Spis treści

.....2
Instalacja PostgreSQL w wersji 9.4 na przykładzie CENTOS 6.7.....3
Proces instalacji i inicjalizacji klastra.....3
Automatyczny start klastra bazodanowego razem z serwerem.....8
Ustawianie hasła użytkownika postgres.....9
Dostęp do klastra z sieci.....10
Instalacja i konfiguracja PhpPgAdmin.....13
Korzystanie z PSQL.....15
Łączenie się z bazą danych.....15
Konfiguracja PSQL.....16
Uruchamianie zewnętrznych skryptów.....17
Sprawdzanie struktury tabeli i dostępnych tabel.....18
Uruchamianie i zatrzymywanie klastra.....19
Zmiana parametrów.....23
Zmiana parametrów dla sesji.....23
Przywracanie wartości parametrów.....25
Trwała zmiana parametrów dla klastra.....26
Zmiana parametrów dla innych użytkowników, oraz wybranych baz danych.....26
Sprawdzanie własności bazy danych.....27
Struktura fizyczna i logiczna bazy.....31
Struktura fizyczna – katalogi i pliki.....31
Struktura logiczna.....36
Bazy danych – informacje podstawowe.....36
Sprawdzanie dostępnych baz.....36
Tworzenie baz danych.....37
Bazy danych a struktura katalogów i pliki związane z obiektami bazy.....39
Kasowanie baz danych.....41
Przestrzenie tabel (tablespace).....42
Schematy.....46
Użytkownicy i uprawnienia.....48
Użytkownicy – tworzenie, kasowanie i autoryzacja.....48
Tworzenie użytkownika.....48
Kasowanie użytkownika.....50
Zmiana własności użytkownika.....51
Grupy użytkowników i masowe zarządzanie uprawnieniami.....52
Uprawnienia.....54
Nadawanie uprawnień.....54
Odbieranie uprawnień.....56
Sprawdzanie uprawnień.....57
Sesje użytkowników i ich rozłączanie.....59
Transakcje, poziomy izolacji i blokady.....61
Zarządzanie transakcjami.....61
Niepożądane zjawiska związane z transakcyjnością.....63
Brudny odczyt.....63
Odczyty nie dające się powtórzyć.....63
Odczyty widmo.....63
Poziomy izolacji.....64
Blokady.....65

Zasada działania blokad....65
Jawne blokowanie wierszy i tabel....65
Mechanizmy wewnętrzne transakcyjności i operacja VACUUM....67
Zarządzanie obiekami....69
Typy danych....69
Typy znakowe....69
Typy liczbowe....70
Typ logiczny – boolean....70
Typy daty i czasu....71
Specjalne typy PostgreSQL i typ Blob....71
Tabele....72
Tworzenie tabel....72
Ograniczenia kolumn....73
Ograniczenia tabel....74
Kasowanie tabel....74
Tabele tymczasowe....74
Ograniczenia kluczy obcych....75
Ograniczenie ON DELETE/UPDATE CASCADE i ON DELETE SET NULL....75
Ograniczenie DEFERRABLE....77
Widoki....78
Backup i odtwarzanie w PostgreSQL....79
Backup lokalny i zdalny z użyciem narzędzia PG_DUMP....82
Odtwarzanie lokalne i zdalne bazy danych....85
Szybkie kopiowanie baz między dwoma klastrami....88
Błędy podczas odtwarzania....89
Backup i odtwarzanie całego klastra....90
Backup plików danych na poziomie fizycznym....92
Archiwizacja ciągła i przywracanie do puntu tuż przed awarią....96
Przywracanie do punktu w czasie....114

Instalacja PostgreSQL w wersji 9.4 na przykładzie CENTOS 6.7

Proces instalacji i inicjalizacji klastra

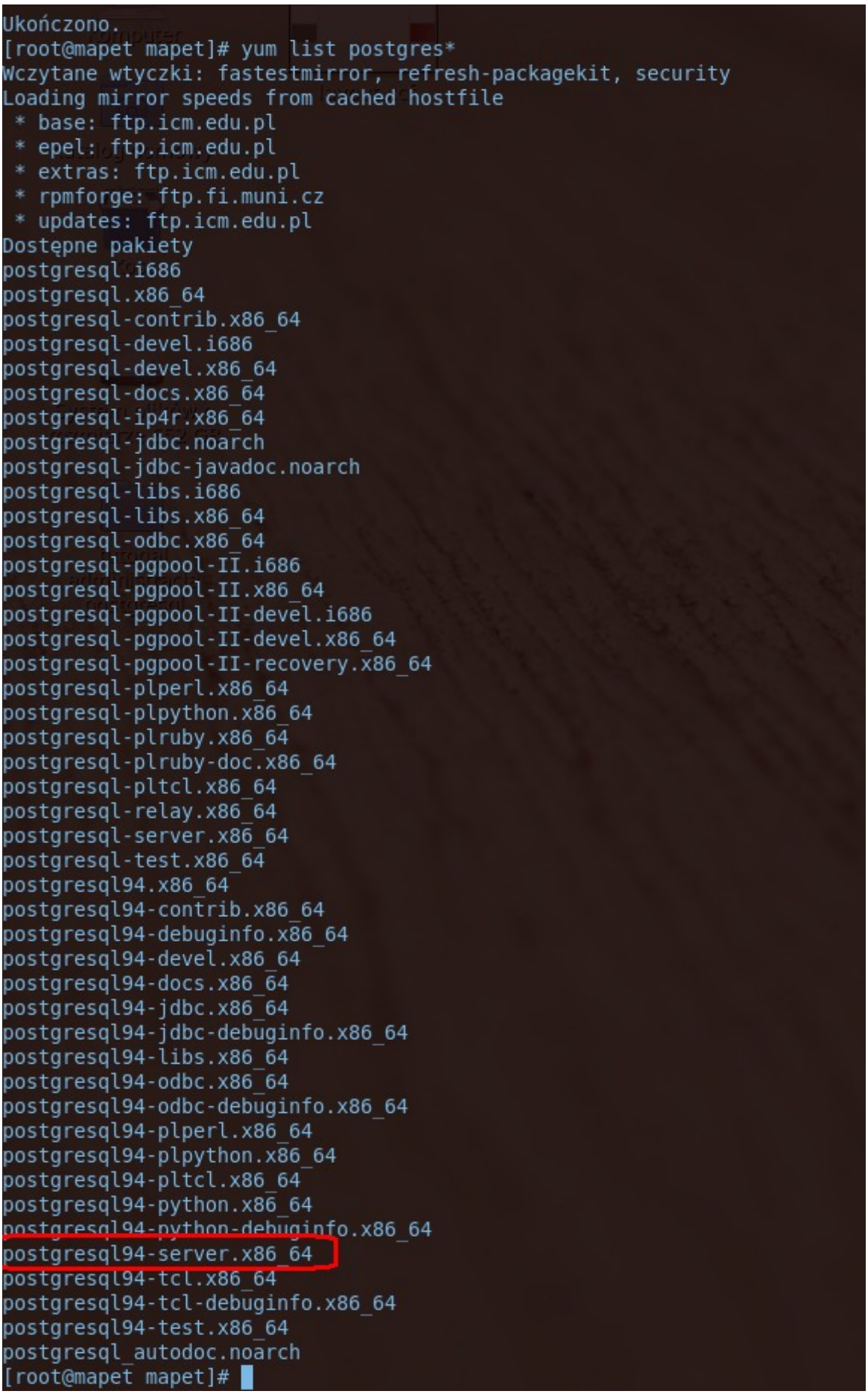
W pierwszej kolejności logujemy się jako root i wykonujemy następujące polecenia, aby zainstalować serwer PostgreSQL.

```
yum localinstall http://yum.postgresql.org/9.4/redhat/rhel-6-x86_64/pgdg-centos94-9.4-1.noarch.rpm
```

Po tej operacji wydając polecenie

```
yum list postgresql94
```

powinieneś widzieć pakiet serwera PostgreSQL w najnowszej wersji:


Ukończono.
[root@mapet mapet]# yum list postgres*
Wczytane wtyczki: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
 * base: ftp.icm.edu.pl
 * epel: ftp.icm.edu.pl
 * extras: ftp.icm.edu.pl
 * rpmforge: ftp.fi.muni.cz
 * updates: ftp.icm.edu.pl
Dostępne pakiety
postgresql.i686
postgresql.x86_64
postgresql-contrib.x86_64
postgresql-devel.i686
postgresql-devel.x86_64
postgresql-docs.x86_64
postgresql-ip4r.x86_64
postgresql-jdbc.noarch
postgresql-jdbc-javadoc.noarch
postgresql-libs.i686
postgresql-libs.x86_64
postgresql-odbc.x86_64
postgresql-pgpool-II.i686
postgresql-pgpool-II.x86_64
postgresql-pgpool-II-devel.i686
postgresql-pgpool-II-devel.x86_64
postgresql-pgpool-II-recovery.x86_64
postgresql-plperl.x86_64
postgresql-plpython.x86_64
postgresql-plruby.x86_64
postgresql-plruby-doc.x86_64
postgresql-pltcl.x86_64
postgresql-relay.x86_64
postgresql-server.x86_64
postgresql-test.x86_64
postgresql94.x86_64
postgresql94-contrib.x86_64
postgresql94-debuginfo.x86_64
postgresql94-devel.x86_64
postgresql94-docs.x86_64
postgresql94-jdbc.x86_64
postgresql94-jdbc-debuginfo.x86_64
postgresql94-libs.x86_64
postgresql94-odbc.x86_64
postgresql94-odbc-debuginfo.x86_64
postgresql94-plperl.x86_64
postgresql94-plpython.x86_64
postgresql94-pltcl.x86_64
postgresql94-python.x86_64
postgresql94-python-debuginfo.x86_64
postgresql94-server.x86_64
postgresql94-tcl.x86_64
postgresql94-tcl-debuginfo.x86_64
postgresql94-test.x86_64
postgresql_autodoc.noarch
[root@mapet mapet]#

Możesz teraz zainstalować serwer:

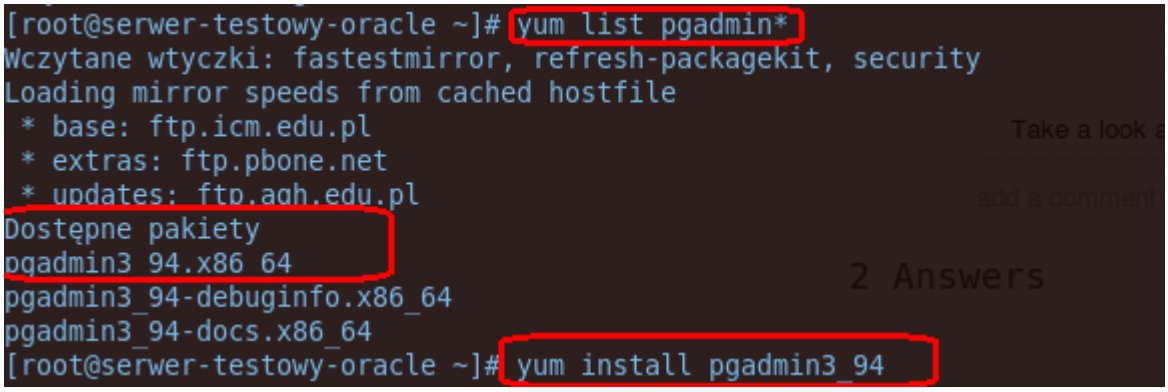
```
yum install postgresql94-server
```

Wraz z serwerem instalowane są podstawowe narzędzia do zarządzania bazą, między innymi program psql, ale ponieważ jest to narzędzie konsolowe to nie jest najwygodniejsze w użyciu. Zainstalujemy więc teraz aplikację kliencką pgadmin3:

```
yum install pgadmin3
```

Gdyby okazało się że nie widzi takiego pakietu dostępnego, wywołaj poniższe polecenie:

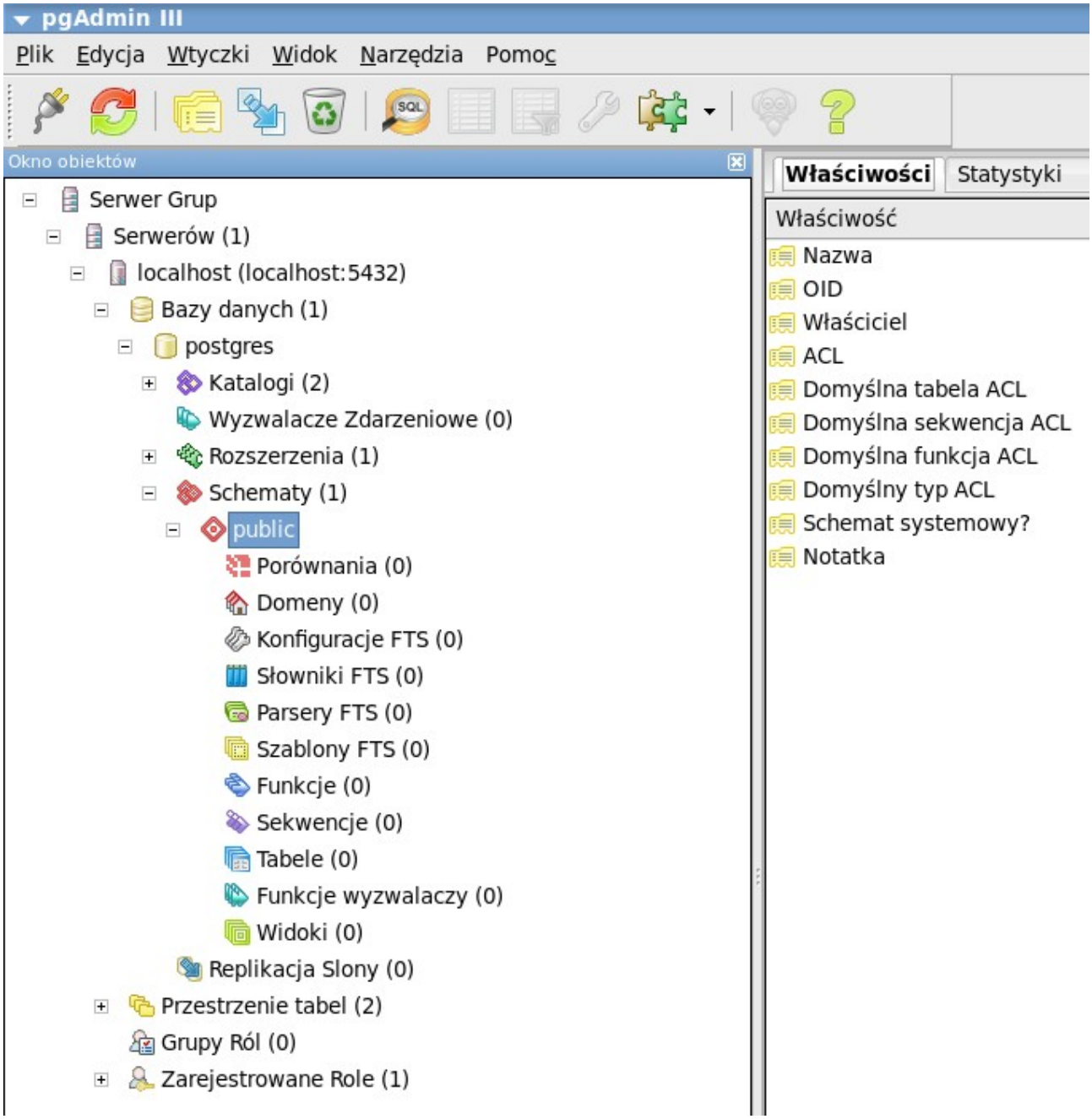
```
yum list pgadmin*
```



Wybierasz teraz pakiet który masz dostępny i go instalujesz. U mnie trzeba było wykonać polecenie:

```
yum install pgadmin3_94
```

Interfejs programu PGADMIN3:



Niezbędne pakiety mamy zainstalowane. Na ten moment nie mamy jeszcze stworzonej żadnej bazy, a katalog przeznaczony na pliki danych jest pusty. Możesz to sprawdzić wydając polecenie:

```
ls /var/lib/pgsql/9.4/data/
```

Gdyby okazało się że z jakiegoś powodu taki katalog nie istnieje wykonaj poniższe polecenia:

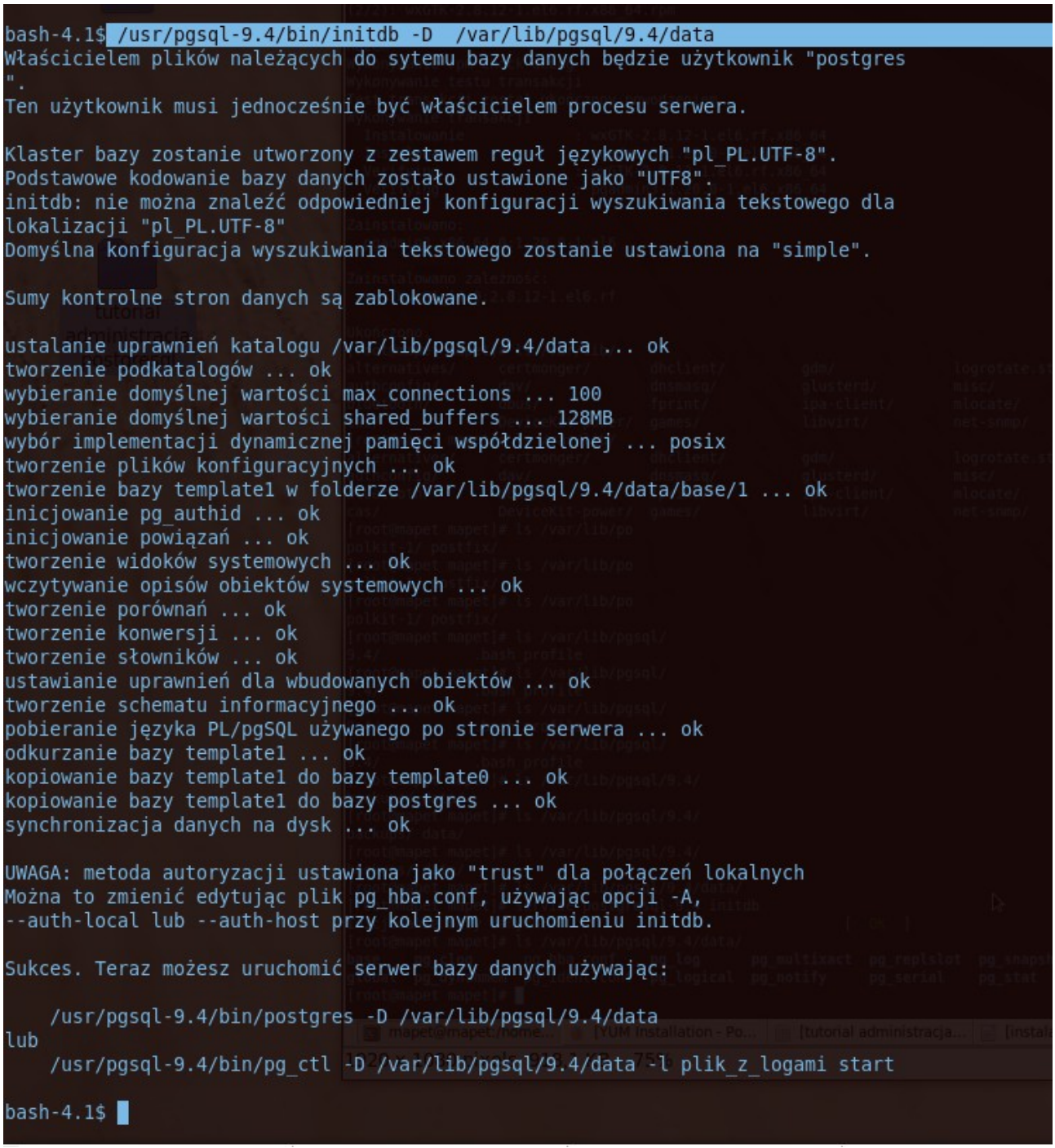
```
mkdir /var/lib/pgsql/9.4/data
chown postgres /var/lib/pgsql/9.4/data
```

Przeloguj się teraz na użytkownika postgres. Dokonamy inicjalizacji. To ważne z jakiego użytkownika odpalamy proces inicjalizacji, ponieważ to do niego będą należały pliki i katalogi

```
service postgresql-9.4 initdb
```

alternatywnie do service postgresql-9.4 initdb tyle że z poziomu użytkownika systemowego postgres:

```
/usr/pgsql-9.4/bin/initdb -D  /var/lib/pgsql/9.4/data
```



Automatyczny start klastra bazodanowego razem z serwerem

Aby klaster Postgresa uruchamiał się wraz z systemem operacyjnym wystarczy wydać polecenie:

```
chkconfig postgresql-9.4 on
```

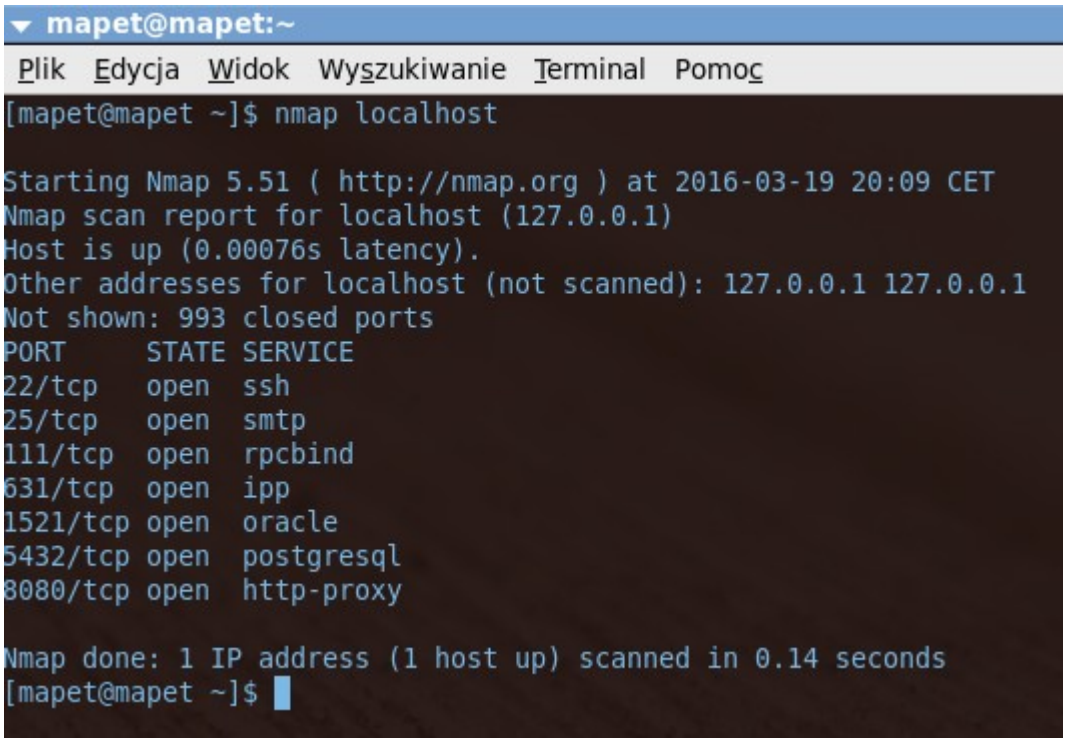
Lub gdy zechcesz to wyłączyć:

```
chkconfig postgresql-9.4 off
```

Zanim zrestartujesz serwer możesz wydać komendę:

```
nmap localhost
```

Na zwróconej liście nie będzie otwartego portu 5432 – taki jest domyślny dla Postgresa. Zrestartuj serwer aby się upewnić że Posgres wstaje. Wydaj ponownie komendę nmap, ale tym razem powinienieś widzieć otwarty port:



Ustawianie hasła użytkownika postgres

Nie mniej ważnym etapem wstępnej konfiguracji jest ustawienie hasła użytkownika postgres. Zaloguj się jako użytkownik systemowy postgres. Uruchom program psql po prostu wpisując jego nazwę w konsolę. Gdyby jednak okazało się że program ten nie jest dostępny (w zasadzie to po prostu nie jest dostępny w zmiennej środowiskowej PATH) to wywołaj go bezpośrednio z katalogu z binariami:

```
/usr/pgsql-9.4/bin/psql
```

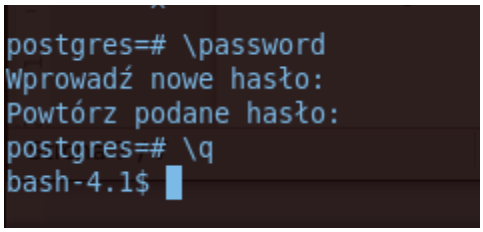
w samym programie psql wprowadź frazę:

```
\password
```

i podaj swoje nowe hasło dwukrotnie. Aby wyjść z programu psql wystarczy wprowadzić

```
\q
```

i zatwierdzić



W tej chwili już powinieneś móc zalogować się do klastra z użyciem programu PgAdmin3.

Dostęp do klastra z sieci

Domyślnie klaster PostgreSQL dostępny jest tylko z lokalnego hosta. Nie ma możliwości dostępu do niego z innych hostów. W tym rozdziale zajmiemy się umożliwieniem dostępu z całej sieci lub z wybranych hostów.

Aby ten efekt osiągnąć musimy wykonać następujące rzeczy:

1. Wprowadzić zmianę w pliku postgresql.conf aby klaster nasłuchiwał połączeń spoza lokalnego kosta
2. Wprowadzić zmianę w pliku pg_hba.conf , a konkretniej dodać regułę która pozwoli na dostęp z wybranych hostów
3. Przeładować konfigurację
4. Zadbać o to by nie blokował nas firewall.

Zarówno plik postgresql.conf jak i pg_hba.conf znajdują się w systemie CentOS w katalogu w którym znajdują się też pliki danych tj. /var/lib/pgsql/data

W pliku postgresql.conf odnajdujemy taki fragment:



i linijkę zawierającą parametr listen_adddresses po pierwsze odkomentowujemy, po drugie w miejsce localhost wprowadzamy *

Poniżej widać też konfigurację portu nasłuchu, który również możemy przy okazji zmienić. Dopóki linijka z portem jest zakomentowana, serwer i tak będzie chodził na domyślnym porcie 5432.

Otwieramy teraz plik pg_hba.conf i wprowadzamy linijkę o kształcie widocznym na samym dole poniższej ilustracji.



Taka konfiguracja sprawi, że dostęp do bazy będzie zewsząd. Pierwszy parametr „host” określa połączenie sieciowe, drugi bazę/bazy danych do których dostęp konfigurujemy, trzeci użytkowników, czarty adresy sieciowe lub podsieć, a piąty metodę autoryzacji. Wartość „md5” oznacza, że logowanie będzie się odbywało w sposób zaszyfrowany. Dla przykładu można też wprowadzić „password”, ale będzie to oznaczało przesyłanie niezaszyfrowanego hasła. Wartość „trust” oznacza logowanie bez autoryzacji. Gdybyśmy zechcieli pozwolić na dostęp tylko z sieci lokalnej, nasz wpis powinien wyglądać przykładowo tak:

host all all 192.168.0.0/24 md5

Konfiguracja wprowadzona. Przyszedł więc czas na przeładowanie ustawień. W tym celu wywołujemy polecenie

/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data/ reload

Taki sposób przeładowania z jednej strony nie powoduje restartu klastra i nie zrywa połączeń, jednak przeładowanie konfiguracji może potrwać nawet kilka minut.

Alternatywnie możemy zatrzymać i uruchomić ponownie klaster w ten sposób z poziomu użytkownika postgres:

/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data/ stop

/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data/ start

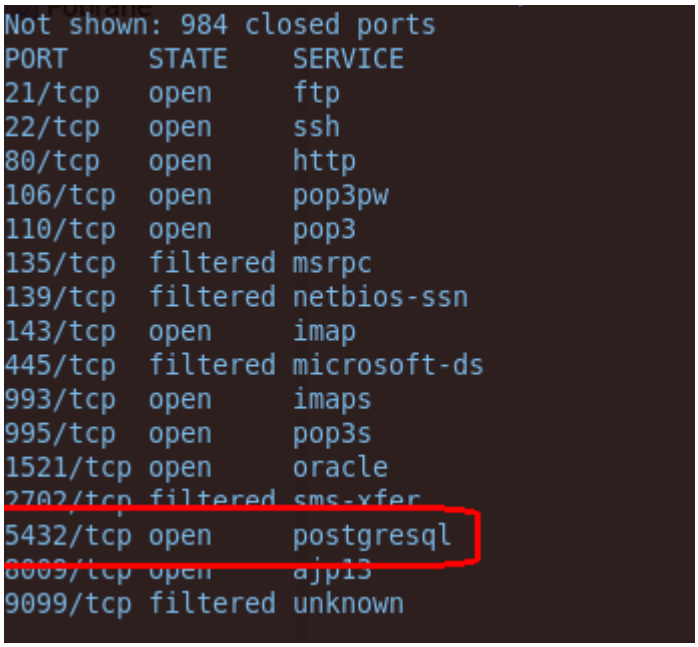
lub z roota:

/etc/init.d/postgresql-9.4 restart

albo

service postgresql-9.4 restart

Po tej operacji możesz przy użyciu narzędzia nmap sprawdzić z innego hosta czy port jest otwarty:



Instalacja i konfiguracja PhpPgAdmin

PhpPgAdmin to klient PostgreSQL dostępny przez przeglądarkę. Bardzo wygodna rzecz jeśli chcemy zarządzać naszym serwerem zdalnie, ale nie chcemy otwierać portu „na świat”.

Zaczniemy od instalacji niezbędnych pakietów. Wielkość liter ma znaczenie!

```
yum install epel-release

yum update
```

```
 yum install  httpd phpPgAdmin
```

i uruchomienia serwera http:

```
service httpd start
```

Aby można się było z użyciem phpPgAdmina zalogować do klastra, musimy go jeszcze skonfigurować:

Z roota: `setsebool -P httpd_can_network_connect_db 1`

Edycja pliku konfiguracyjnego phppgadmin:

```
nano /etc/phpPgAdmin/config.inc.php
```

Musimy teraz znaleźć i podmienić w tym pliku następujące elementy:

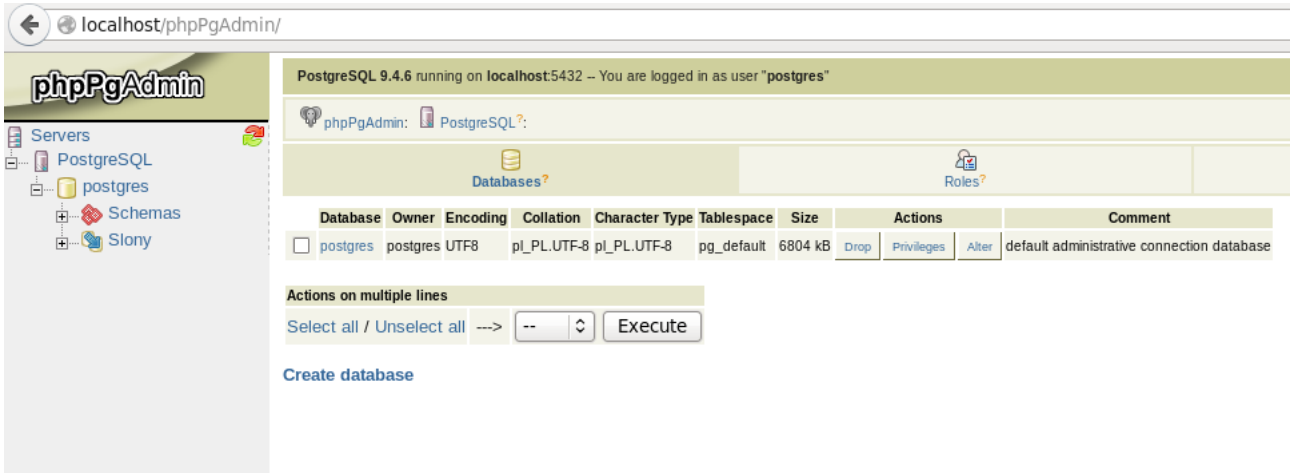
Zamienić:
$conf['servers'][0]['host'] = '';
na
$conf['servers'][0]['host'] = 'localhost';

i
 $conf['extra_login_security'] = true;
na :
$conf['extra_login_security'] = false;

Możemy teraz wejść pod poniższy adres:

http://localhost/phpPgAdmin/

i powinniśmy zobaczyć widok jak poniżej.


localhost/phpPgAdmin/
phpPgAdmin
Servers
PostgreSQL
postgres
Schemas
Slony
PostgreSQL 9.4.6 running on localhost:5432 -- You are logged in as user "postgres"
phpPgAdmin: PostgreSQL?:
Databases?
Roles?
Database Owner Encoding Collation Character Type Tablespace Size Actions Comment
postgres postgres UTF8 pl_PL.UTF-8 pl_PL.UTF-8 pg_default 6804 kB Drop Privileges Alter default administrative connection database
Actions on multiple lines
Select all / Unselect all ---> -- Execute
Create database

Korzystanie z PSQL

Łączenie się z bazą danych

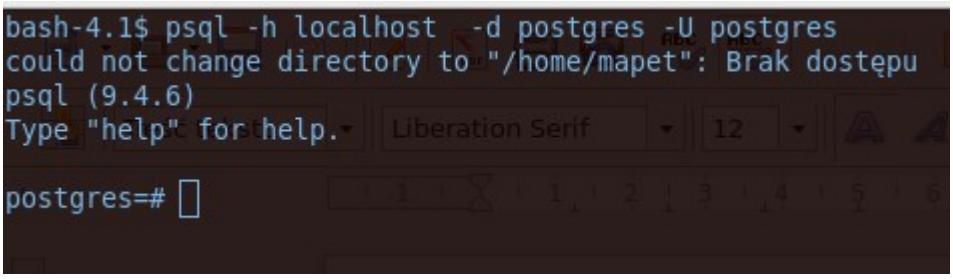
Do czasu kiedy nie ustawimy hasła do użytkownika postgres, wystarczy w konsoli po prostu wpisać :

psql

i to nas automatycznie zaloguje. Jeśli jednak ustawimy hasło, a zrobiliśmy to podczas instalacji musimy wykonać nieco bardziej złożone polecenie. Omówimy je za chwilę, najpierw jednak mała uwaga odnośnie dostępu do programu psql. Korzystać z tego programu najlepiej z poziomu użytkownika systemowego postgres. Gdyby się okazało że nie mamy tego programu w zmiennej $PATH, program ten znajduje się pod ścieżką /usr/pgsql-9.4/bin/

Poniższa komenda łączy się z klastrem zainstalowanym na lokalnym serwerze (-h localhost) i łączy do bazy danych postgres (-d postgres) logując się jako użytkownik bazodanowy postgres (-U postgres).

psql -h localhost -d postgres -U postgres



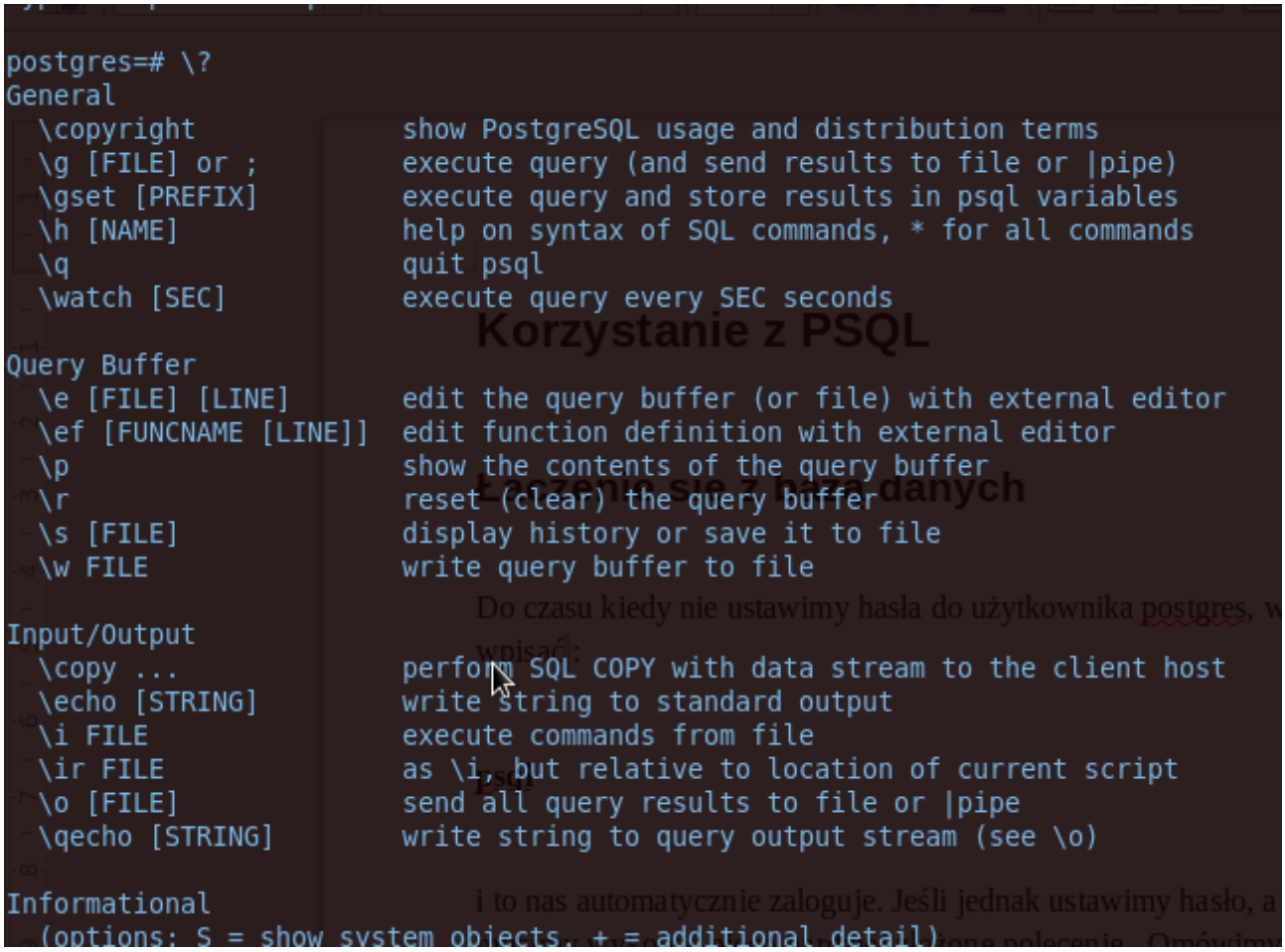
Zostaniemy zapytani o hasło, po czym zalogujemy się do bazy. Aby wyjść z programu wystarczy wydać polecenie **\q**

Jeśli PostgreSQL nasłuchuje na innym niż domyślnym porcie (5432), możesz dodać przy łączeniu dodatkowy parametr -p i określić numer portu w ten sposób:

psql -h localhost -p 5432 -d postgres -U postgres

Konfiguracja PSQL

Aby zobaczyć dostępne komendy programu psql wystaczy wpisać \?



