@linux / # echo 134217728 >/proc/sys/kernel/shmall
root@linux / # echo 134217728 >/proc/sys/kernel/shmmax
```

Diese Kommandos muss man natürlich so ablegen, dass sie beim Systemstart vor dem Start von *PostgreSQL* ausgeführt werden.

Ist das Programm `sysctl` installiert, kann man alternativ folgendes in die Datei `/etc/sysctl.conf` eintragen:

/etc/sysctl.conf

```
kernel.shmall = 134217728
kernel.shmmax = 134217728
```

Durch Ausführen von

```
root@linux / # sysctl -p
```

werden dann die Einträge der Datei `/etc/sysctl.conf` übernommen.

3.2 Authentifizierung

Über die Datei `pg_hba.conf` (hba: host based access, hostbasierter Zugriff) kann eingestellt werden, von welchen Systemen aus welche Authentifizierung durchgeführt werden muss. So läßt sich beispielsweise einstellen, dass Verbindungen vom Webserver nur auf eine bestimmte Datenbank erfolgen dürfen.

Eine sehr schöne Funktion ist auch das **Usermapping**. Es kann eingestellt werden, dass bestimmte Benutzer von bestimmten Maschinen aus nur auf ein bestimmtes Benutzerkonto zugreifen können (zum Beispiel, `www-run` des Webservers bekommt den Benutzer `wwwro`, dieser darf dann nur lesen). Diese Funktion steht leider nur zur Verfügung, wenn `ident` verwendet wird, eine Authentifizierung, von der man leider abraten sollte, da sie nur Sinn macht, wenn man den Administratoren dieser Server vertraut.

Aus Performanzgründen wird diese Datei bei neueren *PostgreSQL* Versionen nur noch einmalig beim Start und nicht mehr bei jedem Verbindungsaufbau geladen.

Um *PostgreSQL* Änderungen an dieser Datei mitzuteilen, kann man als Benutzer **root** auch folgenden Befehl eingeben, statt das DBMS komplett neu zu starten:

```
root@linux / # su -l postgres -s /bin/sh -c "/usr/bin/pg_ctl reload -D
$PGDATA -s"
```

Hat man das Programmpaket der Distribution installiert, kann man auch einfach:

```
root@linux / # /etc/init.d/postgresql reload
```

eingeben.

Jede Zeile ist eine Regel. Eine Regel besteht aus mehreren Teilen, die durch Leerzeichen getrennt sind. Der erste Teil gibt dabei den Regeltyp an.

Der wichtigste Regeltyp `host` gilt für Netzwerkadressen. Er hat das Format:

```
host Datenbankname IP-Adresse Netzmaske Authentifizierung
```

Daneben gibt es beispielsweise noch den Typ `local`, der für Verbindungen über Unix-Domain-Sockets verwendet wird.

Auf dedizierten Datenbankservern, also Servern, die nur die Datenbank fahren und vor allem keine lokalen Benutzerkonten besitzen, verwendet man hier auch oft die Authentifizierung `trust`, also Anmeldung ohne Passwort, da es hier nur **root** gibt, und der darf eh alles. Plant man `cron jobs`, so ist hier `trust` angebracht, da `cron` natürlich keine Passwörter eingibt. Dies ist jedoch problematisch, wenn es Benutzerkonten auf dem System gibt.

Mögliche Werte für **Authentifizierung**:

`trust`

Keine Authentifizierung, der Benutzername wird akzeptiert (evtl. Passwort gilt als korrekt).

`password`

Klartext-Passwort Authentifizierung. Optional kann eine Passwortdatei angegeben werden.

`crypt`

Verhält sich wie password, über das Netzwerk werden jedoch die Passwörter verschlüsselt übertragen

`md5`

Neuere Versionen bieten MD5 Passwörter an. Diese Option benutzt einen anderen und besseren Algorithmus zur Verschlüsselung als crypt.

`ident`

Der Ident-Daemon wird gefragt. Es ist möglich, über eine Datei `pg_ident.conf` ein Benutzernamen-Mapping durchzuführen.

`reject`

Die Verbindung wird in jedem Fall abgelehnt.

Eine Beispielkonfiguration:

pg_hba.conf

```
# TYPE  DATENBANK    IP-ADRESSE         NETZMASKE          TYP
#

#Uber Unix-Domain-Sockets darf mit Klartextpasswort verbunden werden
#   Auf dedizierten Datenbankservern verwendet man hier auch oft
#   trust, siehe Text
local   all                                               password

#Von localhost darf mit Klartextpasswort verbunden werden
host    all          127.0.0.1          255.255.255.255    password

#192.168.1.3 ist ein Webserver und darf nur auf wwwdb
host    wwwdb        192.168.1.3        255.255.255.255    crypt

#192.168.1.1 ist ein Router und darf gar nichts
host    all          192.168.1.1        255.255.255.255    reject

#Der Admin sitzt auf 192.168.1.4
host    all          192.168.1.4        255.255.255.255    md5

#Die Infodatenbank ist für das ganze Netz erlaubt (außer 1.1, siehe oben)
host    info         192.168.1.0        255.255.255.0      crypt

#Die Auftragsabteilung 192.168.2.x fuettert die wwwdb
host    wwwdb        192.168.2.0        255.255.255.0      crypt
```

3.3 Starten und Stoppen

Nach der Grundkonfiguration kann man die Datenbank starten. Wie man das macht, hängt davon ab, ob man ein Distributionspaket verwendet, oder selbst kompiliert hat.

Verwendet man ein Distributionspaket, so kann die Datenbank vermutlich sofort gestartet werden oder läuft sogar bereits.

Das DBMS-Backend von *PostgreSQL* heißt `postmaster`. Dieses nimmt Verbindungsanfragen an, startet für jede Verbindung einen eigenen `postgres` Prozess, der die eigentliche Arbeit erledigt, und koordiniert die Kommunikation zwischen den einzelnen `postgres` Instanzen.

Hat man selbst kompiliert, so startet man beispielsweise mit

```
root@linux / # su -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
postgres
```

oder als Postgres-Systembenutzer mit:

```
root@linux / # postgres$ pg_ctl start -D /usr/local/pgsql/data -l
serverlog
```

In der Regel schreibt man sich ein Skript, dass beim Booten ausgeführt wird. Distributionen installieren in der Regel so ein Skript bereits. Dann startet man beispielsweise über

```
root@linux / # rcpostgres start # SuSE
```

oder

```
root@linux / # service postgresql start # RedHat
```

oder

```
root@linux / # /etc/init.d/postgres* start # generisch
```

das DBMS.

Details finden sich sicherlich im Handbuch.

Sollte dies der erste Start nach dem Update sein, ist dies vermutlich ein guter Zeitpunkt, um das Backup wieder einzuspielen:

```
root@linux / # psql -d template1 -f backup.sql
```

Das Herunterfahren der Datenbank erledigt man analog zum Starten:

```
root@linux / # su -c 'pg_ctl stop'
```

oder einem Distributionskommando wie zum Beispiel

```
root@linux / # rcpostgres stop
```

Man kann auch Signale verwenden. Das Signal `SIGKILL` sollte hier unter allem Umständen vermieden werden, da in diesem Fall die Datenbank nicht geschlossen wird - Datenverluste sind fast unvermeidlich.

Das Signal SIGTERM veranlasst *PostgreSQL*, so lange zu warten, bis alle Clients ihre Verbindungen beendet haben und ist somit die schonendste Methode. Das Signal `SIGINT` beendet alle Clientverbindungen, und fährt die Datenbank sofort sauber herunter. Letzlich kann man noch `SIGQUIT` verwenden, was die Datenbank sofort beendet, ohne sie sauber herunterzufahren. Dieses Signal sollte daher nicht verwendet werden. Mit einer automatischen Reparatur ist beim Starten anschließend zu rechnen.

Ein Beispielaufruf:

```
root@linux / # killall -INT postmaster
```

3.4 Mit der Datenbank arbeiten

An dieser Stelle wird nur der Vollständigkeit halber `psql` genannt. An späterer Stelle wird genauer darauf eingegangen.

`psql` ist das **Interaktive Terminal**, eine Art Shell für die Datenbank. Hier kann man SQL Kommandos absetzen. So kann man Datenbanken anlegen, füllen, benutzen und administrieren.

`psql` erfordert als Parameter den Namen der Datenbank, zu der verbunden werden soll. Es gibt meistens mindestens die Datenbank `template1`. Über Optionen kann man angeben, auf welchen Server die Datenbank läuft und welcher Benutzername verwendet werden soll. Möchte man beispielsweise als Administrator `postgres` zu der Datenbank `template1` auf `localhost` verbinden, kann man schreiben:

```
root@linux / # psql -h localhost -U postgres template1

Password:
Welcome to psql, the PostgreSQL interactive terminal.

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

template1=#
```

Unten sieht man das Prompt (das den Datenbanknamen beinhaltet). Hier kann man SQL Kommandos eingeben, beispielsweise:

```
template1=# SELECT version();
                            version
-------------------------------------------------------------
 PostgreSQL 7.0.2 on i686-pc-linux-gnu, compiled by gcc 2.95.2
(1 row)
```

Hier läuft also die Version 7.0.2 (welches mal ein Update vertragen könnte). `psql` kennt zusätzlich zu den SQL Kommandos sogenannte **interne** Kommandos. Diese beginnen mit einem `\` (backslash). Diese lassen sich mit `\?` auflisten. Mit `\h` kann man auf umfangreiche Hilfe zurückgreifen, eine SQL Referenz. Mit `\q` beendet man das Programm.

3.5 Datenbanken planen

Zunächst sollte man natürlich seine Datenbank planen. Vielleicht erstellt man ein paar **Entity-Relationship-Diagramme** (ER Modelle). Diese kann man dann in eine Normalform übertragen (die Tabellen normieren), bis man in etwa die **3. Normalform** erreicht hat.

Dann überlegt man sich Wertebereiche, Gültigkeiten und Abhängigkeiten. Aus den **Standard-Use-Cases** kann man oft recht einfach die erforderlichen Berechtigungen und zu erzeugenden Views ableiten.

Hat man das erledigt, kann man beginnen, die Datenbank zu erzeugen und die Tabellen anzulegen. Oft schreibt und testet man Konsistenzprüfungsfunktionen wie **Trigger** vor dem Anlegen der Tabellen. Auch die Dokumentation sollte man nicht vergessen. Im Internet findet man Hilfen zur Datenbankplanung (die Planung ist ja nicht *PostgreSQL* spezifisch).

Nun sollte man Testdaten erzeugen. Diese sollten vom Umfang her fünf- bis zehnmal mächtiger als die zu erwartenden Daten sein, wenn möglich. Nun testet man das System und optimiert gegebenenfalls. In einem frühen Stadium ist die Optimierung oft noch einfach - später wird es dann kompliziert, weil man oft **Kompatiblitätsmodus-Views** und ähnliche Workarounds benötigt, da die Anwendungen selten alle auf einen Schlag angepasst werden können.

Wenn man die Datenbank entwickelt und nicht ständig Skripte nachpflegen möchte, kann man nach dem Erzeugen des Schemas (also der Tabellen und was so dazugehört) mit dem Programm `pg_dump` das Schema in ein Skript schreiben, und dieses kommentieren:

```
root@linux / # pg_dump --schema-only database -f schema.sql
```

Das ist bei kleinen Projekten oder in frühen Stadien oft eine nützliche Hilfe.

3.6 Datenbanken erzeugen und löschen

Es gibt zwei Möglichkeiten, neue Datenbanken zu erzeugen. Man kann das Programm `createdb` verwenden. Dieses verwendet das SQL Kommando `CREATE DATEBASE`, um eine neue Datenbank zu erzeugen.

`createdb` versteht etliche Optionen, die sehr ähnlich zu denen von `psql` sind. Man kann auch `psql` verwenden, und dann mit dem SQL Kommando `CREATE DATEBASE` eine Datenbank erzeugen. Als einzigen geforderten Parameter gibt man den Namen der zu erzeugenden Datenbank an. Beispiel:

```
template1=# CREATE DATABASE test;
CREATE DATABASE
```

Datenbanken kann man mit `DROP DATABASE` löschen. Achtung, diese Kommandos sind eigentlich kein SQL Kommandos (kein Abfragekommando, also nicht **query language**), sondern ein sogenannte Strukturkommandos. Diese lassen sich nicht in Transaktionen ausführen und damit insbesondere nicht rückgängig machen!

Versucht man ein `DROP DATABASE test;` in einer Transaktion, so wird das Kommando ignoriert.

3.7 Benutzer und Gruppen

Analog zu Datenbanken kann man Benutzer über das Programm `createuser` oder über `psql` anlegen. Das SQL (Struktur-) Kommando heißt `CREATE USER`. Hier gibt es eine Vielzahl von Parametern; beispielsweise, ob der Benutzer Datenbanken anlegen darf (`CREATEDB`) oder nicht (`NOCREATEDB`), ob der Benutzer weitere Benutzer anlegen darf (`CREATEUSER`) oder nicht (`NOCREATEUSER`), welches Passwort er bekommt (`PASSWORD` **geheim**), in welchen Gruppen er ist (`IN GROUP` gruppe1, gruppe2, ...) und wie lange er gültig ist (`VALID UNTIL` **Zeitstempel**).

Dieses Kommando legt einen Benutzer **steffen** mit einem sehr schlechten Passwort an:

```
template1=# CREATE USER steffen WITH PASSWORD '123' NOCREATEDB
NOCREATEUSER;
CREATE USER
```

Auch hier gibt es ein `DROP USER`.

Änderungen werden über das Kommando `ALTER USER` durchgeführt:

```
template1=# ALTER USER steffen PASSWORD 'geheim';
ALTER USER
```

Gruppen werden mit `CREATE GROUP` erzeugt. Man kann die Benutzer aufzählen, die Mitglied werden sollen (`USER` benutzer1, benutzer2, ...). Es gibt auch `DROP GROUP`, um Gruppen zu löschen.

Alle hier genannten Kommandos sind Strukturkommandos und unterliegen nicht (ganz) den Transaktionsregeln. Ein Rollback auf ein `DROP USER` funktioniert nicht (vollständig).

Zum Hinzufügen bzw. Entfernen von Benutzern zu Gruppen stehen die Kommandos