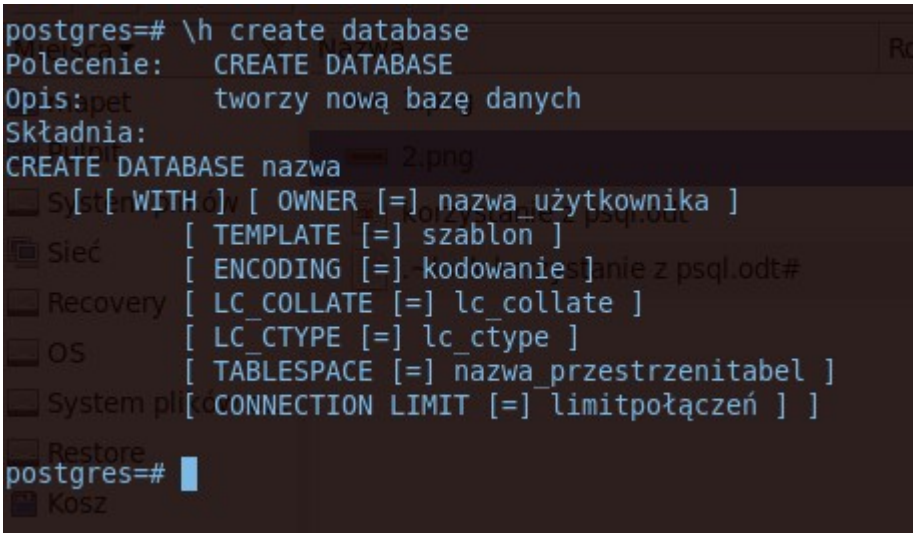
```
postgres=# \?
General
  \copyright             show PostgreSQL usage and distribution terms
  \g [FILE] or ;         execute query (and send results to file or |pipe)
  \gset [PREFIX]         execute query and store results in psql variables
  \h [NAME]              help on syntax of SQL commands, * for all commands
  \q                     quit psql
  \watch [SEC]           execute query every SEC seconds

Query Buffer
  \e [FILE] [LINE]       edit the query buffer (or file) with external editor
  \ef [FUNCNAME [LINE]]  edit function definition with external editor
  \p                     show the contents of the query buffer
  \r                     reset (clear) the query buffer
  \s [FILE]              display history or save it to file
  \w FILE                write query buffer to file

Input/Output
  \copy ...              perform SQL COPY with data stream to the client host
  \echo [STRING]         write string to standard output
  \i FILE                execute commands from file
  \ir FILE               as \i, but relative to location of current script
  \o [FILE]              send all query results to file or |pipe
  \qecho [STRING]        write string to query output stream (see \o)

Informational
  (options: S = show system objects, + = additional detail)
```

Jeśli chcemy poznać składnię jakiegoś polecenia SQL możemy wywołać \h polecenie sql:



```
postgres=# \h create database
Polecenie:   CREATE DATABASE
Opis:        tworzy nową bazę danych
Składnia:
CREATE DATABASE nazwa
    [ [ WITH ] [ OWNER [=] nazwa_użytkownika ]
           [ TEMPLATE [=] szablon ]
           [ ENCODING [=] kodowanie ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] nazwa_przestrzenitabel ]
           [ CONNECTION LIMIT [=] limitpołączeń ] ]

postgres=#
```

Uruchamianie zewnętrznych skryptów

Aby zalogować się do bazy i wykonać jakiś skrypt wystarczy przy wywoływaniu psql dodać parametr -f i nazwę pliku np.:

psql -d postgres -U postgres -h localhost -f /home/mapet/skrypt.sql

W podanym przykładzie w skrypcie znajduje się jedynie komenda „select version()” zwracająca nam informacje o wersji serwera. Efekt działania:



Możemy też zamiast umieszczać polecenie w skrypcie podać je z użyciem parametru -c:

psql -d postgres -U postgres -h localhost -c "select version()"



Jeśli już jesteś zalogowany do psql, możesz wywołać wykonanie skryptu z użyciem komendy \i np.:

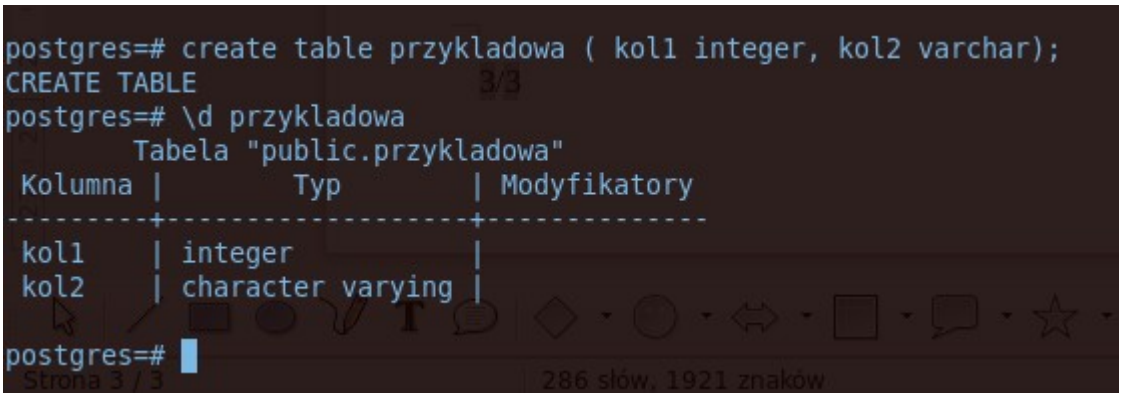
\i /home/mapet/skrypt.sql



Sprawdzanie struktury tabeli i dostępnych tabel

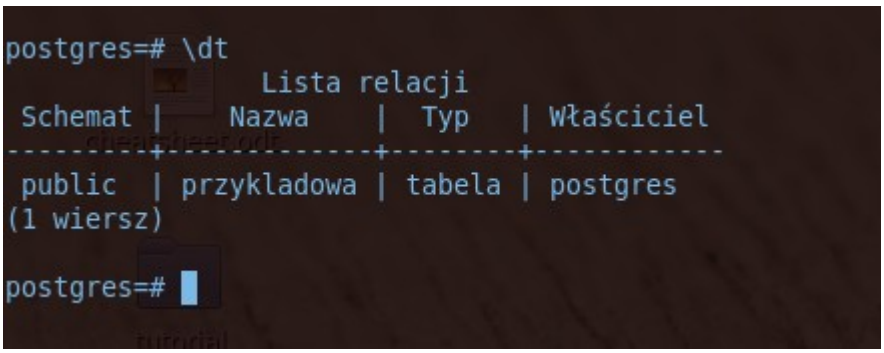
Na potrzeby tego przykładu stworzyłem przykładową tabelkę zawierającą dwie kolumny. Aby wyświetlić listę kolumn i ich właściwości wystarczy w psql wpisać:

\d nazwatabeli



Jeśli chcemy się dowiedzieć jakie w ogóle mamy tabele dostępne w danej bazie wydajemy komendę :

\dt



Uruchamianie i zatrzymywanie klastra

Klaster możemy uruchamiać i zatrzymywać na różne sposoby i z różnym skutkiem. Możemy to zrobić tak jak dla każdej usługi w systemie Linux:

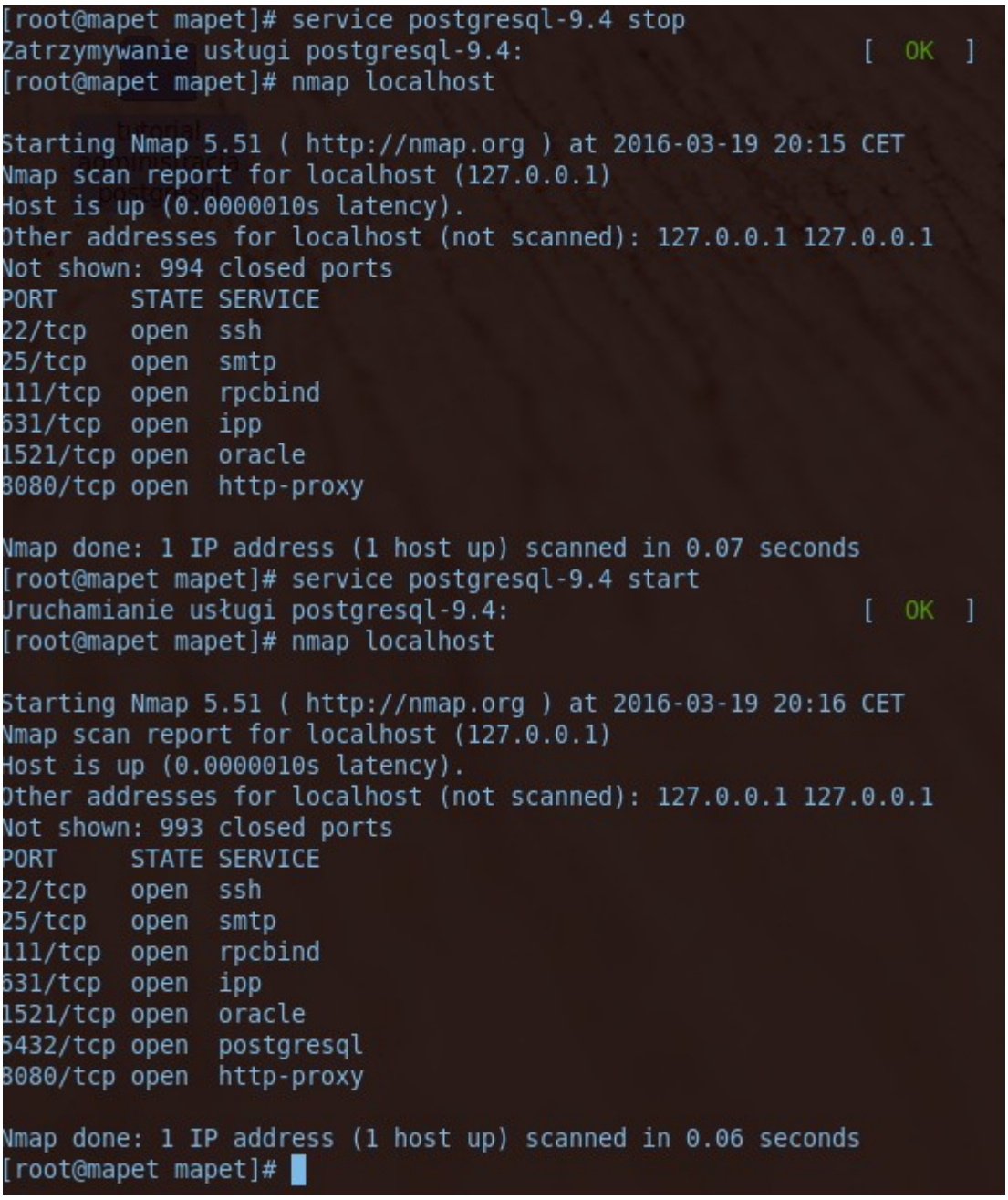
Zatrzymywanie klastra:

```
service postgresql-9.4 stop
```

Startowanie klastra:

```
service postgresql-9.4 start
```

jednak musimy pamiętać że rozłączy to wszystkie połączenia jakie będą podpięte w danym momencie do bazy. Klaster nie poczeka na rozłączenie się użytkowników, ani nie pozwoli im dokończyć aktualnie wykonywanej operacji. Przy zatrzymywaniu i uruchamianiu możesz sprawdzać przy użyciu narzędzia nmap czy port jest otwarty czy nie. Poniżej zrzut ekranu z zatrzymywania i uruchamiania klastra z następującym sprawdzeniem otwartości portu. Jak się nie trudno domyślić, aby wykonać operacje na usługach trzeba mieć do tego uprawnienia systemowe.


[root@mapet mapet]# service postgresql-9.4 stop
Zatrzymywanie usługi postgresql-9.4: [OK]
[root@mapet mapet]# nmap localhost
Starting Nmap 5.51 (http://nmap.org) at 2016-03-19 20:15 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000010s latency).
Other addresses for localhost (not scanned): 127.0.0.1 127.0.0.1
Not shown: 994 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp open smtp
111/tcp open rpcbind
631/tcp open ipp
1521/tcp open oracle
8080/tcp open http-proxy
Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
[root@mapet mapet]# service postgresql-9.4 start
Uruchamianie usługi postgresql-9.4: [OK]
[root@mapet mapet]# nmap localhost
Starting Nmap 5.51 (http://nmap.org) at 2016-03-19 20:16 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000010s latency).
Other addresses for localhost (not scanned): 127.0.0.1 127.0.0.1
Not shown: 993 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp open smtp
111/tcp open rpcbind
631/tcp open ipp
1521/tcp open oracle
5432/tcp open postgresql
8080/tcp open http-proxy
Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds
[root@mapet mapet]#

Alternatywnie do ww ale z poziomu użytkownika postgres możemy zatrzymywać i uruchamiać klaster poniższymi komendami:

Poniższy sposób zatrzymywania będzie czekać na rozłączenie się użytkowników, ale nie pozwolą na podpinanie nowych sesji:

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data stop
```

uruchamianie:

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data start
```

A taka wersja rozłączy wszystkie sesje i wyłączy serwer szybko:

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data -m fast stop
```

Poniższe polecenie spowoduje zamknięcie klastra w podobny sposób jakbyśmy po prostu odłączyli prąd z serwera:

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data stop -m immediate
```

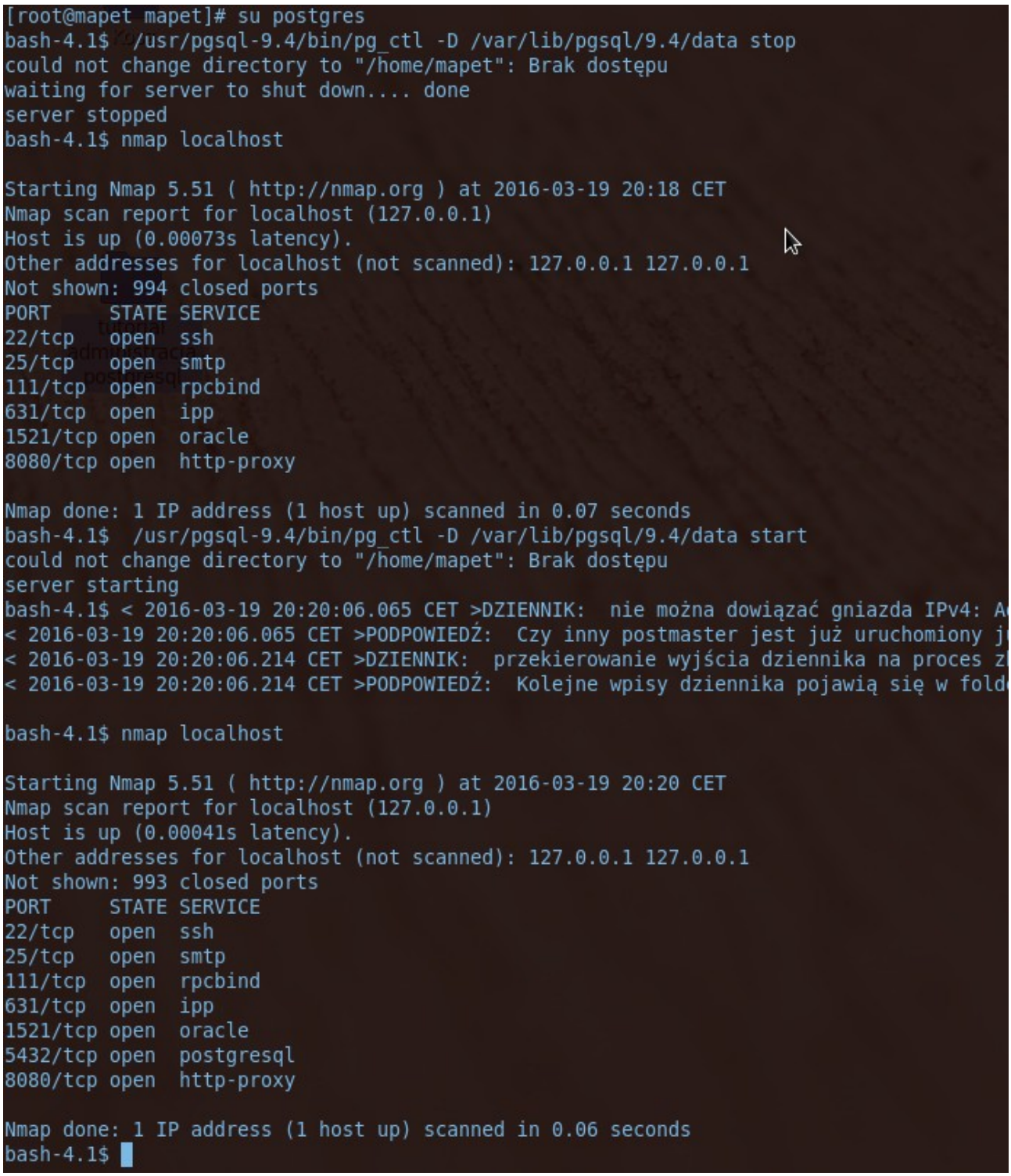
Klaster możemy też restartować:

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data restart
```

lub

```
/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data restart -m fast
```

Pierwsze czeka na odłączenie się wszystkich użytkowników, drugie restartuje serwer bez oczekiwania na zakończenie sesji użytkowników.

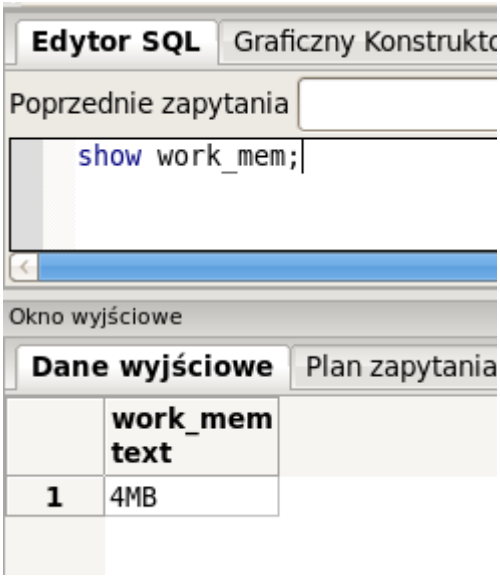

[root@mapet mapet]# su postgres
bash-4.1$ /usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data stop
could not change directory to "/home/mapet": Brak dostępu
waiting for server to shut down.... done
server stopped
bash-4.1$ nmap localhost
Starting Nmap 5.51 (http://nmap.org) at 2016-03-19 20:18 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00073s latency).
Other addresses for localhost (not scanned): 127.0.0.1 127.0.0.1
Not shown: 994 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp open smtp
111/tcp open rpcbind
631/tcp open ipp
1521/tcp open oracle
8080/tcp open http-proxy
Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
bash-4.1$ /usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data start
could not change directory to "/home/mapet": Brak dostępu
server starting
bash-4.1$ < 2016-03-19 20:20:06.065 CET >DZIENNIK: nie można dowiązać gniazda IPv4: A
< 2016-03-19 20:20:06.065 CET >PODPOWIEDŹ: Czy inny postmaster jest już uruchomiony j
< 2016-03-19 20:20:06.214 CET >DZIENNIK: przekierowanie wyjścia dziennika na proces z
< 2016-03-19 20:20:06.214 CET >PODPOWIEDŹ: Kolejne wpisy dziennika pojawią się w fold
bash-4.1$ nmap localhost
Starting Nmap 5.51 (http://nmap.org) at 2016-03-19 20:20 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00041s latency).
Other addresses for localhost (not scanned): 127.0.0.1 127.0.0.1
Not shown: 993 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp open smtp
111/tcp open rpcbind
631/tcp open ipp
1521/tcp open oracle
5432/tcp open postgresql
8080/tcp open http-proxy
Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds
bash-4.1$

Zmiana parametrów

Zmiana parametrów dla sesji

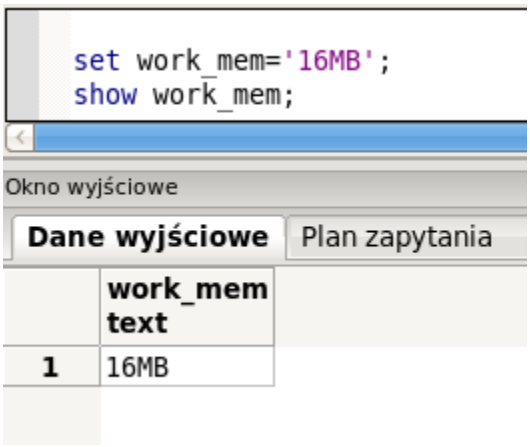
Aby sprawdzić aktualną wartość parametru systemowego wystarczy posłużyć się komendą SHOW:

show nazwa_parametru;



Chcąc zmienić wartość parametru dla sesji korzystamy z komendy SET. Pamiętaj jedynie że zmiana w ten sposób będzie obowiązywała tylko do momentu restartu!

set work_mem='16MB';



Gdybyśmy zechcieli zmienić jakiś parametr tylko dla trwającej transakcji stosujemy komendę SET

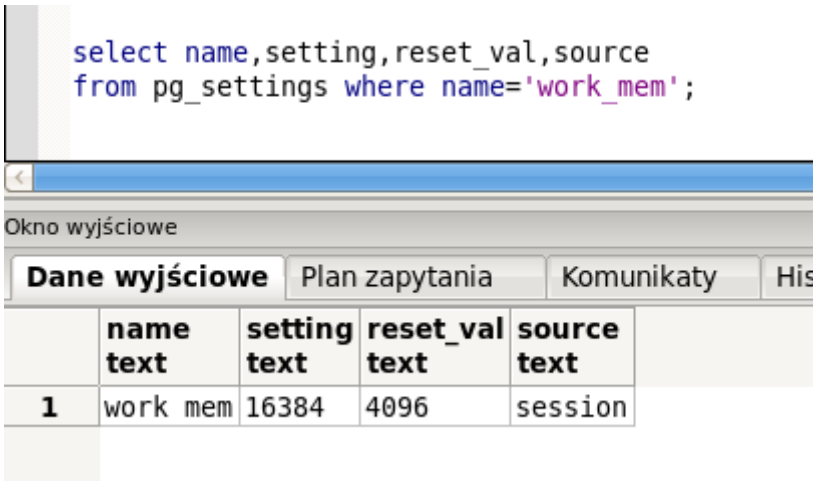
LOCAL:

SET LOCAL work_mem='64MB';

Wartość parametru i zakres jego obowiązywania można sprawdzić odwołując się do zawartości tabeli pg_settings:

select name,setting,reset_val,source

from pg_settings where name='work_mem';



Ograniczyłem wyświetlanie wyników tylko do zmienianego parametru. W kolumnie source zobaczysz zakres obowiązywania parametru.

Wykonaj też zapytanie : select * from pg_settings; Zobaczysz wylistowane wszystkie parametry wraz z opisami czego dotyczą!

Przywracanie wartości parametrów

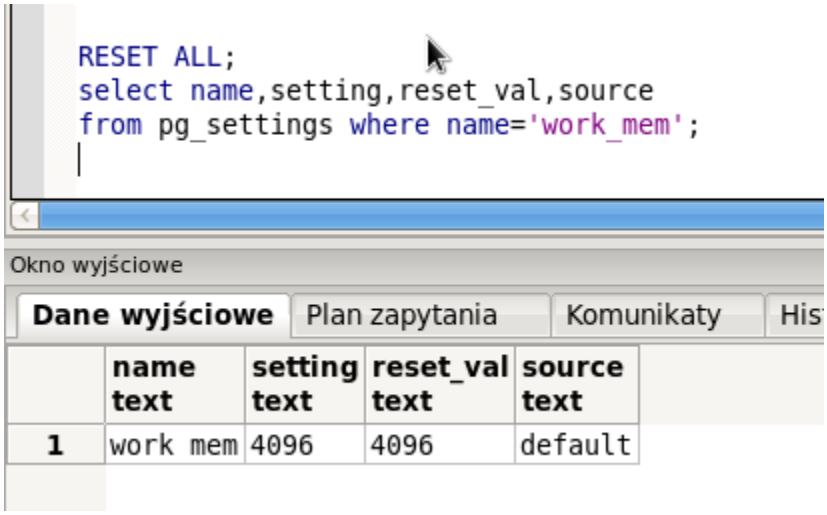
Przywrócić wartość parametru do pierwotnej wartości możemy z użyciem komendy:

RESET work_mem;

Albo dla wszystkich parametrów:

RESEL ALL;

Po wykonaniu tej ostatniej możesz ponownie zajrzeć do słownika pg_settings i przyjrzeć się zawartości kolumny source. Tym razem zobaczysz tam „default” co oznacza że wartość parametru została przywrócona do wartości domyślnej:



Trwała zmiana parametrów dla klastra

Gdybyś zechciał zmienić jakiś parametr w sposób trwały, zmień go w pliku parametrów postgresql.conf i przeładuj konfigurację. Gdyby się zdarzyło że przypadkowo dwukrotnie wprowadziłeś ten sam parametr do pliku, zostanie uwzględnione ostatnie w kolejności ustawienie.

Ciekawostka: w pliku parametrów postgresql.conf można stosować dyrektywę INCLUDE dzięki której możesz „wciągać" zewnętrzne pliki. Pozwala to na przykład odseparować tematycznie konfiguracje do osobnych plików.

Zmiana parametrów dla innych użytkowników, oraz wybranych baz danych

Jeśli zechcesz zmienić parametr dla wybranej bazy danych wywołaj:

alter database postgres set work_mem='32MB';

Dla wybranego użytkownika który jest podłączony:

alter role mapet set work_mem='32MB';

Dla wybranego użytkownika w wybranej bazie danych:

alter role mapet in database postgres set work_mem='32MB';

Sprawdzanie własności bazy danych

Sprawdzanie wersji serwera:

select version();

```
postgres=# select version();
                                                  version
-------------------------------------------------------------------------------------------------------------
 PostgreSQL 9.4.6 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-16), 64-bit
(1 wiersz)
```

Sprawdzanie daty i godziny uruchomienia klastra:

select pg_postmaster_start_time();

```
postgres=# select pg_postmaster_start_time();
   pg_postmaster_start_time
-------------------------------
 2016-03-22 14:58:35.59026+01
(1 wiersz)
```

Sprawdzanie aktualnej daty i godziny:

select current_timestamp;

```
postgres=# select current_timestamp;
              now
-------------------------------
 2016-03-22 15:20:36.72709+01
(1 wiersz)
```

Sprawdzanie od jakiego czasu uruchomiony jest klaster:

select current_timestamp-pg_postmaster_start_time();

```
postgres=# select current_timestamp - pg_postmaster_start_time();
    ?column?
-----------------
 00:22:38.242609
(1 wiersz)

postgres=#
```

Sprawdzanie dostępnych w klastrze baz:

psql -l -U postgres -h localhost -d postgres

```
bash-4.1$ psql -l -U postgres -h localhost -d postgres
                                  Lista baz danych
   Nazwa    | Właściciel | Kodowanie | Porównanie  |    Ctype    |  Uprawnienia dostępu
------------+------------+-----------+-------------+-------------+-----------------------
 postgres   | postgres   | UTF8      | pl_PL.UTF-8 | pl_PL.UTF-8 |
 template0  | postgres   | UTF8      | pl_PL.UTF-8 | pl_PL.UTF-8 | =c/postgres          +
            |            |           |             |             | postgres=CTc/postgres
 template1  | postgres   | UTF8      | pl_PL.UTF-8 | pl_PL.UTF-8 | =c/postgres          +
            |            |           |             |             | postgres=CTc/postgres
(3 wiersze)
```

Alternatywnie będąc już zalogowanym:

select datname from pg_database;

```
bash-4.1$ psql  -U postgres -h localhost -d postgres
psql (9.4.6)
Wpisz "help" by uzyskać pomoc.

postgres=# select datname from pg_database;
  datname
-----------
 template1
 template0
 postgres
(3 wiersze)
```

Sprawdzanie wielkości bazy danych:

select pg_database_size(current_database());

```
postgres=# select pg_database_size(current_database());
 pg_database_size
------------------
          6975252
(1 wiersz)
```

Wartość podana jest w bajtach, więc wyliczymy sobie z tego megabajty:

select pg_database_size(current_database())/1024/1024;

```
postgres=# select pg_database_size(current_database())/1024/1024;
 ?column?
----------
        6
(1 wiersz)
```

Takie sprawdzanie pokazuje jednak tylko wielkość bazy do której jesteśmy akurat podpięci. Gdybyśmy zechcieli sprawdzić wielkość wszystkich dostępnych baz:

select datname, pg_database_size(datname) from pg_database group by datname;

```
postgres=# select datname, pg_database_size(datname) from pg_database group by datname;
  datname  | pg_database_size
-----------+------------------
 postgres  |          6975252
 template0 |          6840836
 template1 |          6967060
(3 wiersze)
```

lub wersja podana w megabajtach:

select datname, pg_database_size(datname)/1024/1024 from pg_database group by datname;

```
postgres=# select datname, pg_database_size(datname)/1024/1024 from pg_database group by datname;
  datname  | ?column?
-----------+----------
 postgres  |        6
 template0 |        6
 template1 |        6
(3 wiersze)

postgres=#
```

Jeśli zechciałbyś wiedzieć gdzie znajduje się Twój katalog z plikami danych, wystarczy wydać polecenie:

SHOW DATA_DIRECTORY;

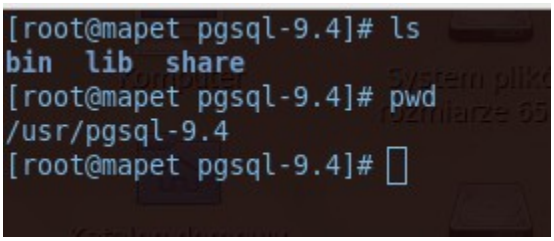
Struktura fizyczna i logiczna bazy

Struktura fizyczna – katalogi i pliki

Po instalacji klastra PostgreSQL pliki są rozłożone w dwóch lokalizacjach.

W **/usr/pgsql-9.4/** i **/var/lib/pgsql/9.4/data**

W /usr/pgsql-9.4/ znajdują się trzy katalogi. Katalog BIN zawiera binaria, znajdziesz tam programy takie jak pgsql czy pg_ctl oraz inne narzędzia. Katalog LIB zawiera biblioteki niezbędne do działania Postgresa, raczej nie będziemy tam zaglądać. Katalog SHARE zawiera przykłady plików konfiguracyjnych.

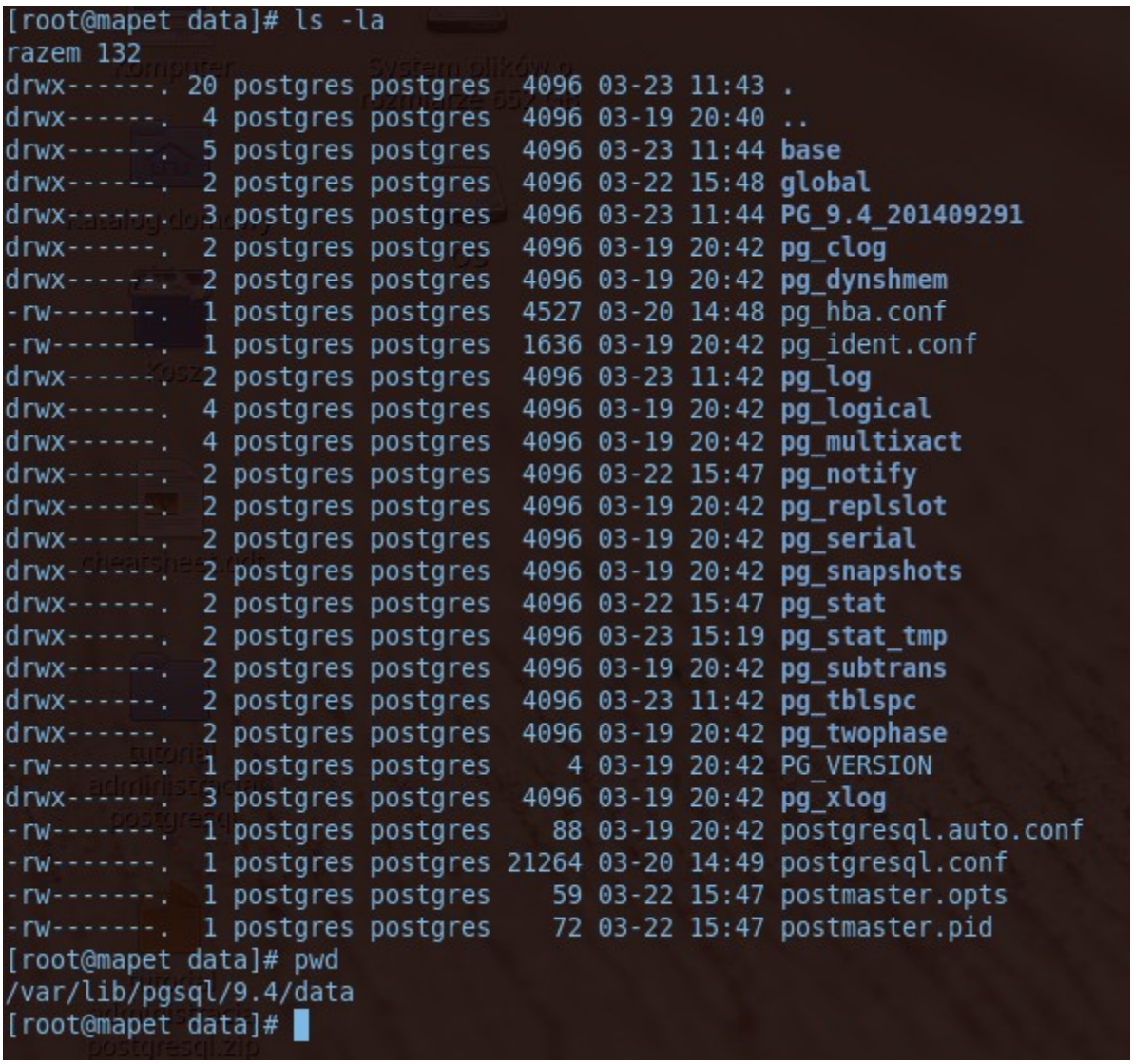


W katalogu /var/lib/pgsql/9.4/data mamy sporo podkatalogów oraz kilka plików tekstowych.

Poniżej ich zestawienie.

Pliki:

pg_hba.conf	Plik zawierający ustawienia dotyczące tego z jakich hostów/sieci można łączyć się z klastrem, oraz jakiego rodzaju uwierzytelniania to wymaga.
pg_ident.conf	Pozwala mapować użytkowników systemowych na bazodanowych
PG_VERSION	Zawiera informacje o wersji klastra PostgreSQL
Postgresql.auto.conf	Dodatkowy plik parametrów. Domyślnie czytane są parametry z pliku postgresql.conf, ale jeśli zmieniamy jakieś parametry z użyciem ALTER SYSTEM, zmiana ta jest zapisywana w pliku postgresql.auto.conf i plik ten jest czytany dodatkowo poza plikiem postgresql.conf przy starcie serwera lub przeładowaniu konfiguracji
Postgresql.conf	Plik konfiguracyjny klastra
Postmaster.opts	Plik zawierający opcje wiersza poleceń które są implementowane podczas startu procesu postmaster
Postmaster.pid	Zawiera identyfikator systemowego procesu postmaster. Istnieje tylko kiedy serwer działa lub gdy jego działanie zostało przerwane w wyniku awarii

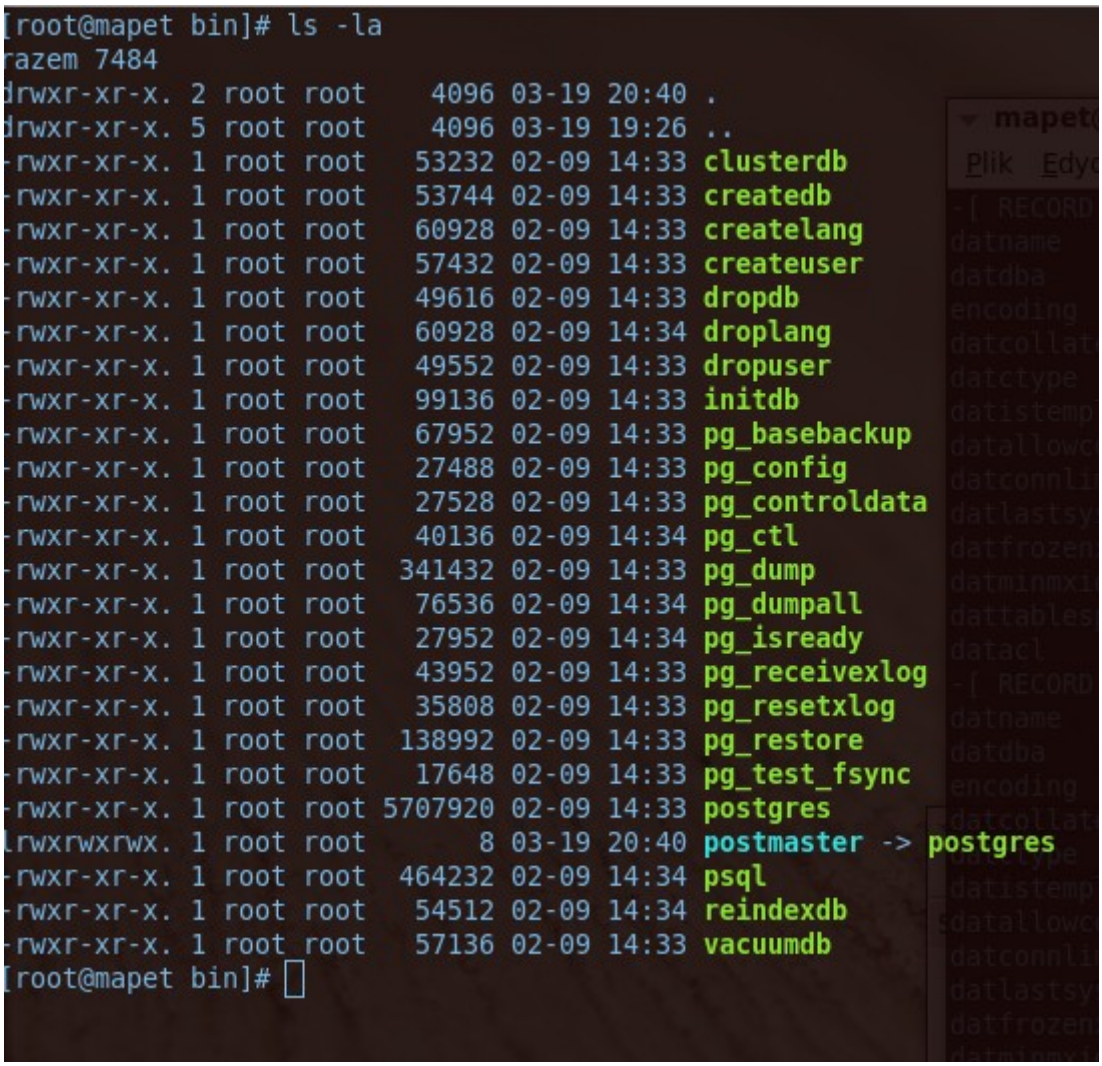


Katalogi:

base	Główny katalog danych. Każda baza danych posiada swój katalog danych, w którym przechowywane są pliki tabel i innych obiektów. Katalogi z plikami danych poszczególnych baz będą podkatalogami tego katalogu. Każda baza będzie posiadała swój podkatalog na pliki o nazwie odpowiadającej OID bazy.
Global	Pliki danych tabel klastra bazy danych tj. słowników nie związanych z konkretną bazą a z klastrem.
pg_clog	Zawiera pliki statusu transakcji
pg_dynshem	Zawiera pliki wykorzystywane przez system zarządzania pamięcią
pg_log	Zawiera pliki logów serwera. To właśnie tutaj zaglądamy gdy coś się dzieje.

pg_logical	Dane związane z replikacją (wprowadzone od wersji 9.4)
pg_multixact	Pliki statusu blokad na poziomie wierszy
pg_notify	
pg_replslot	Dane potrzebne do replikacji
pg_serial	Zawiera informacje na temat zatwierdzonych traksakcji serializowanych
pg_snapshots	Snapshoty
pg_stat	Trwałe pliki systemu statystyk
pg_stat_tmp	Tymczasowe pliki systemu statystyk
pg_subtrans	Pliki statusu podtransakcji
pg_tblspc	Łącza do zewnętrznych przestrzeni tabel
pg_twophase	Status zatwierdzania dwufazowego , lub przygotowanej transakcji
pg_xlog	Dziennik transakcji WAL

Binaria:



clusterdb	Narzędzie do odtwarzania klastrowania w tabelach
createdb	Narzędzie do tworzenia baz danych
createlang	Narzędzie do tworzenia obsługi rozszerzeń dla języków programowania
createuser	Narzędzie do tworzenia użytkowników
dropdb	Narzędzie do kasowania baz danych
droplang	Narzędzie do kasowania obsługi rozszerzeń dla języków programowania
dropuser	Narzędzie do kasowania użytkowników
initdb	Narzędzie do inicjalizowania klastra
pg_basebackup	Narzędzie do tworzenia fizycznej kopii zapasowej klastra na działającym systemie PostgreSQL bez wpływu na podłączonych

	użytkowników. Tak wytworzona kopia zapasowa może być wykorzystana np. do przywracania baz do punktu w czasie, lub jako punkt startowy replikacji
pg_config	Narzędzie do raportowania konfiguracji PostgreSQL
pg_controldata	
pg_ctl	Narzędzie do uruchamiania, zatrzymywania i przeładowywania konfiguracji klastra PostgreSQL
pg_dump	Narzędzie do tworzenia kopii zapasowych pojedynczych baz danych
pg_dumpall	Narzędzie do tworzenia kopii zapasowych wszystkich baz danych
pg_isready	Narzędzie do sprawdzania czy da się podpiąć do klastra
pg_receivexlog	Narzędzie do strumieniowania logów transakcyjnych na potrzeby replikacji
pg_restore	Narzędzie do odtwarzania bazy danych z kopii zapasowej

Struktura logiczna

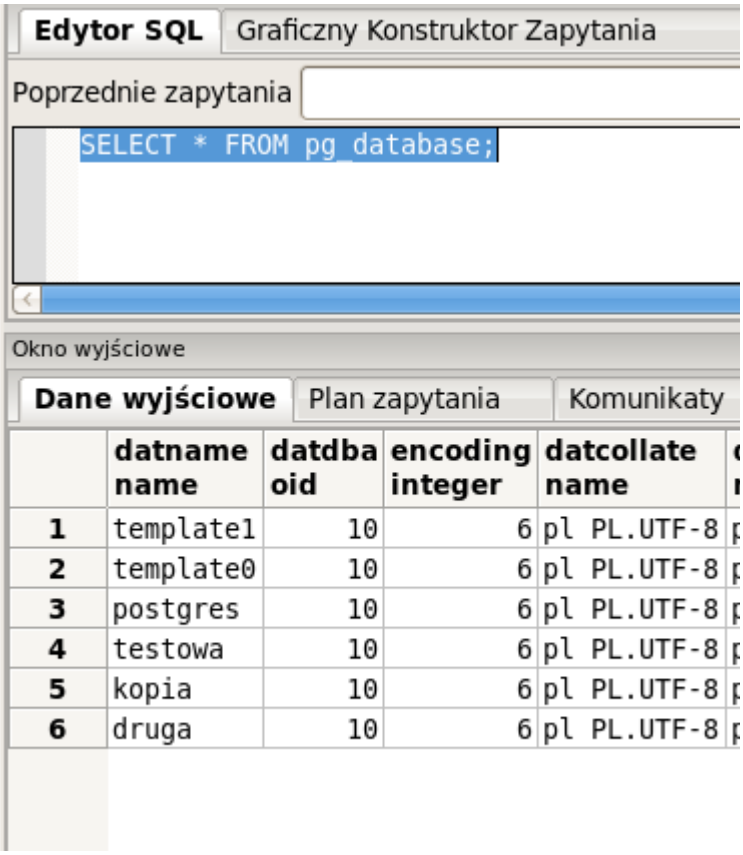
Bazy danych – informacje podstawowe

Bazy danych są strukturą logiczną, choć wiążą się ze strukturą fizyczną. Każda baza danych ma swój osobny katalog na serwerze. Nic nie stoi na przeszkodzie by wszystkie dane trzymać w ramach jednej np. domyślnej bazy danych postgres, ale różnych schematach. Mając dane związane z osobnymi systemami zgromadzone w osobnych bazach, możemy wykonywać osobne kopie zapasowe, osobno odtwarzać (np. do punktu w czasie), a także rozdzielnie konfigurować niektóre parametry.

Sprawdzanie dostępnych baz

Aby sprawdzić jakie mamy stworzone bazy danych w ramach klastra wystarczy wywołać:

SELECT * FROM pg_database;



	datname name	datdba oid	encoding integer	datcollate name
1	template1	10	6	pl PL.UTF-8
2	template0	10	6	pl PL.UTF-8
3	postgres	10	6	pl PL.UTF-8
4	testowa	10	6	pl PL.UTF-8
5	kopia	10	6	pl PL.UTF-8
6	druga	10	6	pl PL.UTF-8

Tworzenie baz danych

Ogólna konstrukcja polecenia tworzenia bazy danych wygląda tak:

CREATE DATABASE NAZWA_BAZY

WITH

OWNER = NAZWAUZYTKOWNIKA

TEMPLATE = BAZA_ROBIĄCA_ZA_SZABLON

ENCODING = KODOWANIE

TABLESPACE = NAZWA_PRZESTRZENI

CONNECTION LIMIT = X

Najprostszy sposób stworzenia bazy:

create database kolejnabaza;

W ramach takiej bazy utworzy się jeden schemat o nazwie public. Możesz też utworzyć bazę danych na podstawie innej bazy:

create database trzecia with template=druga;

Stworzona w ten sposób kopia będzie zawierała te same obiekty co wzorcowa. Ciekawy sposób na produkowanie baz developerskich czy testowych. Jedna uwaga – nie możesz w ten sposób stworzyć kopii bazy do której jest podłączona jakakolwiek sesja.

Możesz również umieścić bazę danych w określonej przestrzeni (co za tym idzie i lokalizacji) w ten sposób:

create database czwarta tablespace=mapetowa;

Aby stworzyć bazę której właścicielem będzie określony użytkownik wykonaj takie polecenie:

create database piata owner=mapet;

Możesz także zmienić właściciela istniejącej bazy w ten sposób:

alter database piata owner to postgres;

Gdybyś zechciał zmienić nazwę bazy danych:

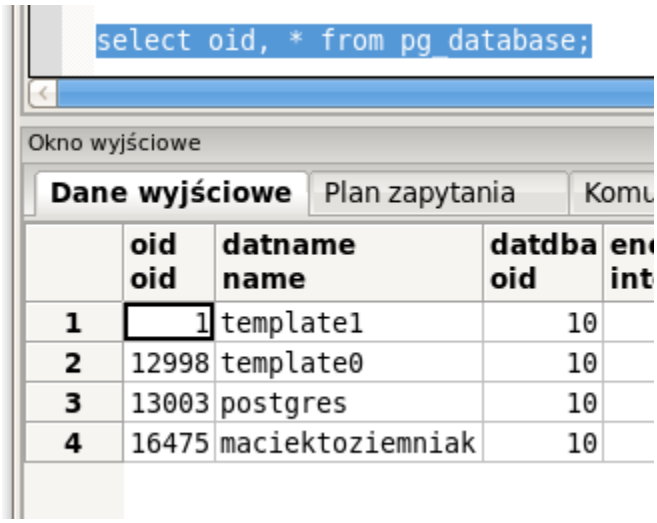
alter database piata rename to innanazwa;

Bazy danych a struktura katalogów i pliki związane z obiektami bazy

W związku z tworzeniem bazy danych, w katalogu $PGDATA/base powstanie nowy katalog o nazwie takiej jaki jest OID nowej bazy.

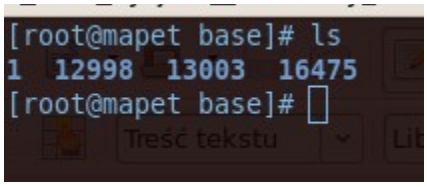
Możesz sprawdzić OID baz np. w ten sposób:

select oid, * from pg_database;



	oid oid	datname name	datdba oid	en int
1	1	template1	10	
2	12998	template0	10	
3	13003	postgres	10	
4	16475	maciektoziemniak	10	

A teraz przyjrzyjmy się zawartości katalogu base:

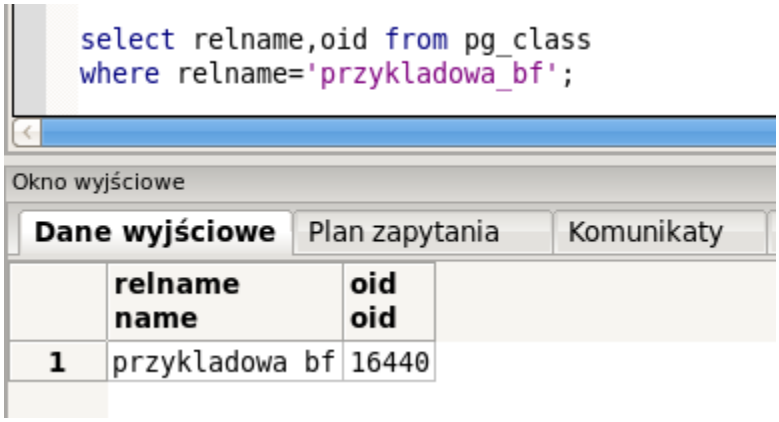


Z jakiegoś powodu wartość kolumny oid nie jest wyświetlana w tabeli pg_database, dlatego ważne abyś ją wymienił osobno.

W bazach PostgreSQL przechowywanie danych związanych z tabelami jest rozwiązane w ten sposób, że każda tabela posiada osobny plik danych. Jak więc odnaleźć plik odpowiedzialny za konkretny obiekt bazodanowy? Trzeba odpytać słownik pg_class wymieniając ukrytą kolumnę OID:

select relname,oid from pg_class

where relname='przykladowa_bf';



Tabelka przykladowa_bf leży w bazie danych postgres której katalog w $PGDATA/BASE to 13003 (wg OID bazy), zaś OID tabeli to 16440. Plik danych związany z tą tabelą to $PGDATA/BASE/13003/16440. Gdyby Cię nawet z ciekawości podkusiło by stworzyć sobie jakąś tabelkę, wstawić coś do niej i podejrzeć zawartość pliku, to najpierw wygeneruj checkpoint aby dane trafiły do pliku. Robi się to komendą :

checkpoint;

Jest to konieczne, bo może się okazać że informacja o nowym wierszu znajdzie się tylko w plikach WAL a nie trafi jeszcze do plików danych. Ciekawostka – Jeśli stworzysz nową zupełnie pustą tabelkę i nic do niej nie wstawisz, jej plik będzie istniał ale będzie kompletnie pusty. Możesz się o tym przekonać uruchamiając cat na tym pliku. Wynika to z tego, że wszelkie informacje nagłówkowe związane z tą tabelą są przechowywane w słownikach systemowych te są gromadzone w katalogu global w $PGDATA.

Kasowanie baz danych

Kasowanie baz odbywa się za pomocą komendy:

drop database nazwabazy;

Do kasowanej bazy nie może być podłączona żadna sesja. Jeśli jest, trzeba będzie ją najpierw rozłączyć. Kiedy kasujemy bazę danych, kasowany jest również powiązany z nią katalog danych (zajrzyj do $PGDATA/base przed i po skasowaniu bazy).

Przestrzenie tabel (tablespace)

Przestrzenie tabel fizycznie są katalogami. Zakłada się je np. by trzymać część danych na osobnej partycji. Tutaj mała uwaga do osób przyzwyczajonych do baz Oracle – tutaj to nie przestrzenie tabel są związane z tabelami i nie baza zawiera przestrzenie tabel, a bazy są przywiązane do przestrzeni tabel i to dla baz ustawiamy domyślną przestrzeń tabel w której mają lądować dane bazy.

Aby stworzyć nową przestrzeń tabel musimy wykonać poniższe kroki:

1. Stworzyć fizycznie katalog – katalog w którym mają znaleźć się obiekty danej przestrzeni tabel musi fizycznie istnieć.
2. Nadać uprawnienia do zapisu w nim i odczytu użytkownikowi z poziomu którego działa demon PostgreSQL. Możesz także zmienić własność z użyciem komendy chmod.
3. Stworzyć przestrzeń z poziomu SQL:

create tablespace dane_produkcyjne

location

'/home/mapet/tablespaces_postgresql/dane_produkcyjne';

Dopiero teraz możesz z tej przestrzeni korzystać. We wskazanym katalogu powstanie jeszcze podkatalog i dopiero w nim znajdą się obiekty tworzone w tej przestrzeni lub przenoszone do niej. Aby stworzyć nową bazę której obiekty domyślnie będą tworzone w tej przestrzeni tabel wywołaj:

create database produkcja tablespace = dane_produkcyjne;

Po tej operacji zostanie utworzony podkatalog w lokalizacji związanej z przestrzenią „dane_produkcyjne” na potrzeby nowej bazy danych.

Możesz również migrować istniejącą bazę danych do nowej przestrzeni tabel:

alter database stara_produkcja set tablespace dane_produkcyjne;

Aby domyślną przestrzeń tabel w której będą tworzone nowe tabele wykonaj polecenie:

alter database stara_produkcja set default_tablespace='dane_produkcyjne';

Nie ma przeszkód by wiele baz miało jako domyślną ustawioną tę samą przestrzeń tabel, lub by przenieść kilka baz do jednej przestrzeni.

Możesz utworzyć nową tabelę umieszczając ją w wybranej przestrzeni tabel w ten sposób:

create table nowa (x integer) tablespace dane_produkcyjne;

Ciekawostka – Tworząc tabelę w przestrzeni tabel innej niż domyślna dla danej bazy danych, w katalogu tej przestrzeni utworzy się jeszcze podkatalog o nazwie takiej jak OID owej bazy.

Możesz przenosić istniejące tabele do innych przestrzeni tabel np. w sytuacji gdy chcesz ową tabelkę przenieść na inny dysk. Przenoszone są również pliki danych związane z tą tabelką. Poniżej przykład tworzenia dużej tabelki i przeniesienia jej do innej przestrzeni tabel.

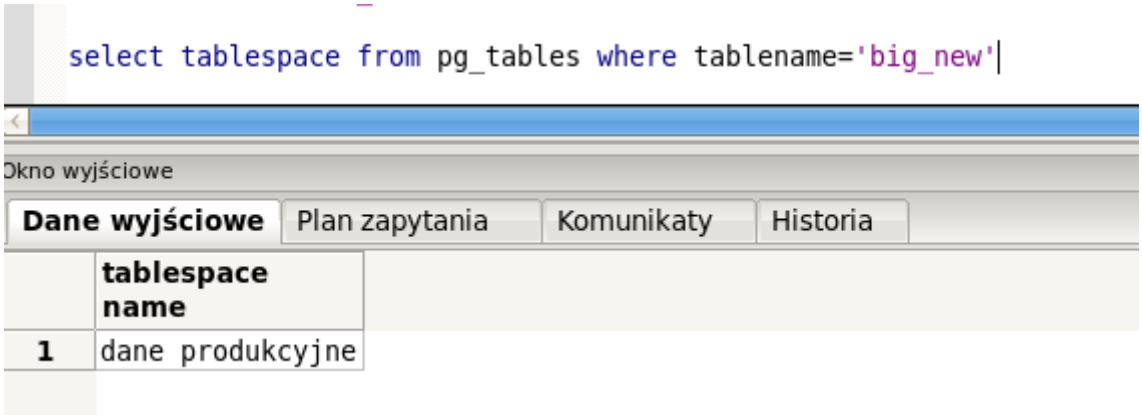
```
create table big(x integer,y varchar);
/
do
$$
begin
for x in 1..1000000 loop
        insert into big values (x,x||' owca');
end loop;
end;
$$
/
alter table big set tablespace  dane_produkcyjne;
```

Taka tabela zostanie przeniesiona do podkatalogu o nazwie takiej jak OID bazy z której przenosimy tabelę w katalogu przestrzeni tabel. Przeniesienie tabel nie pociąga za sobą przeniesienia indeksów. Wynika to z tego, że indeksy mogą leżeć w innych przestrzeniach tabel. Chcąc przenieść indeks należy wykonać polecenie:

alter index nazwa set tablespace dane_produkcyjne;

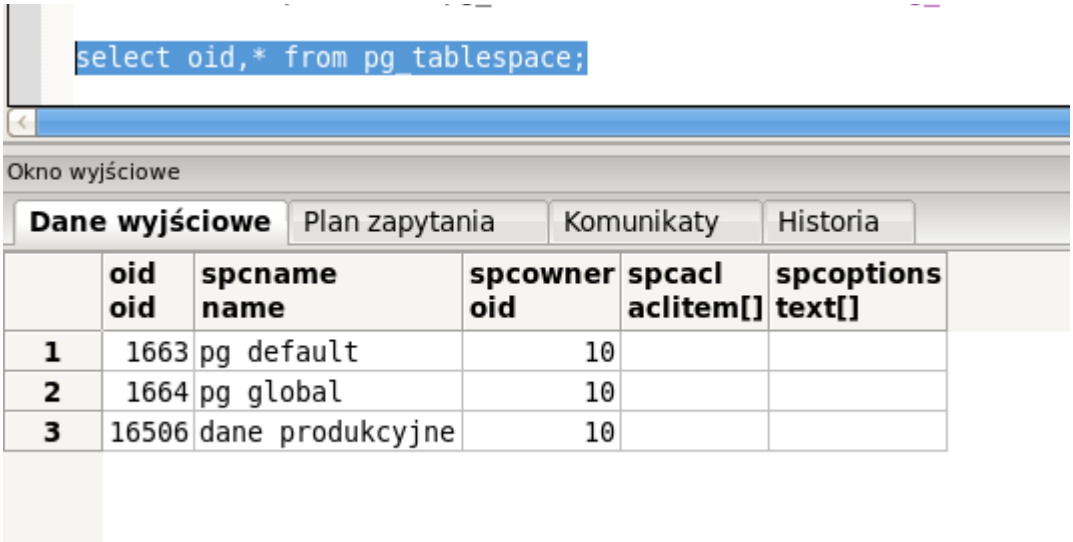
Jeśli chcesz sprawdzić w jakiej przestrzeni tabel leży dana tabela wykonaj polecenie

select tablespace from pg_tables where tablename='big_new';



Gdybyś zechciał sprawdzić jakie masz dostępne przestrzenie tabel, lub jakie są ich OID wystarczy wykonać polecenie:

select oid,* from pg_tablespace;



Skasować przestrzeń tabel możesz korzystając z polecenia :

drop tablespace dane_produkcyjne;

jednak ta przestrzeń tabel musi być pusta. Jeśli będzie, katalog związany z tą przestrzenią zostanie usunięty. Możesz wcześniej przenieść wszystkie tabele z tej przestrzeni tabel do innej wykonując polecenie:

alter tablespace dane_produkcyjne move all tables to postgres;

Schematy

Schematy to w PostgreSQL przestrzenie nazw. Służą odseparowaniu tabel związanych z różnymi systemami. Możesz mieć kilka tabel o takiej samej nazwie w ramach jednej bazy danych czy przestrzeni tabel, pod warunkiem że leżą one w różnych schematach. Schematy nie wiążą się ze strukturami fizycznymi.

Tworzenie schematów :

create schema nowy;

Jeśli właścicielem schematu ma być inny użytkownik:

create schema nowy authorization innyuser;

Aby usunąć pusty schemat zastosuj polecenie:

drop schema nowy;

Jeśli w schemacie znajdują się jakieś tabele, PostgreSQL nie pozwoli Ci go usunąć. Chcąc go skasować razem z zawartymi w nim obiektami stosujemy:

drop schema nowy cascade;

Jeśli chciałbyś podejrzeć jakie masz dostępne w danej bazie danych schematy wystarczy zawołać:

select schema_name from information_schema.schemata;

Gdyby przyszło Ci jeszcze do głowy zweryfikować jakie masz tabele w wybranym schemacie krzyknij:

SELECT * FROM information_schema.tables WHERE table_schema = 'nowy';

Jeszcze parę drobiazgów na temat których warto po prostu wiedzieć. Możesz przenieść obiekt między schematami w taki sposób:

alter table przykladowa set schema nowy;

Możesz też mieć ochotę zmienić nazwę schematu:

alter schema nowy rename to stary;

Gdy odwołujesz się do obiektu PostgreSQL będzie go szukał w schemacie public. Jeśli chcesz odwołać się do obiektu znajdującego się w innym schemacie, musisz jego nazwę poprzedzić nazwą tego schematu np.:

select * from nazwaschematu.nazwatabeli;

Możesz też na czas sesji zmienić domyślny schemat wyszukiwania w ten sposób:

set search_path=nowy;

Użytkownicy i uprawnienia

Użytkownicy – tworzenie, kasowanie i autoryzacja

Domyślnie po instalacji tworzony jest jeden użytkownik o nazwie postgres. Użytkownicy bazodanowi nie są tożsami z użytkownikami systemu operacyjnego. Domyślnie jednak podczas logowania jeśli nie podamy nazwy użytkownika – system przedstawi Postgresowi nazwę systemu użytkownika operacyjnego jako użytkownika bazodanowego – spróbuj przykładowo w systemie Linux zalogować się do PostgreSQL z poziomu roota i bez podania loginu.

Tworzenie użytkownika

Ogólna konstrukcja komendy tworzącej konto użytkownika prezentuje się następująco:

CREATE USER NAZWA_UZYSZKODNIKA

WITH PASSWORD 'JAKIESHASLO'

CREATEDB/NOCREATEDB

CREATEUSER/NOCREATEUSER

LOGIN/NOLOGIN

CONNECTION LIMIT X

IN GROUP NAZWAGRUPY

VALID UNTIL 'CZASWAZNOSCIKONTA'

Tak naprawdę obligatoryjne jest tylko podanie nazwy użytkownika. Tak! W systemie PostgreSQL możemy utworzyć użytkownika bez hasła i się na niego zalogować, pod warunkiem że zrobimy to z hosta dla którego w pliku pg_hba.conf ustawiony jest typ autoryzacji „trust”. Przy takiej sytuacji możemy nawet podać błędne hasło i też się zalogujemy!

With password i pozostałe klauzule są opcjonalne. CREATEDB/NOCREATEDB pozwala nam skonfigurować czy użytkownik może tworzyć bazy danych, CREATEUSER/NOCREATEUSER pozwala skonfigurować czy użytkownik może tworzyć inne konta użytkowników. Gdzieniegdzie w tutorialach i dokumentacji można spotkać jeszcze konstrukcję z CREATE ROLE. W tej chwili oba polecenia są aliasami. Jedyna różnica polega na tym, że tworząc użytkownika domyślnie tworzymy go z LOGIN, a tworząc rolę z NOLOGIN. Te opcje z kolei określają czy użytkownik może się logować czy nie. CONNECTION LIMIT pozwala określić ilość jednoczesnych sesji danego użytkownika. Jeśli damy wartość -1 nie będzie limitu połączeń i takie jest ustawienie domyślne

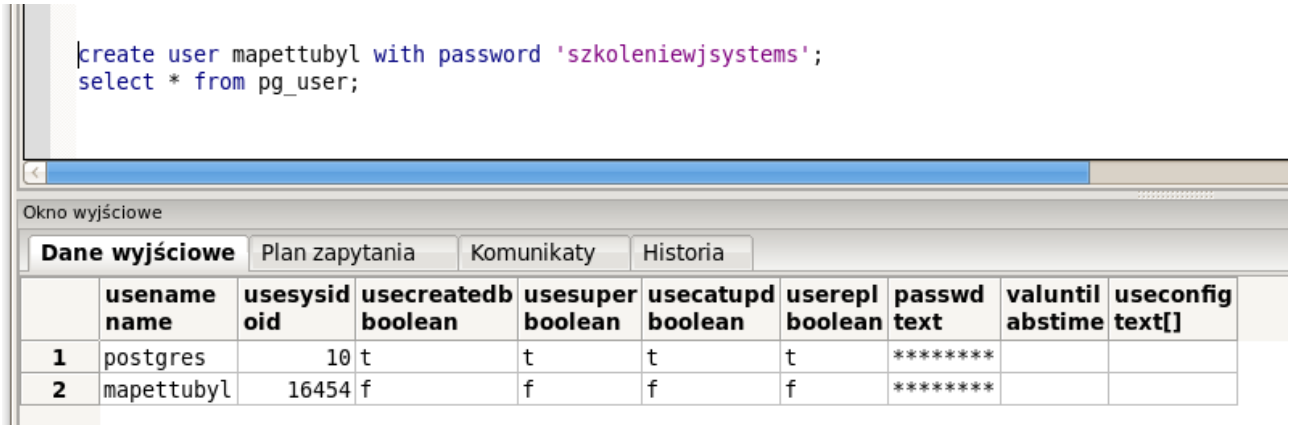
gdybyśmy tego parametru nie określili. IN GROUP pozwala przypisać użytkownika od razu do jakiejś grupy. Grupy tworzy się by łatwiej zarządzać zbiorczo uprawnieniami użytkowników. Zajmiemy się tym nieco później. VALID UNTIL określa okres ważności hasła. Przykładowy zapis:

VALID UNTIL 'May 4 12:00:00 2016 +1'

Jeśli byśmy natomiast zechcieli ustawić nieskończenie długi okres ważności hasła (a tak jest jeśli pominiemy ten parametr) :

VALID UNTIL 'infinity'

Ważna uwaga! Jeśli nie zdefiniujesz inaczej, nowo tworzony użytkownik będzie miał uprawnienia do tworzenia baz danych, będzie superużytkownikiem, będzie mógł aktualizować katalogi, a także włączać i wyłączać bazę w tryb backupu i włączać replikację (wymienione wg kolejności kolumn w słowniku pg_user od kolumny 3). Użytkownicy domyślnie nie mogą zaglądać do ani modyfikować tabel których nie utworzyli, niezależnie od tego w jakiej bazie danych czy schemacie owe tabele się znajdują. Nie dotyczy to użytkownika postgres który może buszować po czym chce. Słów kilka jeszcze o zasięgu użytkowników. Tworząc użytkownika tworzysz go na poziomie klastra. To oznacza, że ma on dostęp do wszystkich baz w ramach klastra, nie ma jednak dostępu do niczego czego nie stworzył w innych bazach, podobnie zresztą jak i w bieżącej. Jeśli byś zechciał mu nadać uprawnienia do obiektu znajdującego się w innej bazie tego samego klastra, musisz zalogować się do tej konkretnej bazy jako superużytkownik i nadać odpowiednie uprawnienie.



	usename name	usesysid oid	usecreatedb boolean	usesuper boolean	usecatupd boolean	userepl boolean	passwd text	valuntil abstime	useconfig text[]
1	postgres	10	t	t	t	t	********		
2	mapettubyl	16454	f	f	f	f	********		

Najczęściej będziesz tworzył użytkowników tak:

create user nazwauzytkownika with password 'haslouzytkownika';

albo tak:

create user nazwauzytkownika;

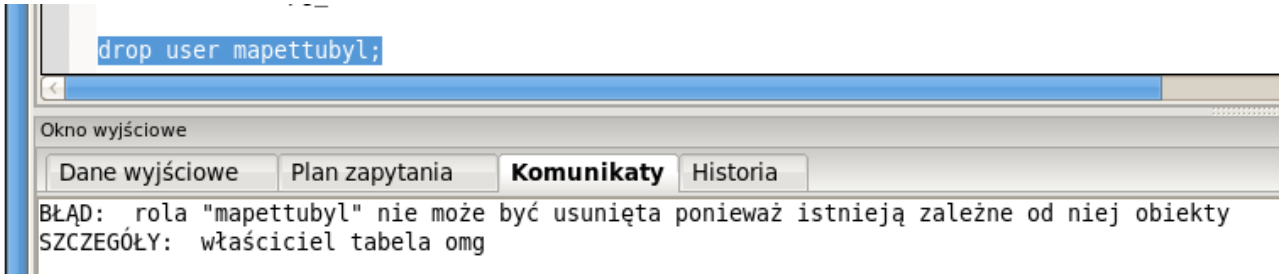
Zależnie od tego czy chcesz aby się autoryzował hasłem czy nie i jakie masz ustawienia w pg_hba.conf.

Kasowanie użytkownika

Użytkownika który nie jest twórcą i właścicielem żadnych obiektów skasować łatwo, ponieważ wystarczy wykonać komendę :

drop user nazwauzytkownika;

Sprawa jednak się nieco komplikuje gdy stworzy on jakieś obiekty, albo zostaną one mu przypisane. W przypadku próby skasowania takiego użytkownika dostaniemy komunikat jak poniżej:



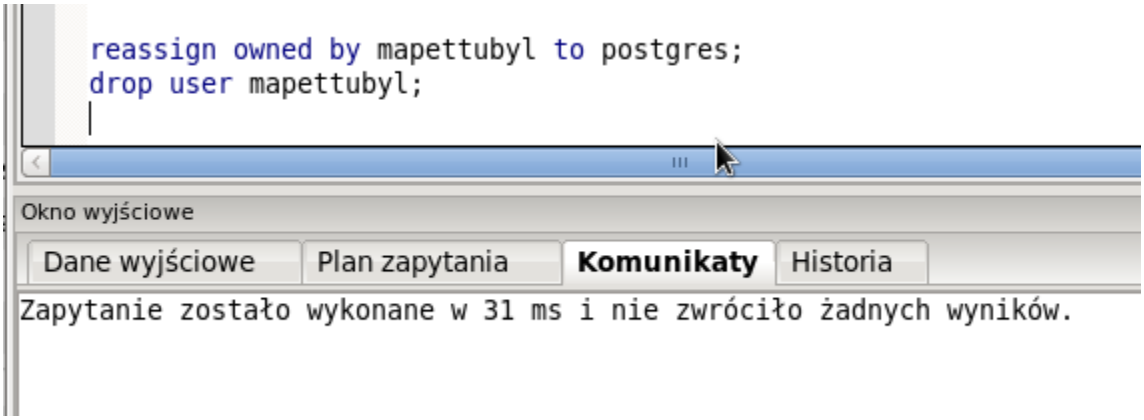
Jeśli zależy Ci po prostu na uniemożliwieniu logowania temu użytkownikowi możesz po prostu wykonać poniższe polecenie:

alter user mapettubyl nologin;

Użytkownik nie będzie mógł się logować, choć jako taki nadal będzie istniał w bazie danych. Gdybyś zechciał skasować go całkowicie, musisz wcześniej przenieść własność jego obiektów na innego użytkownika. Wykonać to możesz korzystając z komendy reassign:

reassign owned by mapettubyl to postgres;

Po przeniesieniu własności obiektów stworzonych przez użytkownika mapettubyl na użytkownika postgres, udało się bez problemu go skasować:



Zmiana własności użytkownika

Wszystkie własności z poniższych

WITH PASSWORD 'JAKIESHASLO'

CREATEDB/NOCREATEDB

CREATEUSER/NOCREATEUSER

LOGIN/NOLOGIN

CONNECTION LIMIT X

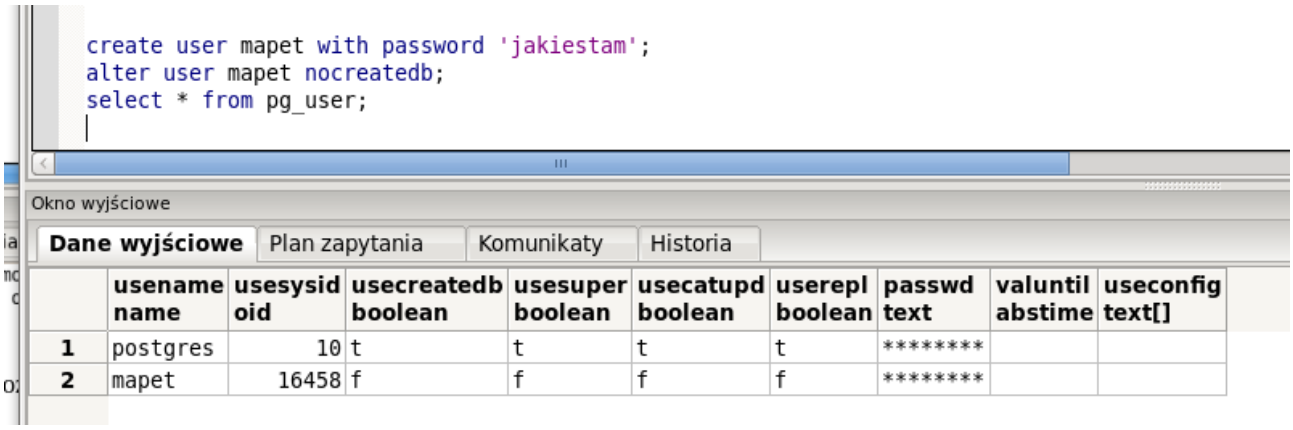
IN GROUP NAZWAGRUPY

VALID UNTIL 'CZASWAZNOSCIKONTA'

możesz modyfikować z użyciem komendy ALTER USER np.

alter user mapet nocreatedb;

spowoduje wyłączenie możliwości tworzenia baz danych przez użytkownika:



	usename name	usesysid oid	usecreatedb boolean	usesuper boolean	usecatupd boolean	userepl boolean	passwd text	valuntil abstime	useconfig text[]
1	postgres	10	t	t	t	t	********		
2	mapet	16458	f	f	f	f	********		

Grupy użytkowników i masowe zarządzanie uprawnieniami

Do masowego zarządzania uprawnieniami użytkowników w systemie PostgreSQL służą grupy. Można to porównać do ról w bazach danych Oracle. Dzięki nim możemy zbiorczo nadać lub odebrać uprawnienia wszystkim użytkownikom mającym przypisaną daną grupę. Grupę tworzymy przy użyciu komendy CREATE GROUP :

create group programisci;

lub jeśli chcemy dodać od razu jednego użytkownika:

create group programisci user mapet;

lub jeśli chcemy dodać od razu wielu użytkowników:

create group programisci user mapet,longer,maciek;

Możemy także do istniejącej grupy dodać kolejnych użytkowników komendą :

alter group programisci add user longer,maciek,ajhor;

albo ich z danej grupy usunąć:

alter group programisci drop user longer;

Poniżej przykład tworzenia nowej grupy, kilku użytkowników, a następnie dodania do niej tych nowo tworzonych użytkowników pojedynczo i usunięcie z niej jednego z nich. Po zajrzeniu do słownika pg_group zobaczymy identyfikatory OID tych użytkowników.

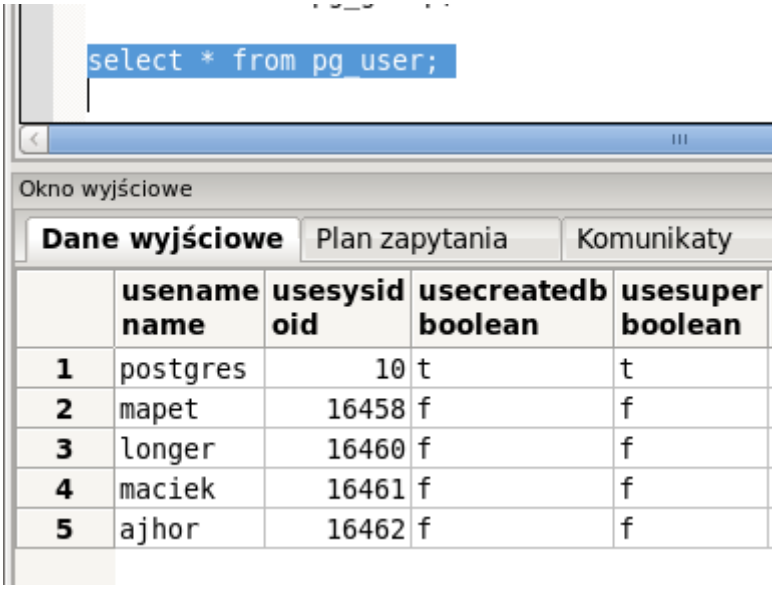
```
create group programisci user mapet;
create user longer with password 'kfc';
create user maciek with password 'java';
create user ajhor  with password 'mamajfona';
alter group programisci add user longer,maciek,ajhor;
alter group programisci drop user longer;
select * from pg_group;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	groname name	grosysid oid	grolist oid[]
1	bezhasela	16451	{}
2	programisci	16459	{16458,16461,16462}