```
template1=# ALTER GROUP gruppe1 ADD USER steffen, elvira
template1=# ALTER GROUP gruppe1 DROP USER elvira
```

zur Verfügung.

Für den Datenbank Superuser oder Administrator gelten Sonderregeln, die im Abschnitt ▶ Privilegien kurz erklärt werden.

3.8 Privilegien

Privilegien sind Zugriffsrechte. *PostgreSQL* unterstützt hier verschiedene Arten:

`SELECT` das Leserecht

INSERT	darf neue Datensätze einfügen
UPDATE	darf Datensätze ändern und Sequenzen verwenden
DELETE	darf Datensätze löschen
RULE	darf Regeln für Tabellen erzeugen (eine *PostgreSQL* Erweiterung)
REFERENCES	darf einen Schlüssel dieser Tabelle als Fremdschlüssel verwenden
TRIGGER	darf Trigger an der Tabelle erzeugen
CREATE	darf Objekte in Datenbank (ab 7.3 auch Schemata) anlegen
TEMPORARY	darf temporäre Tabelle in Datenbank anlegen
EXECUTE	darf Funktion ausführen
USAGE	darf Sprache (z.B. PL/pgSQL) oder Objekte in Schema (ab 7.3) benutzen
ALL	darf alles

In SQL werden Privilegien über GRANT erlaubt und mit REVOKE entzogen. Das Grant-Kommando ist *SQL92* konform. Um sicherzugehen, dass nicht bereits andere Rechte gesetzt sind, führt man vor einem GRANT manchmal auch ein REVOKE aus, um alle Rechte erstmal zu löschen.

Der Benutzer wwwro darf statistics nur lesen:

```
template1=# REVOKE ALL ON statistics FROM wwwro;
template1=# GRANT SELECT ON statistics TO wwwro;
```

Die Gruppe stats darf alles auf dieser Tabelle:

```
template1=# GRANT ALL ON statistics TO GROUP stats;
```

Der Eigentümer hat automatisch immer alle Bereichtigungen. Den Eigentümer kann man über beispielsweise mit:

```
template1=# ALTER TABLE statistics OWNER TO steffen;
```

einstellen.

Es gibt einen besonderen Benutzer, den Superuser oder Administrator. Dieser darf jegliche Aktion immer durchführen; die Privilegien werden nicht ausgewertet. Der Superuser darf auch über ein Kommando jede andere Identität einstellen, das ist bei Tests sehr sinnvoll. Dieses Kommando läßt sich in etwa mit dem Unix-Kommando su vergleichen. Das geschieht wie folgt:

```
template1=# SET SESSION AUTHORIZATION 'steffen';
```

Einige Aktionen können nur vom Superuser durchgeführt werden, beispielsweise das Installieren neuer Sprachen mit Systemberechtigungen.

3.9 Vacuum

Gelöschte bzw. geänderte Datensätze sind lediglich als solche markiert und noch auf der Festplatte vorhanden. Darum ist es erforderlich, regelmäßig den Speicher aufzuräumen. Dies sollte man machen, wenn die Datenbank gerade wenig zu tun hat, beispielsweise nachts. Hierzu dient das SQL Kommando `VACUUM`. Dieses gibt nicht mehr benutzten Speicher frei. Es gibt auch ein Programm `vacuumdb`, das man als `cronjob` einrichten kann.

Eine Option von `VACUUM` ist `ANALYZE`, die die Statistiktabellen für den Planer (Optimizer) aktualisiert.Da die Geschwindigkeit, mit der *PostgreSQL* agiert, entscheidend von diesen Daten abhängt, sollte man die Analyse häufiger, auf jeden Fall aber nach einer größeren Anzahl von `INSERTs` oder `UPDATEs` durchführen. Eine reine Analyse belastet den Rechner auch weniger und kann daher auch stündlich durchgeführt werden.
Hierzu kann z.B. folgendes Skript dienen, welches vom Benutzer postgres ausgeführt werden muss:

analyzedbs

```
#!/bin/bash
# analyzedbs (c) 2003 by SelfLinux.de
# analysiert PostgreSQL-Datenbanken ohne Vacuum
#

PSQL=/usr/bin/psql
dbs=`$PSQL -U postgres -q -t -A -d template1 \
        -c 'SELECT datname FROM pg_database WHERE datallowconn'`

for db in $dbs ; do
    $PSQL -q -c "SET autocommit TO 'on'; ANALYZE" -d $db
done
```

```
root@linux / # su -l postgres -c analyzedbs
```

Hat man keine besonderen Anforderungen, führt man die `ANALYZE` zusammen mit `VACUUM` aus. Ein Beispielaufruf:

```
root@linux / # vacuumdb --all --analyze --full --username=postgres
```

Seit Version 7.2 werden Tabellen während `VACUUM` nicht mehr komplett gesperrt, man kann dieses aber durch die Angabe von `--full` erzwingen und so eine bessere Kompression des Datenbestandes erreichen.

Verwendet man einen `cron-Job`, so sollte in `pg_hba.conf` für Typ **local** die Authentifizierung **trust** verwenden, da `cron` keine Passwörter eingibt.

Man kann beispielsweise in die Datei `/etc/cron.d/postgresql` eintragen:

/etc/cron.d/postgresql

```
0 0  * * *     postgres vacuumdb --all --full --analyze
5 *  * * *     postgres /usr/local/bin/analyzedbs
```

Hier muss man beachten, dass `cron` nicht die `/etc/profile` auswertet, und damit `vacuumdb` nicht

unbedingt im Pfad liegt. Hier sollte man lieber absolute Pfade angeben.

3.10 Sprachen installieren

Ähnlich zu Datenbanken und Benutzern kann man Sprachen, genauer gesagt, prozedurale Sprachen, in die Datenbank installieren. Etliche Sprachen sind Lieferumfang von *PostgreSQL*. Diese Sprachen liegen als Systembibliotheken im `lib` Verzeichnis, also beispielsweise `/usr/local/lib`.

Hier gibt es ein Programm `createlang`. Dieses verwendet das SQL Kommando `CREATE LANGUAGE`, um die Sprache zu installieren, führt jedoch zusätzlich etliche Prüfungen durch, und wird daher empfohlen.

Um Beispielsweise die Sprache *PL/pgSQL* auf einem SuSE 7.0 System zu installieren, genügt folgendes Kommando:

```
root@linux / # createlang --username=postgres --pglib=/usr/lib/pgsql/
plpgsql
```

Der Pfad `/usr/lib/pgsql/` muss angepasst werden. Neben *PL/pgSQL* sind auch noch *PL/TCL* (*TCL* für *PostgreSQL*) und *PL/Perl* (Perl-Sprache) sehr beliebt und mit *PostgreSQL* verfügbar.

Es gibt zwei Möglichkeiten, Sprachen zu installieren: `trusted` und `untrusted`. Da die wörtlichen Übersetzungen nicht weiterhelfen, folgt eine Erklärung. Eine `untrusted` Sprache darf mehr, als eine `trusted` Sprache. Von einer `trusted` Sprache wird erwartet, dass über diese keine normalerweise verbotenen Aktionen durchgeführt werden können. *PL/pgSQL* ist ein Beispiel.

PL/Perl kann auch im `untrusted` Modus installiert werden (wird dann oft *plperlu* genannt). Dann kann der komplette Sprachumfang von Perl verwendet werden. So kann z.B. eine Perlfunktion erstellt werden, die eine Mail verschickt. Dies gibt dem Benutzer damit automatisch die Berechtigungen des Unix-*PostgreSQL* Benutzers postgres. Daher können `untrusted` Sprachen nur vom Datenbank Superuser installiert werden. Die Funktionen in solchen Spachen müssen selbst für Sicherheit sorgen.

Die *PostgreSQL* Sprachen (wie *PL/Perl*) haben natürlich Einschränkungen zu den normalen Versionen (wie Perl). Dies sind zum einen Sicherheitseinschränkungen von trusted Modus Sprachen, und zum anderen Dinge, die aus technischen Gründen nicht gehen (in *PL/Perl* kann man beispielsweise **noch** nicht andere *PL/Perl* Funktionen aufrufen).

3.11 Backup

Es gibt mehrere Arten, Backups anzufertigen. Es gibt die Programme `pg_dump` und `pg_dumpall`. Das erste schreibt eine Datenbank in eine Datei, das zweite sichert alle Datenbanken. Beide kennen eine Vielzahl von Parametern. So kann man sich beispielsweise eine Folge von `INSERT` Kommandos erzeugen lassen, was hilfreich ist, wenn man dieses Backup auch in anderen Datenbanken verwenden möchte, die nicht *PostgreSQL* basiert sind (oder wenn man zu anderen DBMS wechseln möchte/muss).

3.11.1 pg_dump

Es wird als Parameter der Datenbankname erwartet. Die Ausgabe (den `Dump`) schickt man in eine Datei. Man kann über Optionen einstellen, wie der Dump aussehen soll, ob nur bestimmte Tabellen ausgelesen werden sollen oder alle und vieles mehr.

Wichtige Optionen sind:

`--file`=datei, `-f` datei	Ausgabe in Datei
`--inserts`	`INSERT` im Dump verwenden
`--attribute-inserts`	`INSERT` mit Attributen verwenden
`--host` servername	Zu diesem Server verbinden
`--quotes`	Viele Bezeichner quotieren
`--schema-only`	Nur die Struktur, nicht die Daten
`--table` tabelle	Nur diese Tabelle tabelle
`--no-acl`	Berechtigungen auslassen

Möchte man die Datenbank wwwdb sichern, so schreibt man:

```
root@linux / # pg_dump wwwdb -f backup.sql
```

Leider werden so keine **large objects** (große Objecte, ein Datentyp) gesichert. Hier eröffnet ein Blick in die *PostgreSQL* Dokumentation mehrere Lösungsmöglichkeiten, die den Rahmen an dieser Stelle sprengen.

3.11.2 pg_dumpall

Dieses Programm ruft `pg_dump` für alle Datenbanken auf, und wird daher meistens für Backups verwendet. Es ist die empfohlene Art. Ein Backup kann man beispielsweise mit folgendem Kommando durchführen:

```
root@linux / # pg_dumpall -f backup.sql
```

Erfreulicherweise ist auf Grund des später erklärten `MVCC` die Datenbank während des Backups vollständig verwendbar (möglicherweise gibt es einige spezielle Einschränkungen, ein `DROP DATABASE` zum Löschen einer Datenbank wird wohl nicht funktionieren).

3.11.3 Automatisiert

Den Aufruf von `pg_dumpall` zu automatisieren, fällt nicht schwer. Wenn man noch den Wochentag in den Dateinamen aufnimmt, wird das Backup nur wöchentlich überschrieben. Falls man sich mal vertippt hat, kann es sehr hilfreich sein, etwas ältere Backups zu haben. Ein ganz einfaches Skript, dass man täglich über `cron` als Unix-Benutzer postgres starten kann:

backup.sh

```
#!/usr/bin/bash
#muss als postgres gestartet werden

#Sicherheitshalber standard locale
export LC_ALL=C

#Wohin mit den Backups
cd /home/postgres/db_backups/

#Ergibt "Sun", "Mon" usw.
DAY=`date +%a`

#Alternativ: JJJJ-MM-TT
#DAY=`+%Y-%m-%d`

#Man kann alte Backups automatisch löschen, um Platz zu sparen,
#   beispielsweise alles löschen, was älter als 14 Tag ist:
#find /home/postgres/db_backups/ \
#       -iname 'dump_all-*.sql' -mtime +14 \
#       | xargs --no-run-if-empty rm -f

#Die Pfade müssen natürlich angepasst werden.
/usr/bin/pg_dumpall > dump_all-$DAY.sql

#Damit man nicht noch einen extra cron job für Vacuum machen muss:
/usr/bin/vacuumdb --all --analyze
```

3.11.4 Wiederherstellung

Eine Datenbank aus einem mit `pg_dumpall` erstellten File wiederherzustellen, ist sehr einfach. Man muss dafür sorgen, dass die Datenbank **template1** vorhanden ist und das DBMS läuft. Dann übergibt man die Backupdatei einfach dem `psql` Interpreter als SQL-Programm:

```
root@linux / # psql -d template1 -f backup.sql
```

In seltenen Fällen kann man die Backupdatei auch mit einem Editor öffnen, und nur Teile daraus in `psql` eingeben (um Teile wiederherzustellen, beispielsweise). Auch kann man sich so Skripte erstellen, die beispielsweise neue Datenbanken anlegen (wenn man die Testdatenbank für ein Frontend erzeugt hat).

Es gibt noch ein weiteres Programm, `pg_restore`, welche speziell für diesen Zweck entwickelt wurde. Dieses verfügt über einige Zusatzfunktionen, beispielsweise können so nur Teile wiederhergestellt werden. Man kann so einzelne Tabellen oder Funktionen wiederherstellen lassen. Ein Beispielaufruf:

```
root@linux / # $ pg_restore -d template1 backup.sql
```

3.11.5 Dateisystem

Natürlich kann man auch das Verzeichnis sichern, in dem *PostgreSQL* seine Daten aufbewahrt. Dazu muss die Datenbank unbedingt sauber heruntergefahren werden. Dann kann man das Verzeichnis einfach mit tar oder ähnlichem sichern. Verwendet man LVM, kann man die Datenbank stoppen, einen Snapshoot ziehen und die Datenbank wieder starten. Man sichert dann den Snapshoot, und die Datenbank muss nur kurz heruntergefahren werden.

Dies hat den großen Nachteil, dass man unbedingt absolut genau die gleiche Version benötigt, um mit dem Backup etwas anfangen zu können. Diese ist insbesondere bei alten Bändern schwierig (welche Version hatte man damals eigentlich?). Ein weiterer Nachteil ist, dass man nicht nur einzelne Tabellen bzw. Datenbanken rücksichern kann (es geht wirklich nicht, da die commit logs auch benötigt werden!) oder anderes.

Vor dem Wiederherstellen muss die Datenbank natürlich ebenfalls heruntergefahren werden.

Ein Backup über pg_dumpall ist in den meisten Fällen günstiger und sollte vorgezogen werden. Selbst wenn man über das Dateisystem sichert, sollte hin- und wieder ein `Dump` gezogen werden.

3.11.6 Grenzen

Leider ist das Backup mit `pg_dump` nicht perfekt. `pg_dump` wertet nicht aus, ob Tabellen Funktionen benutzen. Es geht davon aus, das Funktionen Tabellen verwenden (man beachte die Reihenfolge!).

Daher kommt es zu Problemen, wenn man Funktionen als Voreinstellung von Tabellenspalten verwendet. In solchen Fällen kann man die Funktion einfach per Hand anlegen (die Backupdatei mit einem Editor öffnen, Funktion übertragen, oder mit `pg_restore` diese Funktion zuerst wiederherstellen) und dann das Backup einspielen. Das gibt zwar eine Warnung, da die Funktion schon existiert, funktioniert aber.

Hat man zirkuläre Abhängigkeiten, so wird es etwas komplizierter, hier hilft nur Handarbeit. Solche Situationen sind meistens jedoch Fehler und unerwünscht.

`pg_dump` sichert auch keine **large objects** (große Objekte, ein Datentyp), wenn keine besonderen Optionen verwendet werden. Hier eröffnet ein Blick in die *PostgreSQL* Dokumentation mehrere Lösungsmöglichkeiten, die den Rahmen an dieser Stelle sprengen.

3.11.7 Update von älteren Versionen

Vor einem Update sollte mit pg_dumpall ein Backup erstellt werden. Diese kann man dann in die neue Version wiederherstellen. Eine Konvertierung der Daten-Dateien ist leider nicht vorgesehen.

Man kann auch die alte und neue Datenbankversion parallel laufen lassen, und dann die Daten einfach über das Netzwerk kopieren. Die neue Datenbank muss dazu natürlich ein eigenes Datenverzeichnis verwenden.

Angenommen, man startet die neue Datenbank auf Port 5433. Dann kann man mit folgender Kette den gesamten Datenbestand kopieren:

```
root@linux / # pg_dumpall -p 5432 | psql -p 5433
```

3.12 Transaktionsprotokolle

Ein Transaktionsprotokoll darf keinesfalls mit Protokolldateien mit Textmeldungen verwechselt werden. In einem Transaktionsprotokoll stehen Änderungen von Daten. Wird eine Transaktion **committed**, also erfolgreich beendet, so werden diese in ein Protokoll eingetragen und erst bei Gelegenheit in die `normalen` Dateien gespeichert. Das ist ein performantes Vorgehen, was auch bei Abstürzen funktioniert: in solchen Fällen wird das Log durchgearbeitet, und die noch nicht überspielten Änderungen werden durchgeführt (siehe auch Abschnitt Datenbankreparatur).
Write ahead logging (WAL) ist ein - wenn nicht das - Standardverfahren für Transaktionsprotokolle und wird von *PostgreSQL* verwendet.

3.13 Datenbankreparatur

Stellt *PostgreSQL* (genauer gesagt, das `postmaster` Programm) beim Start fest, dass die Datenbank nicht sauber heruntergefahren wurde, wird automatisch eine Reparatur begonnen. Hier wird im Wesentlichen das WAL (write ahead log) durchgearbeitet. Dieses Verhalten ist ähnlich dem Journal, über das moderne Filesysteme wird ext3 und Reiser-FS verfügen (diese Technik kommt aus dem Datenbankbereich, aber durch Diskussionen ist die Funktion bei Dateisystemen inzwischen scheinbar fast bekannter). *PostgreSQL* hat also kein separates Standard-Reparatur-Programm, sondern erledigt diese Aufgaben automatisch beim Start.

Unter ganz seltenen Umständen kann es jedoch sein, dass dieser Mechanismus nicht funktioniert. Diese können entstehen, wenn Arbeitsspeicher defekt ist (und einzelne Bits **umkippen**), ein Sicherungsband geknittert wurde, und so kleine Teile fehlen oder fehlerhaft sind und möglicherweise auch durch ganz ungünstige Stromausfälle. Dann kann es vorkommen, dass die automatische Reparatur abbricht, und die Datenbank gar nicht startet.

Selbst in solchen Fällen kann man oft noch viel retten, jedoch muss man dazu unangenehme Sachen machen, beispielsweise das WAL zurücksetzen. Hat man ein solches Problem, sucht man am besten in Mailinglisten Hilfe, denn hier muss man sehr vorsichtig sein, um nicht noch mehr zu zerstören.

3.14 Migration

Migriert man von anderen DBMS, so erstellt man sich in der Regel eine SQL Kommandodatei mit einem Backup, und bearbeitet diese per Hand oder mit Skripten so, dass sie von den anderen DBMS gelesen werden kann.

Portiert man ein System von anderen Datenbanken auf *PostgreSQL*, so ist je nach Art und Komplexität des Systems etliches an Arbeit zu erwarten.

Grundsätzlich kann man davon ausgehen, Daten relativ unproblematisch übernehmen zu können. Tabellen sind oft auch gut handhabbar. Dann wird es aber leider schnell schwierig. Stored Procedures beziehungsweise Datenbankfunktionen müssen in der Regel neu geschrieben werden. Erschwerend kommt hinzu, dass *PostgreSQL* keine Stored Procedures, sondern nur Funktionen kennt, die jedoch die Flexiblität von ersteren haben. Zwar ist `CREATE FUNCTION` Teil von *SQL99*, allerdings sind die Sprachen, in denen die Funktionen geschrieben sind, nicht standardisiert.

Systeme, die viel in der Datenbank machen, sind natürlich aufwendiger in der Portierung. Da man die Konsistenz grundsätzlich in der Datenbank regeln sollte, muss damit gerechnet werden, dass alle `Trigger`, Regeln und Stored Procedures neu implementiert werden müssen.

Der Aufwand für die Anwendungen selbst hängt maßgeblich davon ab, wie nahe diese dem Standard sind. Selbst wenn diese Anwendungen gut standardkonform sind, kann natürlich immer noch nicht mit **Plug'n'Play** gerechnet werden. Bei Anwendungen, die von Fremdfirmen geschrieben wurden, sollte nach Möglichkeit unbedingt Unterstützung durch diese Firmen verfügbar sein.

Das Migrations-Projektteam sollte über Testsysteme mit beiden Datenbanken verfügen und Spezialisten für alle beteiligten Systeme besitzen.

3.14.1 Umstieg von mySQL

mySQL ist eine sehr verbreitete OpenSource Datenbank. Da zu vermuten ist, dass viele bereits mit *mySQL* Erfahrungen haben, ist diesen Umsteigern hier ein eigenes Kapitel gewidmet.

In den sogenannten **Tech Docs** von *PostgreSQL* finden sich Informationen, wie man von *mySQL* zu *PostgreSQL*

migriert. Es gibt Skripte, die *mySQL* SQL-Kommandodateien zu weiten Teilen automatisch so umwandeln, dass sie von `psql` gelesen werden können.

In der Praxis sind allerdings einige Änderungen zu erwarten. So kann es beispielsweise sein, das man Probleme mit der Quotierung bekommt (*mySQL* verwendet beispielsweise **backticks**, um Systembezeichner zu quoten, was bei anderen Datenbanken zu Syntaxfehlern führt). Des weiteren ist *PostgreSQL* bei Zeichenketten `case-sensitiv`, dass heißt, die Groß/Kleinschreibung wird grundsätzlich unterschieden. Bei Tabellen und Spaltennamen ist *PostgreSQL* nicht `case-sensitiv`, es sei denn, man erzwingt dies durch die Verwendung von doppelten Anführungszeichen. Bei *mySQL* hängt dies von der verwendeten Plattform ab. Der Operator || wird in *PostgreSQL* so verwendet, wie in ANSI (*mySQL* kennt hier einen ANSI-Modus, der jedoch vermutlich selten verwendet wird). Man muss daher die `||` in `OR` und die `&&` in `AND` ändern; "`||`" ist der Konkatenierungsoperator (wie in *mySQL's* ANSI-Modus).

Der Umfang von SQL ist bei *mySQL* kleiner, dafür gibt es etliche, nicht standardkonforme Erweiterungen. *mySQL* verwendet # als Kommentarzeichen. ANSI schreibt `--` vor.

Steigt man auf *PostgreSQL* um, sollte man daran denken, die nun zur Verfügung stehenden Funktionen auch sinnvoll zu nutzen, beispielsweise `Trigger` und `Views`. Die Arbeit mit Transaktionen kann verbessert werden, da jetzt die ISO Transaktionslevel `read commited` und `serializable` zur Verfügung stehen.

Ein entscheidendes Detail ist die Verwendung von Fremdschlüsseln. *mySQL* unterstützt diese zwar syntaktisch, jedoch ohne Funktion. Daher ist zu erwarten, dass Daten nicht einfach übernommen werden können, da vermutlich viele Fremdschlüsselintegritäten verletzt sind.

Betrachtet man Vergleiche zwischen den beiden DBMS, so muss man diese sehr vorsichtig bewerten. So gibt es beispielsweise Seiten, die die nicht standardkonforme Verwendung des `||` Operators als Vorteil preisen, oder die Möglichkeit von stored procedures in *mySQL* nennen (die man dann in `C` schreiben muss, und als **root** zur Datenbank dazu linken muss). Eine andere Seite suggerierte es fast als Vorteil, keine Fremdschlüsselbedingungen zu prüfen.

3.14.2 Umstieg von anderen Systemen

Je nach Standard-Konformität zu SQL ist es mehr oder weniger aufwendig, das DBMS zu wechseln. Natürlich spielt auch eine große Rolle, wie viele Spezialfunktionen man verwendet, und wie anspruchsvoll die Anwendungen sind.

Es gibt in den **Tech Docs** von *PostgreSQL* Informationen hierzu. Hier findet man Hilfen für die Migration von MS-SQL Server, Oracle und anderen zu *PostgreSQL*.

Oracle-Erfahrene werden sich freuen, mit *PL/pgSQL* eine zu *PL/SQL* ähnliche Sprache zu finden.

3.15 Hardware

Der Vollständigkeit halber ein paar Worte zur Hardware. *PostgreSQL* fühlt sich auf handelsüblichen PCs mit i386 Architektur wohl. Eine kleinere, gut geplante Datenbank mit weniger als einer Millionen Datensätzen läuft auf einem PC mit vielleicht 1Ghz, 256 MB RAM und normalen Platten wohl zügig genug.

Je nach zu erwartender Last, Größe und Effizienz steigt der Hardwarebedarf schnell an. Abschätzungen lassen sich hier nur schwer treffen, zu groß ist beispielsweise der Unterschied, ob Indizes effizient arbeiten, oder nicht. Bei Datenbanken spielt oft die Geschwindigkeit der Festplatten eine große Rolle. SCSI Platten haben oft eine geringere Zugriffszeit und unterstützen Tagged Command Queuing - gerade Datenbanken profitieren von diesen Eigenschaften.

Ist man der Meinung, die Festplatten sind zu langsam, so kann man den Einsatz von RAID, beispielsweise RAID0+1, erwägen. Je nach Konfiguration kann man gleichzeitig auch eine erhöhte Ausfallsicherheit erreichen. Deshalb ist RAID0+1 beliebt: Man stript über einen Mirror (das ist etwas ausfallsicherer, als über Stripes zu spiegeln, da in letzterem Fall der zweite Plattenausfall weniger wahrscheinlich tötlich ist. Aufmalen!). Ein RAID0+1 mit insgesamt vier Platten erreicht (in der Theorie) die doppelte Schreib- und sogar die dreifache (der Faktor drei ist hier ein Praxiswert) Lesegeschwindigkeit, bietet in jedem Fall Schutz vor einem Plattenausfall und ermöglicht es, 50% der Plattenkapazität zu nutzen - oft ein guter Kompromiss. In solchen Konfigurationen sind SCSI RAID Controller sinnvoll, jedoch stoßen die preiswerteren Controller schnell an Performanzgrenzen (dann bremst der Controller die Platten aus). Hier sollte man sich vor dem Kauf informieren.

Je nach Art der Daten kann auch eine Verdopplung des Hauptspeichers viel Performanz bringen. Hier muss man die im Abschnitt Konfiguration beschriebenen Änderungen durchführen und etwas mit den Werten spielen, bis man günstige Kombinationen gefunden hat. Hat man viel Speicher, so kann es sogar Sinn machen, *PostgreSQL* mehr als 50% zu geben (auf dedizierten Systemen natürlich).

Rechenleistung ist bei vielen Anwendungen weniger ein Thema. Das Unix-Programm **top** hilft einem bei der Analyse. Sollte sich herausstellen, dass man eine sehr rechenintensive Datenbank hat, oder hat man einfach genügend Hauptspeicher, um die Plattenaktivität in den Griff zu bekommen, hilft vielleicht eine weitere CPU.

4 Benutzung

Dieser Abschnitt ersetzt keine SQL Referenz und kein *PostgreSQL* Handbuch. Es wird nur exemplarisch auf einige Details eingegangen. Dabei stehen *PostgreSQL*-spezifische Eigenschaften im Vordergrund.

4.1 psql

Der bereits kurz erwähnt interaktive Kommandointerpreter ist sicherlich das wichtigste Programm.

4.1.1 Kommandozeilenoptionen

psql versteht etliche Optionen:

`-d` Datenbank	Zu dieser Datenbank verbinden
`-h` Servername	Über TCP/IP zu diesem Server verbinden
`-p` Port	Diesen Port verwenden (Voreinstellung 5432)
`-U` Benutzer	Als Benutzer anmelden
`-c` Kommando	Dieses Kommando ausführen
`-f` Datei	Diese SQL Datei ausführen
`-o` Datei	Ausgaben die Datei schreiben
`-s`	Einzelschrittmodus: jedes SQL Kommando bestätigen
`-E`	zeigt das ausgeführte SQL-Kommando bei internen Befehlen (z.B. \d) an.

Nach den Optionen gibt man eine Datenbank an, sofern man nicht `-d` verwendet. Dahinter kann man noch einen Benutzernamen schreiben, sofern man nicht `-U` verwendet.

Um als Superuser postgres zur Datenbank test zu verbinden, schreibt man also beispielsweise:

```
root@linux / # psql -U postgres -d test
```

Je nach Einstellung der Authentifizierung wird nun nach einen Passwort gefragt. Es erscheint das Datenbankprompt.

Hat man *PostgreSQL* mit der readline-Unterstützung übersetzt, kann man ebenso wie in der Bash die Tabulator-Taste drücken, um Befehle und Objekte zu erweitern

4.1.2 Interaktion

Am Prompt kann man SQL Befehle eingeben:

```
test=# CREATE TEMPORARY TABLE temp
test-# ( feld1 int UNIQUE NOT NULL,
test(# feld2 varchar(100000) DEFAULT NULL );
NOTICE:  CREATE TABLE / UNIQUE will create implicit index
'temp_feld1_key' for table 'temp'
CREATE
```

Man sieht, das SQL Kommandos mit Semikolon abgeschlossen werden und dann automatisch ausgeführt werden. Das Prompt zeigt an, ob man in einer Klammer ist, eine kleine Hilfe. Das Beispielkommando hat nun eine einfach Testtabelle erzeugt. Diese kann man nun mit Daten füllen:

```
test=# INSERT INTO TEMP (feld1, feld2) VALUES (1234, 'hallo');
INSERT 1532564 1
```

Die Ausgabe enthält eine merkwürdige Nummer. Das ist der `OID`, der object identifier. Diese sollte man nicht weiter beachten (es handelt sich um eine Art automatisches Indexfeld, ist aber höchst unportabel, und wird nur intern benötigt).

Über `psql` kann man auch in Transaktionen arbeiten:

Die Tabelle enthält einen Datensatz:

```
test=# SELECT count(*) FROM temp;
 count
-------
     1
(1 row)
```

Transkation beginnen:

```
test=# BEGIN;
BEGIN
```

Tabelle temp leermachen (alles löschen):

```
test=# DELETE FROM temp;
DELETE 1
```

Die Tabelle ist jetzt auch Sicht der Transaktion leer:

```
test=# SELECT count(*) FROM temp;
 count
-------
     0
(1 row)
```

Transaktion abbrechen:

```
test=# ROLLBACK;
ROLLBACK
```

Es ist nichts geändert worden:

```
test=# SELECT * FROM temp;
 feld1 | feld2
------+-------
 1234  | hallo
(1 row)
```

Die Temporäre Tabelle verfällt automatisch, wenn man die Verbindung schließt.

4.1.3 Interne Kommandos

`psql` verfügt über eine Reihe sogenannter interner Kommandos. Diese beginnen mit einem `\` (Backslash). Einige der wichtigesten internen Kommandos sind:

`\?`	kurze Hilfe zu allen Backslash Kommandos
`\d` Objekt	Objekt beschreiben. Ist Objekt beispielsweise eine Tabelle, so werden die Spalten und Typen angezeigt. Auch definierte Indizes werden aufgelistet. Wird Objekt nicht angegeben, werden alle Tabellen aufgelistet, die existieren (außer natürlich temporäre Tabellen).
`\d`Kürzel	Liste die zu Kürzel passenden Objekte: Tabellen (`t`), Indizes (`i`), Sequenzen (`s`), Views (`v`), Privilegien (`p`), Systemtabellen (`S`), große Objekte (`l`), Aggregatfunktionen (`a`), Kommentare (`d`; Objektname muss folgen), Funktionen (`f`), Operatoren (`o`) und Datentypen (`T`). Durch ein Leerzeichen kann man noch ein Objekt angeben. `\dp temp` zeigt beispielsweise die Privilegien für die Tabelle temp an (was nur funktioniert, wenn es keine temporäre Tabelle ist).
`\e` Datei	Öffnet das letzte Kommando oder Datei im Editor. Hilfreich, um lange Kommandos wie `CREATE TABLE` zu bearbeiten und zu speichern.
`\l`	Listet alle Datenbanken auf.
`\q`	Beendet `psql`
`\x`	Erweiterte Ausgabe
`\H`	HTML Ausgabe
\c Datenbank \c - Benutzer	Verbindet zu einer neuen Datenbank oder zur aktuellen mit einem neuen Benutzer. Dies ist in etwa mit dem **USE** vergleichbar, das andere DBMS verwenden.

Es folgt ein Beispiel für das Ausgabeformat. Zunächst soll die Ausgabe der oben erwähnten Testtabelle nicht feld1 und feld2 beinhalten, sondern Nummer und Textfeld. Wenn man diese Bezeichner **case-sensitiv** haben möchte (Tabellen- und Feldnamen sind sonst case-insensitiv, das heißt, Groß-/Kleinschreibung wird nicht beachtet), muss man diese quoten:

```
test=# SELECT feld1 AS "Nummer", feld2 AS "Textfeld" FROM temp;
```

```
 Nummer | Textfeld
--------+----------
   1234 | hallo
(1 row)
```

Nach `\x` sieht die Ausgabe so aus:

```
test=# SELECT feld1 AS "Nummer", feld2 AS "Textfeld" FROM temp;
-[ RECORD 1 ]---
Nummer   | 1234
Textfeld | hallo
```

Dies macht bei großen Tabellen Sinn, wenn nicht mehr alle Spalten nebeneinander auf den Bildschirm passen.

4.1.4 Verwendung

Neben der interaktiven Verwendung kann man `psql` dazu verwenden, SQL Skripte auszuführen, beispielsweise Skripte, die Datenbankschemata erzeugen. Man kann `psql` sogar dazu verwenden, Shell-Skripte mit rudimentärer Datenbankfunktionalität zu versehen; hier ist die Verwendung von `Perl::DBI` oder anderen Methoden jedoch oft einfacher und sauberer.

4.2 pgaccess

`pgaccess` ist eine graphisches Frontend, mit dem etliche Standardaufgaben erledigt werden können. Das Anlegen von Tabellen beispielsweise macht sich mit diesem Frontend wesentlich besser, also mit `psql` .

Über das Menü kann man zu einer Datenbank verbinden. Im folgenden Dialog können Server- und Datenbankname sowie ein Benutzerkonto angegeben werden.

Im Hauptfenster kann man rechts die anzuzeigende Objekte wählen. Hier kann man beispielsweise zwischen Tabellen, Views und Sequenzen auswählen. Im linken Teil werden dann die entsprechenden Objekte aufgelistet und können ausgewählt werden.

Nach einem Doppelklick auf eine Tabelle bekommt man ein Fenster, in dem der Inhalt dargestellt wird und geändert werden kann. Klickt man eine Tabelle nur einmal an, so kann man weitere Funktionen anwenden, beispielsweise **Design**. Hier öffnet sich ein Fenster, in dem man komfortabel Indizes hinzufügen kann oder neue Felder anhängen kann.

Seit Version 7.3 ist pgaccess nicht mehr Bestandteil der *PostgreSQL*- Distribution und muss separat von http://www.pgaccess.org besorgt werden.

4.3 RedHat Database Admin

RedHat vertreibt eine eigene Version von *PostgreSQL*. Diese entspricht ungefähr der Version 7.2.3 und ist unter http://www.redhat.com/software/database/ erhältlich. *RedHat* stellt alle Änderungen am DBMS und auch sein graphisches Administrationsfrontend unter die GPL. Dieses läuft auch mit einer **konventionellen** *PostgreSQL* Installation und ist unter http://sources.redhat.com/rhdb/ zu finden. Es ist **hübscher** als `pgaccess` und bietet im Bereich der Verwaltung mehr Optionen als dieses, kann dafür aber nicht zur

Definition von TCL-Formularen herangezogen werden.

Die neuen Funktionen der Version 7.3 (Schemata) werden allerdings noch nicht unterstützt.

4.4 phpPgAdmin

Dies ist ein Webfrontend und setzt einen Webbrowser voraus. Dieses Frontend verfügt über sehr viele nützliche Funktionen. Tabellendaten können als HTML Tabelle betrachtet und editiert werden, beliebige Abfragen können erstellt und ausgeführt werden.

Tabellen selbst können einfach und komfortabel bearbeitet werden, so können neue Felder hinzugefügt oder gelöscht werden. Weiterhin stehen Kopier- und Dumpfunktionen bereit. Auch Berechtigungen können komfortabel verwaltet werden. Die zur Verfügung stehenden Optionen sind sinnvoll in Auswahlfeldern aufgelistet. Bei Bedarf ist es auch möglich, eigene SQL Kommandos einzugeben und ausführen zu lassen.

Eine weitere schöne Funktion ist die Verlinkung zu jeweils passenden Seiten der *PostgreSQL* Dokumentation.

Wer Webfrontends mag, wird dieses Frontend wohl lieben. Es lohnt sich allemal, sich dieses zu installieren. Natürlich muss unbedingt darauf geachtet werden, den Zugang zu diesem Frontend zu schützen, da der Zugriff auf das Frontend Zugriff auf die Datenbank gestattet - und zwar als Superuser!

4.5 Transaktionen

Dieser Abschnitt geht kurz auf Transaktionen ein. Transaktionen sind notwendig, um Änderungen atomar, dass heißt, ganz oder gar nicht, durchführen zu können.

Im Folgenden wird oft der englische Ausdruck lock verwendet. Wörtlich übersetzt bedeutet er in etwa **sperren**. Hier ist gemeint, ein Objekt so zu benutzen, dass es niemand anders gleichzeitig benutzen kann. Lock wird später noch genauer erklärt.

4.5.1 Einführung

Das klassische Beispiel für den Bedarf ist das Buchungsbeispiel. Angenommen, es existieren zwei Kontotabellen. Möchte man nun eine Buchung gegen diese beiden Tabellen machen, muss in jede Tabelle ein neuer Datensatz angelegt werden. Dazu muss man zwei `INSERT INTO SQL` Kommandos ausführen lassen.

Nun könnte es ja passieren, dass eines der beiden Kommandos klappt, das andere jedoch nicht. In diesem Fall würden die Konten nicht mehr stimmen, da die Summen nicht mehr passen. Man hätte inkonsistente Daten und ein Problem.

Daher fasst man beide Kommandos zu einer Transaktion zusammen. Eine Transaktion klappt entweder ganz, oder gar nicht. Geht also eines der SQL Kommandos schief, so hat auch das andere automatisch keinen Effekt (es wird gegebenenfalls **rückgängig gemacht**).

Transaktionen sind für andere erst sichtbar, wenn sie abgeschlossen wurden. Das bedeutet im Beispiel, dass nach dem Ausführen der ersten Kommandos ein anderer Client diese Änderung überhaupt nicht sieht. Erst wenn das andere Kommando erfolgreich war und die Transaktion beendet wurde, werden die Änderungen sichtbar. Somit stimmen die Summen zu jedem Zeitpunkt.

Wenn innerhalb einer Transaktion Daten gelesen werden, und von einer anderen Transaktion in dieser Zeit geändert werden, so wird die Transaktion automatisch abgebrochen. Auch hier kann es nicht passieren, dass Daten versehentlich zurückgeschrieben werden, die inzwischen an anderer Stelle geändert wurden.

4.5.2 Multiversion Concurrency Control

Implementiert wird ein sogenanntes **Multiversion Concurrency Control** (MVCC). Das bedeutet, das Abfragen einer Transaktion die Daten so sehen, wie sie zu einem bestimmten Zeitpunkt waren, unabhängig davon, ob sie inzwischen von einer anderen Transaktion geändert wurden. Dies verhindert, dass eine Transaktion einen Teil Daten vor und einen anderen nach einer nebenläufig abgeschlossenen Transaktion lesen kann und verhindert so inkonsistentes Lesen: die Transaktionen werden von einander isoliert. Der Hauptunterschied zu **Lock** Verfahren ist, dass MVCC Locks für das Lesen nicht mit Locks für das Schreiben in Konflikt stehen. Somit blockiert das Schreiben nie das Lesen und das Lesen nie das Schreiben.

Eine wichtige Einschränkung gibt es: Transaktionen können in *PostgreSQL* nicht geschachtelt werden (es gibt also keine `Untertransaktionen`).

4.5.3 Transaktionslevel

PostgreSQL unterstützt zwei Transaktionslevel: **read committed** und **serializable**. Verwendet eine Transaktion **read committed**, so kann es vorkommen, dass sie Daten erneut liest, aber andere Daten erhält als beim ersten Lesen (nicht-wiederholbares Lesen, non-repeatble reads). Auch sogenanntes Phantom-Lesen (phantom reads) kann vorkommen. Vom Phantom-Lesen spricht man, wenn sich in einer Transaktion die Ergebnissätze von Suchbedingungen ändern können. Sogenanntes **schmutziges Lesen** (dirty reads), also das Lesen von Änderungen aus anderen, nicht abgeschlossenen Transaktionen kann jedoch nicht auftreten. Dieser Transaktionslevel ist die Voreinstellung. Er ist einfach anzuwenden, schnell und für die meisten Anwendungen ausreichend.

Verwendet eine Transaktion **serializable**, können diese beiden unerwünschten Effekte nicht auftreten. Man benutzt diesen Level für Anwendungen, die komplexe Abfragen und Änderungen durchführen.

4.5.4 Anwendung

Transaktionen werden durch das SQL Kommando `BEGIN` eingeleitet. Dies ist nicht standardkonform; ANSI fordert, das immer implizit eine Transaktion begonnen wird. *PostgreSQL* bietet jedoch wie viele andere DBMS auch eine sogenanntes `auto commit` Funktion an, dies ist auch das Standardverhalten. Jedes SQL Kommando wird dann so aufgefasst, als wäre es eine einzelne Transaktion (es wird also sozusagen ein implizites `COMMIT` nach jedem SQL Kommando ausgeführt). Möchte man nun eine aus mehreren Anweisungen bestehende Transaktion beginnen, schreibt man einfach `BEGIN` als erstes Kommando. Dies passt auch gut zum eingebettetem SQL, da die SQL Kommandos dadurch in einen schicken `BEGIN` - `END` Block eingeschlossen sind.

Grundsätzlich gibt es zwei Möglichkeiten, eine Transaktion zu beenden. Eine Anwendung kann eine Transaktion selbst abbrechen. Hierzu dient das Kommando `ROLLBACK`. Keine der Änderungen der Transaktion wird ausgeführt. Eine Anwendung kann die Transaktion auch positiv beenden. Dazu wird `END` oder `COMMIT` verwendet. Die Transaktion wird genau dann durchgeführt, wenn sie fehlerfrei war. In diesem Fall werden alle Änderungen (oder die eine komplexe Transaktionsänderung) übernommen (sichtbar). Trat in der Transaktion ein Fehler auf, so gibt es natürlich keine Möglichkeit, sie doch noch positiv zu beenden, da dies zu Inkonsistenzen führen würde. In solchen Fällen kann die Anwendung (je nach Art des Fehlers) die Transaktion wiederholen. Dies ist natürlich nicht sinnvoll, wenn beispielsweise ein Tabelle fehlt. Dann wird auch die Wiederholung fehlschlagen.

So ist also sichergestellt, dass Transaktionen nur vollständig (und vollständig erfolgreich), oder überhaupt nicht durchgeführt werden.

Hat man mit `BEGIN` eine Transaktion begonnen, so ist zunächst die Datenbankvoreinstellung des Transaktionslevel (**read committed**) aktiv. Solange die Transaktion noch nicht begonnen wurde, kann der Transaktionslevel noch geändert werden. Dazu wird das SQL Kommando `SET TRANSACTION ISOLATION LEVEL` verwendet. Als Parameter wird `READ COMMITTED` oder `SERIALIZABLE` angegeben. Damit ist der Transaktionslevel eingestellt. Ein Client kann auch einen eigene Voreinstellung setzen, wenn beispielsweise Transaktionen grundsätzlich **serializable** sein sollen. Das SQL Kommando lautet `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL` und erwartet die gleichen Parameter wie das vorherige Kommando.

4.5.5 Mögliche Effekte

Verwendet man Transaktionen, so kann es natürlich vorkommen, dass eine Transaktion vom DBMS beendet wird, weil eine andere Transaktion Daten geändert hat; insbesondere, wenn **serializable** verwendet wird. In solchen Fällen wird die Transaktion in der Regel einfach von vorn beginnend vollständig wiederholt (die Anwendung führt diese also erneut aus).

Derartige Effekte minimiert man oft, in dem man Datensätze, von denen man schon weiß, dass man sie ändern muss, schon mal für selbiges vormerkt. Dies geschieht mit dem SQL Kommando `SELECT FOR UPDATE`. Nun weiß das DBMS, dass diese Datensätze **der Transaktion gehören**. Möchte eine andere Transaktion hier auch Daten ändern, so wartet diese automatisch, bis die erste Transaktion beendet wurde (also bestätigt oder abgebrochen). Dann erst wird die Aktion ausgeführt. Mit dem SQL Kommando `LOCK TABLE` können auch komplette Tabellen gesperrt werden. Verwendet man diese Mechanismen sorgfältig, vereinfacht sich die Handhabung; spätere Transaktionsabbrüche treten nicht auf, da die Daten ja bereits verwendet werden.

Es kann passieren, dass sich Transaktionen gegenseitig ausschließen. Würde beispielsweise Transaktion A die Tabelle A sperren und Transaktion B Tabelle B und anschließend Tabelle A sperrt, kommt es zu einer solchen Situation, wenn Transaktion A auch versucht, Tabelle B zu sperren. Transaktion B kann ja Tabelle A nicht sperren, weil diese schon von Transaktion A bereits gesperrt ist und blockiert, bis Transaktion A beendet wurde. Transaktion A wiederum wartet auf Transaktion B, um Tabelle B sperren zu können. Man spricht von einem Deadlock - beide Transaktionen haben sich gegenseitig blockiert.

PostgreSQL erkennt solche Fälle automatisch. Eine der beiden Transaktionen wird mit einem entsprechendem Deadlock-Fehler abgebrochen, woraufhin die andere durchgeführt werden kann. Auch hier wiederholt die Anwendung einfach die Transaktion. Da nun keine andere mehr läuft, wird es diesmal klappen.

Bei der Arbeit mit komplexen Transaktionen muss man damit rechnen, dass eine Transaktion durch solche oder ähnliche Gründe abgebrochen wird. In der Software ist also vorzusehen, Transaktionen wiederholen zu können. Da im Falle eines Transaktionsabbruches ja überhaupt keine Daten geändert werden, geht das unproblematisch. Man beginnt einfach von vorn.

4.5.6 Sperren für Tabellen

Sperre oder Lock bedeutet, dass der Inhaber oder Eigentümer dessen davor geschützt ist, dass jemand anders eine Sperre erzeugt, der dieser widerspricht. Es gibt verschiedene Arten von Sperren. Lese-Locks beispielsweise schließen sich nicht gegenseitig aus (es können ja problemlos mehrere Transaktionen die gleichen Daten lesen), jedoch schließt ein Lese-Lock einen Schreib-Lock aus. Schreib-Locks schließen sich und Lese-Locks aus. Letztere nennt man daher auch **exklusiv**, keine anderere Sperre kann neben einem Schreib-Lock ausgeführt sein.

Die folgende Aufstellung ist unvollständig.

AccessShareLock	(lesender Zugriff) Lese-Lock, der automatisch auf Tabellen gesetzt wird, aus denen gelesen wurden. Schließt AccessExclusiveLock aus.

RowShareLock	(lesender Zugriff auf Zeilen) Wird durch `SELECT FOR UPDATE` und `LOCK TABLE IN ROW SHARE MODE` gesetzt. Schließt ExclusiveLock und AccessExclusiveLock Modi aus.
RowExclusiveLock	(exklusiver Zugriff auf Zeilen) Wird durch `UPDATE`, `DELETE`, `INSERT` und `LOCK TABLE IN ROW EXCLUSIVE MODE` gesetzt. Schließt ShareLock, ShareRowExclusiveLock, ExclusiveLock und AccessExclusiveLock Modi aus.
AccessExclusiveLock	(exklusiver Zugriff) Gesetzt durch `ALTER TABLE`, `DROP TABLE`, `VACUUM FULL` und `LOCK TABLE`. Schließt alle Modi aus. Selbst `SELECT` in anderen Transaktionen blockiert in diesem Fall.

4.5.7 Sperren für Datensätze

Datensätze werden mit `SELECT FOR UPDATE` gesperrt. Dies schließt Änderungen an genau diesen Datensätzen aus. Wie bereits angedeutet, schließt dies kein Lesen aus (Schreiben blockiert kein Lesen).

4.5.8 Transaktionsbeispiel

Wie bereits gesagt, werden Transaktionen bei Fehlern automatisch abgebrochen. Alle Kommandos werden ignoriert:

Transaktion beginnen:

```
test=# BEGIN;
BEGIN
```

Es Kommando geht schief, zum Beispiel weil syntaktisch falsch:

```
test=# SYNTAX ERROR;
ERROR:  parser: parse error at or near "SYNTAX"
```

Die Transaktion ist abgebrochen worden. Alle Kommandos werden ab jetzt ignoriert:

```
test=# DELETE FROM temp;
NOTICE:  current transaction is aborted, queries ignored until
end of transaction block
*ABORT STATE*
```

Selbst wenn man versucht, die Transaktion positiv zu beenden, wird nichts geändert (die Transaktion wird also trozdem abgebrochen):

```
test=# COMMIT;
COMMIT
```

Die Antwort `COMMIT` heißt nicht, dass wirklich etwas committed wurde. Hier wurde ja ein `Rollback` durchgeführt. Dieses Verhalten ist bei Skripts sehr nützlich. Die Kommandos schreibt man einfach ein einen `BEGIN;` - `END;` Block (`End` ist das gleiche wie `Commit`). Bei einem Fehler wird keine Änderung ausgeführt - die Datenbank sieht genauso aus, wie vorher. Man kann das Skript korrigieren und erneut ausführen.

An dieser Stelle sei noch einmal daran erinnert, dass Strukturkommandos (wie `CREATE` und `DROP`) nicht den den Transaktionsregeln unterliegen.

4.5.9 Arbeiten mit Bedingungen

Es ist möglich, Bedingungen (`CONSTRAINTS`) an Tabellen zu definieren. Beispielsweise könnte man fordern, dass die Summe über alle Felder einer Tabelle null sein muss. Möchte man nun zu einem Datensatz drei addieren, muss man also von einem anderen drei abziehen. Doch kurz dazwischen ist die Bedingung ja verletzt, denn die Summe ist ja dann nicht mehr null, sondern drei!

Bedingungen können daher in Transaktionen aufgeschoben werden (`DEFERRED`). Das bedeutet, sie werden erst am Ende der Transaktion geprüft. Eine Bedingung kann dies aber auch verhindern. Bedingungen können so definiert werden, dass sie immer sofort geprüft werden. Bedingungen können aber auch so definiert werden, dass die Prüfung per Voreinstellung aufgeschoben wird, oder das die Bedingung explizit aufgeschoben werden kann.

Um Bedingungen aufzuschieben, die sofort geprüft werden sollen, aber auch aufgeschoben werden dürfen, verwendet man das SQL Kommando `SET CONSTRAINTS ALL DEFERRED`. Anstatt ALL kann man auch den Namen der Bedingung angeben (das wird auch oft gemacht). Anstatt `DEFERRED` kann auch `IMMEDIATE` eingestellt werden. Damit hat das den Gegenteiligen Effekt. Bedingungen, die per Voreinstellung aufgeschoben werden, werden dennoch sofort ausgeführt.

Schiebt man also eine Prüfung auf, so wird diese am (bisher positiven) Ende der Transaktion durchgeführt. Stellt sich nun heraus, dass die Bedingung verletzt ist, wird die Transaktion abgebrochen (und die Bedingung bleibt dadurch erfüllt).

4.6 Variablen und Zeitzonen

Es gibt einige Variablen, die das Verhalten des DBMS (für den entsprechenden Clienten) beeinflussen. Über Variablen wird beispielsweise gesteuert, wie Datumsangaben aussehen. Dies ist nicht standard konform (mit Ausnahme von `TIME ZONE`, hier wurde der Standard erweitert).

Variablen werden mit `SET` gesetzt und mit `SHOW` abgefragt. Mit `SET` wird eine Variable auf einen Wert gesetzt. Zwischen der Variable und dem Wert steht `TO` (oder ein Gleichheitszeichen).

Hier werden nur zwei wichtige Variablen erwähnt. Die Variable `DATESTYLE` setzt die Form der Datumsrepräsentation. Mögliche Werte sind German, ISO und andere.

Auch die Zeitzone kann man setzen. Hier verwendet man `SET TIME ZONE`. ANSI erlaubt als Parameter nur eine Zahl, beispielsweise `SET TIME ZONE 2`. Dies ist natürlich ungünstig, da die Sommer- und Winterzeit Unterscheidung von der Anwendung getroffen werden muss (Ist Berlin nun gerade -1 oder -2? Das hängt vom Datum ab!). *PostgreSQL* erlaubt jedoch auch `SET TIME ZONE 'Europe/Berlin'`.

An einem Beispiel wird gezeigt, wie man die aktuelle Uhrzeit mit Datum in Californien (Zeitzone PST8PDT) im ISO Format (amerikanische Notation) und in Berlin (Zone CET, Central European Time, deutsche Notation) ausgeben lassen kann.

```
test=# SET TIME ZONE 'PST8PDT'; SET DATESTYLE TO ISO; SELECT now();
SET VARIABLE
SET VARIABLE
              now
-------------------------------
 2003-01-02 11:30:17.698728-08
(1 row)
```

```
test=# SET TIME ZONE 'CET'; SET DATESTYLE TO German; SELECT now();
SET VARIABLE
SET VARIABLE
              now
--------------------------------
 02.01.2003 20:32:46.387261 CET
(1 row)
```

4.7 Datentypen

PostgreSQL unterstützt unter anderem die *SQL92* Datentypen. Ingesammt werden viele Typen unterstützt und eigene können definiert werden. Beispiele sind int (Ganzzahlen), double precision (8 Byte Fließkomma), serial (Autoinkrementeller int), varchar (variable lange Zeichenketten), bytea (Binäre Zeichenkette, wie ANSI BLOB), timestamp (Datum und Uhrzeit), boolean (Wahrheitswert) und viele andere.

Typ-Umwandlungen werden durchgeführt, in dem man den Zieltyp durch zwei Doppelpunkte `::` getrennt an den Typ anfügt: `'123'::int`.
Dies konvertiert die Zeichenkette 123 in einen Ganzzahltyp mit dem Wert einhundertdreiundzwanzig.

4.8 Operatoren

Neben den normalen Operatoren (`OR`, `AND`, `+`, `-`, `*`, `||` usw.) gibt viele weitere, beispielsweise Quadratwurzel `(|/)`, `LIKE` und `ILIKE` (Patternmatching wie bei `LIKE`, aber case-insensitiv) auch reguläres Patternmatching (`~`, `~*` und andere). Die Operatoren verhalten sich je nach Datentyp korrekt. Addiert man mit dem Operator `+` beispielsweise ein timestamp und ein intervall (also `now() + intervall '2 hours'`), kommt das erwartete Ergebnis heraus.

4.9 Vordefinierte FunktioneN

PostgreSQL stellt viele Funktionen bereit. Viele mathematische Funktionen sind verfügbar (`sin()`, `cos()`, `abs()`, `random()` usw). Daneben gibt es viele Zeichenkettenfunktionen (`lower()`, `substring()`, `initcap()`, `translate()`, `encode()`, um nur einige zu nennen). Auch die Zeit- und Datumsfunktionen sind sehr interessant und leistungsfähig. Beispielsweise gibt es `current_timestamp` (oder auch kurz `now`, eine klassische *PostgreSQL*-Erweiterung), extract (liefert Datumsteile, `SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40'`); führt also zu 38) und age (berechnet die Differenz zwischen zwei Zeitstempeln).

4.10 Datenbanken

Das Erzeugen und Planen von Datenbanken findet sich im Abschnitt Administration.

4.11 Tabellen

Wie in jedem anderen RDBMS werden natürlich auch Tabellen unterstützt. Diese werden mit `CREATE TABLE` erzeugt. Dieses Kommando ist gut ANSI konform. Es gibt temporäre Tabellen, die automatisch gelöscht werden. Tabellen und Spalten können Bedingungen besitzen, das sind beispielsweise Funktionen, die es verhindern können, sinnlose Daten einzutragen (2 stellige Postleitzahlen beispielsweise). Wie bereits in Arbeiten mit Bedingungen genannt, können die Prüfungen gegebenenfalls auf das Transaktionsende verschoben werden.

Fremdschlüssel sind Sonderformen von Bedingungen und werden auch unterstützt. Hiermit kann man gewährleisten, dass in eine Tabellenspalte nur solche Werte eingetragen werden können, die bereits in der Spalte einer anderen Tabelle definiert sind. Hat man z.B. eine Tabelle mit Herstellern und eine mit Teilen, in welcher der Hersteller vermerkt wird, kann sichergestellt werden, dass kein ungültiger Hersteller in letzterer eingetragen wird). Bei Fremdschlüsseln kann beispielsweise eine Aktion angegeben werden, die ausgeführt werden soll, falls der Fremdschlüssel gelöscht wird: `NO ACTION, RESTRICT` (dann ist das ein Fehler), `CASCADE` (die den Schlüssel referenzierenden Datensätze auch automatisch löschen, Vorsicht, dass können dann evtl. eine ganze Menge sein!), `SET NULL` (Wert auf `NULL` setzen), `SET DEFAULT` (auf Voreinstellung setzen).

Beispiele ähnlich denen aus der PostgreSQL Dokumentation