Nie trudno teraz zweryfikować jacy użytkownicy należą do danej grupy zaglądając do słownika pg_user:



	usename name	usesysid oid	usecreatedb boolean	usesuper boolean
1	postgres	10	t	t
2	mapet	16458	f	f
3	longer	16460	f	f
4	maciek	16461	f	f
5	ajhor	16462	f	f

Skasować grupę również łatwo:

drop group programisci;

Fakt że do kasowanej grupy są przypisani jacyś użytkownicy nam nie przeszkadza. Grupa zostanie skasowana mimo to, choć użytkownicy pozostaną w systemie (najwyżej pozbawieni części uprawnień wynikających z owej grupy).

Uprawnienia

Nadawanie uprawnień

Ogólna składnia komendy do nadawania uprawnień wygląda następująco:

GRANT UPRAWNIENIE ON NAZWA_OBIEKTU

TO PUBLIC / GROUP NAZWA_GRUPY/ NAZWA_UŻYTKOWNIKA

Uprawnienia jakie możemy nadawać są następujące:

SELECT	Umożliwia odczyt tabeli
UPDATE	Umożliwia aktualizację wierszy
INSERT	Umożliwia wstawianie nowych wierszy
DELETE	Umożliwia kasowanie wierszy
RULE	Umożliwia tworzenie reguł
ALL	Daje uprawnienia do wszystkiego powyższego
TRUNCATE	Umożliwia wykonywanie komendy truncate na wskazanej tabeli
CREATE	Nadane dla bazy pozwala na tworzenie nowych schematów w ramach tej bazy. Nadane dla schematu pozwala na tworzenie obiektów w ramach danego schematu. Nadane dla przestrzeni tabel pozwala na tworzenie obiektów w ramach danej przestrzeni tabel, oraz wskazywania danej przestrzeni tabel jako domyślnej dla baz danych
CONNECT	Umożliwia łączenie się ze wskazaną bazą
EXECUTE	Pozwala wykorzystać wskazaną funkcję
REFERENCES	Pozwala zakładać klucze obce na tabelach. Aby móc założyć klucz obcy trzeba mieć nadane to uprawnienie na kolumnach po obu stronach. Można to uprawnienie nadawać dla pojedynczych kolumn i dla całych tabel.
TRIGGER	Pozwala na tworzenie wyzwalaczy na obiekcie
USAGE	Nadane dla języków proceduralnych pozwala na wykorzystanie danego języka proceduralnego do tworzenia funkcji. Dla sekwencji pozwala na pobieranie z nich wartości z użyciem currval i nextval
ALL PRIVILEGES	Wszystkie możliwie przywileje na danym typie obiektu

Nadając uprawnienia z użyciem klauzuli PUBLIC oznacza nadanie uprawnienia wszystkim użytkownikom – w tym także tym którzy jeszcze nie istnieją w systemie! Analogicznie jak nadawanie uprawnień roli PUBLIC w bazach Oracle.

Przykładowo nadanie uprawnień w ten sposób:

grant select on baza_firm.przykladowa_bf to public;

spowoduje że wszyscy użytkownicy jacy istnieją w bazie lub w przyszłości będą w niej istnieli, będą mieli możliwość odpytywania zawartości tabeli przykladowa_bf leżącej w schemacie public.

Nadanie uprawnień odczytu do tabeli leżącej w schemacie public (a taki jest domyślny schemat przeszukiwania) tworzonej wcześniej w tym rozdziale grupie programiści nadamy tak:

grant select on przykladowa_public to group programisci;

Można by się więc spodziewać że nadanie uprawnień do tabeli przykladowa_bf leżącej w schemacie baza_firm sprowadzać się będzie do wydania polecenia:

grant select on baza_firm.przykladowa_bf to group programisci; --NIE!

Niestety tabela ta leży w innym niż publiczny schemacie, więc aby użytkownicy mający przypisaną grupę programiście mieli do niej dostęp trzeba najpierw wydać polecenie analogiczne do poniższego:

grant all on schema baza_firm to group programisci;

A dopiero po tym uprawnienie do odczytu tabeli przykładowa_bf będzie działać. Patrząc na to polecenie można się spodziewać, że w takim razie użytkownicy z grupy programiści będą mieli uprawnienia do robienia wszystkiego z wszystkimi obiektami w tym schemacie. Okazuje się że nie, jednak wydanie takiego polecenia jest konieczne jeśli obiekt do którego nadajemy uprawnienia leży w schemacie innym niż public. Dotyczy to zarówno nadawania uprawnień przez grupę jak i bezpośrednio użytkownikowi. Trochę dziwne, ale co poradzimy. Sprawdzone organoleptycznie, trust me ;)

Gdyby zechcieli nadać uprawnienia do odczytu wszystkich obiektów w jakimś schemacie grupie lub użytkownikowi, musimy wydać polecenie dla grupy:

grant select on all tables in schema baza_firm to programisci;

a dla użytkownika:

grant select on all tables in schema baza_firm to maciek;

Odbieranie uprawnień

Ogólna konstrukcja służąca odbieraniu uprawnień wygląda tak:

REVOKE UPRAWNIENIE ON NAZWA_OBIEKTU

FROM PUBLIC / GROUP NAZWA_GRUPY/ NAZWA_UŻYTKOWNIKA

Bazując na poprzednim podrozdziale - dotyczącym nadawaniu uprawnień wystarczy zapamiętać że chcąc odebrać jakieś uprawnienie w miejsce grant wstawiamy revoke, a w miejsce to wstawiamy from. Kilka przykładów odnoszących się do nadawania uprawnień z poprzedniego podrozdziału:

revoke select on baza_firm.przykladowa_bf from public;

revoke select on przykladowa_public from programisci;

revoke all on przykladowa_public from programisci;

revoke select on all tables in schema baza_firm from maciek;

Jest jednak kilka unikalnych zasad odnoszących się do odbierania uprawnień o których warto wspomnieć. Jeśli np. nadaliśmy rząd uprawnień do jakiegoś obiektu czy schematu, a nie mamy ochoty wymieniać ich wszystkich przy odbieraniu możemy wydać taką komendę:

revoke all on przykladowa_public from programisci;

Można też w drugą stronę. Jeśli nadaliśmy uprawnienia na zasadzie GRANT ALL, można też odbierać je pojedynczo np.:

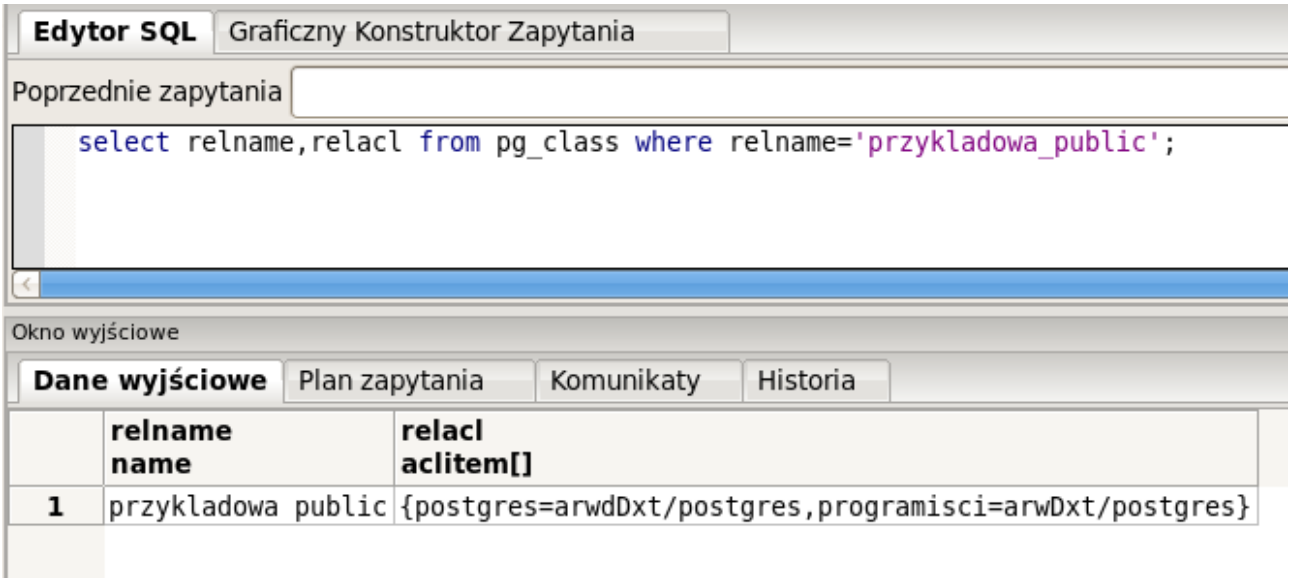
revoke delete on przykladowa_public from programisci;

Sprawdzanie uprawnień

Po pewnym czasie pracy z użytkownikami i ich uprawnieniami możemy po prostu nie pamiętać jakie uprawnienia i komu nadaliśmy. Możemy to sprawdzić z użyciem komend psql, albo korzystając ze słowników systemowych.

Możemy zajrzeć do słownika pg_class:

select relname,relacl from pg_class where relname='nazwa_obiektu';

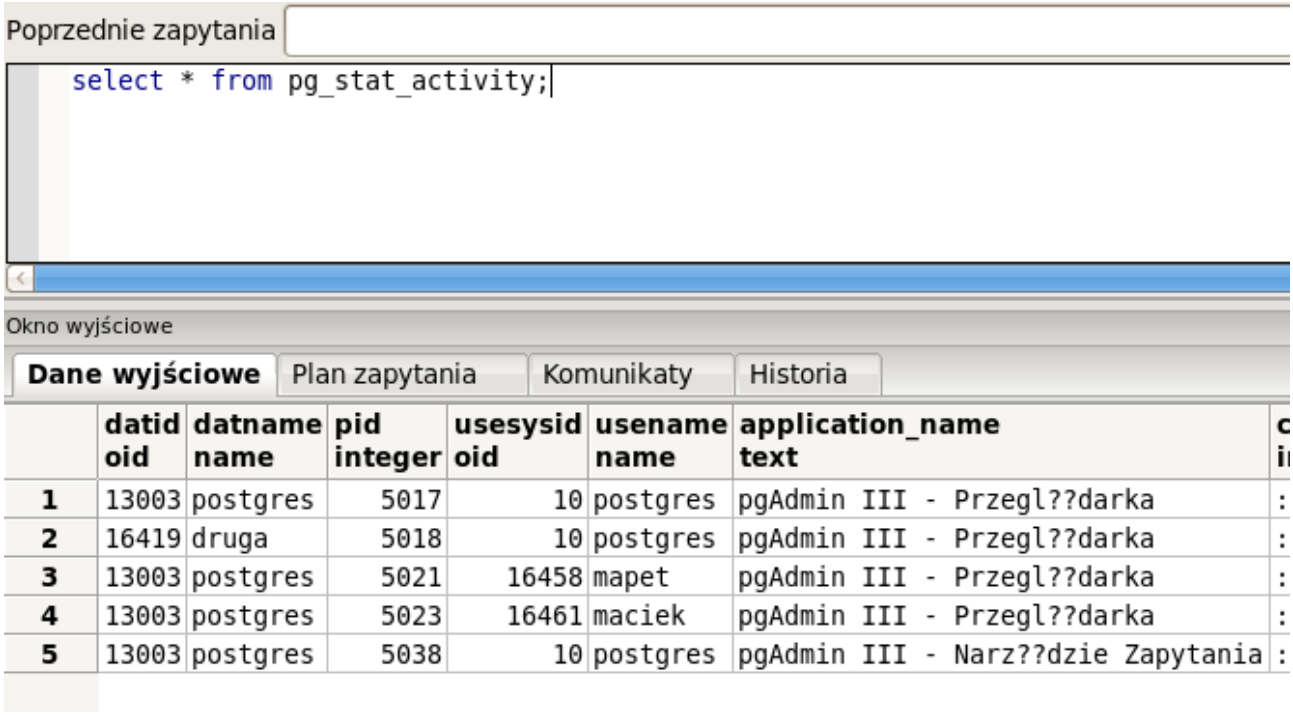


W kolumnie relname znajdziemy nazwy obiektów których uprawnienia dotyczą. W kolumnie reacl nazwy użytkowników i grup oraz rodzaje uprawnień jakie do danego obiektu posiadają. Tylko jak interpretować owe wpisy? Poniższa tabelka to wyjaśnia:

r	Odczyt - select
w	Zmiana - update
a	Dodawanie - insert
d	Kasowanie - delete
D	Kasowanie - truncate
x	Klucze obce - references
t	Zakładanie wyzwalaczy - trigger
X	Wykonywanie - execute
U	Użycie - usage
C	Tworzenie - create
c	Inicjalizacja połączenia - connect
arwdDxt	Wszystko – all privileges
*	Możliwość przekazywania uprawnień dalej dla wymienionych

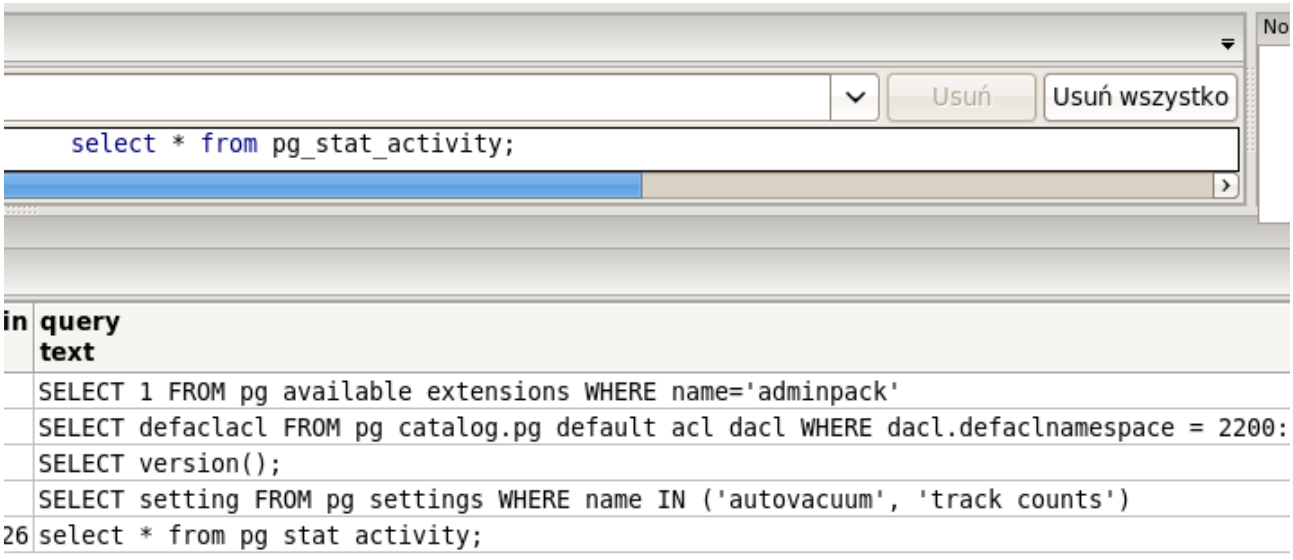
Sesje użytkowników i ich rozłączanie

Sprawdzić listę podłączonych sesji możesz w pg_stat_activity:



	datid oid	datname name	pid integer	usesysid oid	usename name	application_name text	c i
1	13003	postgres	5017	10	postgres	pgAdmin III - Przegl??darka	:
2	16419	druga	5018	10	postgres	pgAdmin III - Przegl??darka	:
3	13003	postgres	5021	16458	mapet	pgAdmin III - Przegl??darka	:
4	13003	postgres	5023	16461	maciek	pgAdmin III - Przegl??darka	:
5	13003	postgres	5038	10	postgres	pgAdmin III - Narz??dzie Zapytania	:

Mamy tu informację o tym do jakiej bazy i z użyciem jakiego programu łączy się użytkownik, ale też kiedy dana sesja się rozpoczęła, kiedy zostało rozpoczęte dane ostatnie zapytanie danego użytkownika, a nawet ostatnie zapytanie jakie puścił:

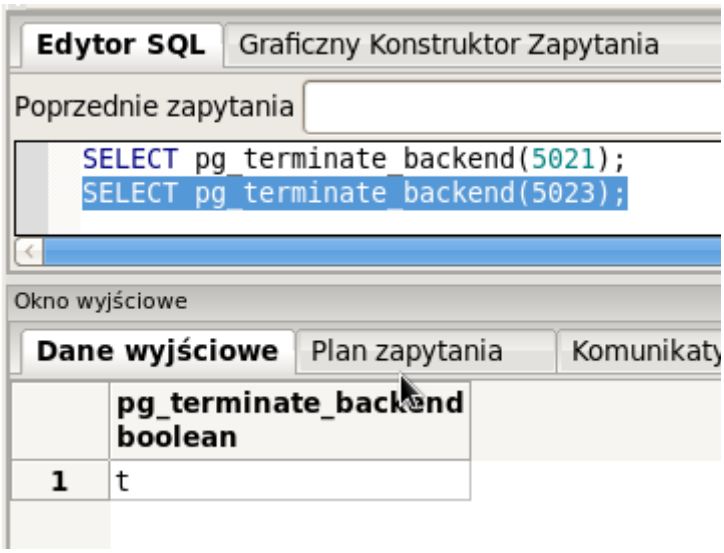


in	query text
	SELECT 1 FROM pg available extensions WHERE name='adminpack'
	SELECT defaclacl FROM pg catalog.pg default acl dacl WHERE dacl.defaclnamespace = 2200:
	SELECT version();
	SELECT setting FROM pg settings WHERE name IN ('autovacuum', 'track counts')
26	select * from pg stat activity;

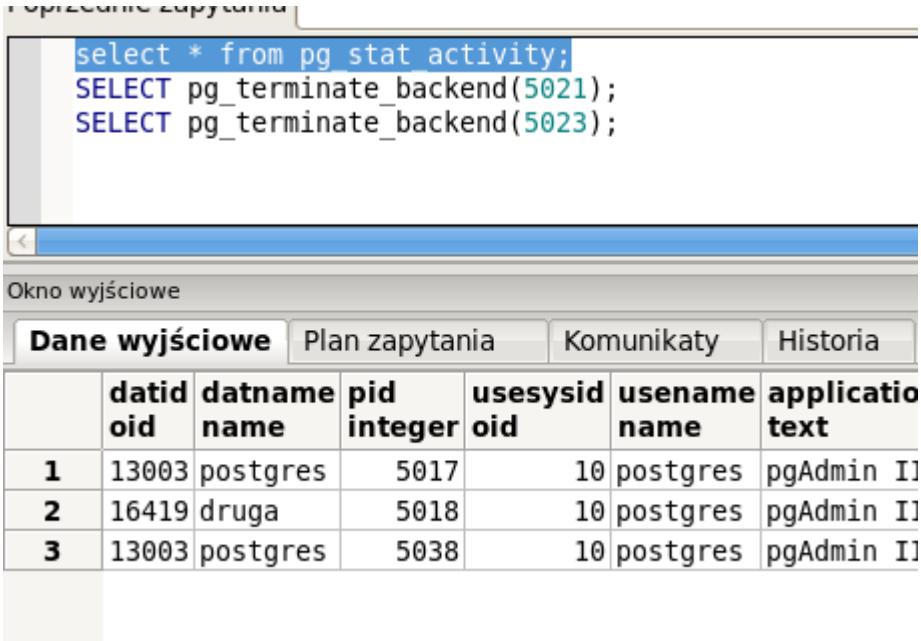
Wycinać sesje możesz taką komendą:

SELECT pg_terminate_backend(5023);

Jako parametr podajesz zawartość kolumny PID danej sesji z przeglądanego przed momentem słownika pg_stat_activity. Tym sposobem wyciąłem sesje użytkowników maciek i mapet. Po wywołaniu komendy dostaniemy wynik „t” albo „f” w zależności od tego czy uda się wyciąć daną sesję czy nie. „f” może się pojawić np. jeśli podasz niewłaściwy PID.



Po wycięciu sesje znikają z listy :



Transakcje, poziomy izolacji i blokady

Zarządzanie transakcjami

Czym są transakcje? Przytoczę tutaj sztampowy przykład przelewów w banku. Wyobraźmy sobie taką sytuację – jeden klient przelewa pewną kwotę drugiemu. System już zdążył pobrać kwotę z konta przelewającego, po czym następuje awaria. Kwota nie pojawia się na koncie beneficjenta. Czy może to tak działać? Dla banku pewnie byłaby to korzystna sytuacja, trochę mniej dla klientów. Obie operacje – pomniejszenia stanu konta jednego klienta i powiększenia innego muszą wykonać się razem, albo żadna z nich. Do tego celu potrzebujemy właśnie transakcji. Osoby pracujące dotąd z bazami Oracle mogą mieć przyzwyczajenie że każda operacja DML rozpoczyna transakcję, a kończy ją jawny bądź niejawny COMMIT lub ROLLBACK. W systemach Oracle domyślnie do czasu zatwierdzenia transakcji, inne sesje nie widzą zmian w bazie. Tu jest podobnie, z tą różnicą że domyślnie w bazach PostgreSQL włączone jest zatwierdzanie. Wykonując więc operację DML musimy się liczyć z tym że zostanie ona od razu zatwierdzona. Jeśli chcemy rozpocząć transakcję która nie zostanie zatwierdzona tak długo jak długo nie zrobimy tego jawnie, wydajemy polecenie:

begin work;

bądź po prostu :

begin;

Słówko „work” nie jest obligatoryjne. Aby zatwierdzić transakcję wykonaj komendę :

commit work;

lub

commit;

Aby ją wycofać :

rollback work;

lub

rollback;

Zatwierdzenie lub wycofanie transakcji będzie dotyczyć wszystkich operacji w ramach niej.

Jeszcze słówko do użytkowników Oracle – mogliście przywyknąć do tego, że operacje DDL wysyłają niejawny commit, tutaj jest to postawione na głowie. Operacje DDL są objęte transakcjami , czyli jeśli jawnie rozpoczniesz transakcję, stworzysz tabelę a następnie transakcję wycofasz – zostanie wycofane stworzenie tabeli O.o (!!!!) Aby się o tym przekonać wykonaj poniższe polecenia:

begin work;

create table public.tab (x integer);

insert into public.tab values (0);

select * from public.tab;

rollback work;

select * from public.tab;

Niepożądane zjawiska związane z transakcyjnością

Do takich zjawisk zaliczyć możemy :

- Brudny odczyt
- Odczyt nie dający się powtórzyć
- Odczyty widmo
- Utracone aktualizacje

Brudny odczyt

Domyślnie inne sesje nie będą widziały zmian które wykonami a nie zatwierdzimy (jawnie lub nie). Gdyby mogły – mielibyśmy do czynienia z brudnym odczytem. PostgreSQL nie umożliwia operacji brudnego odczytu.

Odczyty nie dające się powtórzyć

Domyślnie jeśli zatwierdzisz (jawnie lub nie) jakąś operację, inne sesje będą widziały Twoje zmiany zmiany. Nie jest to zależne od tego czy inna sesja ma w tej chwili trwającą transakcję czy nie. Nazywamy to odczytem nie dającym się powtórzyć, PostgreSQL domyślnie pozwala na tego typu transakcje. Gdyby taki odczyt był zabroniony, inne sesje widziałyby zmiany dopiero po zakończeniu swoich transakcji – o ile oczywiście mają jakieś trwające.

Odczyty widmo

Wyobraź sobie taką sekwencję czynności:

- Sesje A i B rozpoczynają swoje transakcję
- Sesja A wykonuje UPDATE na wszystkich wierszach tabeli PRODUKTY
- Sesja B dodaje nowy wiersz do tabeli PRODUKTY
- Sesje A i B zatwierdzają swoje transakcje.

Czy zmiana wykonana przez sesję A będzie dotyczyła również wiersza dodanego przez sesję B? Nie! Dzieje się tak ponieważ w momencie określania przez sesję A które wiersze należy zaktualizować, nowo dodany przez sesję A wiersz dla sesji B jeszcze nie istnieje. Sesja A zobaczy nowy wierz dopiero po zatwierdzeniu transakcji przez sesję B.

Poziomy izolacji

To jakie zjawiska będziemy obserwować zależy od poziomu izolacji jaki będziemy mięli ustawiony poziom izolacji. Domyślnym trybem jest READ COMMITED. Mamy do wyboru dwa – READ COMMITED i SERIALIZABLE (tak jak w Oracle). Różnica polega na tym, że w trybie SERIALIZABLE nie będziemy mięli do czynienia ze zjawiskiem odczytu nie dającego się powtórzyć ani odczytami widmo. Możemy to zmienić za pomocą komendy:

set transaction isolation level read commited / serializable;

Blokady

Zasada działania blokad

Istnieją dwa tryby blokad :

- blokada współdzielona - taka blokada pozwala innym sesjom odczytywać, ale nie pozwala na zmianę zablokowanych danych
- blokada wyłączna – nie pozwala ani czytać ani zmieniać zablokowanych danych

Domyślnie w PostgreSQL działa blokada współdzielona. Jeśli jakaś sesja rozpocznie transakcję i zmieni dane, inne sesje do czasu zatwierdzenia owej transakcji będą mogły odczytywać oryginalną postać danych, ale nie będą mogły tych danych zmieniać.

Wiersze i tabele mogą być blokowane tylko w ramach transakcji. Po jej zakończeniu są natychmiast odblokowywane.

Może dojść do sytuacji zakleszczenia. Przykładowo sesje A i B rozpoczynają transakcję. Sesja A zmienia wiersze w tabeli X, sesja B zmienia wiersze w tabeli Y. Następnie sesja A usiłuje zmieniać wiersze zablokowane przez sesje B w tabeli Y, a sesja B usiłuje zmieniać wiersze zablokowane przez sesję A w tabeli X. Taki pat nazywamy zakleszczeniem. Obie sesje blokują się wzajemnie. Po pewnym czasie w jednej z sesji dostaniemy komunikat „Deadlock detected”. Nie jesteśmy w stanie przewidzieć w której, dokumentacja milczy na ten temat.

Jawne blokowanie wierszy i tabel

W celu uniknięcia zakleszczeń, lub aby w „międzyczasie” możemy jawnie zablokować wiersze lub tabelę. Dzięki temu będziemy mieli pewność że zmiany wprowadzane przez inne sesje nie będą kolidowały z naszymi.

Wiersze możemy zablokować komendą np. :

select * from produkty where cena>500 for update;

Zadziała to jednak tylko i wyłącznie jeśli będziemy mieli rozpoczętą transakcję! Wiersze pozostaną zablokowane do czasu zakończenia naszej transakcji.

Możesz też zablokować całą tabelę:

lock table produkty;

Tutaj mamy jednak aż dwa obostrzenia. Po pierwsze działa to tylko w ramach transakcji, po drugie blokuje wszelki dostęp do tabeli – w tym odczyt!

Mechanizmy wewnętrzne transakcyjności i operacja VACUUM

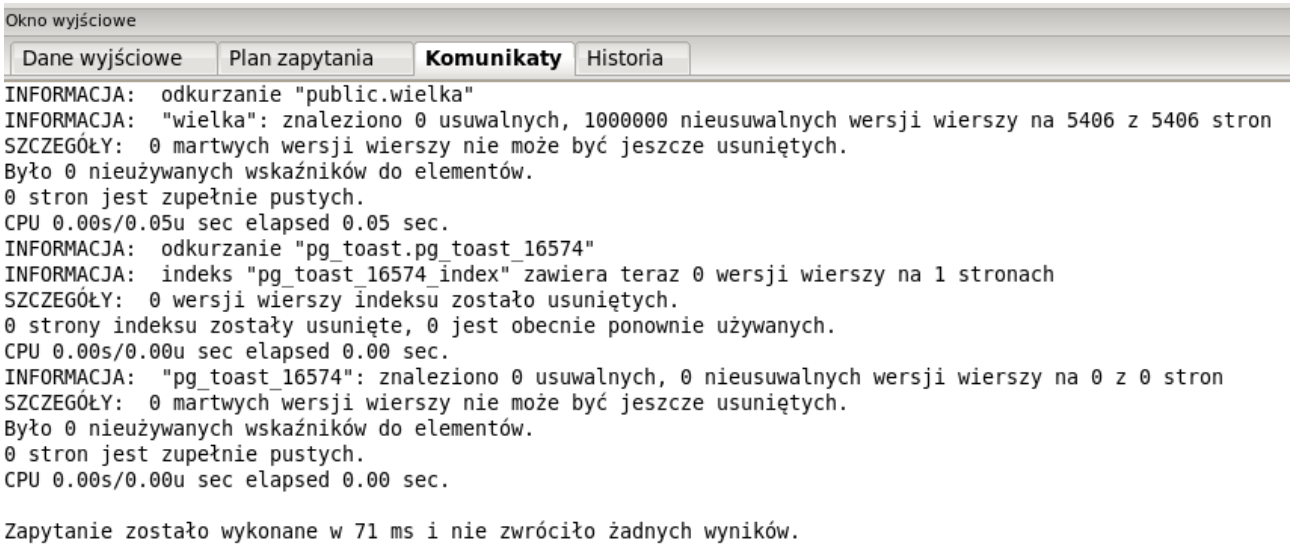
Ktoś dociekliwy mógłby się zacząć zastanawiać jak to w ogóle działa? Jak to jest możliwe że dwie sesje widzą zupełnie różne rzeczy w sytuacji gdy jedna z nich zmieni dane w ramach transakcji ale nie zatwiedzi? Gdy zmieniasz jakiś wiersz, PostgreSQL tworzy nową kopię wiersza na której Ty jako zmieniający operujesz. Kopia tego wiersza znajduje się w ramach tabeli w której oryginalny wiersz się znajduje. Dla zapytań odpytujących tę tabelę dostępne są stare wiersze. Po zakończeniu transakcji wszystkie zapytania wszystkich transakcji korzystają już z nowych wierszy. Taki sposób działania sprawia, że w plikach związanych z tabelami z czasem powstaje ogromna ilość nieużywanych wierszy wierszy do których już nawet nie ma dostępu. To z kolei powoduje rozrost plików danych. Miejsce zajmowane przez takie wiersze można odzyskać za pomocą polecenia VACUUM. Polecenie to poza odzyskiwaniem wiersza może też odświeżać statystyki – ale nie jest to tematem tego rozdziału.

Sprawdźmy więc działanie tego polecenia. Wykonaj sekwencję poniższych komend aby stworzyć przykładową tabelę i zapełnić ją danymi.

```
create table wielka (x integer,y varchar);
do
$$
begin
for x in 1..1000000 loop
        INSERT INTO WIELKA VALUES (X,'X='||X);
end loop;
end;
$$;

VACUUM VERBOSE WIELKA;
```

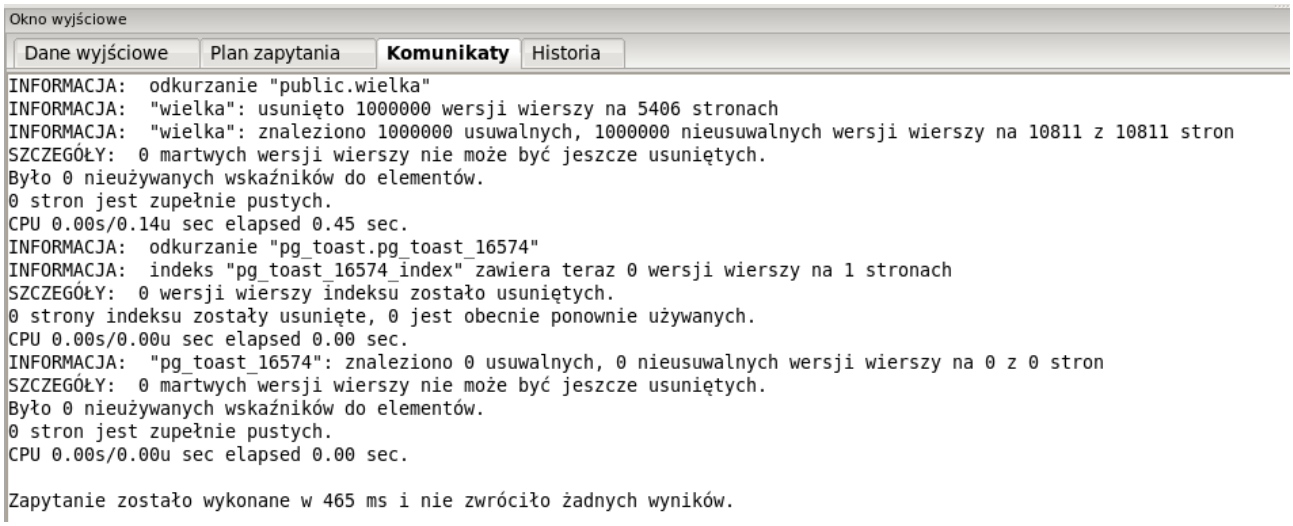
Na końcu wywoływany jest vacuum który powinien nam zwrócić mniej więcej taki wynik:



Jak widać mamy milion nieusuwalnych wierszy – to są te przed momentem wstawione, oraz 0 usuwalnych. Nie dokonaliśmy żadnych zmian jak dotąd, więc po prostu nie ma niepotrzebnych wierszy które można usunąć. Zmieńmy więc dane tak by takie wiersze powstały i ponownie przeprowadźmy czyszczenie:

UPDATE WIELKA SET Y='COS TAKIEGO';

VACUUM VERBOSE WIELKA;



Tym razem już było co usunąć, co też zostało wykonane. Zwolniło się sporo miejsca na nowe dane. Możesz też wykonać czyszczenie dla całej bazy danych:

vacuum verbose;

Zarządzanie obiekami

Typy danych

Mamy 6 typów danych w PostgreSQL

- Znakowe
- Liczbowe
- Logiczne
- Daty i czasu
- Specjalne typy PostgreSQL
- BLOB

Typy znakowe

Typy znakowe to:

- **varchar(X)** Ciąg znaków zniennej długości o maksymalnej długości wskazanej przez parametr. W przypadku wstawienia ciągu o długości np. 10 znaków do kolumny zdefiniowanej jako varchar(20), ciąg ten nie będzie uzupełniany spacjami – w przeciwieństwie do typu char
- **char(X)** Ciąg tekstwoy złożony z X znaków. W przypadku wstawienia ciągu o długości np. 10 znaków do kolumny zdefiniowanej jako char(20) - ciąg zostanie uzupełniony spacjami do 20 znaków. Jeśli przekroczymy maksymalną zdefiniowaną liczbę znaków, ciąg zostanie obcięty beż żadnego komunikatu
- **char** Pojedynczy znak
- **text** Nieograniczonej długości ciąg tekstowy. Podobny do varchar, z tą różnicą że jest to typ niestandardowy i niezgodny z ANSI. Mogą pojawić się problemy z rzutowaniem w niektórych językach programowania.

Jedna z pierwszych myśli jaka przychodzi mi do głowy odnośnie typów znakowych to pytanie: jaka jest maksymalna długość wstawianych ciągów tekstowych? Teoretycznie ogranicza nas jedynie maksymalna długość wiersza która od wersji 7.1 wynosi 1GB.

Typy liczbowe

Typów liczbowych mamy 7 i są to:

- **smallint –** dwubajtowa liczba całkowita (-32 768 do 32 767)
- **int –** czterobajtowa liczba całkowita (-2 147 483 648 do 2 147 483 647)
- **serial –** czterobajtowa liczba całkowita której wartość jest nadawana automatycznie przez PostgreSQL (dlatego najczęściej jest wykorzystywana do kluczy głównych).
- **Float(X)** - ośmiobajtowa liczba zmiennoprzecinkowa o minimalnej precyzji X znaków
- **real –** ośmiobajtowa liczba zmiennoprzecinkowa podwójnej precyzji
- **numeric(x,y) –** Liczba rzeczywista. Maksymalnej długości X (przy czym X określa całkowitą długość, włącznie z częścią ułamkową), oraz precyzją określoną przez Y
- **money –** specyficzny dla PostgreSQL typ danych. Tak naprawdę to numeric(9,2), a służy do przechowywania wartości monetarnych

Zarówno typ float jak i real stosują zaokrąglenia! Nie dotyczy to innych typów.

Typ logiczny – boolean

Typ mogący przechowywać wartość logiczną prawda/fałsz. Ciekawostka: jako prawda zostaną zinterpretowane następujące wartości: TRUE, '1','yes','t','true'. Jako fałsz: FALSE,'0','no','n', 'false','f'. Jeśli zacząłeś się już zastanawiać - to liczby 1 i 0 muszą być objęte znakami ' , bez nich PostgreSQL nie przyjmie 0 ani 1 do typu boolean.

Typy daty i czasu

Typy daty i czasu dostępne w PostgreSQL:

- **Date –** przechowuje datę
- **Time –** przechowuje czas
- **Timestamp –** przechowuje datę i czas
- **Interval –** przechowuje różnicę między wartościami typu timestamp

Specjalne typy PostgreSQL i typ Blob

W bazach PostgreSQL są dostępne jeszcze typy takie jak box,line,point,lseq,polygon i odnoszą się one do figur geometrycznych, ale nie są one tematem niniejszego podręcznika – stosowane są np. w PostGis. Typ blob służy do przechowywania danych binarnych, takich jak np. filmy czy zdjęcia.

Tabele

Tworzenie tabel

Tabele tworzymy z użyciem instrukcji CREATE TABLE w taki sposób:

```
create table przykladowa(
x serial,
y real,
z varchar2(50)
);
```

gdzie x,y,z są nazwami kolumn. Można też utworzyć tabelę w innym schemacie poprzedzając jej nazwę nazwą schematu. Musimy mieć oczywiście prawa do tworzenia obiektów we wskazanym schemacie:

```
create table nazwa_schematu.przykladowa(
x serial,
y real,
z varchar2(50)
);
```

Ograniczenia kolumn

Dotychczas tworzyliśmy tabele z kolumnami bez jakichkolwiek ograniczeń. Czasem jednak przydaje się np. możliwość wymuszenia wstawiania wartości do wybranej kolumny, lub unikalności wartości. Poniżej zestawienie ograniczeń dostępnych w bazach danych PostgreSQL:

- **NOT NULL –** w kolumnie z tym ograniczeniem nie możemy wprowadzać NULL. Zawsze trzeba będzie tu wrzucić jakąś wartość.
- **UNIQUE –** wymuszenie unikalności. Specyficzne jest zachowanie w PostgreSQL tego ograniczenia w zestawieniu z NULLami. Zgodnie ze standardem ANSI powinna móc wystąpić tylko jedna wartość NULL w takiej kolumnie, w PostgreSQL nie ma takiego ograniczenia.
- **DEFAULT –** pozwala skonfigurować wartość domyślną dla kolumny
- **CHECK(WARUNKI) –** Pozwala na sprawdzanie spełnienia określonego warunku logicznego dotyczącego danych podczas modyfikacji lub wprowadzania wierszy
- **PRIMARY KEY –** Powoduje założenie klucza głównego na kolumnie.
- **REFERENCES –** zakłada klucz obcy

Przykład użycia ograniczeń:

```
create table dzialy (
id serial primary key,
nazwa varchar
);
create table pracownicy (
        id serial primary key,
        imie varchar not null,
        nazwisko varchar not null,
        email varchar not null unique,
        pensja float check(pensja>2000),
        data_zatrudnienia date default current_date,
        dzial int references dzialy(id)
);
```

Ograniczenia tabel

Niekiedy chcemy zastosować warunek logiczny dla kilku kolumn, albo np. skomponować klucz główny składający się z kilku kolumn. W takich sytuacjach możemy użyć ograniczeń na poziomie tabeli. Przykład:

create table tabelka (

kol1 varchar,

kol2 varchar,

constraint omg_ale_unikalnosc unique(kol1,kol2)

);

Kasowanie tabel

Kasowanie tabel jest bardzo proste i sprowadza się do wydania instrukcji:

drop table nazwa_tabeli;

Tabele tymczasowe

Tabele tymczasowe to tabele które są automatycznie usuwane po zakończeniu sesji. Wykorzystuje się takie cudo np. w procesach ETL. Tworzy się je tak samo jak zwykłe tabele, dodając jedynie słówko „temporary”:

create temporary table tymczasowa (x int,y varchar);

Ograniczenia kluczy obcych

Klucze obce mogą referować wyłącznie do kolumn unikalnych. Wynika to koniecznością jednoznacznego określenia wiersza w tabeli B do którego referuje wiersz z tabeli A. Wyobraźmy sobie że np. mamy tabelę samochody_firmowe i tabelę uzytkownicy. W tabeli samochody_firmowe jest założony klucz obcy do tabeli uzytkownicy, a zawierający ID użytkownika który aktualnie jeździ danym samochodem. Ktoś rozbija firmowego Merca, sprawdzamy kto nim jeździ i widzimy wartość ID=50. Zaglądamy do tabelki uzytkownicy a tam 3 osoby mają taki identyfikator. Kto płaci? ;)

Ograniczenie ON DELETE/UPDATE CASCADE i ON DELETE SET NULL

Wróćmy na moment do naszych przykładowych tabel działy i pracownicy. Pracownicy pracują w działach firmy, a to który w którym determinuje wartość w kolumnie dzia tabeli pracownicy, referującej do kolumny id w tabeli działy. Co powinno się stać z pracownikami kiedy skasujemy dział? Wartości w kolumnie dzial tabeli pracownicy tych pracowników którzy pracują w kasowanym dziale powinny zostać ustawione na NULL, czy też wiersze tych pracowników powinny być skasowane razem z wierszem działu? To zależy od konfiguracji. Domyślnie PostgreSQL nam na to nie pozwoli. W przypadku takiego zapisu:

```
create table pracownicy (
        id serial primary key,
        imie varchar not null,
        nazwisko varchar not null,
        email varchar not null unique,
        pensja float check(pensja>2000),
        data_zatrudnienia date default current_date,
        dzial int references dzialy(id)         on delete cascade
);
```

pracownicy zostaną usunięci razem z działem. W przypadku takiego:

```
create table pracownicy (
        id serial primary key,
        imie varchar not null,
        nazwisko varchar not null,
        email varchar not null unique,
        pensja float check(pensja>2000),
        data_zatrudnienia date default current_date,
        dzial int references dzialy(id)         on delete set null
);
```

wartości kolumny dzial zostaną ustawione na NULL. Ci pracownicy nie będą po prostu przypisani do żadnego działu. Wersja z UPDATE działa podobnie, jednak skasuje wiersze lub ustawi NULL w kolumnie referującej w przypadku zmiany wartości kolumny do której referujemy w drugiej tabeli.

Ograniczenie DEFERRABLE

Z kluczami obcymi wiąże się jeszcze jeden problem. Przypomnijmy sobie tabele pracownicy i działy, oraz łączące je zależności. Wyobraźmy sobie że mamy dział o numerze 10 i pracownika który w nim pracuje. Teraz musimy zmienić numer tego działu na 20. Jeśli spróbujemy najpierw zmienić ID działu, nie uda nam się ponieważ PostgreSQL zauważy że zostałby nam pracownik referujący „w kosmos”. Jeśli spróbujemy najpierw zmienić wartość w kolumnie dział tabeli z pracownikami na 20, też nam się nie uda ponieważ pracownik referowałby do nieistniejącego jeszcze działu. I co teraz? Możemy posłużyć się taką oto konstrukcją:

```
create table pracownicy (
        id serial primary key,
        imie varchar not null,
        nazwisko varchar not null,
        email varchar not null unique,
        pensja float check(pensja>2000),
        data_zatrudnienia date default current_date,
        dzial int references dzialy(id)         deferrable
);
```

Taka konstrukcja pozwoli na naruszenie ograniczenia klucza głównego, ale tylko w ramach transakcji! Update obu tabel będzie musiał być przeprowadzony w ramach jednej transakcji.

Widoki

Widoki są po prostu nazwanymi zapytaniami. Korzystamy z nich tak samo jak z tabel. Można nawet poprzez widoki wstawiać dane do tabel z których korzysta zapytanie na podstawie którego tworzony jest widok. Jedynym ograniczeniem we wstawianiu wierszy poprzez widok jest to by zapytanie nie posiadało : złączeń tabel, agregatów, grupowania, oraz aby wymienione były wszystkie kolumny z ograniczeniem NOT NULL oraz PRIMARY KEY.

create view prac_view as select id,imie,nazwisko,email from pracownicy;

select * from prac_view;

insert into prac_view values (1,'x','y','x@y.pl');

Kasowanie widoków odbywa się z użyciem polecenia:

drop view nazwa_widoku;

Backup i odtwarzanie w PostgreSQL

Istnieją trzy sposoby wykonywania kopi zapasowych w PostgreSQL.

- Z użyciem pg_dump lub pg_dumpall – jest to zrzut aktualnego stanu danych do pliku SQL, nie zawiera informacji o strukturze fizycznej. Są to tylko struktury danych (tabele, indeksy etc), bez informacji o np. przestrzeniach tabel czy strukturze plików i katalogów. Ewentualne przywrócenie bazy będzie możliwe tylko do momentu z którego będzie pochodził zrzut. Bardzo przydatne narzędzie do wykonywania kopii danych np. z serwera produkcyjnego na testowy.
- Backup na poziomie fizycznym. W zasadzie sprowadza się to do skopiowania niezbędnych plików przy wyłączonej bazie. Tutaj również odtworzenie będzie możliwe tylko do punktu w czasie z którego pochodzi backup.
- Archiwizacja ciągła. Jest to najbardziej złożony sposób, ale daje też najwięcej możliwości. Umożliwia między innymi przywrócenie bazy do punktu przed awarią, lub wskazanego przez administratora punktu w czasie.

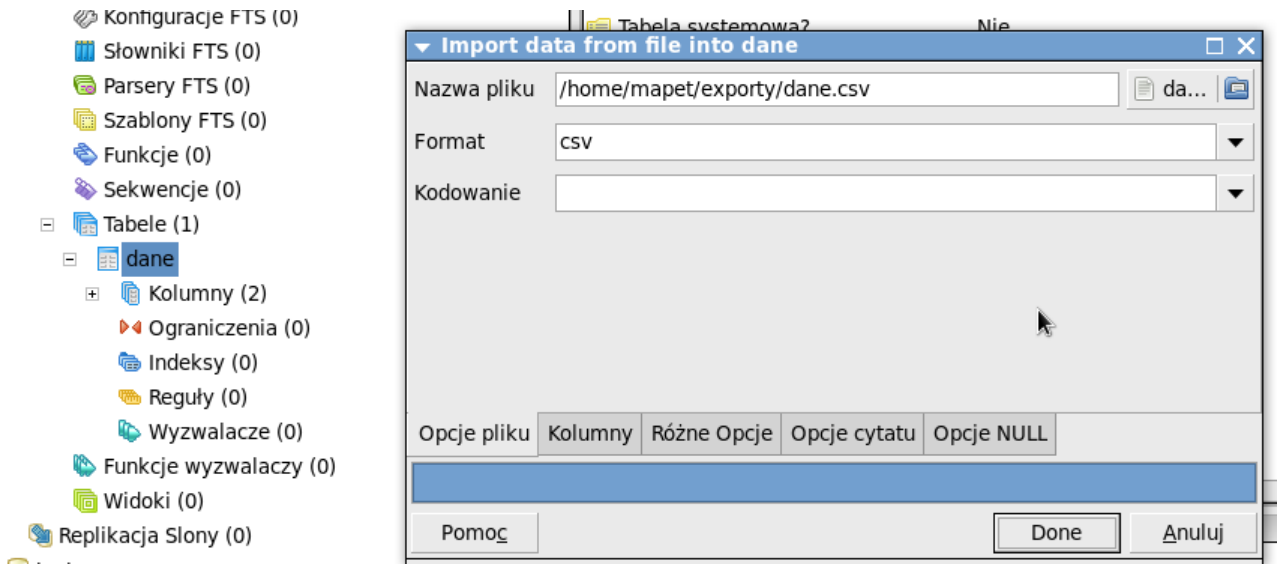
Zajmiemy się tymi trzema metodami po kolei i zastanowimy się nad plusami i minusami wykorzystania poszczególnych.

Na początek stworzyłem sobie tabelkę zawierającą dwie proste kolumny i wrzucam do niej dane, aby w ogóle było co backupować :)



Wrzucam do tej tabeli 161 MB bzdurnych danych. Jeśli masz ochotę iść za mną krok w krok, to wrzuciłem plik który importuję pod adres : http://www.jsystems.pl/storage/postgres/dane.csv

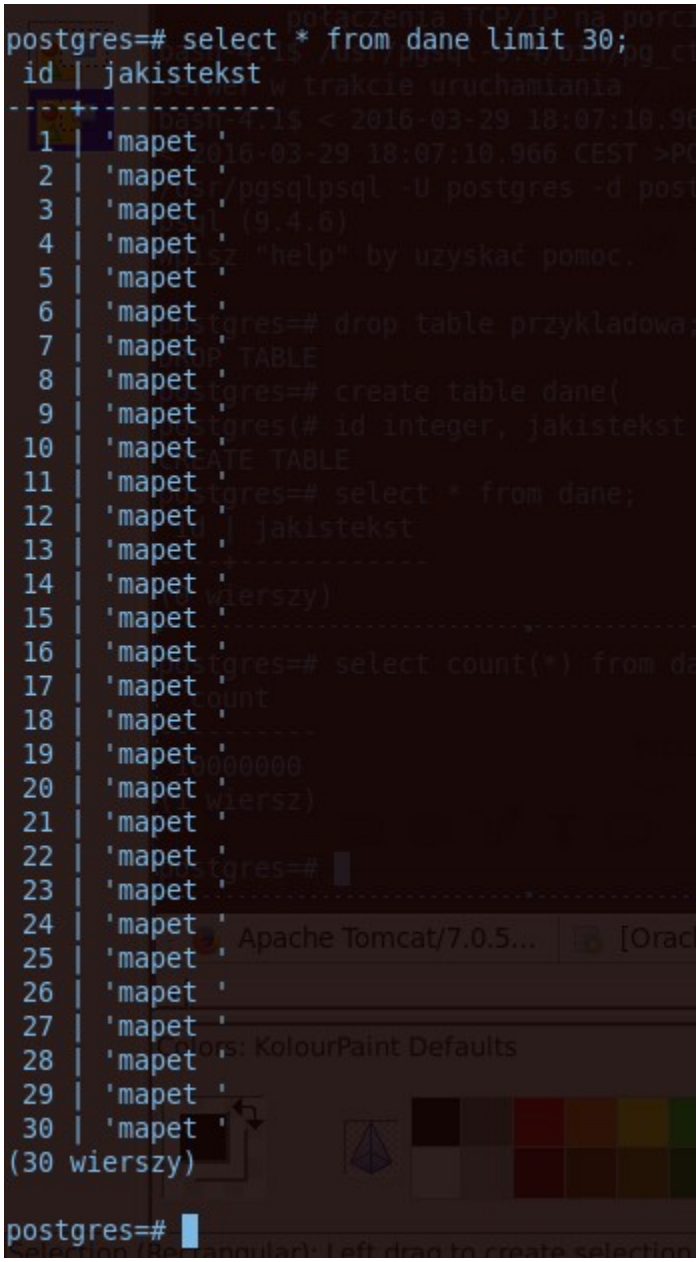
Importuję dane z poziomu PgAdmina. Klikamy prawym przyciskiem na nazwę tabeli, wybieramy „importuj”. W okienku które wyskoczy podajemy położenie naszego pliku, format ustawiamy na CSV, a w zakładce „różne opcje” jako rozdzielacz wybieramy średnik.



Upewniam się jeszcze że dane zostały wrzucone:



I zawartość tabeli:



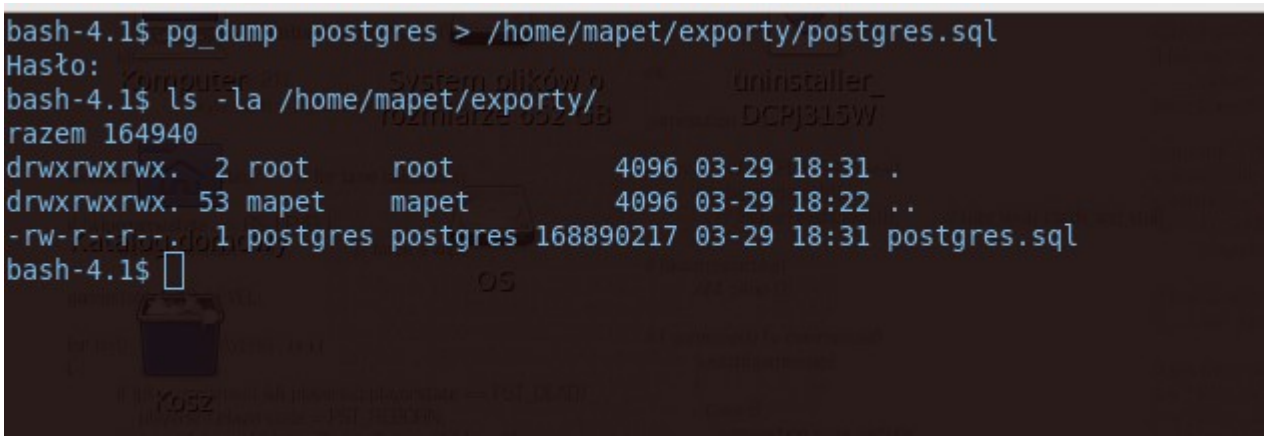
i tak w naszej tabelce znalazło się 10 milionów ponumerowanych mapetów ;)

Backup lokalny i zdalny z użyciem narzędzia PG_DUMP

Korzystając z tego narzędzia utworzymy skrypt który będzie zawierał zwyczajne polecenia SQL tworzące tabele i ładujące dane. To jest kopia na poziomie logicznym!

Aby wykonać backup wybranej bazy podajemy jako parametry dla pg_dump'a nazwę bazy której kopię chcemy wykonać, oraz ścieżkę do pliku do którego ma zostać zrzucona.

pg_dump postgres > /home/mapet/exporty/postgres.sql



Jeśli otworzysz teraz ten plik z użyciem jakiegoś edytora tekstu (ale lepiej przygotuj się na to, że otwieranie tego pliku może chwilę potrwać) powinieneś zobaczyć mniej więcej taką zawartość:

postgres.sql

```
--
-- PostgreSQL database dump
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;

--
-- Name: postgres; Type: COMMENT; Schema: -; Owner: postgres
--

COMMENT ON DATABASE postgres IS 'default administrative connection database';


--
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;


--
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
--

COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';


SET search_path = public, pg_catalog;

SET default_tablespace = '';

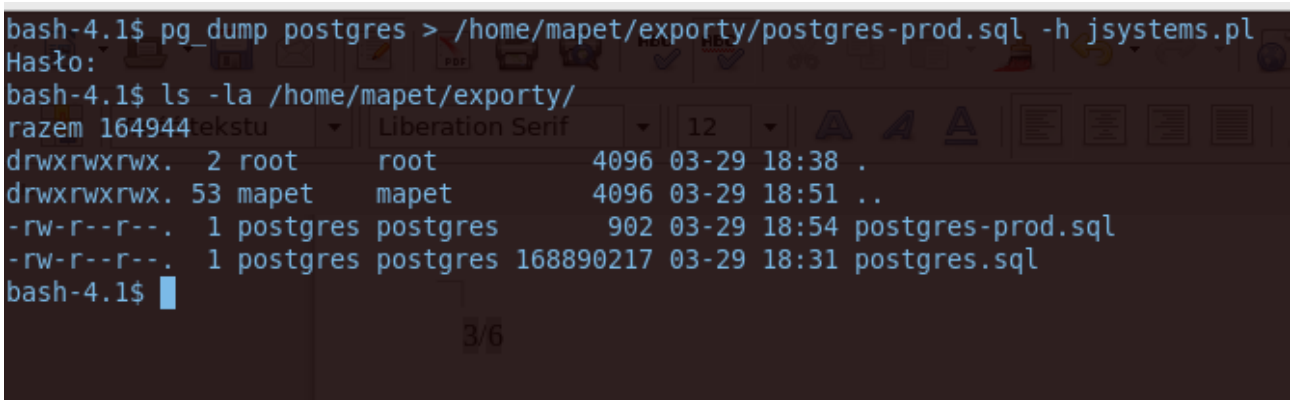
SET default_with_oids = false;

--
-- Name: dane; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE dane (
    id integer,
    jakistekst character varying
);
```

Jak widzisz są to komendy tworzące struktury danych i ładujące dane. Zapewne już się domyślasz w jaki sposób odtwarzamy bazę z takiego pliku ;)

Zanim jednak do tego dojdziemy chciałbym omówić jeszcze kilka ważnych elementów związanych z wykonywaniem tego typu zrzutów. Eksportować możesz również bazy znajdujące się na innym hoście – wystarczy dodać przełącznik **-h** i adres IP albo nazwę domenową hosta z którego chcemy eksportować:

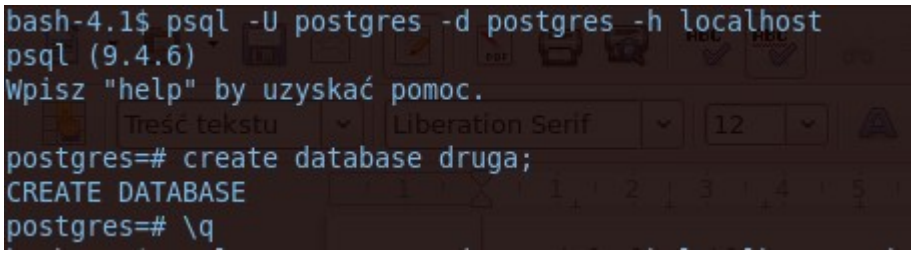


Analogicznie możemy też wykorzystać przełącznik **-p** by ustawić port po którym chcemy się łączyć ze zdalną bazą.