```
-- Eine Tabelle mit Primärschlüssel und einfachem Aufbau
CREATE TABLE films (
    code            CHARACTER(5) CONSTRAINT films_pkey PRIMARY KEY,
    title           CHARACTER VARYING(40) NOT NULL,
    distributors_id  DECIMAL(3) NOT NULL,
    date_prod       DATE,
    kind            CHAR(10),
    len             INTERVAL HOUR TO MINUTE
);
-- Beispieldatensatz
INSERT INTO films (code, title, distributors_id) VALUES ('FilmA', 'Der Film A', 123);

-- Eine Tabelle mit einem Autoinkrement und einer einfachen Bedingung
CREATE TABLE distributors (
     id     DECIMAL(3) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     name   VARCHAR(40) NOT NULL CHECK (name <> '')
);


-- Ein Tabelle mit Bedingung (distributors_id muss größer als 100 sein, der Name
-- darf nicht leer sein, sonst gibt es einen Fehler
-- Das Feld modtime wird automatisch auf "jetzt" gesetzt, wenn ein
-- Datensatz eingefügt wird.
CREATE OR REPLACE TABLE distributors (
    id              DECIMAL(3) UNIQUE,
    name            VARCHAR(40),
    modtime         TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    CONSTRAINT cst_valid_distributors_id CHECK (id > 100 AND name <> '')
);
-- Datensatz einfügen:
INSERT INTO distributors (id, name) VALUES (123, 'Name');
-- Nochmal geht schief, weil id eindeutig sein muss

-- Das geht auch schief:
-- INSERT INTO distributors (id, name) VALUES (001, 'Name');
--   denn: "ExecAppend: rejected due to CHECK constraint cst_valid_distributors_id"
--   (muss ja > 100 sein)


-- Eine Tabelle mit Fremdschlüssel und benannten Bedingungen.
-- "varchar" heißt einfach: kann beliebig lang werden (also fast,
--   bei ca 1000 MB ist Ende)
CREATE TABLE lager (
    id              SERIAL PRIMARY KEY,
    films_code      CHARACTER(5),
    distributors_id  DECIMAL(3),
    info            VARCHAR DEFAULT NULL,
    CONSTRAINT fk_lager_distributors_id FOREIGN KEY (distributors_id) REFERENCES
distributors(id)
        ON DELETE RESTRICT,
    CONSTRAINT fk_lager_films_code FOREIGN KEY (films_code) REFERENCES films(code)
        ON DELETE RESTRICT DEFERRABLE
);
-- Datensatz einfügen
INSERT INTO lager (id, films_code, info) VALUES (123, 'FilmA', 'hallo');

-- Das geht schief:
-- INSERT INTO lager (id, films_code, info) VALUES (124, 'FilmA', 'hallo');
--   denn: "fk_lager_distributors_id referential integrity violation
--         - key referenced from lager not found in distributors"

-- Das geht auch schief:
-- DELETE FROM distributors;
--   denn: "fk_lager_distributors_id referential integrity violation
--         - key in distributors still referenced from lager"
```

Tabellen können mit dem Kommando `ALTER TABLE` geändert werden. Diese Kommando hat viele Formen.

Einige Beispiele:

Beispiele: ALTER TABLE

```
-- Eine Spalte anfügen:
ALTER TABLE lager ADD COLUMN plz VARCHAR(8);

-- Eine Spalte ändern:
ALTER TABLE lager ALTER COLUMN plz SET DEFAULT 'unsortiert';

-- Eine Spalte umbennen:
ALTER TABLE lager RENAME COLUMN plz TO zipcode;

-- Bedingung hinzufügen (PLZ muss fünfstellig sein)
ALTER TABLE lager ADD CONSTRAINT cst_zipchk CHECK (char_length(zipcode) = 5);

-- Bedingung entfernen
ALTER TABLE lager DROP CONSTRAINT cst_zipchk RESTRICT;

-- Tabelle umbennnen
ALTER TABLE lager RENAME TO lagermitplz;
ALTER TABLE lagermitplz RENAME TO lager;

-- Eigentümer ändern
ALTER TABLE lager OWNER TO steffen;
```

Ab Version 7.3 wird endlich auch *SQL 92* `ALTER TABLE DROP COLUMN` unterstützt. Gibt es einen Index, eine Bedingung oder einen Fremdschlüssel der die zu löschende Spalte referenziert, muss die Option `CASCADE` mit angegeben werden.

Für ältere Versionen hat sich folgende Vorgehensweise bewährt: Man muss die Tabelle neu erzeugen. Diese Funktion wird übrigens von `phpPgAdmin` unterstützt (das heißt, es gibt einen DROP Knopf, der im Prinzip das tut). Im Folgenden wird ein Workaround gezeigt. Es werden hier gleich noch ein paar weitere Kommandos demonstriert.

Beispiele

```
-- Workaround für fehlendes:
-- ALTER TABLE lager DROP COLUMN zipcode;

-- Daten in Temp-Tabelle:

BEGIN;

-- Tabelle exklusiv schützen:
LOCK TABLE lager IN ACCESS EXCLUSIVE MODE;
-- LOCK TABLE lager; macht das gleiche (Voreinstellung ist ACCESS EXCLUSIVE)

CREATE TEMPORARY TABLE temp AS SELECT id, films_code, distributors_id, info FROM lager;

-- lager Tabelle neu erstellen
DROP TABLE lager;
CREATE TABLE lager (
    id              SERIAL PRIMARY KEY,
    films_code      CHARACTER(5),
    distributors_id  DECIMAL(3),
    info            VARCHAR DEFAULT NULL,
    CONSTRAINT fk_lager_distributors_id FOREIGN KEY (distributors_id) REFERENCES
distributors(id)
        ON DELETE RESTRICT,
    CONSTRAINT fk_lager_films_code FOREIGN KEY (films_code) REFERENCES films(code)
        ON DELETE RESTRICT
        DEFERRABLE
);
-- Achtung, die Berechtigungen und Bedingungen der Tabelle müssen
--   noch gesetzt werden!

-- neue Tabelle füllen
INSERT INTO lager SELECT * FROM temp;

-- vielleicht noch prüfen
-- SELECT * FROM lager LIMIT 100;

DROP TABLE temp;
-- nicht unbedingt notwendig, passiert sonst bei Ende der
--   Sitzung automatisch

-- Transaktion abschließen
END;
```

Füllt man (beispielsweise neue) Tabellen mit sehr vielen Daten, so ist `INSERT` langsam. Die schnellste Möglichkeit ist das Füllen über `COPY`. Bei sehr vielen Datensätzen spart es auch Zeit, die `Indizes` zu löschen und anschließend neu zu erzeugen. Traut man den Daten, weil diese beispielsweise aus einem Backup kommen, so bringt es auch oft sehr viel Zeitersparnis, wenn man die `Trigger` und Bedingungen löscht und nach dem Füllen wieder neu anlegt.

Eine Erweiterung ist die Möglichkeit `CREATE TABLE AS`, die eine Tabelle aus einer `SELECT`-Abfrage erzeugt. Das ist äquivalent zu einer `INSERT INTO` Erweiterung, mit der auch Tabellen erzeugt werden können (beides ist nicht Standard-SQL). Um standardkonform zu sein, muss man zunächst ein `CREATE TABLE` machen und diese dann über `INSERT ... SELECT` füllen.

4.12 Views

Views sehen aus Sicht der Anwendung aus wie Tabellen. Manchmal werden sie sogar als **virtuelle Tabellen** bezeichnet. Es sind Sichten auf Tabellen. Eine View stellt eine Abfrage (ein `SELECT` Kommando) dar. Diese Abfrage kann beispielsweise nur einige der Spalten einer Tabelle enthalten. Die Abfrage kann auch über einen

`join` mehrere Tabellen verbinden und so Werte aus verschiedenen Tabellen anzeigen.

Ein großer Vorteil von Views ergibt sich, wenn man sich an die Privilegien erinnert. Über Views kann man es erreichen, dass nur bestimmte Felder sichtbar sind. In diesem Fall definiert man einen View über die erlaubten Felder und gibt dem entsprechenden Benutzer Rechte auf den View - nicht aber auf die Tabelle.

Momentan können Views so erstmal nur zum Lesen von Daten, nicht jedoch zum Ändern benutzt werden. Möchte man Daten auch ändern können, so verwendet man eine *PostgreSQL* Erweiterung, eine Regel. Im später später folgenden Abschnitt zu Regeln wird dies exemplarisch erklärt.

4.13 Cursors

Das Cursorkonzept stammt aus eingebettetem SQL (ESQL). Eingebettet heißt, dass man SQL Anweisungen **direkt** in Programmquelltexte einbettet (diese Programmiertechnik wurde inzwischen durch Standards wie ODBC weitgehend abgelöst; ESQL wird jedoch auch heute noch verwendet und auch von *PostgreSQL* unterstützt). In *PostgreSQL* stehen Cursors unabhängig von der Verwendung von ESQL zur Verfügung. Man kann sie beispielsweise auch über `psql` interaktiv verwenden.

Einem aktiven Cursor ist eine Menge von Datensätzen assoziiert, die über eine Abfrage, also über ein `SELECT` Kommando, ausgewählt wurden. Man kann nun einzelne Datensätze oder Teilmengen der Datensatzmenge über den Cursor holen. Der Cursor merkt sich dabei die Position. Holt man beispielsweise dreimal einen Datensatz aus einem Cursor, so erhält man automatisch die ersten drei Datensätze. Der Cursor zählt sozusagen mit, was auch den Namen erklärt. Eine Besonderheit ist, dass man über Cursors (in *PostgreSQL*, das gilt nicht generell) auch rückwärts gehen kann, also dass man Datensätze mehrfach holen kann.

Cursors funktionieren in *PostgreSQL* nur in Transaktionen. Um einen Cursor zu verwenden, muss dieser zunächst deklariert werden. Man kann sich vorstellen, dass man einer Abfrage einen (temporären) Namen gibt. Dann kann man Datensätze (die Ergebnisse der Abfrage) holen. Man kann den Cursor auch verschieben, beispielsweise, um Datensätze auszulassen oder erneut zu verarbeiten. Wird der Cursor nicht mehr benötigt, so wird er mit `CLOSE` geschlossen.

Besonderheiten in *PostgreSQL* sind, dass aus einem Cursor nicht über absolute Positionen gelesen werden kann und das Cursordaten nicht geändert werden können (es gibt kein `DECLARE FOR UPDATE`). Ein Cursor ist also immer `READ ONLY`. Durch die Transaktionsforderung ist er auch immer `INSENSITIVE`, auch wenn dies nicht explizit mit angeben wurde. Auch `SCROLL` ist nicht notwendig, da ein Cursor immer `SCROLL` kann. Es muss auch kein `OPEN` auf einen Cursor gemacht werden.

Ein einfaches Beispiel folgt.

cursor.sql

```
-- Die Tabelle sieht so aus:
test=> SELECT code, title FROM films WHERE distributors_id = 124;
 code  |      title
-------+------------------
 MM-dt | Mädchen, Mädchen
 IJ1   | Indiana Jones 1
 IJ2   | Indiana Jones 2
 IJ3   | Indiana Jones 3
(4 rows)

-- Transaktion starten
test=> BEGIN;
BEGIN

-- Einen Cursor für Indiana Jones deklarieren.
test=> DECLARE ijfilme INSENSITIVE CURSOR FOR
test->   SELECT code, title FROM films
test->   WHERE code LIKE 'IJ%'
test->   ORDER BY code
test-> FOR READ ONLY;
DECLARE

-- Ersten Datensatz holen
test=> FETCH NEXT FROM ijfilme;
 code  |      title
-------+-----------------
 IJ1   | Indiana Jones 1
(1 row)

-- Zweiten Datensatz holen (1 ist wie NEXT)
test=> FETCH 1 FROM ijfilme;
 code  |      title
-------+-----------------
 IJ2   | Indiana Jones 2
(1 row)

-- Einen Datensatz zurückgehen:
test=> FETCH -1 FROM ijfilme;
 code  |      title
-------+-----------------
 IJ1   | Indiana Jones 1
(1 row)


-- Die nächsten zwei Datensätze holen:
test=> FETCH 2 FROM ijfilme;
 code  |      title
-------+-----------------
 IJ2   | Indiana Jones 2
 IJ3   | Indiana Jones 3
(2 rows)

-- Hier ist Ende:
test=> FETCH 1 FROM ijfilme;
 code | title
------+-------
(0 rows)

-- weit Zurückspringen (an den Anfang)
test=> MOVE -100 FROM ijfilme;
MOVE 3

-- wieder am Anfang
test=> FETCH 1 FROM ijfilme;
 code  |      title
-------+-----------------
 IJ1   | Indiana Jones 1
(1 row)
```

```
-- Rest holen
test=> FETCH ALL FROM ijfilme;
 code  |      title
-------+-----------------
 IJ2   | Indiana Jones 2
 IJ3   | Indiana Jones 3
(2 rows)

-- Den letzen nochmal (wie -1)
test=> FETCH PRIOR FROM ijfilme;
 code  |      title
-------+-----------------
 IJ3   | Indiana Jones 3
(1 row)