Pytanie które samo się nasuwa: co jeśli dane zmienią się podczas wykonywania backupu? Jest tak jakbyśmy sobie tego życzyli – zawartość zrzutu będzie konsystentna. To znaczy że stan wszystkich danych będzie pochodził z momentu rozpoczęcia backupu. Nic więc nam się nie „rozjedzie”. Wykonywanie takiego zrzutu nie blokuje też bazy – wszystkie procesy nadal działają i dane można zmieniać w trakcie wykonywania backupu.

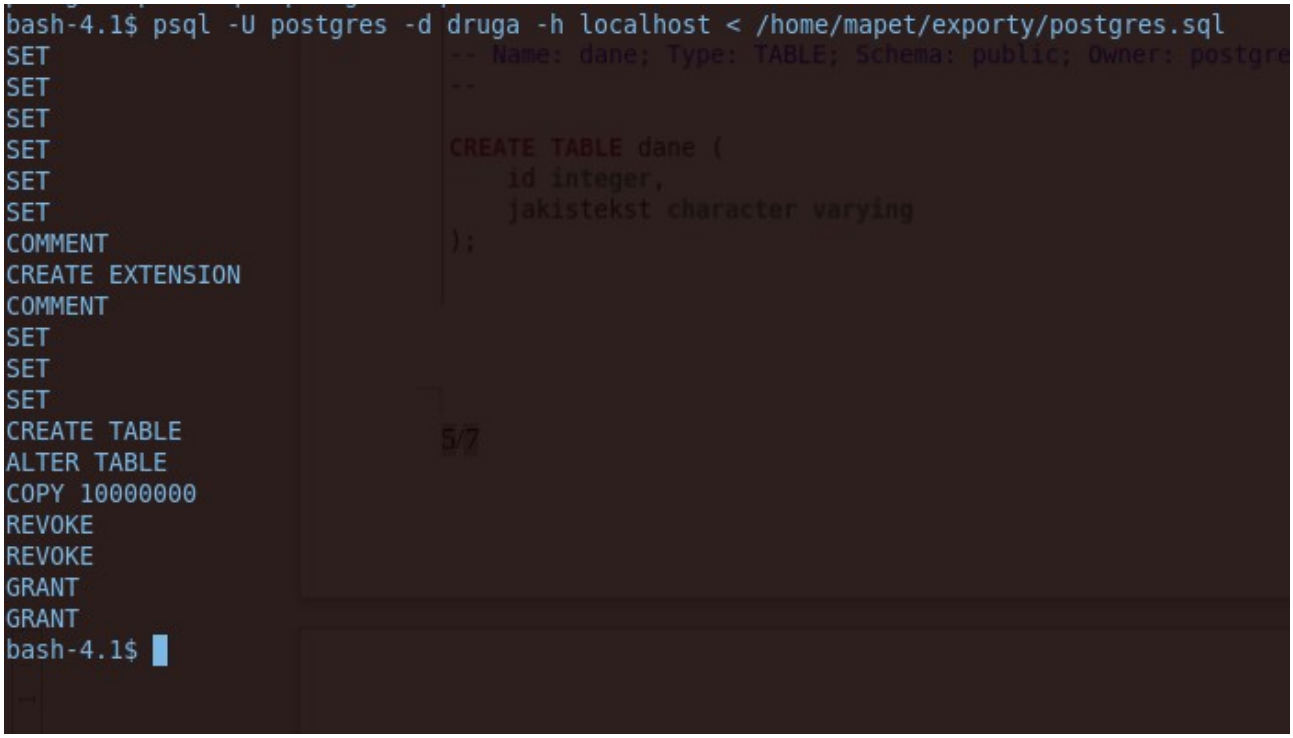
Odtwarzanie lokalne i zdalne bazy danych

Skoro już wyrzuciliśmy sobie dane do pliku, to teraz go załadujmy. Oczywiście możemy dane wrzucić do tej samej bazy danych, ale nic nie stoi też na przeszkodzie by stworzyć nową bazę danych i załadować do niej zrzut danych z innej (tak właśnie zrobię). Bazę będziemy musieli jednak stworzyć sami, nie zostanie ona utworzona automatycznie:

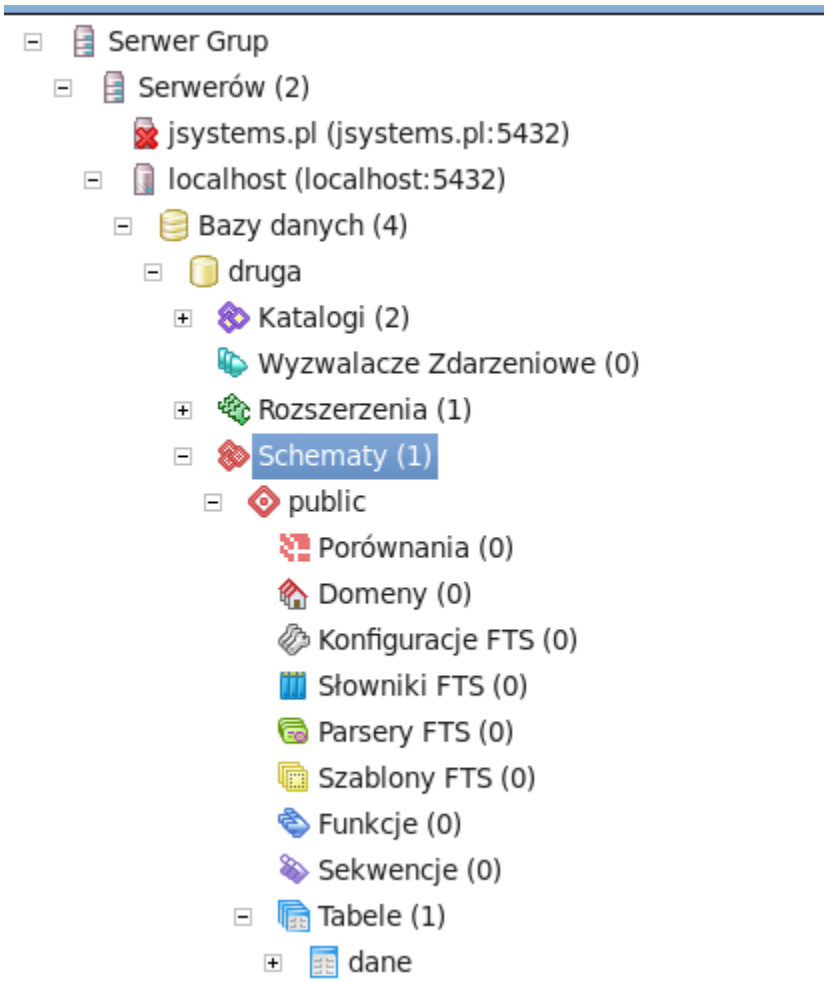


Sam plik zrzutu wciągamy z użyciem dobrze znanego nam już narzędzia psql (wszak plik zrzutu zawiera po prostu instrukcje SQL):

psql -U postgres -d druga -h localhost < /home/mapet/exporty/postgres.sql



Sprawdźmy więc czy w nowej bazie danych pojawiła się nasza tabela:



Jeśli wrócimy pamięcią do pierwszych rozdziałów tych materiałów, to przypomnimy sobie że z użyciem narzędzia psql możemy też podpinać się do zdalnej bazy danych. Czemu więc nie odtworzyć bazy na zdalnym hoście?

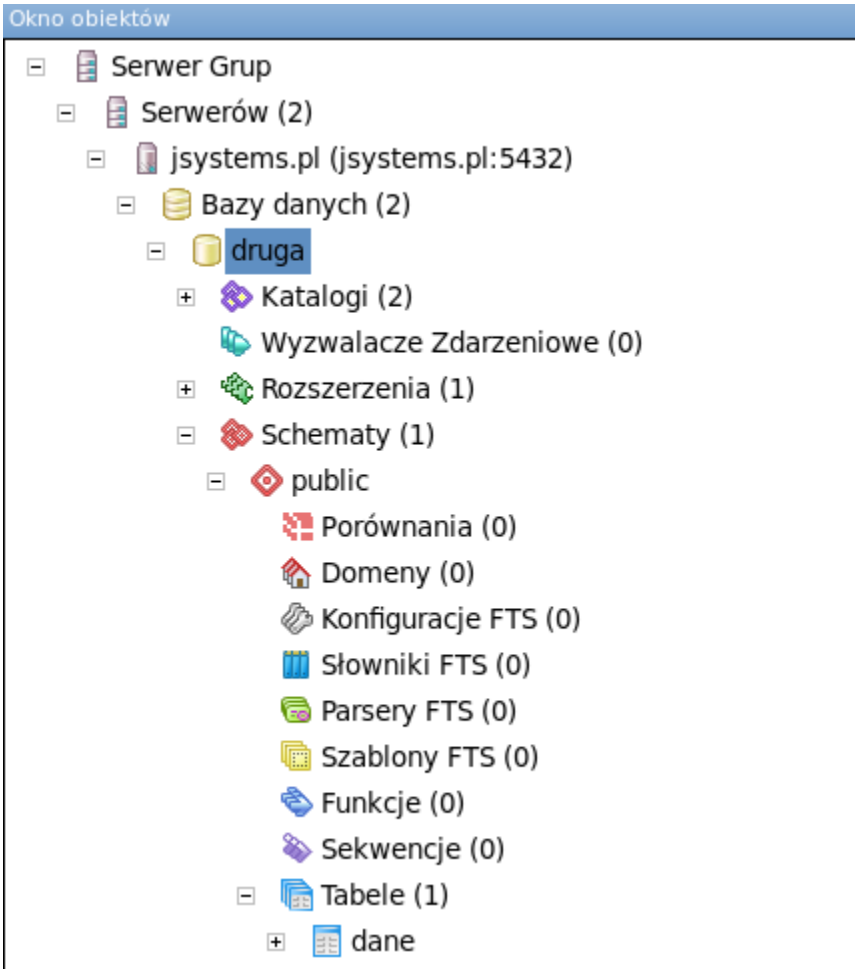
psql -U postgres -d druga -h jsystems.pl < /home/mapet/exporty/postgres.sql

Żaden problem, wystarczy po przełączniku **-h** podać adres IP albo nazwę domenową zdalnego hosta.

```
bash-4.1$ psql -U postgres -d postgres -h jsystems.pl
Hasło użytkownika postgres:
psql (9.4.6)
Wpisz "help" by uzyskać pomoc.

postgres=# create database druga;
CREATE DATABASE
postgres=# \q
bash-4.1$ psql -U postgres -d druga -h jsystems.pl < /home/mapet/exporty/postgres.sql
Hasło użytkownika postgres:
SET
SET
SET
SET
SET
SET
COMMENT
CREATE EXTENSION
COMMENT
SET
SET
SET
CREATE TABLE
```

Sprawdzamy :



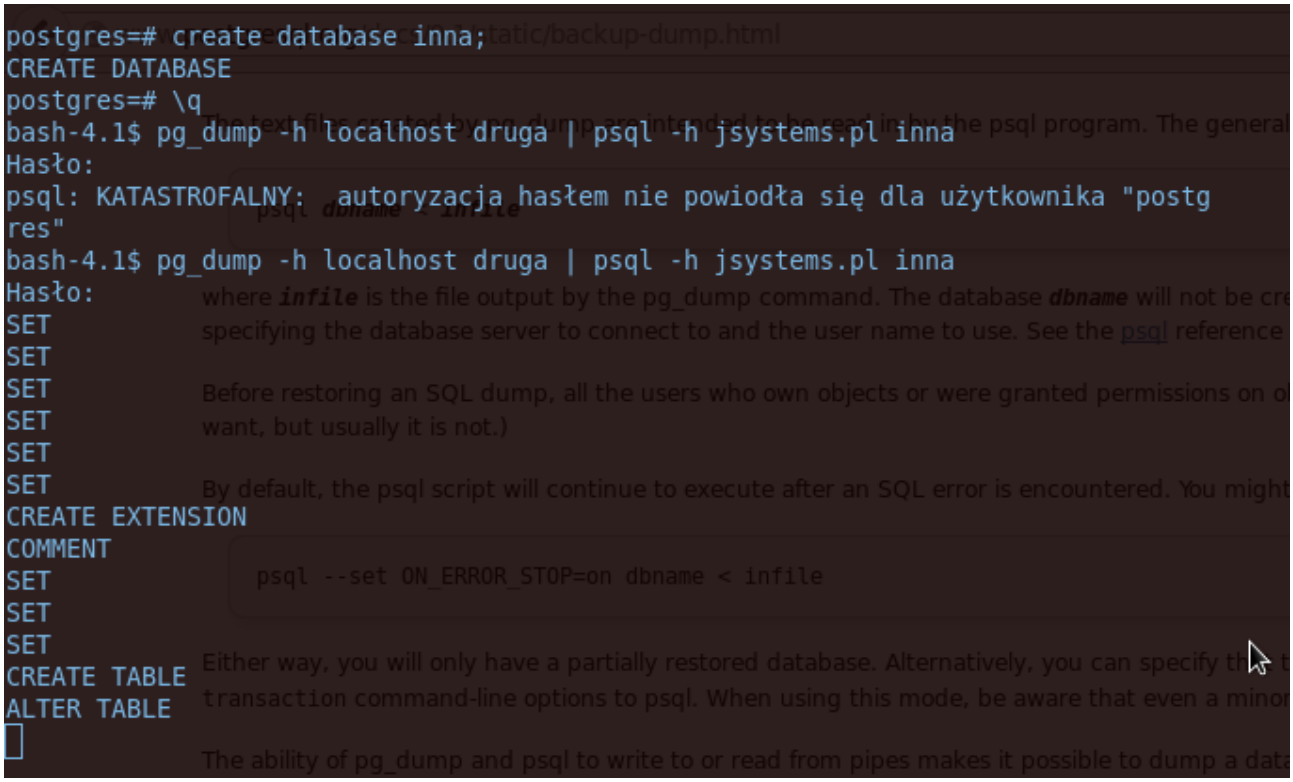
Szybkie kopiowanie baz między dwoma klastrami

No to teraz „z grubej rury” :). Szybkie kopiowanie baz danych między dwoma klastrami z czego jeden jest zdalny. Nic nie stoi oczywiście na przeszkodzie by obie bazy były zdalne.

pg_dump -h localhost druga | psql -h jsystems.pl inna

Zasadniczo sprowadza się to do eksportu z jednej bazy i jednoczesnego wciągnięcia do innej – bez wytwarzania pliku eksportu. Strumień wytwarzany przez pg_dump'a tym razem nie ląduje w pliku tylko jest od razu przekierowany do PSQL którego podpinamy do zdalnej bazy ;) Wcześniej utworzyłem sobie nową bazę o nazwie „inna” na serwerze zdalnym:

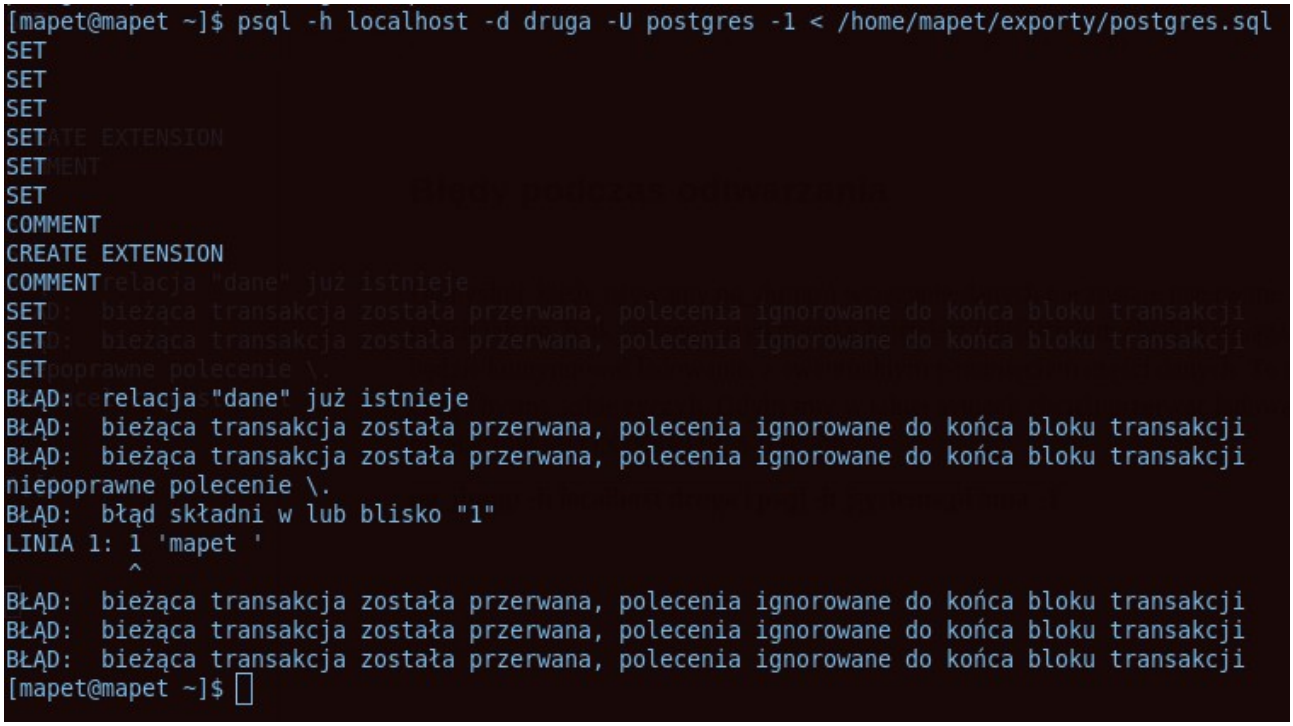
pg_dump -h localhost druga | psql -h jsystems.pl inna



Błędy podczas odtwarzania

Domyślnie kiedy używamy pg_dump'a wciąganie danych nie zostaje przerwane w wyniku błędów takich jak np. brak jakiegoś użytkownika który jest właścicielem obiektu wciąganego. PG_DUMP będzie kontynuował ładowanie, z ewentualnym pominięciem części danych. To nie zawsze jest to czego byśmy sobie życzyli. Gdybyśmy w takiej sytuacji chcieli przerwać ładowanie, należy dodać przełącznik **-1**

psql -h localhost -d druga -U postgres -1 < /home/mapet/exporty/postgres.sql



Ten przełącznik sprawi, że całość skryptu jest wykonywana jako jedna transakcja i jako taka zostanie w całości zatwierdzona albo wycofana.

Bardziej uważni czytelnicy z pewnością wpadną na pomysł, że ponieważ plik zrzutu jest zwykłym edytowalnym plikiem możemy wykomentować to co nie jest nam potrzebne i wciągnąć całą resztę :)

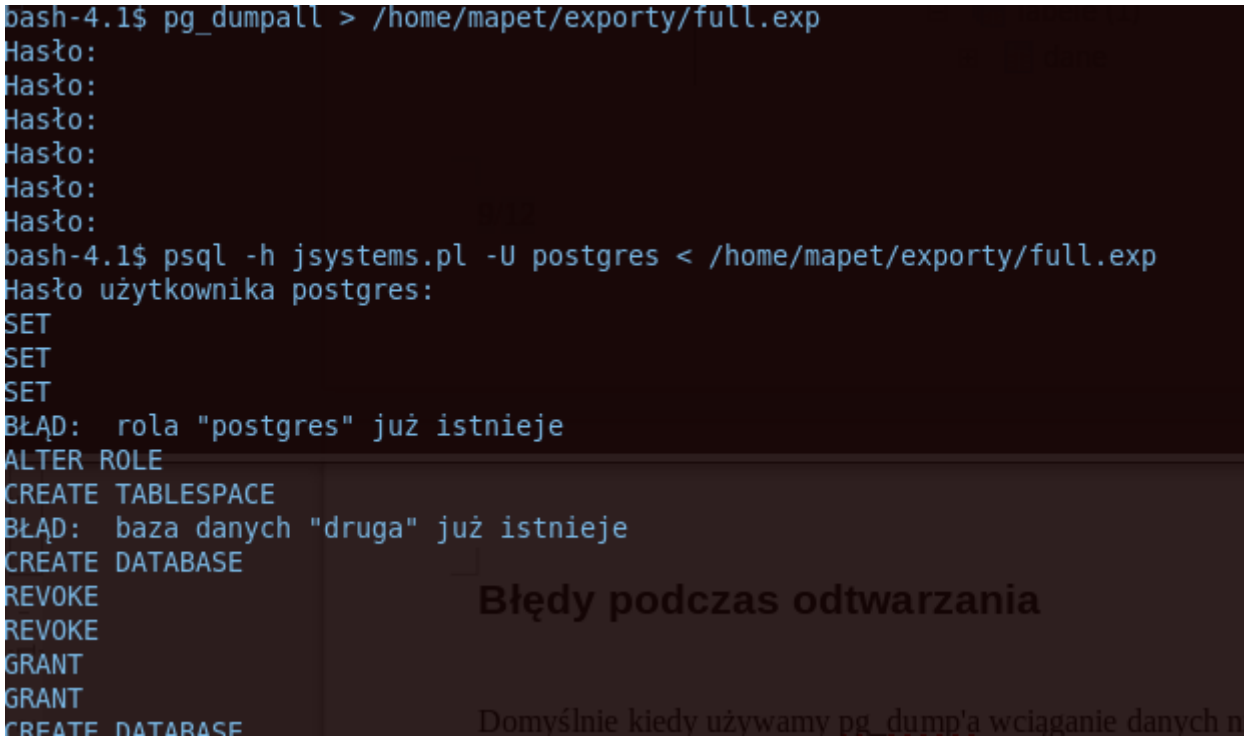
Backup i odtwarzanie całego klastra

Dotychczas wykonywaliśmy kopie zapasowe pojedynczych baz danych. Średnio to wygodne jeśli tych baz mamy dużo, a chcielibyśmy wykonywać regularne kopie zapasowe wszystkich. Ponadto użycie pg_dump powoduje że nie są zrzucane informacje dotyczące całego klastra – jak przestrzenie tabel czy użyszkodnicy. Jeśli chcielibyśmy takie informacje zawrzeć w kopii zapasowej, albo skopiować między dwoma serwerami cały klaster musimy skorzystać z narzędzia pg_dumpall. PG_DUMPALL w pliku zrzutu zawrze też informacje na temat przestrzeni tabel i użytkowników.

Wyrzucenie całego klastra do pliku:

pg_dumpall > /home/mapet/exporty/full.exp

Taki wytworzony plik również (podobnie jak przy pg_dump) wciągamy przy użyciu narzędzia psql:



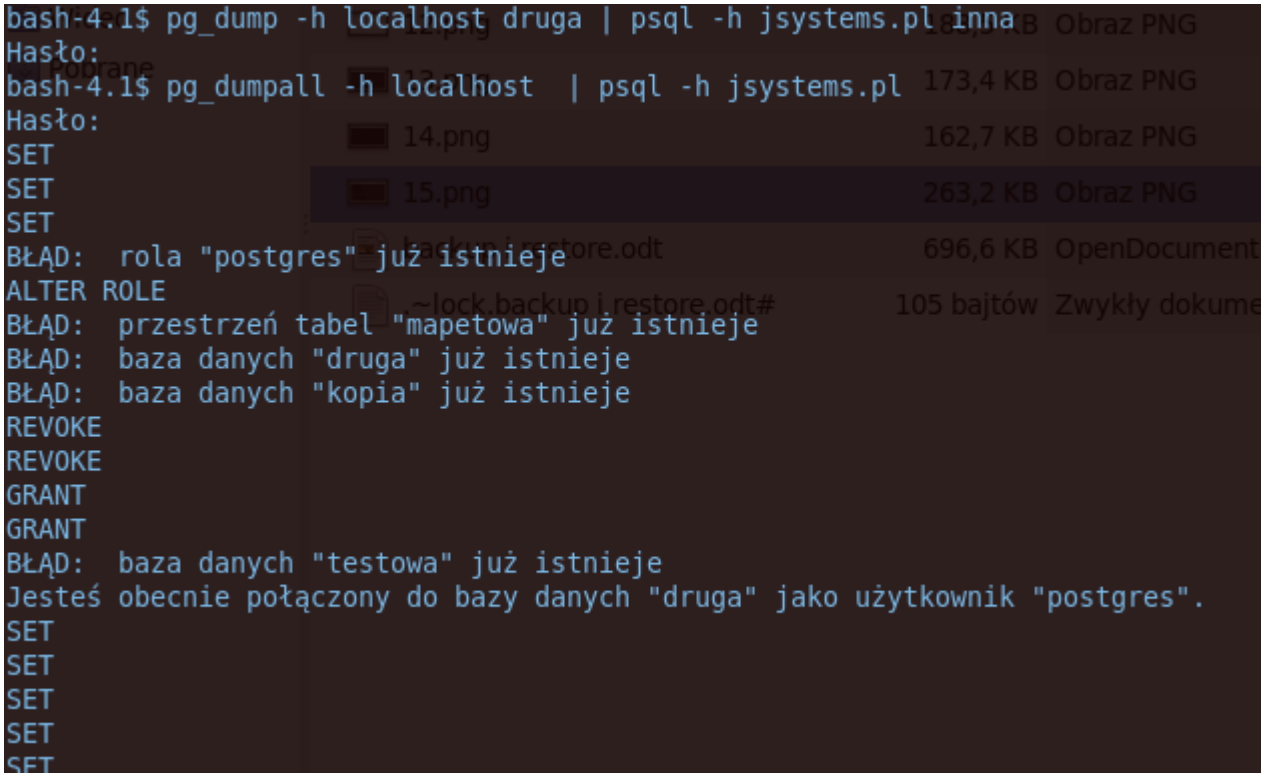
Jak widzimy, podczas importu podejmowana jest też próba tworzenia baz danych i przestrzeni tabel. Jeśli takowe już istnieją w docelowej bazie danych, to psql zgłosi tylko błąd i przejdzie dalej, tak jak widzimy na powyższej ilustracji.

Idąc za ciosem, a pamiętając różne wariacje z eksportem i importem plików pg_dump'a i tutaj mamy rząd możliwości. Gdybyśmy zechcieli zrzucić zdalny klaster do lokalnego pliku:

pg_dumpall -h jsystems.pl -U postgres > /home/mapet/exporty/full2.exp

albo gdybyśmy zechcieli skopiować cały klaster „w locie”:

pg_dumpall -h localhost | psql -h jsystems.pl



Nieco irytująca rzecz związana z pg_dumpall – będzie eksportował bazy liniowo, a przy każdej kolejnej zapyta o hasło....

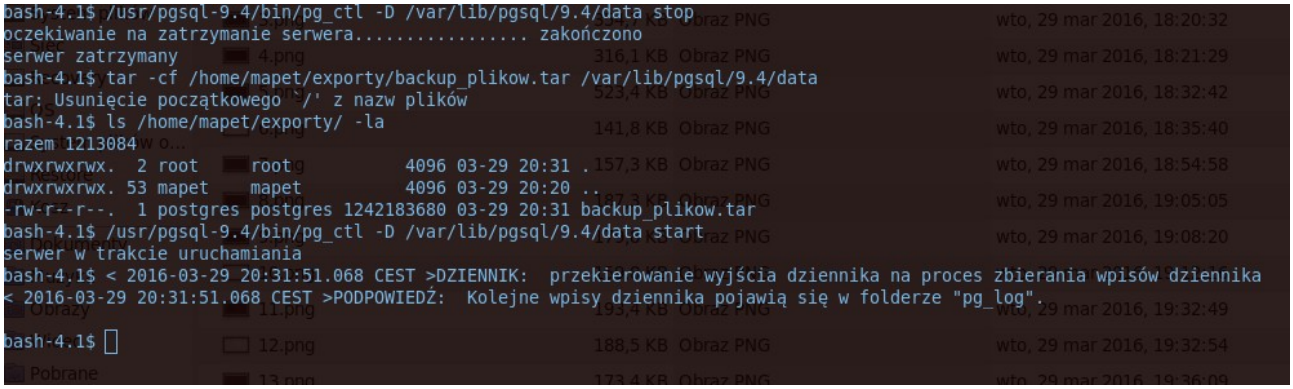
Backup plików danych na poziomie fizycznym

Możemy też wykonać kopię plików danych. Sprowadza się to do skopiowania plików, ja użyłem akurat narzędzia tar aby sobie je jednocześnie skompresować. Ważne by te pliki były konsystentne. Podczas takiego sposobu wykonywania backupu, nic nie przeszkadza klastrowi PostgreSQL w normalnym działaniu, co jednocześnie oznacza że pliki te mogą podlegać zmianom w trakcie kopiowania. Taki backup byłby bezużyteczny. Dlatego też przed wykonaniem backupu należy klaster wyłączyć!

/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data stop

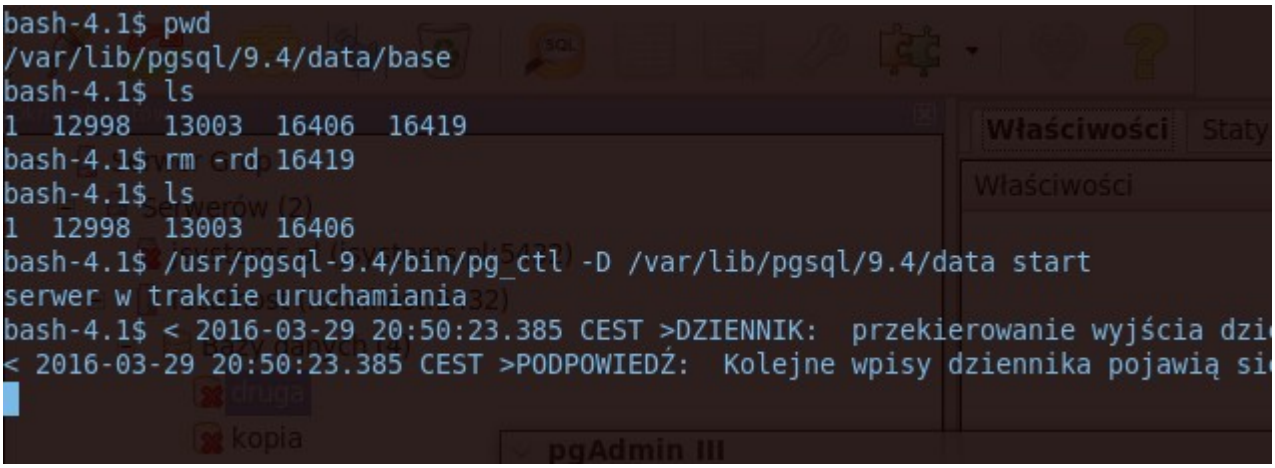
tar -cf /home/mapet/exporty/backup_plikow.tar /var/lib/pgsql/9.4/data

/usr/pgsql-9.4/bin/pg_ctl -D /var/lib/pgsql/9.4/data start

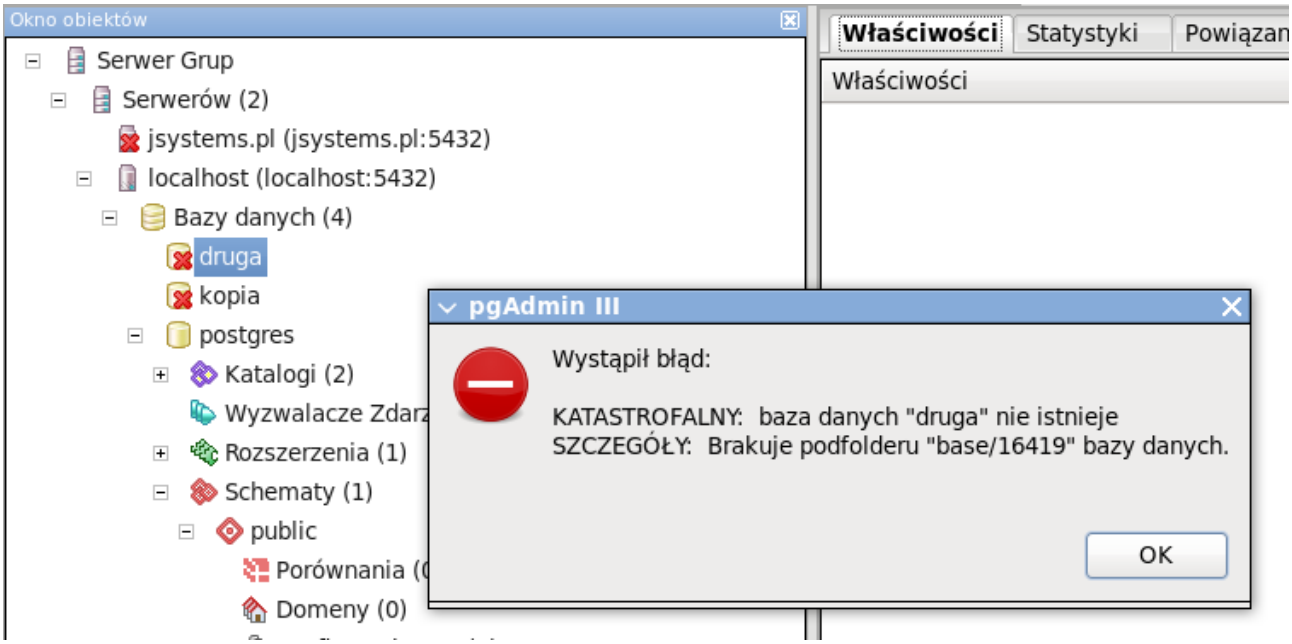


Na powyższym screenie widzimy też utworzony plik backupu we wskazanym miejscu. Taki sposób wykonywania backupu będzie z reguły szybszy niż z użyciem pg_dumpall, ale też jednocześnie będzie zajmował więcej miejsca.

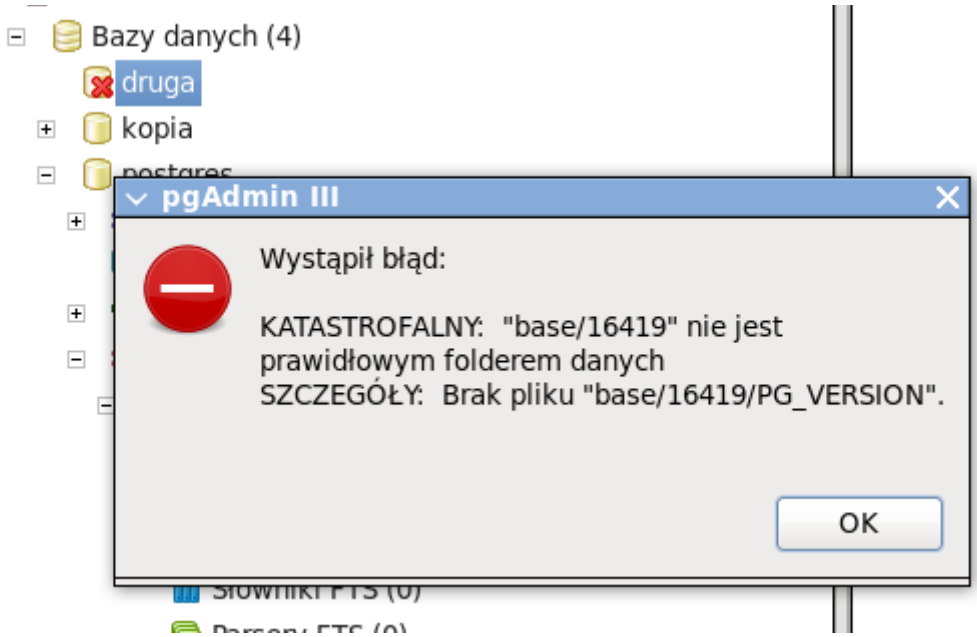
Dla pewności sprawdźmy też jak przebiega odtwarzanie takiego backupu. Najpierw położyłem klaster, a następnie usunąłem katalog z plikami jednej z baz. Podniosłem klaster. Jak widzimy PostgreSQL nie zgłosił sprzeciwu. Problemy zaczną się dopiero pojawiać przy próbie dostępu do danych z tej bazy.



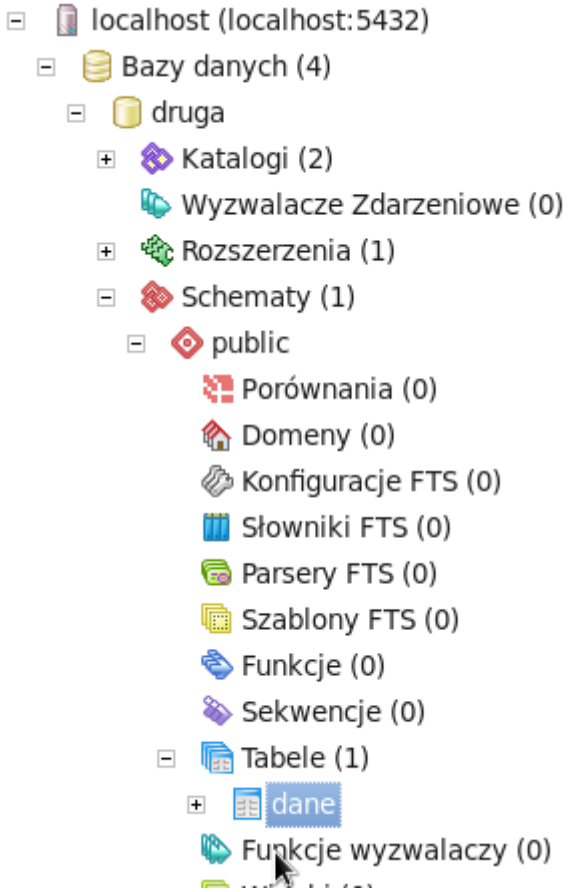
Przy próbie dobrania się do bazy:



KATASTROFALNY nawet błąd :) Ok, brakuje mu katalogu. Stworzę więc go ręcznie i ponowię próbę:



Tym razem będzie zgłaszał problem przy dostępie do poszczególnych plików – tutaj akurat nic ważnego – plik zawierający informacje o wersji oprogramowania naszego klastra. Ponieważ jestem czepliwy i mam obyczaj sprawdzać poprawność dokumentacji (a to wynika akurat z bardzo poważnego błędu w dokumentacji baz Oracle przez który kiedyś najadłem się sporo nerwów i pewnie straciłem nieco włosów), to postanowiłem nieco utrudnić zadanie. Zanim przystąpiłem do odtwarzania, wykonałem wprawdzie na innej bazie spore operacje zmieniające dane. Chodziło mi o to, żeby przewinęło się kilka dzienników WAL, oraz żeby powstało też kilka checkpointów – tak żeby zweryfikować czy PostgreSQL później nie będzie na mnie krzyczał że np. pliki WAL zawierają zatwierdzone wpisy z punktu w czasie przed stworzeniem plików danych czy coś takiego. Dzienniki WAL dotyczą wszak całego klastra, a nie pojedynczych baz danych. Wiecie, przezorny zawsze ubezpieczony. A i jeszcze jedno – odtarowałem i umieściłem przywracane pliki danych podczas gdy cały klaster zadziałał. Słowem – postarałem się o to by całość przebiegała w zbliżonych warunkach co gdyby wystąpiła jakaś normalna awaria – z reguły minęłoby trochę czasu zanim zorientowalibyśmy się że w ogóle coś się dzieje. Ku mojemu zaskoczeniu po wykonaniu tych czynności i ponownej próbie dostępu do danych nie było problemów:


localhost (localhost:5432)
Bazy danych (4)
druga
Katalogi (2)
Wyzwalacze Zdarzeniowe (0)
Rozszerzenia (1)
Schematy (1)
public
Porównania (0)
Domeny (0)
Konfiguracje FTS (0)
Słowniki FTS (0)
Parsery FTS (0)
Szablony FTS (0)
Funkcje (0)
Sekwencje (0)
Tabele (1)
dane
Funkcje wyzwalaczy (0)

Archiwizacja ciągła i przywracanie do puntu tuż przed awarią

W PostgreSQL działa mechanizm podobny do redo i archivelogów w Oracle. W PostgreSQL mowa o mechaniźmie WAL – Write Achead Log. Działa to tak :

1. Użytkownik zmienia jakieś dane
2. Informacja o zmianie trafia do plików WAL – w których przechowywane są informacje o zmianach na wypadek gdyby zmiany nie zostały utrwalone w plikach danych a nastąpiłoby np. odcięcie zasilania.
3. Zmiana danych następuje na poziomie bloków w buforze – od tej pory noszą miano brudnych bloków.
4. Kiedy następuje checkpoint brudne bloki są utrwalane na poziomie plików danych. Checkpoint następuje wtedy gdy upłynie czas określony w parametrze checkpoint_timeout, lub gdy skończy się miejsce w plikach WAL (domyślnie są to trzy pliki o rozmiarze 16MB każdy. Ich ilość określamy w parametrze checkpoint_segments).
5. Jeśli archiwizacja ciągła jest wyłączona (a jest wyłączona domyślnie) bieżące pliki WAL są kasowane z katalogu pg_xlog, a w ich miejsce powstają nowe o kolejnych numerach. Jeśli w wyniku np. długo wykonującego się backupu w katalogu pg_xlog powstanie więcej niż 3 pliki WAL taka ich ilość pozostanie i będzie rotowała, ale nadal w paczkach po tyle plików ile określimy w parametrze checkpoint_segments. Jeśli archiwizacja ciągła jest włączona to przed skasowaniem pliki WAL kopiowane są w miejsce archiwizacji które wskażesz w konfiguracji.

W punktach powyżej pojawia się pojęcie archiwizacji ciągłej. Co to takiego? Wyobraź sobie że zarządzasz dużym systemem którego baza danych oparta jest o PostgreSQL. Robisz backupy raz w tygodniu korzystając z pg_dump lub pg_dumpall. Czy to nam wystarczy? Nie zawsze. Jeśli zrobiłeś backup np. w niedzielę a w środę nastąpi awariaa to z takiego zrzutu odzyskasz stan bazy z niedzieli. Czy to się sprawdzi np. w banku? Dla klienta banku pewnie byłoby to korzystne, o ile backup był zrobiony tuż po wypłatach ;) Dla administratora i samego banku nieco mniej. W takiej sytuacji chcielibyśmy odzyskać stan bazy z momentu tuż przed awarią, a nie sprzed paru dni. Może też nastąpić taka sytuacja że w wyniku ludzkiego błędu będzie trzeba cofnąć stan bazy do wczoraj (a backup jest sprzed przykładowo 5 dni). W obu przypadkach mechanizm archiwizacji ciągłej będzie dla nas rozwiązaniem. Ok, ale jak to działa? Wyobraź sobie że masz kopię zapasową plików danych sprzed tygodnia, ale w ciągu tego tygodnia zbierałeś wszystkie pliki WAL jakie powstawały. Masz więc punkt wyjścia w postaci kopii plików danych i pliki WAL zawierające wszystkie instrukcje zmieniające cokolwiek w bazie od backupu do teraz. Mając taki zestaw możesz przywrócić pliki danych z kopii na pierwotne miejsce, a następnie powtórzyć wszystkie operacje z WAL lub ich część – do wskazanego punktu w czasie. Fajne? Pewnie, bo może nam uratować skórę. Jeśli nie włączysz archiwizacji ciągłej, przywracanie bazy w taki sposób nie będzie możliwe, ponieważ miałbyś backup sprzed jakiegoś czasu – np. kilku dni i pliki WAL zawierające informacje o zmianach z np. ostatniej godziny. Mielibyśmy kilka dni luki. Potrzebujemy całej historii operacji od backupu do teraz, by przywracać bazę do punktu w czasie. Aby włączyć archiwizację ciągłą i umożliwić sobie odtwarzanie baz do punktu w czasie musimy zmienić 3 parametry w pliku postgresql.conf

Edytujemy plik postgresql.conf:

```
bash-4.1$ whoami
postgres
bash-4.1$ nano /var/lib/pgsql/9.4/data/postgresql.conf
```

Odremowujemy parametr WAL_LEVEL i ustawiamy jego wartość na archive lub hot_standby. Jego domyślna wartość to minimal, która pozwala jedynie na odtworzenie bazy po awarii. Wartość archive powoduje dodawanie do WAL informacji które umożliwiają nam wykorzystanie archiwizacji ciągłej, a hot_standby pozwala na uruchamianie serwera w trybie hot_standby i puszczanie zapytań działających w trybie tylko do odczytu.

```
#------------------------------------------------------------------------------
# WRITE AHEAD LOG
#------------------------------------------------------------------------------

# - Settings -

wal_level = archive                     # minimal, archive, hot_standby, or logical
                                        # (change requires restart)
```

Odremowujemy teraz parametr ARCHIVE_MODE i ustawiamy jego wartość na ON. Umożliwia on uruchomienie archiwizacji.

```
# - Archiving -

archive_mode = on               # allows archiving to be done
                                # (change requires restart)
```

Zarchiwizowane pliki WAL muszą gdzieś lądować, więc przygotowujemy sobie odpowiedni katalog. Ważne jest by jego właścicielem był użytkownik systemowy postgres. Zadbaj o to by katalog leżał na jakimś dysku ze sporą ilością miejsca, ponieważ zarchiwizowane pliki WAL sporo go zajmą.

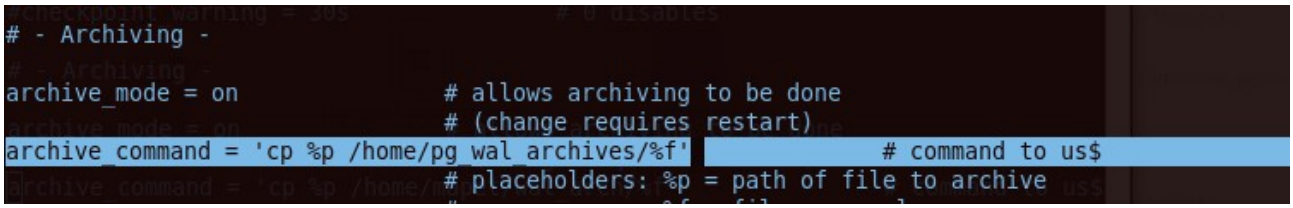
Przygotowanie katalogu do którego będą wrzucane zarchiwizowane pliki WAL:

```
[root@vps-1077604-8206 home]# mkdir /home/pg_wal_archives
[root@vps-1077604-8206 home]# chown postgres /home/pg_wal_archives/
[root@vps-1077604-8206 home]#
```

Została jeszcze jedna rzecz. Pliki WAL same się nie skopiują, więc musimy podać komendę która to

zrobi. Można by sobie zadać pytanie – a dlaczego nie wystarczy wskazać ścieżki do katalogu? Komenda którą podajemy to nie musi być polecenie cp. Dzięki takiej konstrukcji możemy tu wykorzystać rsync, ftp, albo jakikolwiek inny mechanizm.

Odszukujemy parametr ARCHIVE_COMMAND i podajemy komendę kopiującą oraz odremowujemy go:



Ponieważ plik postgresql.conf nie jest czytany „na żywo”, musimy przeładować konfigurację. Przeładowanie konfiguracji:

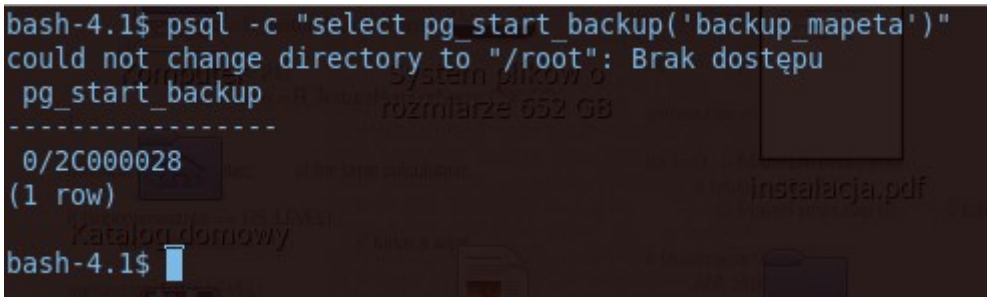
service postgresql-9.4 restart



Zadbaliśmy już o kopiowanie plików WAL, teraz potrzebujemy naszego punktu wyjścia jakim będzie kopia zapasowa. Nie może to być kopia stworzona z użyciem pg_dump ani pg_dumpall ponieważ są to kopie logiczne a nie fizyczne. Aby wykonać kopię zapasową online, musmy przejść do trybu backupu. Sprawia on że zmiany w bazie nie będą utrwalane w plikach danych, a jedynie w plikach WAL. Zostaną one utrwalone dopiero po zakończeniu trybu backupu. Chodzi o spójność danych, nie możemy doprowadzić do sytuacji że coś się pozmienia w tych plikach w trakcie kopiowania.

Tryb backupu:

psql -c „select pg_start_backup('etykieta_backupu')”



Utworzymy sobie też katalog pod backup:

```
[root@vps-1077604-8206 backups]# pwd
/var/lib/pgsql/9.4/backups
[root@vps-1077604-8206 backups]# mkdir 20160404
[root@vps-1077604-8206 backups]# chown postgres 20160404
[root@vps-1077604-8206 backups]# ls -la
razem 12
drwx------ 3 postgres postgres 4096 04-04 17:25 .
drwx------ 4 postgres postgres 4096 03-22 11:41 ..
drwxr-xr-x 2 postgres dba      4096 04-04 17:25 20160404
[root@vps-1077604-8206 backups]#
```

W dalszej kolejności kopiujemy pliki. Możemy wykonać to dowolną komendą systemu operacyjnego.

```
bash-4.1$ tar cfP /var/lib/pgsql/9.4/backups/20160404/backup.tar /var/lib/pgsql/9.4/data
bash-4.1$ ls /var/lib/pgsql/9.4/backups/20160404/
backup.tar
bash-4.1$
```

Po zakończeniu kopiowania wychodzimy z trybu backupu:

psql -c „select pg_stop_backup()”

```
bash-4.1$ psql -c "select pg_stop_backup()"
could not change directory to "/root": Brak dostępu
UWAGA:  pg_stop_backup kompletny, zarchiwizowano wszystkie wymagane segmenty WAL
 pg_stop_backup
----------------
 0/2C0000B8
(1 row)

bash-4.1$
```

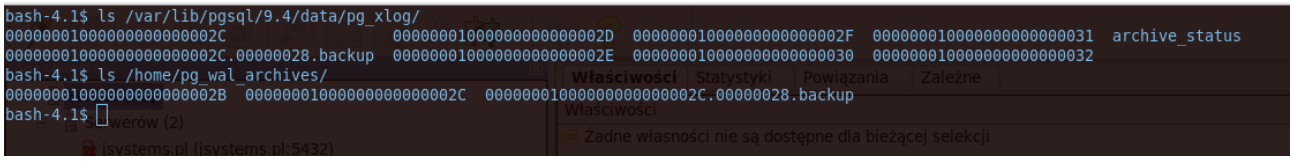
Uruchomiłem tryb backupu i stworzyłem tabelkę do której wrzucam duży wolumen danych. Chodzi mi o to by troszkę plików WAL zostało dodanych do naszego nowego katalogu. Stworzyłem pustą tabelkę i uruchomiłem prosty skrypt w języku Plpg/SQL który do tej nowej tabelki dodaje 10 milionów wierszy.

```
create table pobackupie(
id integer primary key,
cos varchar
);

do
$$
begin
 for x in 1..10000000 loop
        insert into pobackupie values (x,'x po raz '||x);
 end loop;
end
$$;

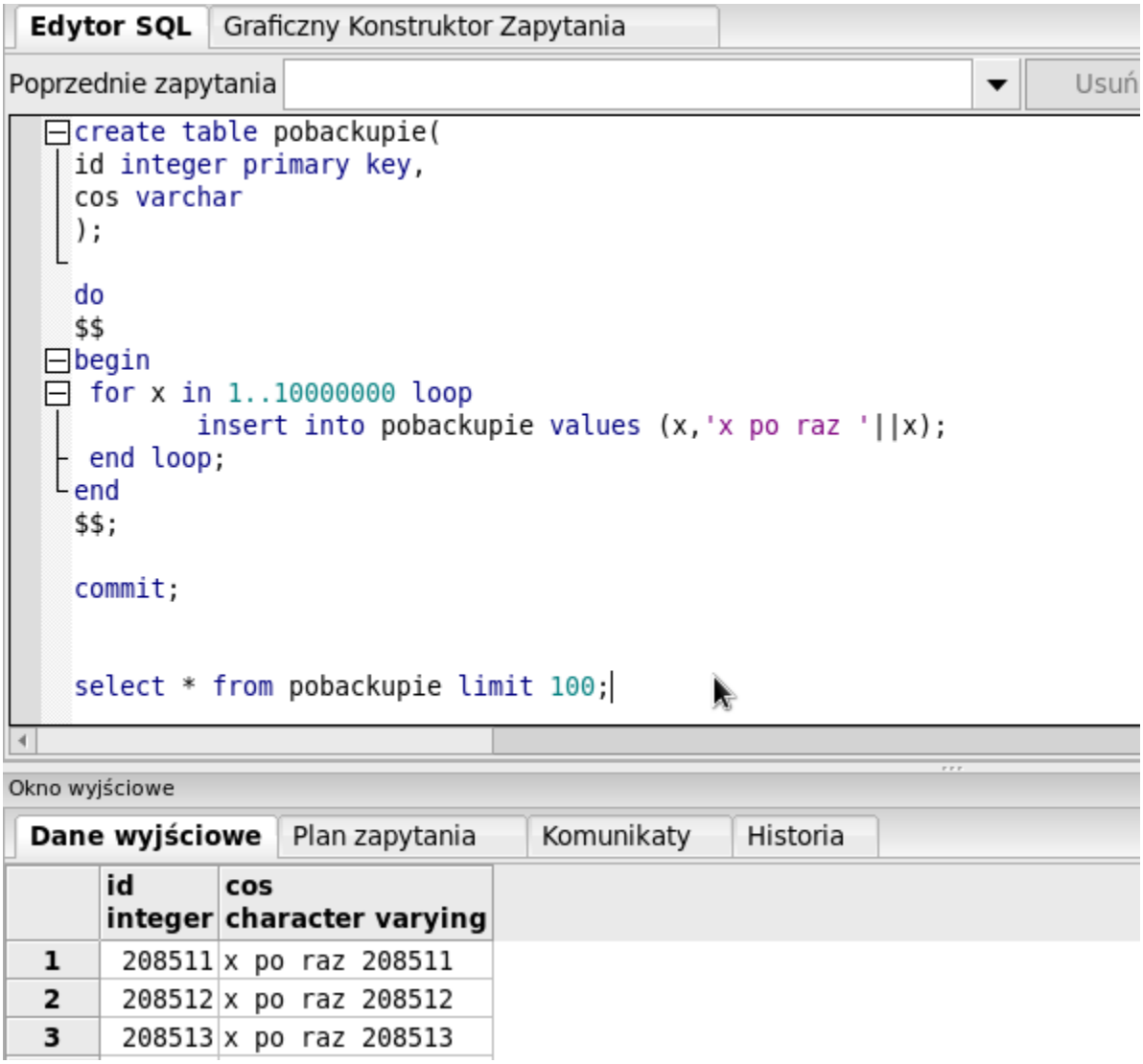
commit;
```

Zawartość katalogu pg_xlog i nowego katalogu na zarchwizowane pliki WAL w trakcie wykonywania powyższego skryptu:

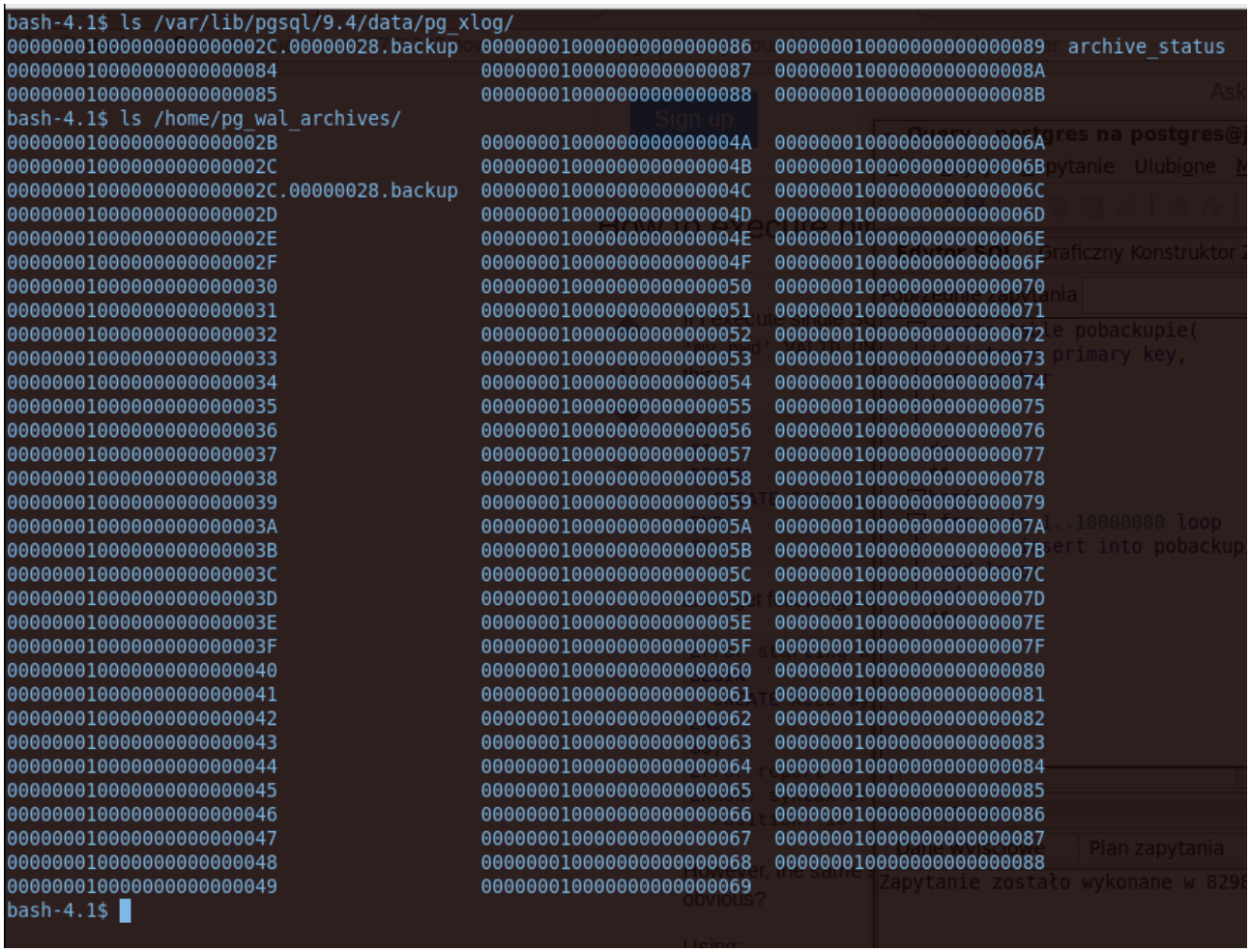


Zwróć uwagę że tabelkę stworzyłem po wykonaniu backupu, a więc w kopii zapasowej nie ma o niej informacji! To ważne bo za chwilę będziemy odtwarzać bazę z użyciem wszystkich zarchiwizowanych plików WAL.

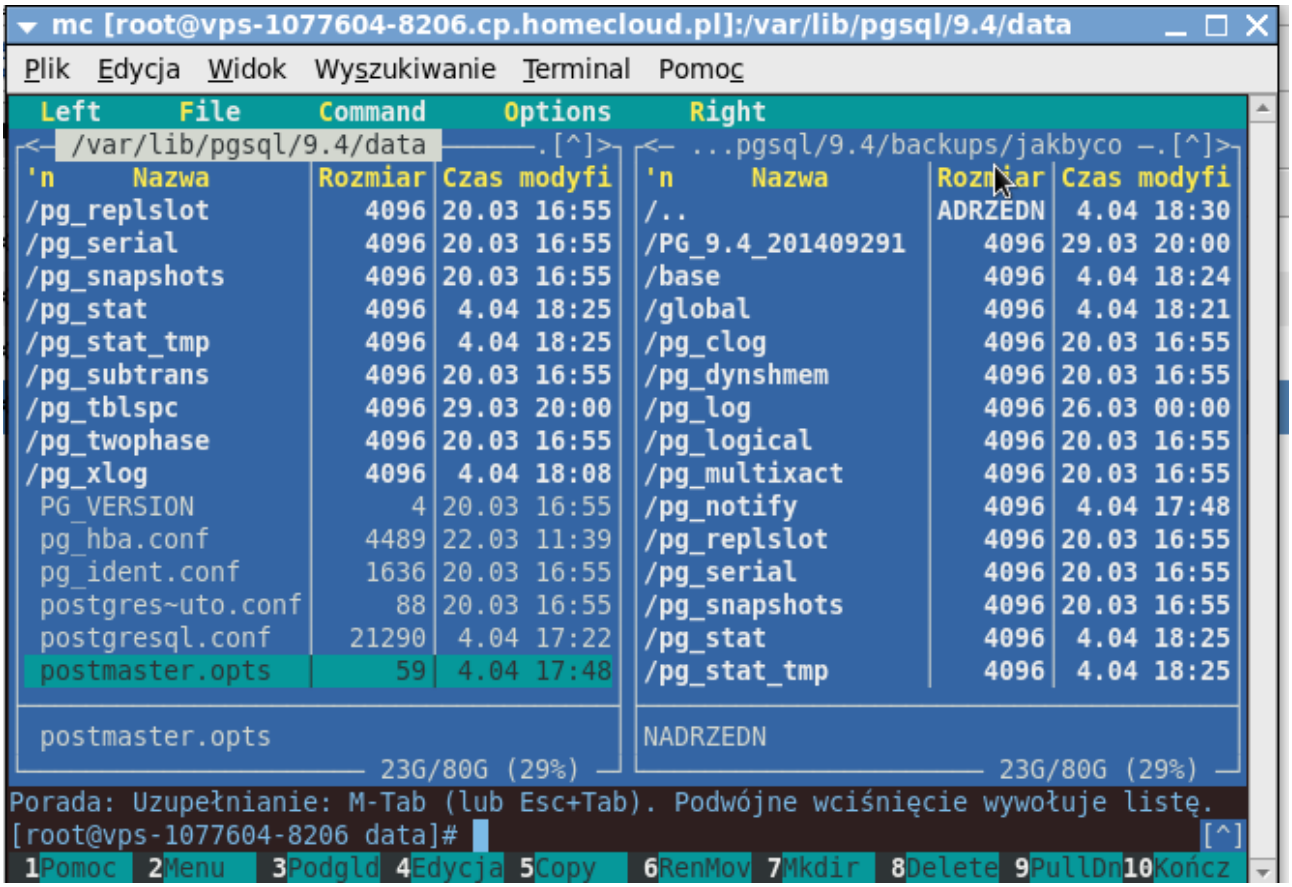
Po zakończeniu mielenia skryptu nasza tabelka zawiera sporo danych:



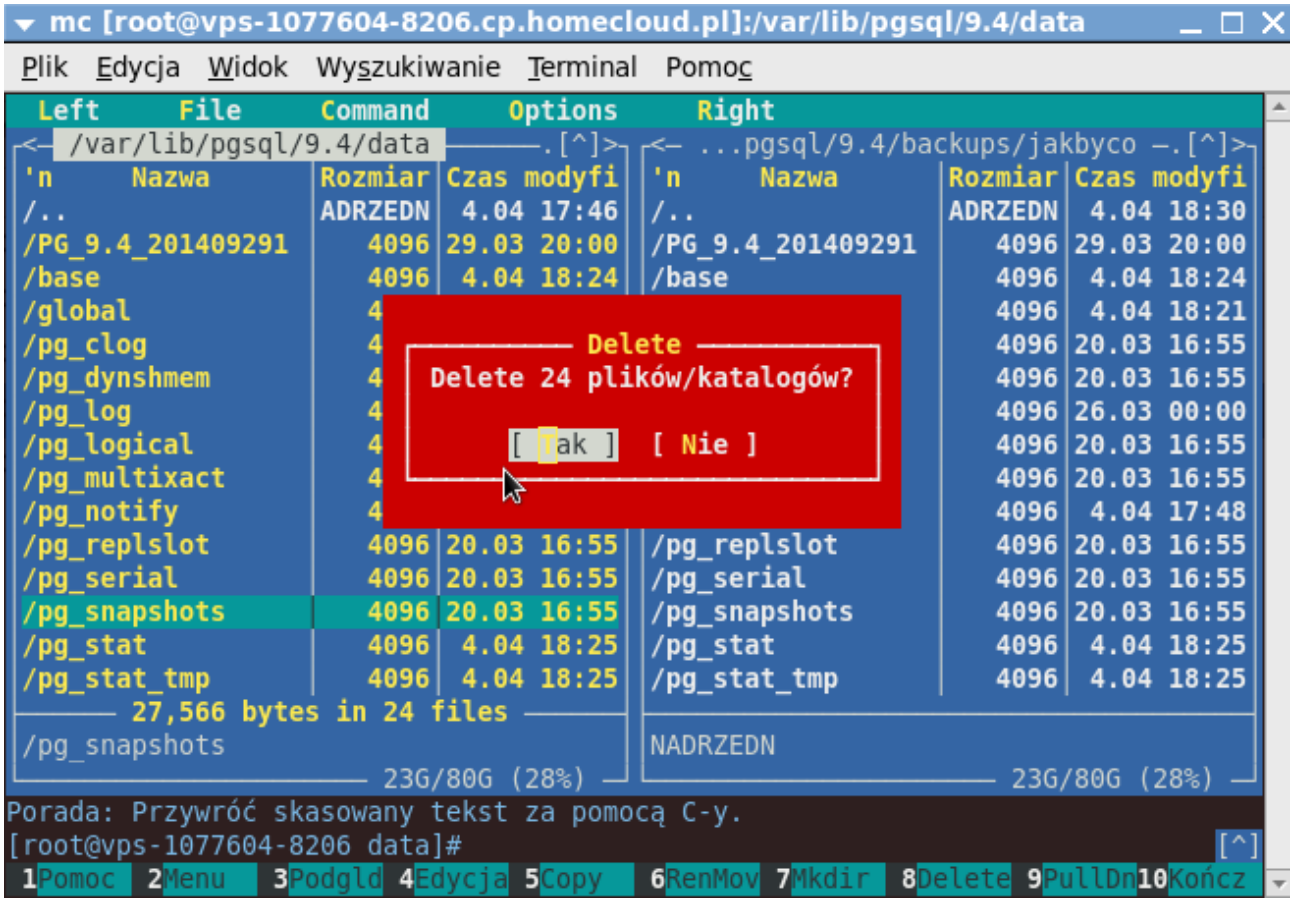
Mój katalog ze zarchiwizowanymi plikami WAL też trochę się wypełnił. Powstało w nim 89 plików, każdy po 16 MB - razem 1424 MB, a więc prawie 1,5 GB!