-- Cursor schließen
test=> CLOSE ijfilme;
CLOSE

-- Transaktion abbrechen
test=> ROLLBACK;
ROLLBACK
```

4.14 Indizes

Ein Index dient dazu, Datensätze mit bestimmten Eigenschaften schnell zu finden. Hat man beispielsweise eine Tabelle films wie im Beispiel **Tabellen** und sucht den Film mit dem code FilmA, so müsste ja die gesamte Tabelle durchsucht werden (und dazu vor allem von Festplatte geladen werden), dann müsste jeder code geprüft werden, ob er denn dem gesuchten entspricht.

Hier verwendet man einen Index. Ein Index gilt für eine bestimmte Tabellenspalte, also beispielsweise für code. Er kann aber auch aus mehreren zusammengesetzten Spalten bestehen. Ein Index ist eine effiziente Speicherung aller code Werte und einem Verweis auf die Stelle, an der der zugehörige Datensatz gespeichert ist. Wie genau die Speicherung funktioniert, hängt vom Typ des Index ab. Es gibt beispielsweise HashIndizes und binäre Bäume.

Sucht man nun FilmA, so wird nur der Index geladen, der ja viel kleiner ist, als die ganze Tabelle. Es wird an der entsprechenden Stelle nachgesehen (bei einem Hash geht das bei einer Gleichoperation mit einem Zugriff), dann direkt die richtige Stelle (oder die richtigen Stellen) der Tabelle geladen. Das ist dann wesentlich schneller.

Indizes sind aber nicht immer günstig. Hat man beispielsweise viele Datensätze, beispielsweise alle, so muss eh sehr viel von der Tabelle geladen werden. Hier bremst es nur, zusätzlich den Index zu laden (der Abfrageplaner würde in solchen Fällen den Index aber automatisch nicht verwenden, weil er das auch weiß, mehr dazu später). Das gleiche Verhalten kann man auch bei kleinen Tabellen erwarten (wenn man beispielsweise 100 aus 1000 Datensätzen liest, ist ein Index oft nicht günstig und wird nicht verwendet). Ein Index verlangsamt auch Änderungen, da nicht nur die Tabelle, sondern auch der Index aktualisiert werden muss.

Ein Index kann auch Eindeutigkeit (`UNIQUE`) fordern. Genauer gesagt, wird Eindeutigkeit in Tabellen garantiert, in dem ein `UNIQUE` Index angelegt wird. Dies sollte man aber lieber durch ein sauberes `ALTER TABLE ... ADD CONSTRAINT` erledigen. Das dann ein Index verwendet wird, ist ein Implementierungsdetail von *PostgreSQL*.

Die bereits kurz erwähnten Speichertypen von Indizes sind: BTREE (Lehman-Yao B-Baum), RTREE (R-Baum mit Guttman's "quadratic split" Algorithmus), HASH (Litwin's lineares hashen) und GIST (Generalized Index Search Trees, verallgemeinerter Index Suchbaum).

BTREE kann bei den Operationen <, <=, =, >=, > verwendet werden. RTREE bei den Operationen <<, &<, &>, >>, @, ~=, && und ein HASH bei =.

Indexes kann man per Hand erzeugen. Dazu gibt es das nicht-standard SQL Kommando `CREATE INDEX`. Zum Löschen gibt es analog `DROP INDEX`. Ein Index auch hat immer einen Namen. Meistens setzt man diesen aus Tabellen- und Feldnamen zusammen. Ein Beispiel für einen Index test1_id_idx über die Spalte id der Tabelle test1:

```
CREATE UNIQUE INDEX test1_id_idx ON test1 USING BTREE (id);
```

Es ist sogar möglich, Indizes für Funktionsergebnisse zu definieren. Verwendet man beispielsweise oft:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

so hilft einem ein Index über col1 hier ja nichts. Man kann aber einen Index für lower(col1) erzeugen, der dann wieder verwendet wird:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Indizes können auch nur über Teile gelegt werden, in dem man eine `WHERE` Bedingung hinzufügt. So kann man beispielsweise sehr häufige Werte ausklammern und von Geschwindigkeitsvorteilen bei seltenen Werten profitieren (bei häufigen Werten werden Indizes oft gar nicht verwendet, weil langsam). Eine genaue Diskussion würde diesen Rahmen hier jedoch sprengen.

Eine Erweiterung ist die Möglichkeit, Indizes neu zu erstellen. Oft kann man diese einfach löschen und neu anlegen, was den Vorteil hat, dass nur die zu lesenden Datensätze gelockt werden. Hat man jedoch kaputte Indizes, kann man diese mit `REINDEX` neu erstellen lassen. Dies wird nur durchgeführt, wenn der Index als kaputt bekannt ist, oder man `FORCE` mit angibt.

Es gibt drei Varianten des Kommandos: `REINDEX INDEX` (erzeugt den folgenden Index neu), `REINDEX TABLE` (erzeugt für die folgend genannte Tabelle alle Indizes neu) und `REINDEX DATABASE` (erzeugt für die folgend genannte Datenbank alle Indizes neu).

```
REINDEX DATABASE my_database FORCE;
```

Hilft bei Problemen also (was in der Praxis jedoch im Prinzip NIE benötigt wird; aber wenn, dann hilft das).

4.15 Funktionen

Man kann sich eigene Funktionen definieren. Hierzu stehen neben SQL noch weitere Sprachen bereit. SQL ist in manchen Punkten beschränkt oder umständlich. Hier hilft einem eine Sprache wie *PL/pgSQL* oder *PL/Perl* weiter.

So kann man sich Funktionen schreiben, die beispielsweise komplizierte Bedingungen prüfen können (vielleicht **Quersumme der ID muss gerade sein**). Funktionen kann man auch direkt aufrufen. In *PostgreSQL* ruft man selbst definierte Funktionen genauso auf, die eingebaute: einfach über `SELECT`. Eine Funktion hallo mit zwei Parametern könnte man beispielsweise aufrufen:

```
SELECT hallo(123, 'hallo parameter');
```

Eine Erweiterung von *PostgreSQL* ist die Möglichkeit, Aggregatfunktionen selbst zu definieren (`CREATE AGGREGATE`). Aggregatfunktionen sind Funktionen wie min oder max, die beispielsweise in Gruppierten `SELECT` Anweisungen verwendet werden.

4.16 Trigger

`Trigger` sind relativ *SQL99* konform (es gibt einfach zu umgehende Ausnahmen). `Trigger` können nicht auf einzelne Spalten angewendet werden. Über einen `Trigger` kann man vor oder nach den Ereignissen `INSERT`, `DELETE` oder `UPDATE` auf eine Tabelle eine Funkion aufrufen (man sagt, der `Trigger` feuert bei einem Ereignis). Diese Funktion kann dann die Daten prüfen, ändern oder sonst was unternehmen. *PostgreSQL* bietet erweitert dazu auch Regeln (Rules).

Über `Trigger` kann man, wie auch mit Bedingungen, Konsistenzbedingungen realisieren. Werden beispielsweise Schlüssel geändert, so kann man über einen `Trigger` vielleicht abhängige Datensätze entsprechend anpassen.

4.17 Regeln (Rules)

Regeln sind eine *PostgreSQL* Erweiterung. Ähnlich wie `Trigger` reagieren sie auf ein Ereignis `SELECT`, `INSERT`, `DELETE` oder `UPDATE` auf eine Tabelle. Optional kann noch eine Bedingung angegeben werden, die ebenfalls erfüllt sein muss, damit die Regel greift. Die Regel definiert dann, ob gar nicht passieren soll (`NOTHING`), ob zusätzlich oder ob anstatt (`INSTEAD`) des eigentlichen Kommandos ein anderes ausgeführt werden soll.

Über Regeln kann man, wie auch mit Bedingungen, Konsistenzbedingungen realisieren.

Regeln werden bei *PostgreSQL* oft in Verbindung mit Views verwendet. *Dr.Ruud* postete eine beispielhafte **Regelschablone**:

rules-template.sql

```
CREATE VIEW <virtual-table> AS SELECT * FROM <actual-table>;

CREATE RULE <virtual-table>_ins AS ON INSERT TO <virtual-table>
DO INSTEAD
INSERT INTO <actual-table> ( <field-1>, <field-2>, ... , <field-n> )
VALUES ( new.<field-1>, new.<field-2>, ... , new.<field-n> );

CREATE RULE <virtual-table>_upd AS ON UPDATE TO <virtual-table>
DO INSTEAD
UPDATE <actual-table>
SET <field-1> = new.<field-1>,
<field-2> = new.<field-2>,
...
<field-n> = new.<field-n>
WHERE <primary-key> = old.<primary-key>;

CREATE RULE <virtual-table>_del AS ON DELETE TO <virtual-table>
DO INSTEAD
DELETE FROM <actual-table>
WHERE <primary-key> = old.<primary-key>;
```

4.18 Sequenzen

Eine Sequenz ist eine Zählfunktion, die hauptsächlich bei Autoinkrementfeldern angewendet werden. Bei jedem Aufruf liefert eine Sequenz einen größeren Wert (dies funktioniert natürlich auch vollständig in Transaktionen). Man kann auch den letzten Wert abfragen, den man in der Transaktion erhalten hat und so herausfinden, welchen Wert der letzte Datensatz im Autoinkrementfeld erhalten hat.

Sequenzen werden beim Feldern vom Typ **serial** automatisch erzeugt. Der Name wird automatisch bestimmt. Es kommt zu einem Fehler beim Anlegen der Tabelle, wenn der Name bereits vergeben ist. Man kann ein Sequenz

auch für mehrere verschiedene Felder verwenden, und so tabellenübergreifend eindeutige Werte erzeugen. Daher werden automatisch erzeugte Sequenzen nicht automatisch mit dem Löschen von Tabellen gelöscht.

Man kann Sequenzen auch explizit über `CREATE SEQUENCE` erzeugen. Die Funktionen `nextval('seq')` und `currval('seq')` liefern den nächsten bzw. aktuellen (zuletzt gelieferte nextval) zurück. Mit `setval('seq', 1234)` kann man den Wert eine Sequenz direkt setzen.

Dies braucht man beispielsweise, wenn man IDs hat, die von der Sequenz noch gar nicht erzeugt wurden, weil jemand einen Wert bei `INSERT` direkt angegeben hat. In solchen Fällen erreicht die Sequenz irgendwann diesen Wert (oder den ersten dieser Werte), daraufhin klappt das `INSERT` nicht, weil die ID sicherlich eindeutig sein muss, die Sequenz wird auch nicht erhöht (Transaktionsabbruch) und man kommt nicht weiter. Hier hilft es, die Sequenz auf den höchsten verwendeten Wert zu setzen. Hat man eine Tabelle lager mit einem Autoinkrementfeld id, so heißt die automatisch erzeugte Sequenz lager_id_seq. Um diese anzupassen, kann man einfach schreiben:

```
test=# SELECT setval('lager_id_seq', (SELECT max(id) FROM lager) );
 setval
--------
      3
(1 row)
```

Danach funktioniert das Autoinkrementfeld wieder. Vor solchen Phänomenen kann man sich schützen, wenn man Regeln verwendet, die ein direktes Setzen solcher Felder verhindern.

4.19 Sprachen

Neben SQL unterstützt *PostgreSQL* weitere Datenbanksprachen. Arbeitet man mit SQL, so kann man bestimmte Dinge teils nur schwierig formulieren.

SQL ist eine sehr mächtige Sprache, wenn man sie beherrscht. Im Gegensatz zu prozeduralen Sprachen beschreibt man jedoch keine Algorithmen. Möchte man beispielsweise alle Werte des Feldes gehalt einer Tabelle mitarbeiter um 10 Prozent erhöhen, würde man prozedural formulieren: **gehe jeden Datensatz durch, und für jeden Wert setze Wert gleich Wert mal 1.1**. In SQL schreibt man das jedoch einfach so hin:

```
UPDATE mitarbeiter SET gehalt = gehalt * 1.1;
```

Man beschreibt also in etwa Änderungen. Zusätzlich kann man hier natürlich auch Bedingungen angeben (**nur, wenn gehalt kleiner als 5000 ist** beispielsweise). Diesen grundlegenden Unterschied muss man unbedingt verstehen, wenn man mit SQL arbeitet. In der Praxis sieht man manchmal Skripte, die die Datensätze einer Tabelle einzeln durchgehen, einen Test machen, und eine Änderung schreiben. So etwas macht man in der Regel einfach mit einem passendem SQL Kommando; das hat noch den angenehmen Nebeneffekt, viel schneller zu sein.

In *PostgreSQL* kann man auch Unterabfragen verwenden:

Beispiel: Unterabfrage

```
UPDATE mitarbeiter
SET gehalt = gehalt +
        (SELECT bonus FROM bonustabelle WHERE art = 'weihnachtsgeld');
```

Mit derartigen Konstrukten kann man Operationen durchführen, die in prozeduralen Sprachen nur sehr umständlich gemacht werden können.

4.19.1 PL/pgSQL

Die Sprache *PL/pgSQL* ist im **Lieferumfang** von *PostgreSQL*. Sie ähnelt *PL/SQL* von *Oracle*. Diese Sprache ist beliebt, um `Trigger`funktionen zu implementieren. In *PL/pgSQL* sind Kontrollstrukturen verfügbar (beispielsweise Schleifen). Diese Sprache ist an SQL angelehnt und daher sehr leicht erlernbar und einfach zu benutzen.

Neben Zuweisungen, der Möglichkeit dynamische SQL Kommandos auszuführen und Bedingungen auszuwerten, stehen mehrere Schleifen zur Verfügung. Mit `FOR` kann gezählt oder über Datensätze iteriert werden, auch mit `LOOP` und `WHILE` kann man Schleifen bilden. Bedingungen sind flexibel (`IF-THEN-ELSIF-ELSE`). Ein Blick in die Dokumentation ist sicherlich interessant, `PL/pgSQL` sollte zum Handwerkszeug eines Datenbankbenutzers gehören.

Als Beispiel folgt eine `Trigger`funktion. Da die gesamte Funktion in einfache Anführungszeichen eingeschlossen ist, müssen innerhalb der Funktion alle einfachen Anführungszeichen durch zwei aufeinanderfolgende ersetzt werden (leider etwas unübersichtlich).

Beispiel: triggerbeispiel.sql

```
-- Eine Beispieltabelle für Angestellte
CREATE TABLE emp (
    empname text,               -- Name
    salary integer,             -- Gehalt
    last_date timestamp,        -- Letztes Datum
    last_user text              -- Letzter Benutzer
);

-- Der Trigger.
-- Es ist eine Funktion, die einen Datensatz zurückliefert.
CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    -- Dieses Begin kommt von PL/pgSQL. Es startet keine
    --   neue Transaktion!
    BEGIN
        -- Prüfen, ob empname und salary (Gehalt) angegeben wurde
        IF NEW.empname ISNULL THEN
            -- RAISE erzeugt einen Fehler (EXCEPTION)
            -- Die Transaktion wird dadruch abgebrochen.
            RAISE EXCEPTION ''empname darf nicht NULL sein'';
        END IF;
        IF NEW.salary ISNULL THEN
            -- anstelle des % steht dann der Name
            RAISE EXCEPTION ''% ohne Gehalt?!'', NEW.empname;
        END IF;

        -- Wer arbeitet für uns und muss dafür bezahlen?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% mit negativen Gehalt?!'', NEW.empname;
        END IF;

        -- Es wird das letze Änderungsdatum und der Änderungsbenutzer gesetzt.
        -- Selbst wenn bei INSERT oder UPDATE last_user angegeben wird, so
        -- wird dennoch immer current_user verwendet. Es ist also
        -- nicht mehr möglich, einen falschen Eintrag zu erzeugen.
        NEW.last_user := current_user;
        -- now sollte hier besser als Funktion und besser gegen
        --   den standardkonformen Namen current_timestamp ersetzt werden:
        --   NEW.last_date := current_timestamp;
        NEW.last_date := ''now'';

        -- Den (geänderten) Datensatz zurückliefern (wird dann eingetragen)
        RETURN NEW;
    END;
    -- Das END kommt - wie auch BEGIN - von PL/pgSQL und beeinflußt
    --   die aktive Transaktion nicht
' LANGUAGE 'plpgsql';

-- Diese Funktion als Trigger setzen. Danach wird sie bei INSERT
-- oder UPDATE automatisch gestartet.
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

4.19.2 PL/Perl

PL/Perl kann auf zwei Arten installiert werden (siehe Abschnitt ▶ Sprachen im administrativen Teil). Im trusted Modus kann die Sprache gefahrlos benutzt werden, darf jedoch nicht alles. So dürfen zum Beispiel keine externen Module geladen werden. Im untrusted Modus geht das. Man kann so beispielsweise Mails verschicken. Da hierdurch jeder, der eine Funktion schreiben und starten darf, die Unix-Rechte des Unix-Benutzers postgres (oder unter welchem Benutzer das DBMS läuft) erhält, muss man hier vorsichtig und sorgfältig arbeiten.

Die Verwendung von *PL/Perl* ist sehr intuitiv. `NULL` Werte werden in Perl als **undef** dargestellt. Parameter werden wie gewohnt über `$_` erreicht. Zusammengesetzte Datentypen werden als Referenzen auf Hashes

übergeben, was eine sehr komfortable Handhabung erlaubt.

Fehler werden durch Aufruf der Funktion elog gemeldet. elog verhält sich analog zu `RAISE`.

Leider gibt es (noch?) einige Einschränkungen bei der Verwendung. So kann *PL/Perl* leider nicht dazu verwendet werden, `Trigger`funktionen zu schreiben. Es ist aber möglich, einen `Trigger` in *PL/pgSQL* zu schreiben, und hier einfach eine *PL/Perl* Funktion aufruft.

Beispiel: plperl.sql

```
-- Eine Funktion, die den größeren Wert zurückliefert.
-- Ist eine der Werte NULL, so wird der andere zurückgeben.
-- Sind beide NULL, ergibt die Funktion auch NULL
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS '
    my ($a,$b) = @_;
    if (! defined $a) {
        if (! defined $b) { return undef; }
        return $b;
    }
    if (! defined $b) { return $a; }
    if ($a > $b) { return $a; }
    return $b;
' LANGUAGE plperl;


-- Ein Beispiel mit einen zusammengesetzten Datentyp (hier employee)
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

-- Als Parameter kommt ein employee, also z.B. ein Datensatz aus
-- dieser Tabelle
CREATE FUNCTION empcomp(employee) RETURNS integer AS '
    my ($emp) = @_;
    return $emp->{''basesalary''} + $emp->{''bonus''};
' LANGUAGE plperl;
```

4.20 Notifikationen (Benachrichtigungen)

Eine *PostgreSQL* Erweiterung erlaubt es, das mehrere Clienten sich synchronisieren. Dazu kann ein Client über `LISTEN` ein Objekt beobachten. Ruft ein anderer Client `NOTIFY` auf diesem Objekt auf, so wird ersterer (und alle anderen `LISTENer`) benachrichtigt. Hat er kein Interesse mehr an Notifikationen, ruft er ein `UNLISTEN` auf das Objekt auf. `LISTEN` ist nicht blockierend; die Notifikation erfolgt asynchron.

Dies wird wohl selten verwendet und ist nicht portabel. Oft kann man ähnliches Verhalten auch über Datensatz-Locks über Tabellen erreichen.

4.21 Statistiken für den Planer

Wie im Abschnitt ▶ Optimierung mit EXPLAIN noch genauer erklärt wird, wird eine Abfrage vom Planer in Abfragepläne umgewandelt. Um sich für den richtigen (also den schnellsten) Abfrageplan entscheiden zu können, muss beispielsweise geschätzt werden, wie viele Daten von Festplatte gelesen werden müssen. Das hängt von der Tabellengröße ab.

Die Tabellengrößen werden von *PostgreSQL* in einer speziellen Tabelle `pg_class` gespeichert. Meistens werden jedoch nicht alle Datensätze benötigt, sondern nur ein Teil. Dieser wird oft über eine Bedingung

definiert. Dadurch wird es schwierig zu schätzen, wie viele Daten geladen werden müssen, da man dazu ja wissen muss, wie oft die Bedingung erfüllt ist.

Um diese Abschätzung durchführen zu können, werden Statistiktabellen geführt, beispielsweise `pg_stats`. In diesen Tabellen werden statistische Informationen über Tabelleninhalte gespeichert, beispielsweise die häufigsten Werte, die Anzahl der `NULL` Werte, die Anzahl der verwendeten Werte (es kann ja Tabellen mit 1000 Einträgen geben, die nur 4 verschiedene Werte verwenden) und andere. Mit diesen Informationen errechnet der Planer seine Abschätzungen.

Die Statistiken werden natürlich nicht ständig aktualisiert, das wäre ja sehr bremsend (um statistische Korrelation zu berechnen, muss ja in jedem Fall jeder Datensatz gelesen werden). Statt dessen werden die Statistiken durch das SQL Kommando `ANALYZE` oder `VACUUM ANALYZE` aktualisiert, dass man demzufolge regelmäßig (zum Beispiels nachts und nach großen Änderungen) ausführen sollte.

4.22 Optimierung mit "EXPLAIN"

Natürlich ist es immer interessant, Abfragen auf Geschwindigkeit zu optimieren. Bei langsamen Abfragen ist es interessant, den Grund zu kennen. Vielleicht fehlt ja nur ein Index oder sitzt ungünstig?

Verarbeitet *PostgreSQL* eine Abfrage, so wird vom DBMS ein Abfrageplan erstellt. Dies wird durch den sogenannten Planer erledigt. Dieser legt fest, in welcher Reihenfolge die Daten organisiert werden und ob (und welche) Indizes verwendet werden. Dazu prüft er die verschiedenen Möglichkeiten auf Effizienz. Er erstellt also erstmal viel Pläne und wählt dann den Plan aus, der die geringsten Kosten hat, also am schnellsten geht.

Es gibt das `EXPLAIN` Kommando, das den Abfrageplan für die Abfrage anzeigt (eine *PostgreSQL* Erweiterung). Man erhält die geschätzten Kosten, bis mit der Ausgabe begonnen werden könnte, und die gesamten Kosten. Als Einheit wird in etwa **Festplattenzugriffe** verwendet. Die anderen beiden Zahlen sind die geschätzte Anzahl an Datensätzen (etwas richtiger ist hier der Begriff Tupel), die zurückgegeben werden, und die geschätzte Größe eines Datensatzes.

Eine Abfrage besteht aus mehreren Teilen. Die Kosten jedes Teiles schließen immer die aller nach unten folgenden Teile ein.

Ein paar Beispiele dazu.

```
regression=# EXPLAIN SELECT * FROM tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

Man sieht: Es ist ein vollständiges durchgehen der Tabellen tenk1 notwendig (Seq Scan heißt sequentiell). Mit der Ausgabe kann sofort begonnen werden, nach 333 Zugriffen ist sie nach 10000 Datensätzen beendet. Die 333 Zugriffe entstehen hier übrigens durch 233 Diskzugriffe und 10000 * cpu_tuple_cost (Voreinstellung ist 0.01), also 233 + 100 == 333.

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1007 width=148)
```

Man sieht, das immer noch die gesamte Tabelle gelesen werden muss. Es werden weniger Datensätze erwartet (natürlich ist der Wert nur geschätzt und nicht wirklich aussagekräftig). Die Kosten sind durch die zusätzlich benötigte Vergleichszeit etwas gestiegen. Es wird immer noch kein Index verwendet, weil er sich nicht lohnt.

Oft liefert Abfragen jedoch nicht solche Mengen an Daten:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..181.09 rows=49
width=148)
```

Hier ist die Bedingung so, dass nur noch 49 Datensätze erwartet werden. Daher entscheidet der Planer, den Index zu verwenden. Da hier nur 50 Datensätze erwartet werden, ist die Verwendung eines Index billiger, obwohl jeder einzelne Datensatz langsamer geladen wird (Festplatten lesen Folgedaten schneller).

```
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 <
50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148)
```

In diesem etwas komplizierteren Beispiel wird zusätzlich eine zweite Tabelle benutzt, die über einen `Join` verbunden ist. Man sieht, das der Planer einen `Indexscan` ausgewählt hat. Durch den `Join` entsteht ein `Loop` mit zwei Teilen. Zunächst wird die Tabelle tenk1 über den Index durchgearbeitet. Die Kosten sind natürlich die gleichen im Beispiel davor (gleiche Bedingung: `WHERE unique1 < 50`). Mit dem Wert unique2, der aus tenk1 gelesen wurde (genauer gesagt, sind das ja insgesamt 49 Werte!), wird nun ein passender Eintrag in tenk2 gesucht. Der Planer erwartet genau einen Treffer und verwendet daher wieder einen Index (der glücklicherweise auch verfügbar ist). Diese Teilabfrage gilt für einen konstanten Wert unique2 (je einen der insgesamt 49). Damit ist der Zugriff vergleichsweise billig (3).

Die zweite Teilabfrage wird nun für jeden der 49 Werte durchgeführt. Die Kosten sind also 49 * 3 == 147. Dazu kommen die 181 des vorherigen Teils (der die 49 Werte überhaupt erstmal lieferte), macht zusammen 147 + 181 == 328. Dazu kommt noch etwas Rechenzeit für den `Join` (hier ca. 2). Macht dann zusammen 330.

330 sind auch die Kosten, die der Planer für den Loop ausgerechnet hat. Es 49 Datensätze (es wird ja erwartet, das jeweils ein Datensatz passt), nur das die etwas größer sind, also vorher (sind ja durch einen `Join` verbunden).

Der Planer hat sich entschieden, einen `nested-loop join` (etwa: **geschachtelte Schleife**) zu verwenden. Man kann über Variablen den Planer beeinflussen. In der Praxis bringt das so gut wie nie Vorteile. Beispielsweise kann man dem Planer sagen, dass er `nested-loop join` nicht verwenden soll:

```
regression=# set enable_nestloop = off;
SET VARIABLE

***layout: achtung, zwei Zeilen Prompt!
regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 <
50
regression-# AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=181.22..564.83 rows=49 width=296)
  ->  Seq Scan on tenk2 t2
              (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=181.09..181.09 rows=49 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
```

Der Planer kommt nun mit einem anderen Plan. Zunächst werden wieder die 49 Datensätze aus tenk1 mit Indexunterstützung geladen. Die Werte werden nun aber erst alle geladen, und in einem Hash gespeichert. Man sieht das gut an den Anfangskosten für den Hash: sie entsprechen den Gesamtkosten für den Indexscan (da der Hash anschließend gebaut wird, und sehr schnell fertig ist).

Anschließend wird die Tabelle tenk2 sequentiell durchsucht, ob irgendwo der Wert unique2 aus der Tabelle zu einem der im Hash gespeicherten Werte passt. Dies muss ja nun für alle 10.000 Datensätze gemacht werden (`nestloop` ist ja **verboten**).

Sobald mit dem Scan über tenk2 begonnen wurde, sind die ersten Treffer zu erwarten. Die Anfangskosten des Joins entsprechen also den Kosten, die anfallen, bis mit tenk2 begonnen werden kann (vorher kommt ja keine Ausgabe), also den Gesamtkosten des ersten Indexscans. Dazu kommen die 333 für den sequentiellen Scan über tenk2, macht 514.09. Die restlichen 50 gehen für Rechenleistung drauf; schließlich muss mit jedem der 10000 Datensätze eine Test auf den vorher gespeicherten Hash gemacht werden. Die erwarten Kosten sind wesentlich höher als vorhin, daher hat der Planer vorhin auch einen `nestloop` verwendet.

Weitere Variablen, die bestimmte Pläne vermeiden, sind: `ENABLE_HASHJOIN`, `ENABLE_INDEXSCAN`, `ENABLE_MERGEJOIN`, `ENABLE_SEQSCAN` `ENABLE_SORT` und `ENABLE_TIDSCAN`. Auch diese können auf **off** gesetzt werden, um anzuzeigen, dass sie zu vermeiden sind. Wie bereits gesagt, lassen sich nur schwer Fälle konstruieren, wo das was bringt.

`EXPLAIN` kann auch um `ANALYZE` erweitert werden. Dann wird die Abfrage tatsächlich ausgeführt, und auch die wirklichen Werte werden ausgegeben.

```
regression=# EXPLAIN ANALYZE
regression-# SELECT * FROM tenk1 t1, tenk2 t2
regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..330.41 rows=49 width=296)
             (actual time=1.31..28.90 rows=50 loops=1)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..181.09 rows=49 width=148)
              (actual time=0.69..8.84 rows=50 loops=1)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..3.01 rows=1 width=148)
              (actual time=0.28..0.31 rows=1 loops=50)
```

```
Total runtime: 30.67 msec
```

Hier wird die tatsächlich benötigte Zeit in Millisekunden angezeigt. Man erkennt auch, dass der Planer sich gering verschätzt hat, anstatt 49 Datensätzen sind es 50. Es läßt sich abschätzen, das ein **Festplattenzugriff** (die Einheit von EXPLAIN) hier in etwa 10 Millisekunden dauert.

Dies mag als Einführung ausreichen. Das Verstehen dieser Ausgaben erfordert Übung. Man kann so erkennen, ob und wann Indexe verwendet werden, ob sie günstig sind, oder vielleicht gar nicht benötigt sind. Dann sollte man sie löschen, dass spart Zeit bei Aktualisierungen.

5 Ausblick

Es gibt etliches an Dokumentation, die mit *PostgreSQL* mitgeliefert wird.

Die *PostgreSQL* Homepage ist http://www.postgresql.org/. Hier finden sich viele Informationen und sehr viel (englischsprachige) Dokumentation. Natürlich ist auch eine **SQL Referenz** vorhanden.