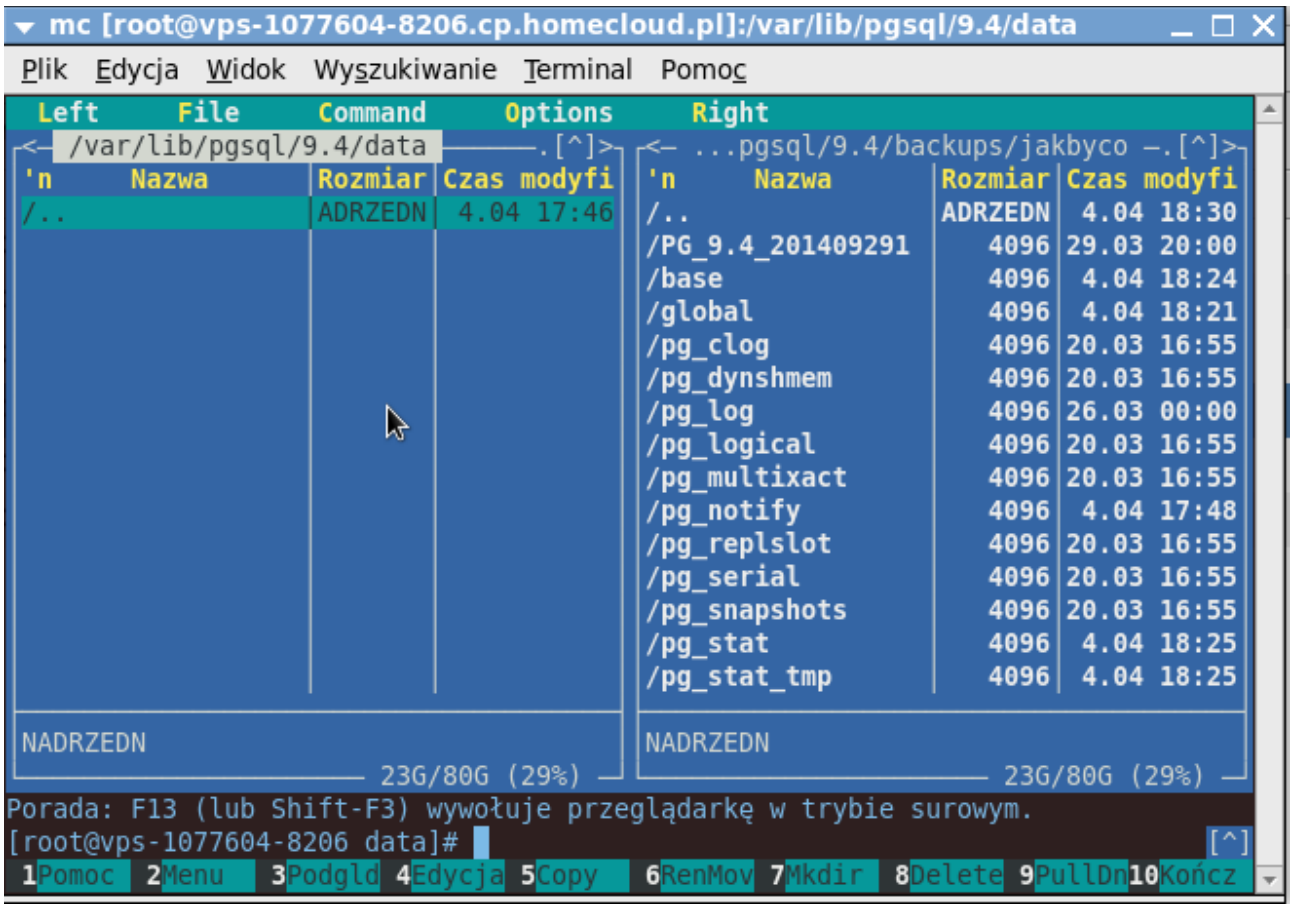


Zasymulujemy teraz awarię która będzie wymagała odtwarzania bazy do punktu tuż przed awarią. Zanim zaczniemy odtwarzać, warto zrobić kopię aktualnego stanu plików danych, nawet jeśli są one uszkodzone. Może się tak zdarzyć że w trakcie odtwarzania coś pójdzie nie tak i jeszcze bardziej napsujemy.

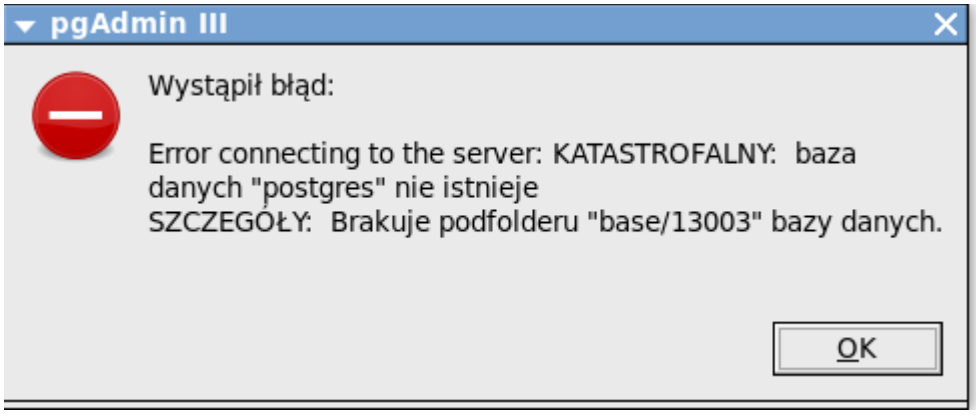


Przed odtwarzaniem usuwamy wszystkie pliki danych z katalogu $PGDATA, będą one zastępowane tymi z kopii zapasowej.





Przypominam że klaster jest cały czas uruchomiony. Próba dostępu do jakiejkolwiek bazy kończy się w tej chwili tak:

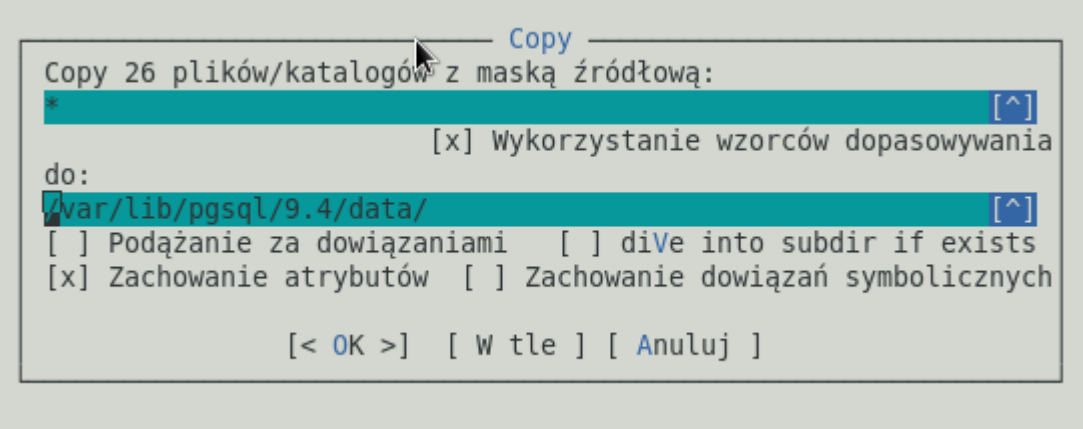


Zatrzymujemy serwer:

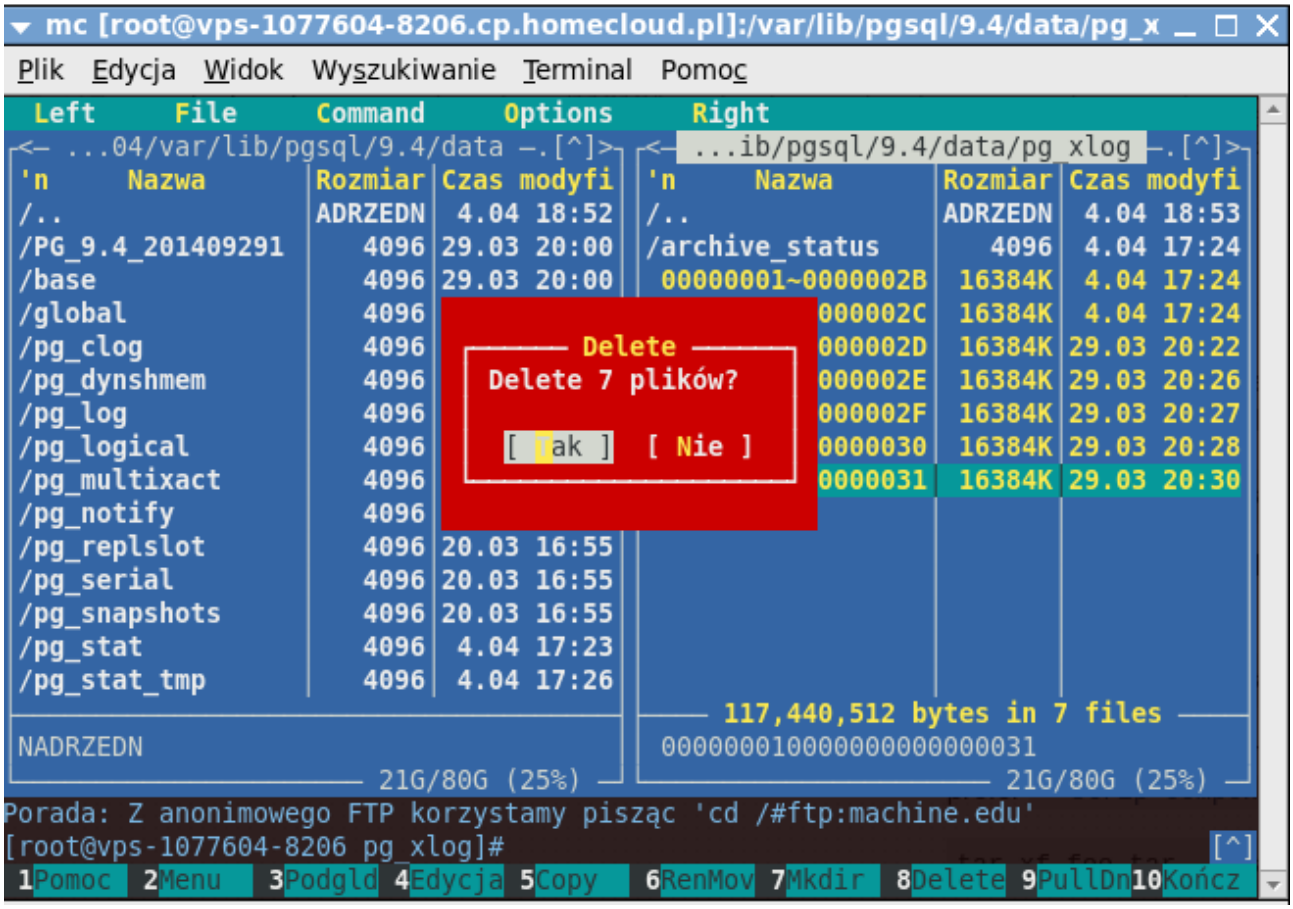
service postgresql-9.4 stop



Kopiujemy pliki z backupu :

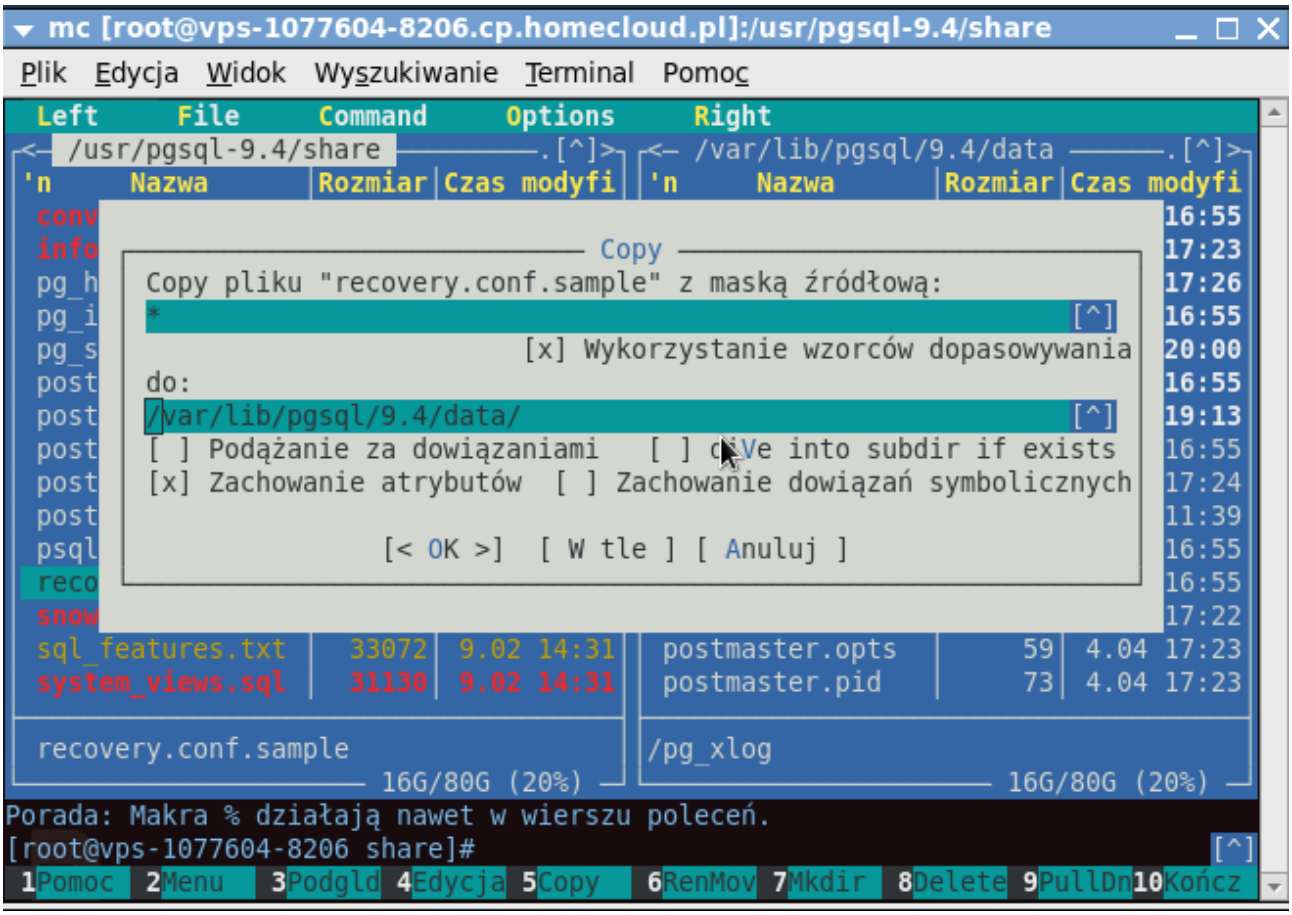


Wywalamy zawartość pg_xlog bo i tak nie koreluje z aktualnym stanem:



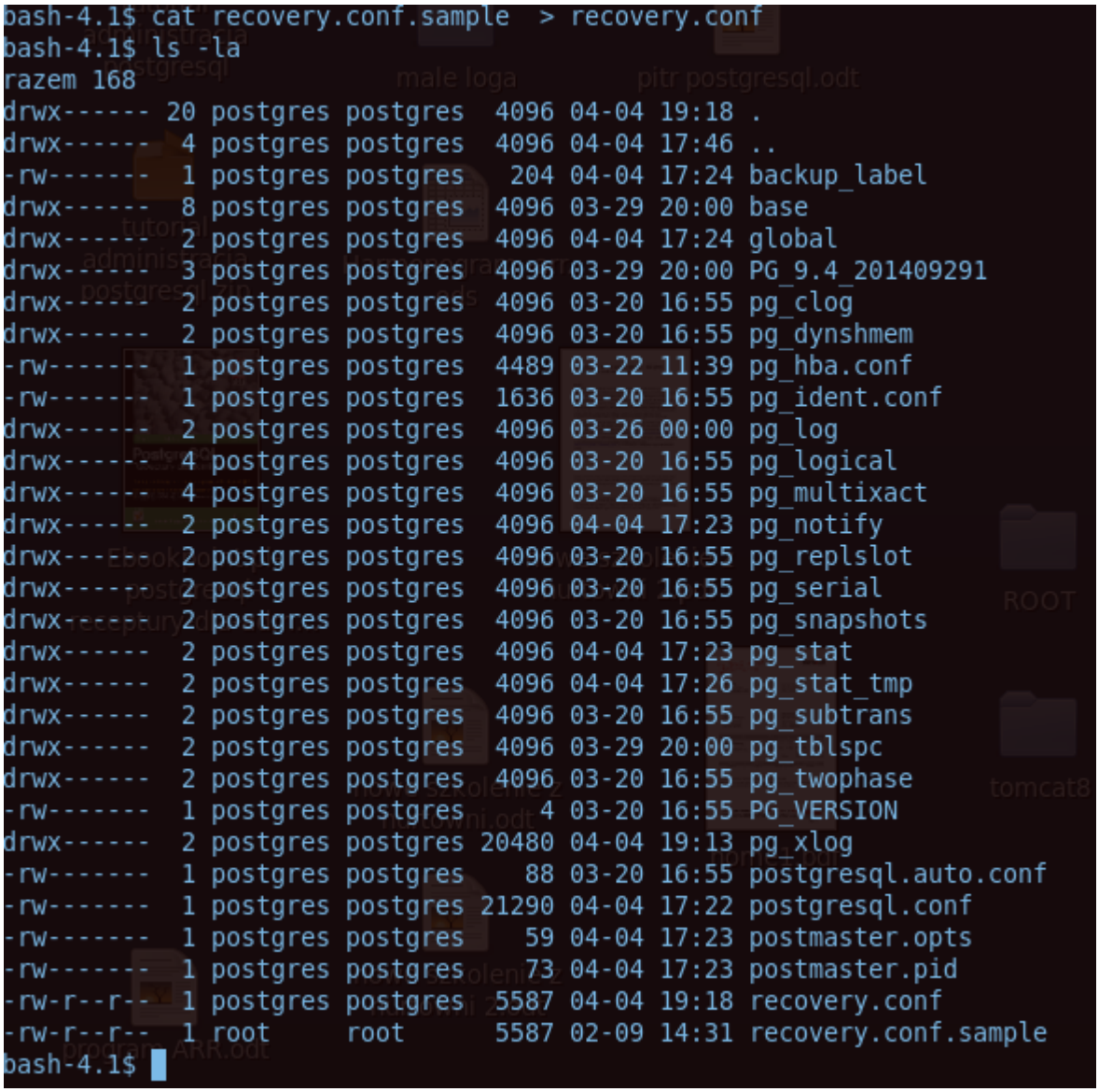
Do odzyskiwania będzie nam potrzebny plik recovery.conf. Jego obecność w katalogu $PGDATA spowoduje że klaster przy uruchamianiu przejdzie w tryb odtwarzania. Poza odtwarzaniem nie powinno go tam oczywiście być. W nim zapisane jest skąd ma zaczytać pliki WAL i jak ma je odtwarzać. Skąd go jednak wziąć? W katalogu /usr/pgsql-9.4/share mamy wiele przykładowych plików konfiguracyjnych, w tym właśnie recovery.conf. Jego nazwa będzie jednak zawierała końcówkę „.sample” którą będziemy musieli usunąć.

Kopiujemy przykładowy plik recovery.conf.sample do katalogu $PGDATA i zmieniamy jego nazwę na recovery.conf.

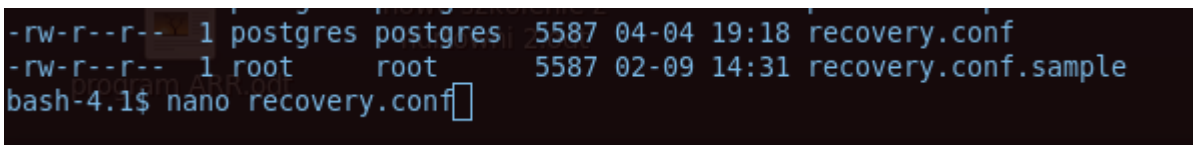


Kopiujemy zawartość pliku recovery.conf.sample do pliku recovery.conf:

cat recovery.conf.sample > recovery.conf



Edytujemy ten plik:



odnajdujemy w nim linijkę z restore_command, odkomentowujemy i piszemy tak:

'cp /home/pg_wal_archives/%f %p'

```
#
# NOTE that the basename of %p will be different from %f; do not
# expect them to be interchangeable.
#
restore_command = 'cp /home/pg_wal_archives/%f %p'              # e.g. 'c
#
```

Konstrukcja tej instrukcji jest łudząco podobna do instrukcji archive_command – owszem i działa w drugą stronę :)

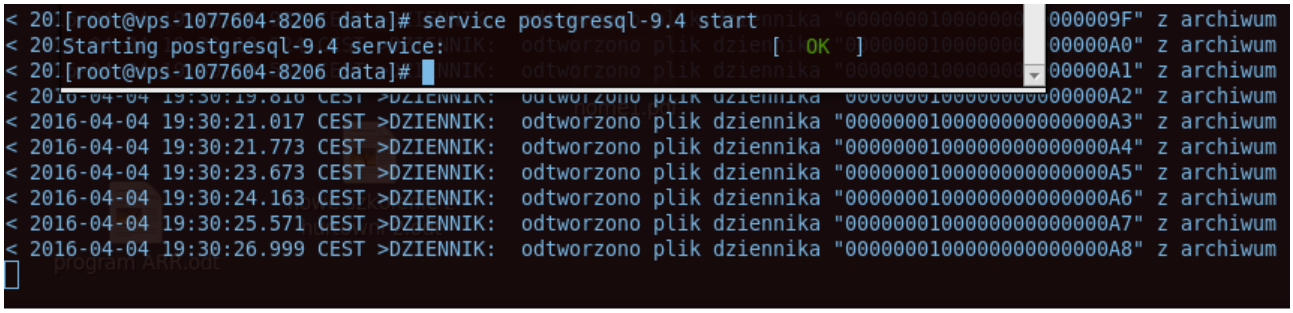
Z użyciem programu tail włączamy sobie podgląd logów :

```
[root@vps-1077604-8206 data]# tail -f /var/lib/pgsql/9.4/data/pg_log/postgresql-Mon.log
< 2016-04-04 02:33:31.780 CEST >DZIENNIK:  niekompletny pakiet uruchomieniowy
< 2016-04-04 17:23:00.106 CEST >DZIENNIK:  odebrano żądanie szybkiego zamknięcia
< 2016-04-04 17:23:00.136 CEST >DZIENNIK:  przerywanie wszelkich aktywnych transakcji
< 2016-04-04 17:23:00.142 CEST >DZIENNIK:  zamknięto program wywołujący autoodkurzanie
< 2016-04-04 17:23:00.226 CEST >DZIENNIK:  zamykanie
< 2016-04-04 17:23:00.538 CEST >DZIENNIK:  system bazy danych jest zamknięty
< 2016-04-04 17:23:01.412 CEST >DZIENNIK:  system bazy danych został zamknięty 2016-04-04 17:23:00 CEST
< 2016-04-04 17:23:01.510 CEST >DZIENNIK:  MultiXact member wraparound protections are now enabled
< 2016-04-04 17:23:01.513 CEST >DZIENNIK:  uruchomiono program wywołujący autoodkurzanie
< 2016-04-04 17:23:01.513 CEST >DZIENNIK:  system bazy danych jest gotowy do przyjmowania połączeń
```

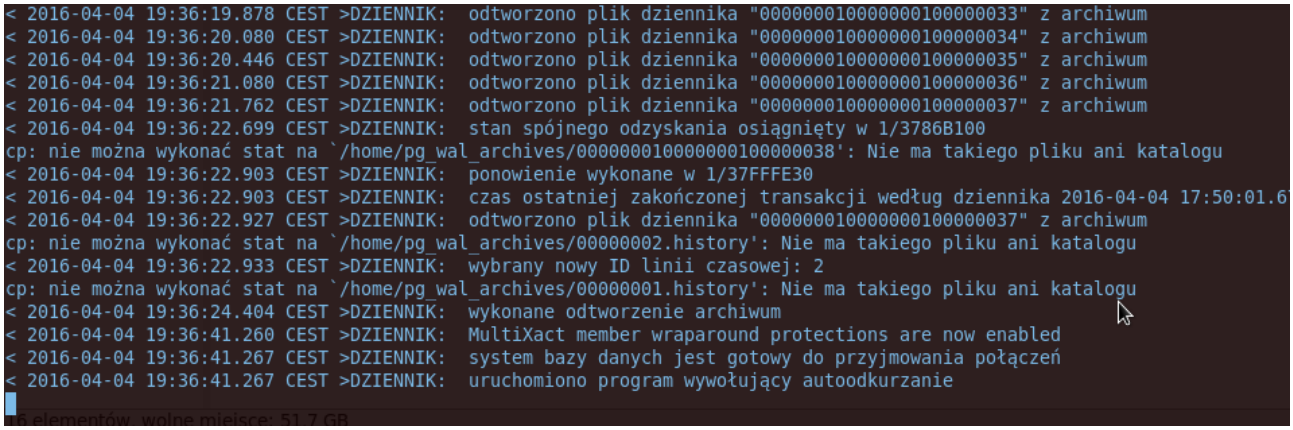
Uruchamiamy usługę, a PostgreSQL widząc istnienie pliku recovery.conf rozpoczyna odtwarzanie:

```
postgresql-Fri.log  postgresql-Mon.log  postgresql-Sat.log  postgresql-Sun.log  postgresql-Thu.log  postgresql-Tue.log
[root@vps-1077604-8206 data]# tail -f /var/lib/pgsql/9.4/data/pg_log/postgresql-Mon.log
< 2016-04-04 02:33:31.780 CEST >DZIENNIK:  niekompletny pakiet uruchomieniowy
< 2016-04-04 17:23:00.106 CEST >DZIENNIK:  odebrano żądanie szybkiego zamknięcia
< 2016-04-04 17:23:00.142 CEST >DZIENNIK:  zamknięto program wywołujący autoodkurzanie
< 2016-04-04 17:23:00.226 CEST >DZIENNIK:  zamykanie
< 2016-04-04 17:23:00.538 CEST >DZIENNIK:  system bazy danych jest zamknięty
< 2016-04-04 17:23:01.412 CEST >DZIENNIK:  system bazy danych został zamknięty 2016-04-04 17:23:00 CEST
< 2016-04-04 17:23:01.510 CEST >DZIENNIK:  MultiXact member wraparound protections are now enabled
< 2016-04-04 17:23:01.513 CEST >DZIENNIK:  uruchomiono program wywołujący autoodkurzanie
< 2016-04-04 17:23:01.513 CEST >DZIENNIK:  system bazy danych jest gotowy do przyjmowania połączeń
< 2016-04-04 19:29:43.784 CEST >DZIENNIK:  działanie systemu bazy danych zostało przerwane; ostatnie znane podniesienie
< 2016-04-04 19:29:43.784 CEST >DZIENNIK:  tworzenie brakującego folderu WAL "pg_xlog/archive_status"
< 2016-04-04 19:29:43.880 CEST >DZIENNIK:  rozpoczęto odzyskiwanie archiwum
< 2016-04-04 19:29:43.899 CEST >DZIENNIK:  odtworzono plik dziennika "00000001000000000000002C" z archiwum
< 2016-04-04 19:29:43.917 CEST >DZIENNIK:  ponowienie uruchamia się w 0/2C000090
< 2016-04-04 19:29:43.918 CEST >DZIENNIK:  stan spójnego odzyskania osiągnięty w 0/2C0000B8
< 2016-04-04 19:29:43.936 CEST >DZIENNIK:  odtworzono plik dziennika "00000001000000000000002D" z archiwum
< 2016-04-04 19:29:44.190 CEST >DZIENNIK:  odtworzono plik dziennika "00000001000000000000002E" z archiwum
< 2016-04-04 19:29:44.450 CEST >DZIENNIK:  odtworzono plik dziennika "00000001000000000000002F" z archiwum
< 2016-04-04 19:29:44.712 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000030" z archiwum
< 2016-04-04 19:29:44.996 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000031" z archiwum
< 2016-04-04 19:29:45.265 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000032" z archiwum
< 2016-04-04 19:29:45.523 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000033" z archiwum
< 2016-04-04 19:29:45.779 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000034" z archiwum
< 2016-04-04 19:29:46.050 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000035" z archiwum
< 2016-04-04 19:29:46.309 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000036" z archiwum
< 2016-04-04 19:29:46.583 CEST >DZIENNIK:  odtworzono plik dziennika "000000010000000000000037" z archiwum
```

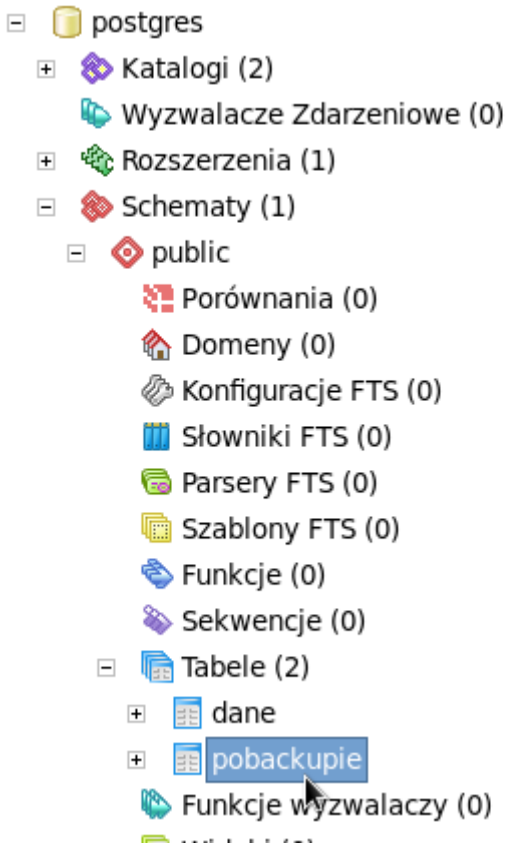
Zwróć szczególną uwagę na to, że mimo że teoretycznie usługa została uruchomiona jako taka, to odtwarzanie cały czas trwa i nadal nie będzie można się podłączyć do bazy.



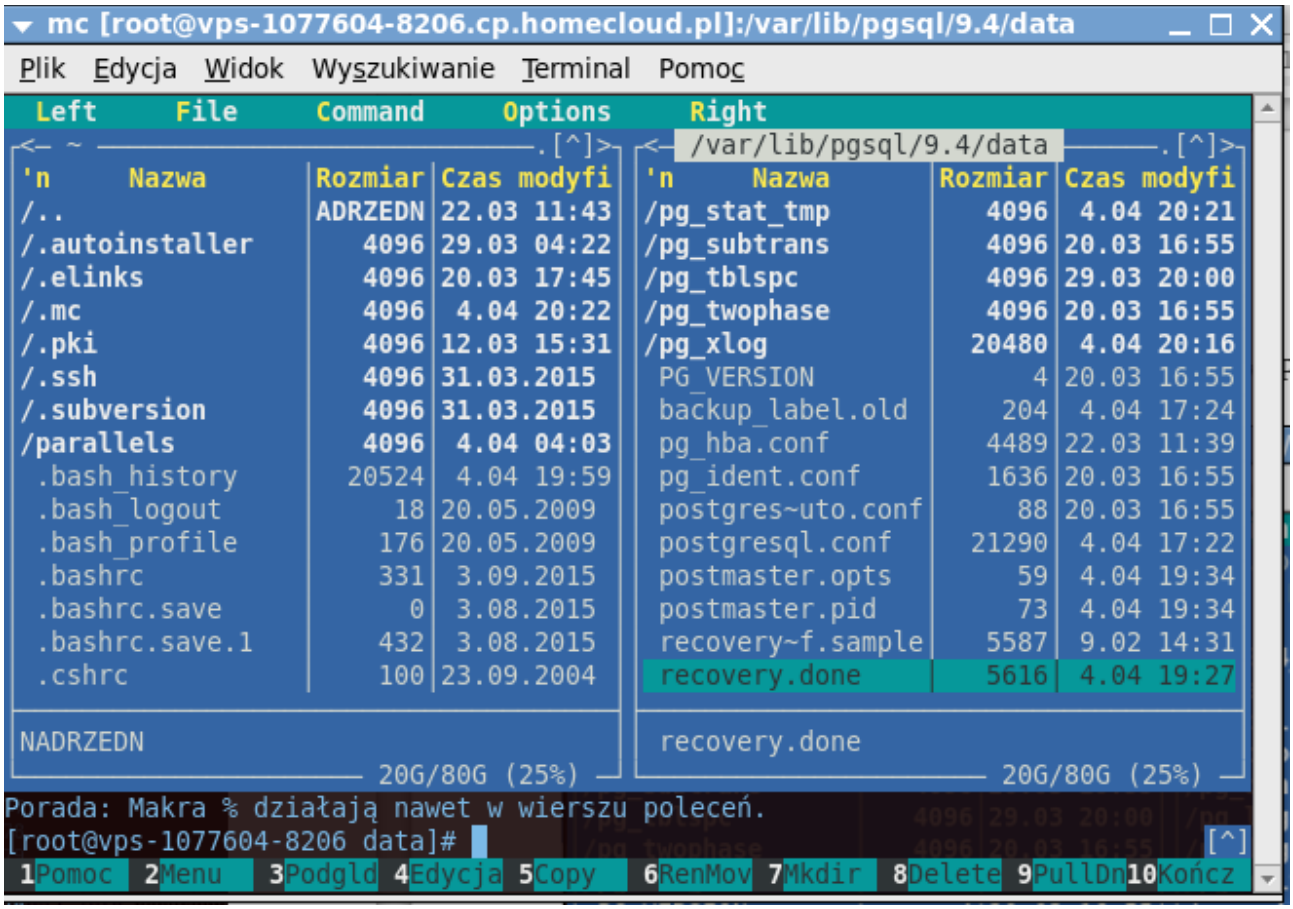
Gdy zakończy odtwarzanie:



Odtwarzanie zostało zakończone, podłączam się więc do bazy i sprawdzam czy istnieje moja tabelka:



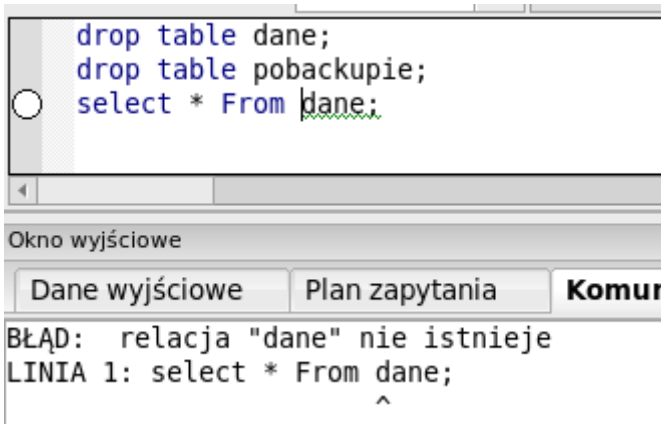
Przy okazji zauważ jedną bardzo ważną rzecz:



Po zakończeniu odtwarzania nazwa pliku recovery.conf została zmieniona na recovery.done
Dzieje się tak po to, by po restarcie serwer nowu nie wszedł w tryb odtwarzania.

Przywracanie do punktu w czasie

Procedura wygląda niemal identycznie, z tą różnicą że do pliku recovery.conf dodajemy jeszcze jeden parametr określający punkt w czasie do którego przywracamy. Aby się upewnić odnośnie działania tej procedury skasuję dwie tabele, a następnie przywrócę bazę do stanu przed ich skasowaniem.



Zmiana i odkomentowanie parametru określającego punkt w czasie:

```
#
#recovery_target_name = ''      # e.g. 'daily backup 2011-01-26'
#
recovery_target_time = '2016-04-04 20:05:00 EST'       # e.g. '2004-07-14 22:39:00 EST'
#
#recovery_target_xid = ''
#
#recovery_target_inclusive = true
```

Wskazałem czas tuż przed skasowaniem tabel. Otwieram bazę jak zwykle, a po zakończonym procesie odtwarzania sprawdzam czy na pewno mam swoje skasowane tabele:

