Laboratorium baz danych, PostgreSQL

Andrzej Leśnicki

Gdańsk 2010

Spis treści

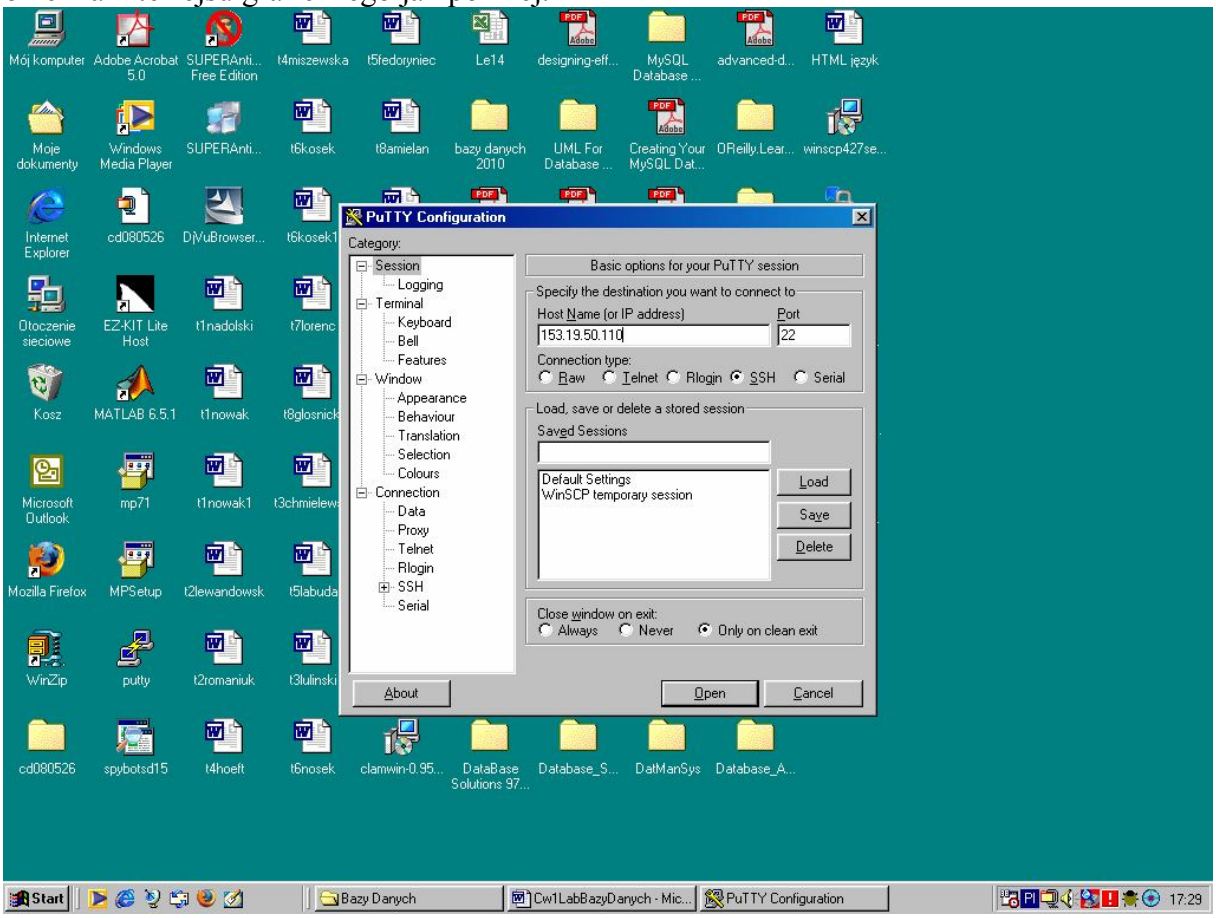
Laboratorium Baz Danych, PostgreSQL

Ćwiczenie 1. Zakładanie bazy danych, podstawy PostgreSQL

Istnieje wiele programów DBMS (tj. systemu zarządzania relacyjnymi bazami danych) używających języka SQL. Wśród nich są takie darmowe programy jak PostgreSQL czy MySQL. W konkretnym programie DBMS język SQL zawsze nieco różni się od wzorcowego języka SQL-99 (ISO/IEC 9075: 1999 „Information Technology – Database Languages – SQL"). Na serwerze laboratoryjnym zainstalowano pod systemem operacyjnym Linux darmowy program PostgreSQL, rozwijany przez PostgreSQL Global Development Group www.postgresql.org/download, gdzie jest też dostępna kompletna dokumentacja z wielu przykładami. Celem pierwszego ćwiczenia laboratoryjnego jest nabycie umiejętności: logowania się do systemu, korzystania z pomocy, zakładania bazy danych, używania podstawowych poleceń SQL.

1. Logowanie

Komputer w Laboratorium jest komputerem klienta. Student ma dostęp do tego komputera jako użytkownik: **`baza,`** hasło: **`danych`**. Każdy student ma już założone konto na serwerze, zna swój login i hasło. Należy połączyć się z serwerem poprzez sieć używającą protokołu TCP/IP. Do nawiązania połączenia posłuż się znajdującym się na pulpicie programem `putty.exe` (jest to niewielki program 444 kB, ściągany za darmo z Internetu, open source). Pozwala on nawiązać szyfrowane połączenie Windowsa z właściwym Linuxowi serwerem SSH (ang. Secure Shell). Uruchomienie programu `putty.exe` powoduje pojawienie się okienka interfejsu graficznego jak poniżej.



Sprawdź pod Window/Translation, czy wybrano właściwą translację polskich znaków: ISO 8859-2: 199 (latin-2 East Europe, jest to niezbędne do swobodnego pisania łańcuchów w języku polskim; w informatyce unikamy pisania polskich znaków, poza łańcuchami). Następnie powróć do Session, wpisz IP serwera 153.19.50.110 i daj Open. Podaj serwerowi swój login (`username`), np. `student123` i swoje hasło (`password`), np. `bor22`. Wszędzie, gdzie poniżej będzie występowała przykładowa nazwa `student123` **podawaj własną, sobie znaną nazwę, swój login**. Serwer podaje ogólne informacje i na koniec potwierdza gotowość do współpracy znakiem zachęty ~$ jak poniżej (tylda ~ oznacza, że znajdujesz się w katalogu domowym):

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/

System information as of Thu Mar 11 17:19:20 CET 2010

System load: 0.0 Memory usage: 10% Processes: 118

Usage of /home: 0.1% of 131.91GB Swap usage: 0% Users logged in: 0

Graph this data and manage this system at https://landscape.canonical.com/

Last login: Thu Mar 11 14:37:20 2010 from knot894.eti.pg.gda.pl
student123@bazyksm:~$

2. Pomoc (help)

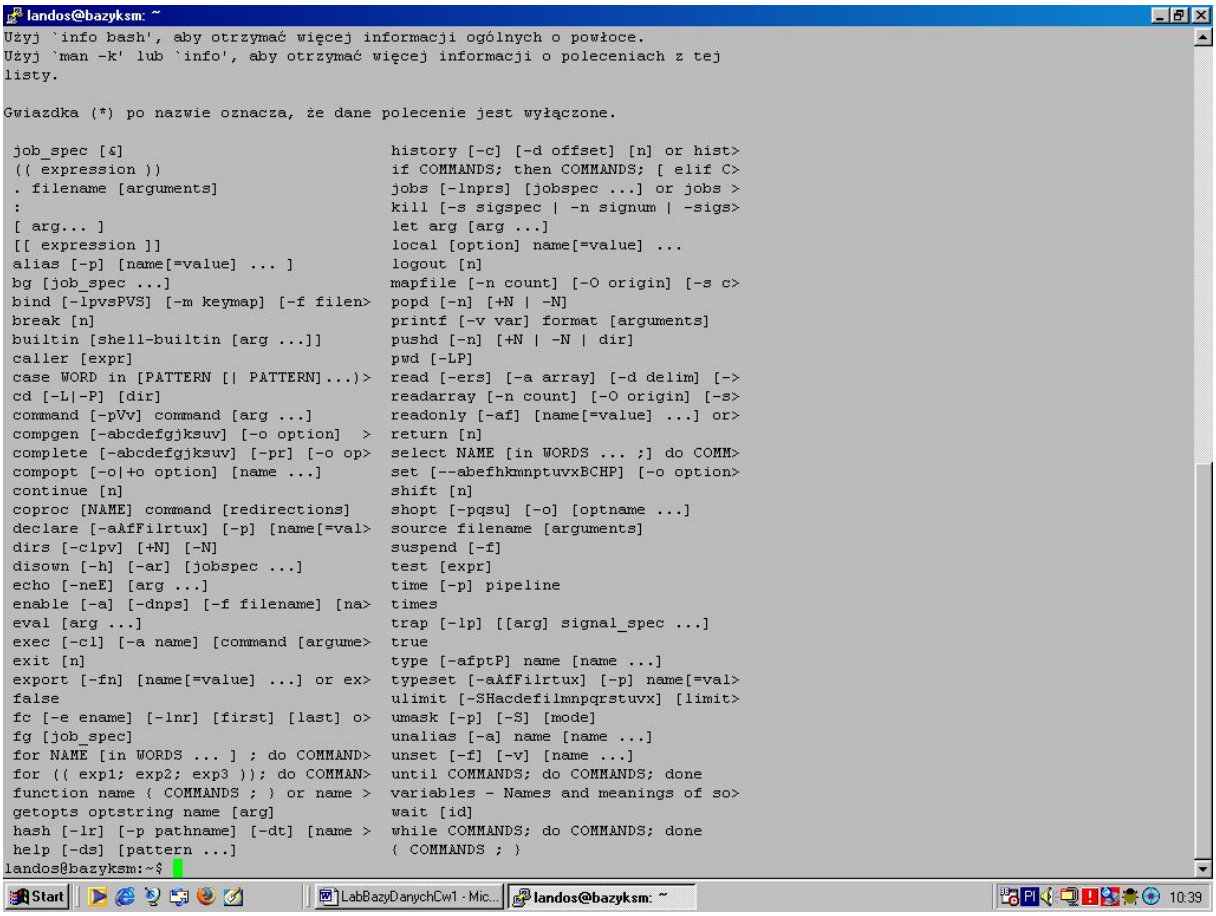
W linii poleceń po znaku zachęty wpisujemy interesujące nas polecenia. Tak jak w każdym innym edytorze linii, możemy cofać się do uprzednio napisanych poleceń strzałkami w górę ↑ i w dół ↓. Jest to pomocne zwłaszcza przy mylnym wpisaniu polecenia, gdy zostanie zasygnalizowany błąd. Nie trzeba wtedy pisać od nowa całego, czasami bardzo długiego polecenia, ale wystarczy powrócić strzałką do mylnie napisanego polecenia i poprawić je.

Jednym z poleceń jest polecenie pomocy. Trzeba zapoznać się z tym poleceniem, gdyż rozpoczynając dopiero pracę w języku PostgreSQL będziemy mogli dzięki wyjaśnieniom zawartym w pomocy rozwiać wiele wątpliwości.

Napisz `help`, czyli

```
~$ help
```

Uzyskasz wskazówki posługiwania się pomocą (`man -k`, `info`) i listę poleceń jak poniżej.



Czasami wyświetlany tekst pomocy lub lista nie mieszczą się na ekranie i w lewym dolnym rogu okna pojawia się znak „:” . **Taki tekst można przewijać strzałkami. Wyjście z trybu przewijania wymaga naciśnięcia q.**

Napisz

```
~$ help pwd
~$ info pwd
~$ man -k pwd
```

Uzyskasz wyjaśnienie, że polecenie `pwd` służy do tego, aby komputer podał bieżący katalog. W podobny sposób uzyskaj pomoc dotyczącą innych interesujących cię poleceń.

Ważnym poleceniem jest polecenie `exit` służące do zakończenia połączenia z serwerem. Przećwicz kończenie i nawiązywanie na nowo połączenia z serwerem, daj `exit` i nawiąż ponownie połączenie z serwerem używając `putty.exe`.

3. Midnight Commander

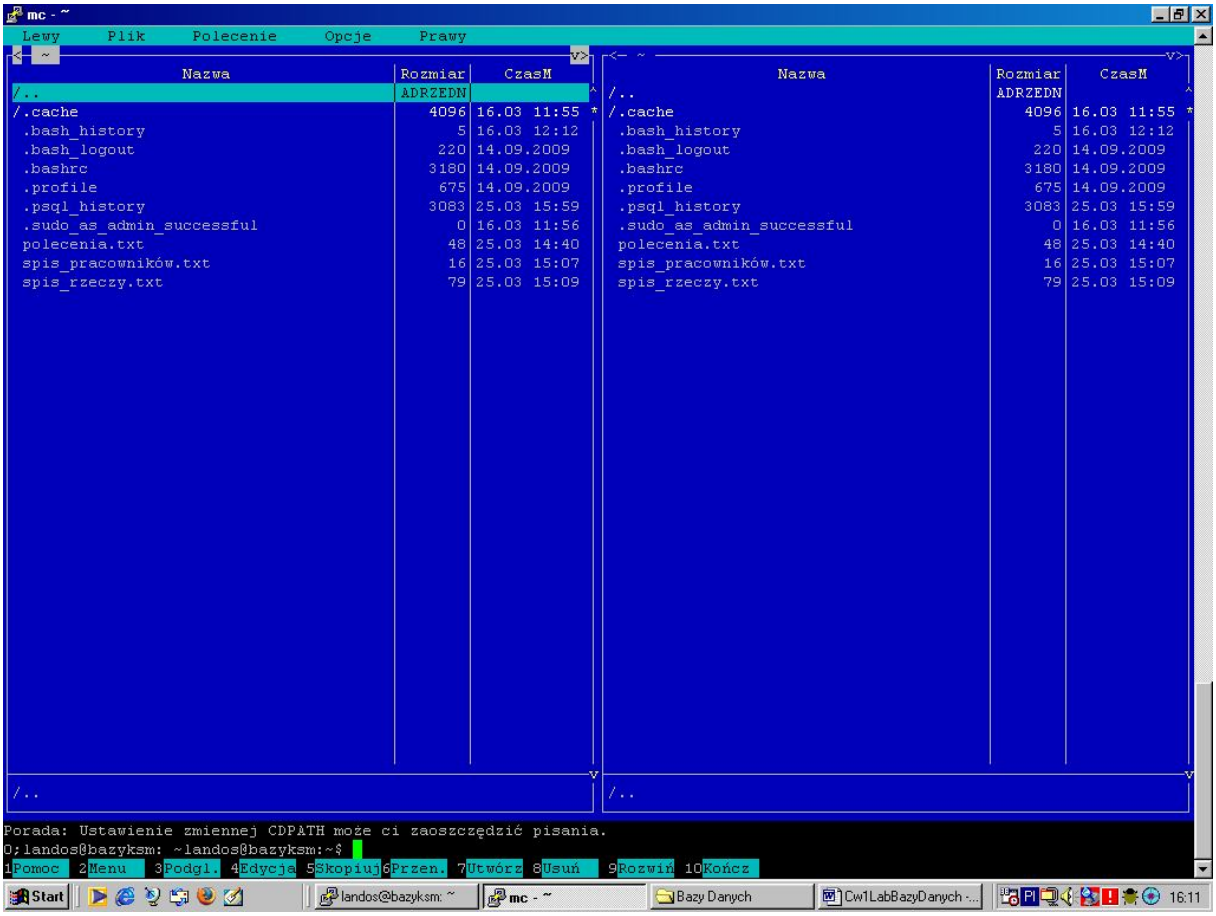
Do przeglądania katalogów dysku serwera jest przydatny program Midnight Commader (odpowiednik Norton Commander). Uruchom program Midnight Commander pisząc `mc` w linii poleceń po znaku zachęty

```
~$ mc
```

Okno uruchomionego programu jest takie jak poniżej. Przekonaj się, że znajdujesz się w katalogu domowym. Jest możliwe kopiowanie, edycja plików. Korzystanie z wewnętrznego edytora będzie możliwe po ustawieniu opcji:

Opcje → Konfiguracje → [x] Wewnętrzny edytor

Z programu Midnight Commander wychodzimy kluczem F10.



4. Zakładanie bazy danych

Serwer z DBMS może zawierać wiele baz danych. Zazwyczaj jeden użytkownik ma jedną bazę danych do jednego projektu i nadaje się domyślnie nazwę bazy danych taką samą, jak nazwa użytkownika (np. `nazwa_bazy_danych =username=student123`). Bazę zakładamy posługując się poleceniem `createdb`. Jest to jedno z poleceń aplikacji klienta (listę tych poleceń zamieszczono w dodatku 1.1). Przeczytaj informację o tym poleceniu (`~$ info createdb`) Załóż własną bazę danych, np. dla użytkownika `student123` będzie to

```
~$ createdb student123
```

lub krótko (zostanie przyjęta domyśla nazwa bazy danych, taka jak nazwa użytkownika)

```
~$ createdb
```

Istniejącą bazę danych usuwa się poleceniem `dropdb`. Założona przed chwilą baza danych jest jeszcze pusta i bez żalu usuniemy ją dla przećwiczenia umiejętności usuwania

```
~$ dropdb student123
```

W praktyce trzeba być bardzo ostrożnym w użyciu `dropdb`, gdyż nawet bardzo duża baza zostanie usunięta błyskawicznie i bezpowrotnie, a komputer nie zapyta (jak to jest w wielu innych programach i do czego jesteśmy przyzwyczajeni), czy jesteśmy pewni, że chcemy usunąć bazę. Nie ma polecenia cofającego `dropdb`.

Załóż od nowa usuniętą przed chwilą bazę danych

```
~$ createdb
```

Wylistuj wszystkie założone bazy danych

```
~$ psql -l
```

Nasza założona przed chwilą baza danych `student123` powinna znaleźć się na tej liście.

5. Dostęp do bazy danych

Dogodny dostęp do założonej bazy danych zapewnia interfejs tekstowy języka PostgreSQL. Jest on uruchamiany poleceniem `psql`. Interfejs ten pozwala wprowadzać, edytować i wykonywać polecenia SQL (przeczytaj informację o tym poleceniu, `~$ info psql`). Uruchom ten interfejs pisząc

```
~$ psql
```

Uzyskaliśmy w ten sposób dostęp do bazy danych o domyślnej nazwie `nazwa_bazy_danych =username=student123`. Gdybyśmy chcieli uzyskać dostęp do bazy danych o innej nazwie, to napisalibyśmy

```
~$ psql -d nazwa_innej_bazy_danych
```

Gotowość interfejsu `psql` do przyjęcia polecenia SQL jest sygnalizowana znakiem zachęty o postaci równości zakończonej strzałką

```
nazwa_bazy_danych =>
```

Tak jest u zwykłego użytkownika. W przypadku superużytkownika (ang. superuser), zakończenie znaku zachęty będzie takie

```
nazwa_bazy_danych =#
```

Teraz po znaku zachęty możemy napisać polecenie w języku SQL. Zwykłe polecenie kończy się znakiem średnika ; a nie ENTER. Jest to w praktyce bardzo korzystne, gdyż polecenia

bywają długie, są pisane w wielu liniach. Wtedy w kolejnych liniach znak zachęty ma postać nie równości, ale kreski ->. **Pisząc polecenie SQL pilnie obserwuj znak zachęty, gdyż będzie on się zmieniał, będzie zawierał podpowiedź, czy nie popełniliśmy powyżej jakiegoś błędu (np. brak nawiasu).**

Oprócz zwykłych poleceń SQL istnieją bardzo krótkie polecenia wewnętrzne (nazywane też meta-poleceniami lub poleceniami sterującymi). Rozpoczynają się one od lewego ukośnika (backslash) i nie kończą się średnikiem. Najczęściej używane polecenia wewnętrzne, to:

`\l`	lista wszystkich baz danych na serwerze
`\dt`	lista tabel w bieżącej bazie danych
`\d [NAME]`	opis wymienionej tabeli
`\h [NAME]`	pomoc na temat wymienionego polecenia SQL
`\copy ...`	dokonaj SQL `COPY` (`FROM` lub `TO`)
`\i [FILE]`	wykonanie poleceń zapisanych w pliku tekstowym
`\q`	quit, czyli wyjście z programu `psql`

Wypróbuj dwa początkowe polecenia wewnętrzne, tj. daj `\l` , a następnie `\dt`.

Listę wszystkich poleceń wewnętrznych języka PostgreSQL, uzyskaną poleceniem wewnętrznym `\?` , zamieszczono na końcu instrukcji w dodatku 1.2.

Listę poleceń SQL, a więc listę poleceń, dla których jest dostępna pomoc `help (\h [NAME])`, zamieszczono w dodatku 1.3 (listę tę uzyskano poleceniem wewnętrznym `\h`). Jeszcze raz przypominamy, że gdy wyświetlany tekst pomocy lub lista nie mieszczą się na ekranie, to w lewym dolnym rogu okna pojawia się znak „:” i tekst można przewijać strzałkami. **Wyjście z trybu przewijania wymaga naciśnięcia q.**

Przećwicz zamykanie i otwieranie interfejsu tekstowego, tj. daj `\q` , a następnie `psql`.

Pojawienie się znaku zachęty => oznacza, że interfejs tekstowy `psql` jest w stanie nasłuchu, oczekuje na polecenie w języku SQL. Przykładowo wydaj następujące polecenia `SELECT`

```
SELECT version();

SELECT pi();

SELECT 2*sqrt(3), pi()^2;

SELECT current_date;

SELECT current_time;
```

Interfejs tekstowy `psql` (i generalnie PostgreSQL) nie jest czuły na małe-duże litery (poza wnętrzami łańcuchów), ale dla lepszej czytelności programu wyrazy kluczowe są pisane dużymi literami.

Użyte powyżej polecenie (zapytanie) SELECT ma ogromne znaczenie praktyczne dla użytkowników bazy danych. Pozwala odzyskiwać dane z bazy danych, jest chyba najczęściej

używanym poleceniem SQL. Tutaj to polecenie zostało użyte w swojej najprostszej postaci, ale pełne możliwości tego polecenia są ogromne (daj `\h SELECT`, aby zobaczyć jaka jest pełna składnia tego polecenia).

6. Zakładanie tabeli

Tabele zakładamy dla danej bazy danych. Baza danych zawiera przynajmniej jedną tabelę. Tabela (relacja) jest realizacją encji, wyróżnia się w niej wiersze i kolumny. Wiersze są rekordami (krotkami, z ang. tuples), a kolumny atrybutami (wydzielają pola w rekordzie). Tabelę zakładamy następującym poleceniem (podglądnij pełną składnię tego polecenia dając `\h CREATE TABLE`)

```
CREATE  TABLE nazwa_tabeli (nazwa_kolumny typ_danej
[DEFAULT wartość] ograniczenia,   ...,   ...);
```

Atrybut, to typ danych z danej kolumny, np.
- integer – liczba całkowita
- real – liczba rzeczywista
- varchar(80) – znak (maksymalna liczba znaków)
- date – data, np. '2010-03-15'

Parametrem kolumny jest
`DEFAULT` - wartość dopisywana domyślnie, gdy nie podano innej

Ograniczenia dla danych w kolumnie:
`NOT NULL` - w kolumnie nie może wystąpić wartość pusta `NULL`
`UNIQUE` - w kolumnie nie może powtórzyć się żadna wartość
`PRIMARY KEY` - klucz pierwotny PK, połączenie `NOT NULL` i `UNIQUE`, unikalna kolumna (lub zestaw kolumn) w tabeli
`CHECK` (warunek) - warunek sprawdzany przy wprowadzaniu lub modyfikacji danych
`REFERENCES` - definicja referencji, odniesienia do innej tabeli (klucz obcy FK)

Załóż dwie poniższe tabele (jeszcze raz przypominamy, że: język SQL nie rozróżnia dużych-małych liter poza wnętrzem łańcucha; zwyczajowo, dla zwiększenia czytelności piszemy wyrazy kluczowe dużymi literami, nazwy i resztę małymi literami; nowa linia nie ma znaczenia, można pisać wszystko w jednej linii, polecenie kończy się średnikiem, a nie nową linią; tak jak w każdym innym edytorze linii, możemy cofać się do uprzednio napisanych poleceń strzałkami w górę ↑ i w dół ↓; obserwuj postać znaku zachęty):

```
CREATE TABLE pracownicy(
     nazwisko          varchar(80),
     numer_pokoju      int,
     ciezar_pracownika real   -- ciężar w kg
   );

CREATE TABLE rzeczy(
     nr_inwentarzowy       int,
     nazwa                 varchar(80),
     ciezar_rzeczy         real,    /* ciężar w kg */
```

```
    data_produkcji        date,
    na_stanie_pracownika  varchar(80)   );
```

Powyższe dwie tabele `pracownicy` i `rzeczy` pozostają ze sobą w związku. Pracownik ma zapisane na swoim stanie różne rzeczy. Rzecz jest zapisana na stanie jednego pracownika. Pracownicy i ich rzeczy są rozmieszczeni w tych samych pokojach.

W tabeli `pracownicy` napisano komentarz „`-- ciężar w kg`". Wszystko po znakach „`--`„ aż do końca linii jest traktowane jako komentarz. Dlatego znaki „`);`" należało koniecznie napisać w nowej linii. Nie będzie tego problemu, gdy damy komentarz ze znakami początku i końca, tak jak w języku C (taki /*komentarz*/ napisano w tabeli `rzeczy`).

Możemy napisać komentarz dla bazy danych, tabeli (oglądnij pełną składnię polecenia `\h COMMENT`), np.

```
COMMENT ON DATABASE student123 IS  ' Ćwiczeniowa baza danych
student123 ';

COMMENT ON TABLE pracownicy IS  ' Tabela zawiera dane
pracowników ';
```

Sprawdź poleceniem wewnętrznym `\dt` , czy założone dwie tabele znajdują się na liście tabel aktualnej bazy danych. Sprawdź poleceniem `\d nazwa_tabeli`, czy tabele mają strukturę taką, jak planowano. Sprawdź poleceniami `\dd pracownicy` i `\dt+ pracownicy`, czy zostanie przytoczony nasz komentarz do tabeli. Polecenie `\l+` wylistuje bazy danych ze szczegółami, w tym z naszymi komentarzami.

Załóż jeszcze dwie własne tabele, które będą związane ze sobą. Tych tabel będziesz wielokrotnie używał w przyszłych ćwiczeniach, dlatego dobierz je z sensem, aby z przyjemnością do nich wracać (niech dają możliwość wykonania ciekawych dla studenta obliczeń).

Założoną tabelę usuwamy bezpowrotnie (wraz z jej ewentualną zawartością, a jak wpisywać dane do tabeli, dowiemy się poniżej w punkcie 7) poleceniem

```
DROP TABLE nazwa_tabeli;
```

Usuń najkrótszą z czterech założonych przed chwilą tabel (zrobimy to bez żalu, gdyż tabela jest jeszcze pusta). Sprawdź, że znikła ona z listy tabel `\dt` . Następnie załóż tę tabelę na nowo (łatwy powrót strzałką ↑ do już raz uprzednio napisanego polecenia).

7. Wpisywanie danych do tabeli

Dane zawsze wpisujemy do tabeli wierszami jako rekordy. Są możliwe różne sposoby wpisywania wierszy z użyciem polecenia `INSERT` (oglądnij pełną składnię tego polecenia `\h INSERT`). Pierwszy sposób zakłada, że użytkownik pamięta kolejność kolumn i ich nazwy

```
INSERT INTO nazwa_tabeli VALUES (val1, val2, . . .);
```

Według drugiego sposobu sami narzucamy kolejność kolumn (nie muszą wystąpić wszystkie możliwe kolumny)

```
INSERT INTO nazwa_tabeli (nazwa_kol1, nazwa_kol8,nazwa_kol3,
...) VALUES (val1, val8, val3, ...);
```

Według trzeciego sposobu wpisujemy wartości całej serii wierszy (tutaj seria trzech wierszy, wszystkie wiersze muszą mieć taką samą liczbę kolumn, w tej samej kolejności)

```
INSERT INTO nazwa_tabeli VALUES (val11, val12, . . .),
                                (val21, val22, . . .),
                                (val31, val32, . . .);
```

Kolejność wpisywania rekordów nie ma znaczenia. Tylko w bardzo dużych tabelach to, czy rekord znajduje się na początku, czy końcu tabeli, będzie miało wpływ na czas odnalezienia rekordu przy odczycie. Podobnie kolejność kolumn nie ma znaczenia (chociaż, gdy już raz została ustalona kolejność, to trzeba się tego porządku trzymać). Mimo podobieństwa do znanej nam z matematyki macierzy, tabela nie ma nic wspólnego z macierzą.

Wpisz do tabeli `pracownicy` dane:

Kowalski, 101, 55.8
Jarosz, 103, 75
Dragan, 108, 63.5
Wolski, 55, 88.7
Lenart, 101, 75.3
Dolasa, 108, 91.4
Rolski, 55, 83.1
Krol, 103, 98
Jarecki, 55, 83.3
Szuba, 24, 72.6
Ekiert, 24, 66.2

Wpisujemy przykładowo pierwszy rekord

```
INSERT INTO pracownicy VALUES ('Kowalski',101,55.8);
```

drugi rekord

```
INSERT INTO pracownicy (numer_pokoju, nazwisko,
ciezar_pracownika) VALUES(103,'Jarosz',75);
```

dalsze trzy rekordy

```
INSERT INTO pracownicy VALUES ('Dragan',108,63.5),
                              ('Wolski',55,88.7),
                              ('Lenart',101,75.3);
```

Uzupełnij tabelę `pracownicy` dalszymi, w tym własnymi rekordami.

Zawartość tabeli możemy podglądnąć dając polecenie

```
SELECT * FROM pracownicy;
```

Znak gwiazdki, mnożenia * oznacza tutaj "wszystkie kolumny".

Jeżeli wyniki zapytań są dla nas na tyle istotne, że chcemy je zapisać w sposób trwały, to zapiszemy je w pliku jak poniżej

```
\o wynik1.txt
SELECT * FROM pracownicy;
```

Sprawdź wspomagając się programem `mc`, że został utworzony plik `wynik1.txt` zawierający wyniki zapytania.

Jeżeli zauważymy, że jakiś wpis jest niepoprawny, to możemy usunąć cały wiersz poleceniem `DELETE` (oglądnij pełną składnię tego polecenia, `\h DELETE`). Na przykład, gdy chcemy usunąć wiersz z nazwiskiem Ekiert, to napiszemy

```
DELETE FROM pracownicy WHERE nazwisko='Ekiert';
```

Gdybyśmy chcieli usunąć wszystkie wiersze z pokojami 24, to napiszemy

```
DELETE FROM pracownicy WHERE numer_pokoju=24;
```

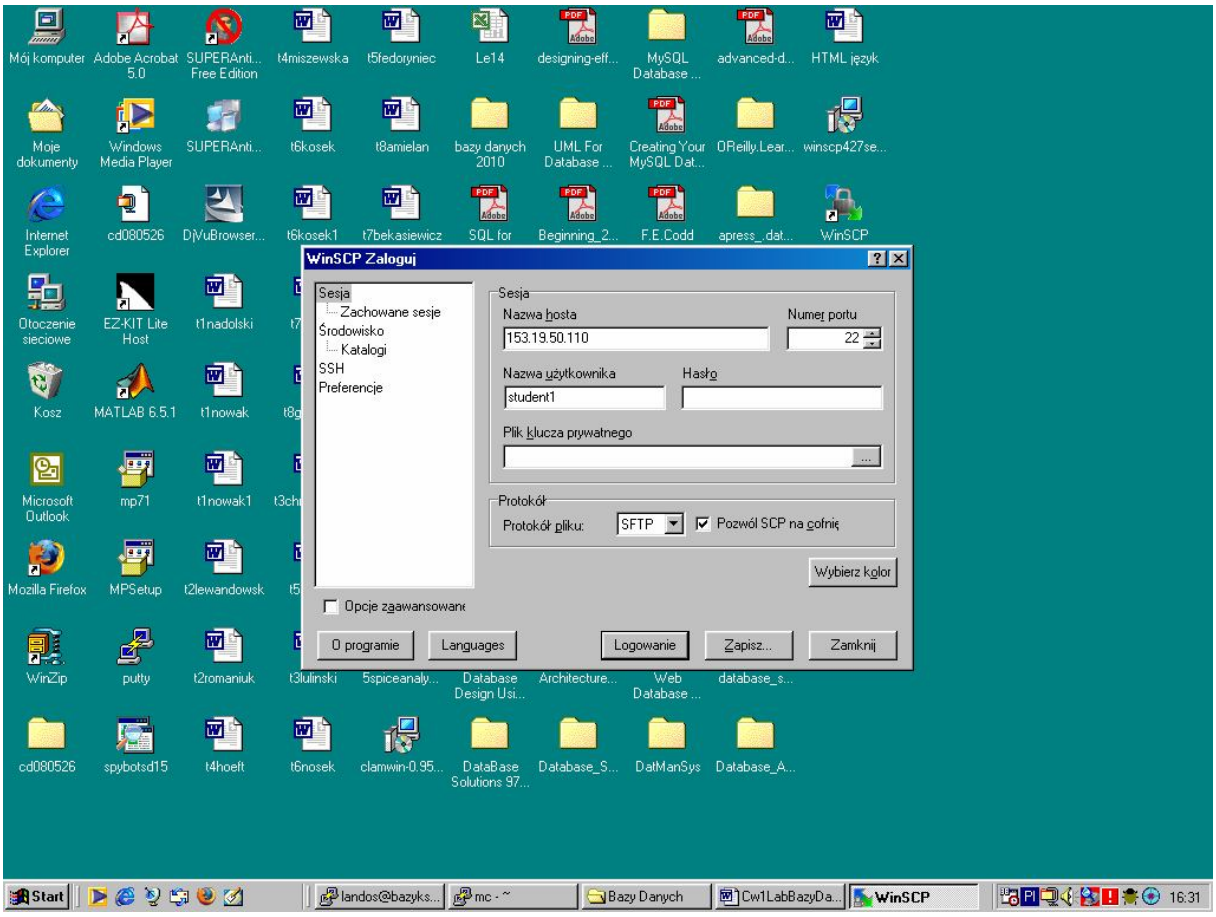
W podobny sposób jak dla tabeli `pracownicy`, wpisz dane do jednej z założonych przez siebie, a jeszcze pustej tabeli.

8. Kopiowanie danych do tabeli

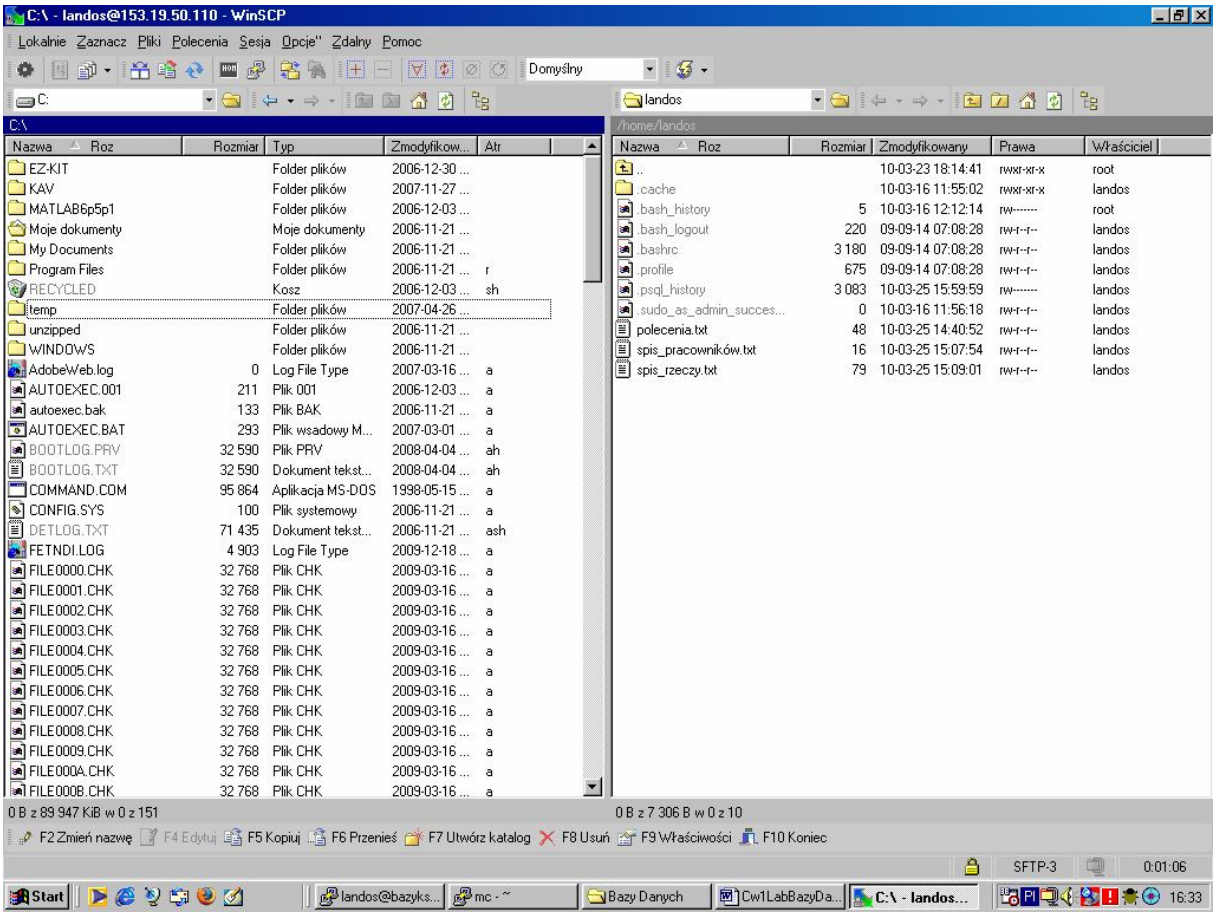
Często zdarzy się, że dane mamy już zgromadzone w pliku i chcemy je przekopiować do tabeli bazy danych (i jest to czwarty sposób wpisywania wierszy do tabeli). Często plik z danymi będzie przygotowany pod innym systemem operacyjnym i trzeba go najpierw przekopiować pod bieżący system operacyjny. Użyjemy do tego celu programu WinSCP (ok. 3MB, open source). Zilustrujemy to na przykładzie wpisania danych do tabeli `rzeczy` z pliku tekstowego `spis_rzeczy.txt` . Zawartość tego pliku została podana w dodatku 1.4.

Mamy już w naszej bazie danych założoną tabelę `rzeczy`. Wychodzimy z programu `psql` dając `\q` i następnie przerywamy połączenie z serwerem dając `exit`. Znajdujemy plik `spis_rzeczy.txt` na pulpicie komputera klienta (OS Windows). Jest to plik napisany z użyciem windowsowego Notatnika, a zapisując ten plik (Zapisz plik) użyto opcji : Wszystkie pliki. Wiemy już, gdzie znajduje się interesujący nas plik.

Uruchamiamy teraz program WinSCP. Pojawia się okno interfejsu graficznego (jak na rysunku poniżej).



Wpisujemy IP serwera, login, hasło. Uzyskujemy połączenie z serwerem. Pojawia się okno podobne do okna Norton Commander, gdzie po lewej stronie widzimy katalog pod OS Windows i po prawej stronie widzimy katalog pod OS Linux.



Kopiujemy plik `spis_rzeczy.txt` przeciągając go myszką z katalogu lewego do katalogu prawego (przekopiuj przy okazji pliki `spis_pracownikow.txt` i `polecenia.txt`). Równie dobrze można przeciągnąć plik z pulpitu do prawego katalogu. Zamykamy program WinSCP. Łączymy się z serwerem poprzez program `putty.exe`. Program Midnight Commander pokazuje przekopiowany plik `spis_rzeczy.txt`. Przechodzimy do programu `psql`. Oglądnij składnię polecenia kopiowania `\h COPY`. Kopiujemy dane z pliku `spis_rzeczy.txt` do tabeli `rzeczy` poleceniem wewnętrznym (nie ma średnika na końcu polecenia)

```
\COPY rzeczy FROM '/home/student123/spis_rzeczy.txt' WITH DELIMITER ','
```

Polecenie

```
SELECT * FROM rzeczy;
```

pozwala zobaczyć, że pusta do tej pory tabela wypełniła się danymi. Gdyby tabela nie była pusta przed kopiowaniem, to do starych rekordów dodałyby się nowe rekordy.

W podobny sposób jak dla tabeli `rzeczy`, przekopiuj dane do jednej z założonych przez siebie, a jeszcze pustej tabeli.

Kopiować możemy również w drugą stronę, z tabeli do pliku tekstowego. Na przykład przekopiujemy tabelę `pracownicy` do pliku tekstowego `pracownicy1.txt`

```
=> \COPY pracownicy TO 'pracownicy1.txt' WITH DELIMITER ','
```

Sprawdź, jaka jest zawartość pliku `pracownicy1.txt`.

W podobny sposób jak z tabeli `pracownicy`, przekopiuj dane z jednej z twoich tabel do pliku tekstowego.

9. Wykonywanie serii poleceń zapisanych w pliku tekstowym

Jeżeli mamy zaplanowaną długą serię poleceń, to wygodnie będzie przygotować je w pliku tekstowym, a następnie zastosować polecenie wewnętrzne `\i <plik>` nakazujące wykonanie poleceń z pliku. Na przykład plik `polecenia.txt` ma następującą zawartość

```
SELECT * FROM pracownicy;
INSERT INTO pracownicy VALUES ('Bilski',108,82.3);
SELECT * FROM pracownicy;
SELECT * FROM rzeczy;
INSERT INTO rzeczy VALUES
(209448,'stojak',20.3,'2008-3-5','Bilski');
SELECT * FROM rzeczy;
```

Mając ten plik w katalogu `/home/student123/polecenia.txt` , dajemy polecenie wewnętrzne

```
\i  polecenia.txt
```

i widzimy na ekranie monitora efekt wykonanych poleceń.

Napisz własny plik `polecenia_student123.txt` (zamiast `student123` daj swoją nazwę użytkownika), z dowolnymi poleceniami dotyczącymi własnych dwóch tabel z punktów 6, 7. Sprawdź, że polecenia z pliku zostały wykonane.

10. Kopiowanie bazy danych

Bazę danych kopiujemy na przykład w celu przeniesienia jej na inny serwer. Możliwość kopiowania bazy danych jest przydatna także wtedy, gdy chcemy bazę danych zlikwidować na danym serwerze na pewien czas (będziemy mieli pewność, że nikt nie ma do niej dostępu) i udostępnić ją (założyć na nowo) w przyszłości.

Skopiujemy naszą bazę danych `student123` użytkownika `student123` (użyj tu swojej nazwy), skasujemy, a następnie założymy od nowa i przywrócimy skopiowane na początku dane.

Sprawdź, jakie bazy danych posiadasz

```
~$ psql -l | grep student123
```

Dokonaj zrzutu bazy danych `student123` do pliku typu SQL-skrypt

```
~$ pg_dump student123 > zrzutstudent123.sql
```

Skasuj swoją bazę danych `student123`

```
~$ dropdb student123
```

Sprawdzamy dając `~$ psql -l`, że bazy danych `student123` nie ma. Załóż od nowa swoją bazę danych

```
~$ createdb student123
```

Przywróć dane do bazy danych, tj. rozładuj plik skryptowy SQL

```
~$ psql -d student123 -f zrzutstudent123.sql
```

Zamiast zrzucać bazę danych do pliku skryptowego SQL, możemy dokonać zrzutu bazy danych do pliku archiwum, na przykład pliku archiwum o zwykłym formacie

```
~$ pg_dump -Fc student123 > zrzutstudent123.dump
```

Teraz bazę danych możemy skasować

```
~$ dropdb student123
```

(sprawdzamy dając `~$ psql -l`, że bazy danych `student123` nie ma), założyć od nowa

```
~$ createdb student123
```

i przywrócić dane do bazy danych

```
~$ pg_restore -d student123 zrzutstudent123.dump
```

11. Porządkowanie bazy danych

Używając bazę danych powodujemy, że wiele wierszy zostaje zmienionych, czy usuniętych. Usuwane wiersze nie znikają fizycznie z tabeli i zajmują niepotrzebnie pamięć. Bazę danych trzeba okresowo porządkować, „odkurzać”. Robi się to za pomocą polecenia `vacuumdb`, usuwającego fizycznie z bazy danych wszystkie niepotrzebne obiekty . Zapoznaj się ze szczegółami dotyczącymi tego polecenia (`~$ info vacuumdb`) . Można porządkować całą bazę danych, wybrane tabele lub wybrane kolumny. Na przykład polecimy „odkurzyć” tabelę `pracownicy` z prośbą o podanie szczegółów i analizę

```
~$ vacuumdb --verbose --analyze -t pracownicy
```

Dokonaj porządkowania własnej tabeli (lub kolumny tabeli lub całej bazy danych).

12. Bezpieczeństwo bazy danych, przywileje

Użytkownikiem bazy danych jest osoba lub stworzona przez nią aplikacja. Użytkownik tworząc obiekty bazy danych staje się automatycznie ich właścicielem, tylko on może je odczytywać, modyfikować, usuwać. Rzadko kiedy baza danych jest tworzona przez jednego użytkownika dla jednego użytkownika. Po stworzeniu bazy danych nadajemy innym, wybranym przez nas użytkownikom przywilej pełnego dostępu do danych lub tylko przywilej odczytu danych z bazy. Przywileje nadajemy poleceniem `GRANT` (przeczytaj jakie są możliwości tego polecenia, daj `\h GRANT`). Listę obiektów z ich przywilejami oglądniemy dając polecenie wewnętrzne `\dp` w `psql`.

Skoro mamy prawo nadawania przywilejów, to mamy także prawo ich odbierania. Nadane przez nas przywileje odbieramy poleceniem `REVOKE` (przeczytaj jakie są możliwości tego polecenia, daj `\h REVOKE`).

W tym laboratorium baza danych jest tworzona przez użytkownika-studenta wyłącznie do celów dydaktycznych. Nie nadawaj do niej uprawnień innym użytkownikom. Musisz mieć pewność, że inny użytkownik choćby przez nieuwagę nie skasuje ci tabeli, nad którą ciężko pracowałeś.

Na zakończenie zajęć daj `\q` (wyjście z `psql`), a następnie `exit` (zakończenie połączenia z bazą danych).

Dodatek 1.1. Lista poleceń aplikacji klienta PostgreSQL

clusterdb -- klaster bazy danych PostgreSQL

createdb -- zakładanie nowej bazy danych PostgreSQL

createlang -- definiowanie nowego języka proceduralnego PostgreSQL

createuser -- definiowanie nowego konta użytkownika PostgreSQL

dropdb -- usunięcie bazy danych PostgreSQL

droplang -- usunięcie języka proceduralnego PostgreSQL

dropuser -- usunięcie konta użytkownika PostgreSQL

ecpg -- wbudowany C preprocesor SQL

pg_config -- odzyskanie informacji o zainstalowanej wersji PostgreSQL

pg_dump -- ekstrakcja bazy danych PostgreSQL w plik skryptowy, czy inny plik archiwum

pg_dumpall -- ekstrakcja klastra baz danych PostgreSQL w plik skryptowy

pg_restore -- odtworzenie bazy danych PostgreSQL z pliku archiwum utworzonego przez pg_dump

psql -- interaktywny terminal PostgreSQL

reindexdb -- reindeksowanie bazy danych PostgreSQL

vacuumdb -- zbieranie i analizowanie „śmieci” z bazy danych PostgreSQL

Wszystkie polecenia aplikacji mogą być wykonane na dowolnym hoście, niezależnie od tego gdzie rezyduje serwer bazy danych.

Szczegóły dotyczące polecenia uzyskamy dając `info`, np.

```
~$ info createdb
```

Dodatek 1.2. Lista poleceń wewnętrznych (meta-polecenia) języka PostgreSQL (uzyskana poleceniem `\?`)

```
General
  \copyright         show PostgreSQL usage and distribution terms
  \g [FILE] or ;     execute query (and send results to file or |pipe)
  \h [NAME]          help on syntax of SQL commands, * for all commands
  \q                 quit psql

Query Buffer
  \e [FILE]              edit the query buffer (or file) with external editor
  \ef [FUNCNAME]         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
  \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)
  \d[S+]                 list tables, views, and sequences
  \d[S+]  NAME           describe table, view, sequence, or index
  \da[+]  [PATTERN]      list aggregates
  \db[+]  [PATTERN]      list tablespaces
  \dc[S]  [PATTERN]      list conversions
  \dC     [PATTERN]      list casts
  \dd[S]  [PATTERN]      show comments on objects
  \dD[S]  [PATTERN]      list domains
  \des[+] [PATTERN]      list foreign servers
  \deu[+] [PATTERN]      list user mappings
  \dew[+] [PATTERN]      list foreign-data wrappers
  \df[antw][S+] [PATRN]  list [only agg/normal/trigger/window] functions
  \dF[+]  [PATTERN]      list text search configurations
  \dFd[+] [PATTERN]      list text search dictionaries
  \dFp[+] [PATTERN]      list text search parsers
  \dFt[+] [PATTERN]      list text search templates
  \dg[+]  [PATTERN]      list roles (groups)
  \di[S+] [PATTERN]      list indexes
  \dl                    list large objects, same as \lo_list
  \dn[+]  [PATTERN]      list schemas
  \do[S]  [PATTERN]      list operators
  \dp     [PATTERN]      list table, view, and sequence access privileges
  \ds[S+] [PATTERN]      list sequences
  \dt[S+] [PATTERN]      list tables
```

```
  \dT[S+] [PATTERN]    list data types
  \du[+]  [PATTERN]    list roles (users)
  \dv[S+] [PATTERN]    list views
  \l[+]                list all databases
  \z      [PATTERN]    same as \dp

Formatting
  \a                   toggle between unaligned and aligned output mode
  \C [STRING]          set table title, or unset if none
  \f [STRING]          show or set field separator for unaligned query output
  \H                   toggle HTML output mode (currently off)
  \pset NAME [VALUE]   set table output option
                       (NAME := {format|border|expanded|fieldsep|footer|null|
                       numericlocale|recordsep|tuples_only|title|tableattr|pager})
  \t [on|off]          show only rows (currently off)
  \T [STRING]          set HTML <table> tag attributes, or unset if none
  \x [on|off]          toggle expanded output (currently off)

Connection
  \c[onnect] [DBNAME|- USER|- HOST|- PORT|-]
                       connect to new database (currently "student123")
  \encoding [ENCODING]  show or set client encoding
  \password [USERNAME]  securely change the password for a user

Operating System
  \cd [DIR]            change the current working directory
  \timing [on|off]     toggle timing of commands (currently off)
  \! [COMMAND]         execute command in shell or start interactive shell

Variables
  \prompt [TEXT] NAME   prompt user to set internal variable
  \set [NAME [VALUE]]   set internal variable, or list all if no parameters
  \unset NAME           unset (delete) internal variable

Large Objects
  \lo_export LOBOID FILE
  \lo_import FILE [COMMENT]
  \lo_list
  \lo_unlink LOBOID     large object operations
```

Dodatek 1.3. Lista poleceń języka PostgreSQL, dla których jest dostępna pomoc (uzyskana poleceniem wewnętrznym `\h`)

Available help:

ABORT	CREATE LANGUAGE	DROP TEXT SEARCH DICTIONARY
ALTER AGGREGATE	CREATE OPERATOR	DROP TEXT SEARCH PARSER
ALTER CONVERSION	CREATE OPERATOR CLASS	DROP TEXT SEARCH TEMPLATE
ALTER DATABASE	CREATE OPERATOR FAMILY	DROP TRIGGER
ALTER DOMAIN	CREATE ROLE	DROP TYPE
ALTER FOREIGN DATA WRAPPER	CREATE RULE	DROP USER
ALTER FUNCTION	CREATE SCHEMA	DROP USER MAPPING
ALTER GROUP	CREATE SEQUENCE	DROP VIEW
ALTER INDEX	CREATE SERVER	END
ALTER LANGUAGE	CREATE TABLE	EXECUTE
ALTER OPERATOR	CREATE TABLE AS	EXPLAIN
ALTER OPERATOR CLASS	CREATE TABLESPACE	FETCH
ALTER OPERATOR FAMILY	CREATE TEXT SEARCH CONFIGURATION	GRANT
ALTER ROLE	CREATE TEXT SEARCH DICTIONARY	INSERT
ALTER SCHEMA	CREATE TEXT SEARCH PARSER	LISTEN
ALTER SEQUENCE	CREATE TEXT SEARCH TEMPLATE	LOAD
ALTER SERVER	CREATE TRIGGER	LOCK
ALTER TABLE	CREATE TYPE	MOVE
ALTER TABLESPACE	CREATE USER	NOTIFY
ALTER TEXT SEARCH CONFIGURATION	CREATE USER MAPPING	PREPARE
ALTER TEXT SEARCH DICTIONARY	CREATE VIEW	PREPARE TRANSACTION
ALTER TEXT SEARCH PARSER	DEALLOCATE	REASSIGN OWNED
ALTER TEXT SEARCH TEMPLATE	DECLARE	REINDEX
ALTER TRIGGER	DELETE	RELEASE SAVEPOINT
ALTER TYPE	DISCARD	RESET
ALTER USER	DROP AGGREGATE	REVOKE
ALTER USER MAPPING	DROP CAST	ROLLBACK
ALTER VIEW	DROP CONVERSION	ROLLBACK PREPARED
ANALYZE	DROP DATABASE	ROLLBACK TO SAVEPOINT
BEGIN	DROP DOMAIN	SAVEPOINT
CHECKPOINT	DROP FOREIGN DATA WRAPPER	SELECT
CLOSE	DROP FUNCTION	SELECT INTO
CLUSTER	DROP GROUP	SET
COMMENT	DROP INDEX	SET CONSTRAINTS
COMMIT	DROP LANGUAGE	SET ROLE
COMMIT PREPARED	DROP OPERATOR	SET SESSION AUTHORIZATION
COPY	DROP OPERATOR CLASS	SET TRANSACTION
CREATE AGGREGATE	DROP OPERATOR FAMILY	SHOW
CREATE CAST	DROP OWNED	START TRANSACTION
CREATE CONSTRAINT TRIGGER	DROP ROLE	TABLE
CREATE CONVERSION	DROP RULE	TRUNCATE
CREATE DATABASE	DROP SCHEMA	UNLISTEN
CREATE DOMAIN	DROP SEQUENCE	UPDATE
CREATE FOREIGN DATA WRAPPER	DROP SERVER	VACUUM
CREATE FUNCTION	DROP TABLE	VALUES
CREATE GROUP	DROP TABLESPACE	WITH
CREATE INDEX	DROP TEXT SEARCH CONFIGURATION	

Dodatek 1.4. Zawartość pliku `spis_rzeczy.txt`

```
177638,komputer,5.5,'1999-10-12',Kowalski
284749,biurko,33.4,'1995-2-9',Wolski
374849,krzesło,2.5,'2005-11-23',Jarosz
475053,fotel,12.3,'2003-2-12,Lenart
946462,monitor,3.2,'1998-12-25',Dolasa
746924,biblioteczka,188.4,'1985-4-24',Wolski
648247,regał,106.3,'2004-9-23',Rolski
378463,szafa,205.6,'1997-5-21',Krol
459347,szafka,28,'1996-12-13',Szuba
823479,drukarka,4.2,'2007-1-26',Jarecki
897345,kserokopiarka,28.4,'2006-6-22',Lenart
645945,krzesło,2.5,'2005-11-23',Krol
389557,komputer,5.5,'1999-1-12',Jarosz
873401,biurko,33.4,'1995-2-9',Dolasa
847041,szafa,158,'2002-12-3',Szuba
783017,regał,85.6,'2006-9-22',Kowalski
730733,krzesło,3,'2004-3-24',Wolski
834791,komputer,4.7,'2007-5-14',Lenart
745682,biurko,48,'2001-4-3',Jarosz
763453,monitor,3.8,'2004-5-21',Kowalski
384751,fotel,9.8,'1998-12-3',Dolasa
734669,komputer,4.7,'2005-2-17',Dolasa
893577,krzesło,3.1,'2009-11-23',Kowalski
346181,biurko,28.7,'2004-2-24',Wolski
834743,krzesło,3.1,'2009-11-23',Dragan
248095,fotel,9.8,'1998-12-3',Lenart
783461,krzesło,3.1,'2009-11-23',Rolski
893942,fotel,9.8,'2002-4-25',Krol
893571,krzesło,3.1,'2009-11-23',Jarecki
746133,fotel,9.8,'2002-4-25',Szuba
763468,krzesło,3.1,'2009-11-23',Ekiert
```

Laboratorium Baz Danych, PostgreSQL

Ćwiczenie 2. Polecenia w języku PostgreSQL

Użytkownik ma dostęp do bazy danych wyłącznie za pośrednictwem DBMS poprzez wydawanie poleceń w języku PostgreSQL. Poleceń tych jest ponad 200 plus opcje (p. dodatki 1.1, 1.2), które w wielu poleceniach są bardzo rozbudowane. Nie jest możliwe przećwiczenie wszystkich poleceń wraz z opcjami. Nie jest to też niezbędne. Z dydaktycznego punktu widzenia wystarczy, że student zapozna się tylko z najważniejszymi poleceniami i ich opcjami, i temu jest poświęcone niniejsze ćwiczenie laboratoryjne. Tak jak w przypadku poznawania każdego innego języka student poznaje niezbędne podstawy, a do najdrobniejszych szczegółów dochodzi się dopiero w praktyce inżynierskiej.

1. Modyfikowanie tabel

Połącz się z serwerem tak jak w ćwiczeniu 1 (dostęp do komputera klienta jako użytkownik: **baza**, hasło: **danych**; dostęp do serwera IP 153.19.50.110 poprzez program `putty`, podaj swój login i hasło). Rozpocznij pracę w języku PostgreSQL w trybie interfejsu tekstowego (daj `~$ psql`). Sprawdź, czy posiadasz w swojej bazie danych tabele `pracownicy`, `rzeczy` i dwie inne tabele założone na poprzednich zajęciach laboratoryjnych (daj `\dt`). Przypomnij sobie budowę tabel (daj `\d pracownicy`, `\d rzeczy`, itd.).

Założoną tabelę możemy modyfikować poleceniem `ALTER TABLE` (oglądnij pełną składnię tego polecenia dając `\h ALTER TABLE`). Zmodyfikujemy tabelę `pracownicy` dopisując w niej nową kolumnę z datą urodzenia (podajemy nazwę i typ kolumny)

```
ALTER TABLE pracownicy ADD COLUMN data_urodzenia  date;
```

Sprawdź, czy kolumna dodała się, dając `\d pracownicy`. Jedno polecenie `ALTER TABLE` pozwala dodać tylko jedną kolumnę. Kolumna doda się jako ostatnia. Wyraz `COLUMN` jest opcjonalny.

Dowolną kolumnę usuwamy podobnym poleceniem jak powyżej tyle, że zamiast `ADD` piszemy `DROP` i podajemy tylko nazwę kolumny bez typu. Usuniemy dodaną powyżej kolumnę (wyraz `COLUMN` jest opcjonalny)

```
ALTER TABLE pracownicy DROP COLUMN data_urodzenia ;
```

Dając `\d pracownicy` widzimy, iż tej kolumny już nie ma.

Poleceniem `ALTER TABLE` możemy dodawanej kolumnie (lub już istniejącej kolumnie) nadać wartość domyślną `DEFAULT` lub narzucić ograniczenie.

Nadamy wartość domyślną `DEFAULT 'nazwisko nieznane'` kolumnie `nazwisko` w tabeli `pracownicy`

```
ALTER TABLE pracownicy ALTER COLUMN nazwisko SET DEFAULT
'nazwisko nieznane' ;
```

Dając `\d pracownicy` widzimy tę zmianę. Dodajmy teraz wiersz bez nazwiska

```
INSERT INTO pracownicy (numer_pokoju, ciezar_pracownika)
VALUES (103, 95.3) ;
```

Po napisaniu

```
SELECT * FROM pracownicy ;
```

widzimy, że w kolumnie `nazwisko` została wpisana wartość domyślna `'nazwisko nieznane'`.

Gdy dojdziemy do wniosku, że wstawianie wartości domyślnej nie jest już potrzebne, to usuniemy nakaz stosowania wartości domyślnej `DEFAULT` poleceniem

```
ALTER TABLE pracownicy ALTER COLUMN nazwisko DROP DEFAULT  ;
```

Dając `\d pracownicy` widzimy tę zmianę. Usuniemy też niepotrzebny wiersz z nieznanym nazwiskiem

```
DELETE FROM pracownicy WHERE nazwisko='nazwisko nieznane' ;
```

Dajemy

```
SELECT * FROM pracownicy ;
```

i widzimy, że nie ma już pracowników o nieznanym nazwisku.

Nie zrobimy tego, ale gdybyśmy chcieli usunąć wszystkie wiersze z tabeli, to dalibyśmy `DELETE FROM pracownicy;` (pozostanie pusta tabela, nie trzeba jej zakładać od nowa). Do tego celu można byłoby też użyć szybszego polecenia `TRUNCATE` (patrz `\h TRUNCATE`).

Narzucimy ograniczenie `UNIQUE` o nazwie `ogr` na kolumnę `nazwisko` w tabeli `pracownicy`

```
ALTER TABLE pracownicy ADD CONSTRAINT ogr UNIQUE(nazwisko);
```

Dając `\d pracownicy` widzimy tę zmianę. Teraz próba wpisania pracownika o już istniejącym nazwisku, np.

```
INSERT INTO pracownicy(nazwisko) VALUES ('Kowalski') ;
```

zakończy się niepowodzeniem (będzie sygnalizacja błędu `ERROR:`). Usuniemy nałożone ograniczenie

```
ALTER TABLE pracownicy DROP CONSTRAINT ogr  ;
```

Dając `\d pracownicy` widzimy tę zmianę (ograniczenia `ogr` już niema).

Narzucimy ograniczenie `NOT NULL` na kolumnę `nazwisko` w tabeli `pracownicy`

```
ALTER TABLE pracownicy ALTER COLUMN nazwisko SET NOT NULL;
```

Teraz próba wpisania wiersza bez nazwiska

```
INSERT INTO pracownicy(numer_pokoju, ciezar_pracownika)
VALUES (103, 75) ;
```

zakończy się niepowodzeniem (`ERROR:)`. Ograniczenie NOT NULL nie ma nazwy i jest usuwane inaczej niż ograniczenie z nazwą

```
ALTER TABLE pracownicy ALTER COLUMN nazwisko DROP NOT NULL ;
```

Dając `\d pracownicy` widzimy, że tego ograniczenia już nie ma.

Polecenie `ALTER TABLE` pozwala też zmienić typ danych kolumny. Na przykład kolumna `nazwisko` miała typ `varchar(80)` , który zmienimy na typ `text`

```
ALTER TABLE pracownicy ALTER COLUMN nazwisko TYPE text ;
```

Moglibyśmy zmienić nazwę kolumny

```
ALTER TABLE nazwa_tabeli RENAME COLUMN stara_nazwa TO nowa_nazwa ;
```

lub nazwę tabeli

```
ALTER TABLE stara_nazwa RENAME TO nowa_nazwa ;
```

ale proszę to przećwiczyć na innych tabelach niż tabele `pracownicy, rzeczy` , gdyż te tabele będą używane w dalszych punktach ćwiczenia laboratoryjnego.

Powtórz wszystkie powyższe czynności na własnych tabelach, założonych na poprzednich zajęciach.

2. Aktualizowanie danych

Dodaj kolumnę `ocena` w tabeli `pracownicy`

```
ALTER TABLE pracownicy ADD COLUMN ocena text ;
```

Zaktualizuj dane podając, czy pracownik ma: niedowagę, w normie, nadwagę. Aktualizacji dokonujemy poleceniem `UPDATE` (oglądnij pełną składnię tego polecenia `\h UPDATE`)

```
UPDATE pracownicy SET ocena='niedowaga' WHERE ciezar_pracownika<60 ;

UPDATE pracownicy SET ocena='w normie' WHERE
(ciezar_pracownika>=60)AND(ciezar_pracownika<=80) ;
```

```
UPDATE pracownicy SET ocena='nadwaga' WHERE ciezar_pracownika>80 ;
```

Oglądnij wynik aktualizacji poleceniem

```
SELECT*FROM pracownicy;
```

Usuń kolumnę `ocena` jako już niepotrzebną

```
ALTER TABLE pracownicy DROP COLUMN ocena ;
```

Sprawdź, że tej kolumny już nie ma (daj `\d pracownicy`).

Powtórz powyższe czynności na własnych tabelach.

3. Zapytania SELECT

Jeszcze raz przypomnij sobie pełną składnię polecenia `SELECT` (`\h SELECT`). Przećwiczymy występujące tam opcje.

Zapytaj o nazwiska z tabeli `pracownicy`

```
SELECT nazwisko FROM pracownicy ;
```

Niech nazwiska zostaną uporządkowane rosnąco

```
SELECT nazwisko FROM pracownicy ORDER BY nazwisko ASC ;
```

Wylistuj numery pokojów (bez powtórzeń, `DISTINCT`) w porządku malejącym

```
SELECT DISTINCT numer_pokoju FROM pracownicy ORDER BY
numer_pokoju DESC ;
```

Użyjemy pięć podstawowych funkcji agregujących (`COUNT, AVG, SUM, MIN, MAX`) w jednym poleceniu dla pracowników z nadwagą

```
SELECT COUNT(ciezar_pracownika), AVG(ciezar_pracownika),
SUM(ciezar_pracownika), MIN(ciezar_pracownika),
MAX(ciezar_pracownika) FROM pracownicy WHERE
ciezar_pracownika>=80 ;
```

Poznaliśmy liczbę pracowników z nadwagą, średni ciężar pracownika z nadwagą, sumę ciężaru pracowników z nadwagą, minimalny ciężar pracownika z nadwagą, maksymalny ciężar pracownika z nadwagą.

Wylistuj pracowników z pokoju 108

```
SELECT nazwisko FROM pracownicy WHERE numer_pokoju=108 ;
```

Wylistuj w porządku alfabetycznym pracowników o nazwiskach rozpoczynających się od litery D

```
SELECT nazwisko FROM pracownicy WHERE nazwisko LIKE 'D%'
ORDER BY ASC;
```

Zamiast 'D%' mogło być $$D%$$ (PostgreSQL dopuszcza tak zwane dolarowe apostrofowanie).

Podaj dane pracowników o nazwiskach rozpoczynających się od litery między 'E' i 'K'

```
SELECT * FROM pracownicy WHERE nazwisko BETWEEN  'E' AND 'K' ;
```

Podaj dane pracowników o nazwiskach kończących się na 'ski'

```
SELECT * FROM  pracownicy WHERE nazwisko LIKE  '%ski' ;
```

Oblicz ciężar pracowników w poszczególnych pokojach

```
SELECT  numer_pokoju,  SUM(ciezar_pracownika) FROM pracownicy
GROUP BY  numer_pokoju;
```

Wylistuj w porządku malejącego ciężaru te pokoje, w których ciężar pracowników przekracza 150kg

```
SELECT  numer_pokoju,  SUM(ciezar_pracownika) FROM pracownicy
GROUP BY  numer_pokoju HAVING  SUM(ciezar_pracownika)>150
ORDER BY  2  DESC ; --dwójka oznacza numer kolumny
```

W zapytaniu SELECT możemy operować danymi z różnych tabel. Wylistuj rzeczy znajdujące się w pokoju 108 w porządku rosnącego numeru inwentarzowego

```
SELECT numer_pokoju, nazwisko, nazwa, nr_inwentarzowy FROM
pracownicy, rzeczy WHERE
(numer_pokoju=108)AND(nazwisko=na_stanie_pracownika)
ORDER BY nr_inwentarzowy ;
```

Nazwiska występowały w tabeli pracownicy w kolumnie nazwisko i w tabeli rzeczy w kolumnie na_stanie_pracownika. Byliśmy przezorni, nie nastąpił konflikt jednakowych nazw kolumn. Gdyby jednak w obydwu tabelach były jednakowe nazwy kolumn nazwisko, to rozróżnilibyśmy je tak jak poniżej, dopisując nazwę tabeli przed kropką

```
SELECT numer_pokoju, pracownicy.nazwisko, nazwa,
nr_inwentarzowy FROM pracownicy, rzeczy WHERE
(numer_pokoju=108)AND(pracownicy.nazwisko=rzeczy.nazwisko)
ORDER BY nr_inwentarzowy ;
```

Jest dopuszczalne posłużenie się podzapytaniem (SELECT w SELECT). Na przykład

```
SELECT DISTINCT numer_pokoju, nazwisko, nazwa FROM pracownicy,
rzeczy WHERE nazwa IN (SELECT nazwa FROM rzeczy WHERE
(na_stanie_pracownika='Kowalski')AND(numer_pokoju=101) ) ;
```

Możemy dokonać sumowania (`UNION`), odejmowania (`EXCEPT`) i przecięcia (`INTERSECT`) zbiorów danych tego samego typu. Na przykład wykryjemy takie same nazwiska w tabelach `pracownicy` i `rzeczy`

```
SELECT nazwisko FROM pracownicy UNION SELECT
na_stanie_pracownika FROM rzeczy ;
```

Podobnie dokonaj odejmowania (to działanie nie jest przemienne)

```
SELECT nazwisko FROM pracownicy EXCEPT SELECT
na_stanie_pracownika FROM rzeczy ;

SELECT na_stanie_pracownika FROM rzeczy EXCEPT SELECT nazwisko
FROM pracownicy ;
```

i przecięcia

```
SELECT nazwisko FROM pracownicy INTERSECT SELECT
na_stanie_pracownika FROM rzeczy ;
```

Wyniki uzyskane poleceniem `SELECT` możemy wstawić do innej tabeli o strukturze dopasowanej do tych wyników (przypomnij sobie pełną składnię polecenia `\h INSERT`)

```
INSERT INTO <nazwa tabeli> [<lista kolumn>] <polecenie SELECT>
```

Załóż tabelę `pracownicy_z_nadwaga` o strukturze takiej samej jak struktura tabeli `pracownicy`

```
CREATE TABLE pracownicy_z_nadwagą(
     nazwisko          text,
     numer_pokoju      int,
     ciezar_pracownika real    );
```

Sprawdź, że masz tę tabelę w bazie danych `\d pracownicy_z_nadwaga`. Teraz można wpisać do tej tabeli pracowników z nadwagą

```
     INSERT INTO pracownicy_z_nadwaga SELECT * FROM pracownicy
     WHERE ciezar_pracownika>80 ;
```

W skrajnym przypadku można w ten sposób przekopiować całą tabelę bez zmian.

Powtórz wszystkie powyższe czynności na własnych tabelach.

4. Widoki VIEW

Widok (perspektywa) jest wirtualną tabelą utworzoną z wierszy i kolumn wziętych z jednej lub wielu tabel. W bazie danych jest pamiętana definicja widoku (w słowniku danych), z

której jest tworzony za każdym razem w miarę potrzeby widok. Widok ma swoją nazwę i można na nim wykonywać polecenia SELECT jak na każdej innej tabeli. Utworzymy widok `stare_rzeczy` wybierając z tabeli `rzeczy` te rzeczy, które zostały wyprodukowane przed rokiem 2000 (podglądnij pełną składnię polecenia `\h CREATE VIEW`)

```
CREATE VIEW stare_rzeczy AS SELECT * FROM rzeczy WHERE
data_produkcji< '2000-1-1' ;
```

Sprawdzamy, czy założono widok `stare_rzeczy` dając `\dv`, a następnie `\d stare_rzeczy`. Oglądamy widok dając

```
SELECT * FROM stare_rzeczy ;
```

Zakładamy widok z rzeczami znajdującymi się w pokoju 108 o nazwie `rzeczy108` z tabel `pracownicy, rzeczy`

```
CREATE VIEW rzeczy108 AS SELECT numer_pokoju, nazwisko, nazwa,
nr_inwentarzowy FROM pracownicy, rzeczy WHERE
(numer_pokoju=108)AND(nazwisko=na_stanie_pracownika)
ORDER BY nr_inwentarzowy ;
```

Oglądamy widok

```
SELECT * FROM rzeczy108 ;
```

Powtórz powyższe czynności na własnych tabelach.

5. Klucz pierwotny PK i klucz obcy FK

Klucz pierwotny PRIMARY KEY jest taką kolumną lub grupą kolumn, która może być użyta do jednoznacznego identyfikowania wierszy w tabeli. Zgodnie z teorią relacyjnych baz danych każda tabela powinna mieć klucz główny. PostgreSQL nie wymusza istnienia klucza głównego, ale zaleca się, aby przestrzegać generalnych zasad (wymóg integralności bazy danych) i nadawać każdej tabeli klucz główny. W przypadku tabeli `rzeczy` naturalnym kandydatem do miana klucza głównego jest kolumna `nr_inwentarzowy` i z niej zrobimy klucz główny. Ponieważ nie zrobiliśmy tego zakładając tabelę

```
CREATE TABLE rzeczy (
     nr_inwentarzowy  int  PRIMARY KEY ,
     nazwa  varchar(80) ,
     ciezar_rzeczy  real ,
     data_produkcji  date,
     na_stanie_pracownika varchar(80)  ) ;
```

to zrobimy to teraz modyfikując tabelę

```
ALTER TABLE rzeczy ADD PRIMARY KEY (nr_inwentarzowy) ;
```

Dając `\d rzeczy` widzimy wprowadzoną modyfikację. Teraz próba wpisania wiersza z już istniejącym numerem inwentarzowym, np.

```
INSERT INTO rzeczy VALUES (177638, 'komputer' , 5.5 ,
'1999-10-12' , 'Kowalski' );
```

lub wiersza bez numeru inwentarzowego, np.

```
INSERT INTO rzeczy(nazwa,ciezar_rzeczy) VALUES ('lampa',2.8);
```

zakończy się niepowodzeniem (sygnalizacja błędu `ERROR:`).

Dla tabeli `pracownicy` niech kluczem głównym będzie kolumna `nazwisko`. Jest to dyskusyjny wybór, ale wynika tylko z uproszczeń dydaktycznych, powinno się operować nazwiskiem z imieniem, a jeszcze lepiej z dwojgiem imion. Najlepszy byłby PESEL lub kolumna będąca identyfikatorem (np. liczba porządkowa), sekwencją `SERIAL` automatycznie zwiększającą się dla kolejnych wpisywanych rekordów (patrz `\h CREATE SEQUENCE`). Ponieważ nie założyliśmy klucza głównego przy zakładaniu tabeli `pracownicy`

```
CREATE TABLE pracownicy (
     nazwisko text  PRIMARY KEY ,
     numer_pokoju  int ,
     ciezar_pracownika  real ) ;
```

to zrobimy to teraz modyfikując tabelę

```
ALTER TABLE  pracownicy  ADD PRIMARY KEY (nazwisko) ;
```

Oglądnij wprowadzoną zmianę dając `\d pracownicy`. Teraz próba wpisania wiersza z już istniejącym nazwiskiem lub bez nazwiska zakończy się niepowodzeniem (sprawdź to).

Zauważamy, że kolumna `na_stanie_pracownika` w tabeli `rzeczy` zawiera nazwiska z kolumny `nazwisko PRIMARY KEY` z tabeli `pracownicy`. Kolumna `na_stanie_pracownika` w tabeli `rzeczy` jest kluczem obcym `FOREIGN KEY` pochodzącym z tabeli `pracownicy`. Ponieważ nie założyliśmy klucza obcego przy zakładaniu tabeli `rzeczy`

```
CREATE TABLE rzeczy (
     nr_inwentarzowy  int  PRIMARY KEY ,
     nazwa  varchar(80) ,
     ciezar_rzeczy  real ,
     data_produkcji  date,
     na_stanie_pracownika varchar(80) REFERENCES
                                    pracownicy(nazwisko));
```

to zrobimy to teraz

```
ALTER TABLE rzeczy ADD FOREIGN KEY (na_stanie_pracownika)
REFERENCES pracownicy(nazwisko) ;
```

Oglądnij wprowadzone zmiany dając `\d rzeczy` i `\d pracownicy`. Teraz nie jest możliwe operowanie w kolumnie `na_stanie_pracownika` nazwiskami, których nie ma w kolumnie `nazwisko` tabeli `pracownicy`. Sprawdź to.

Powtórz powyższe czynności na własnych tabelach.

6. Transakcje

Transakcja jest pojęciem mającym ogromne praktyczne znaczenie w systemie bazy danych. Transakcja jest blokiem poleceń pozwalającym dokonywać transakcji, przesyłać bezpiecznie np. pieniądze z konta jednego użytkownika na konto drugiego użytkownika, chociaż nie zawsze musi chodzić w transakcji o pieniądze. Każda baza danych ma zbiór właściwości ACID, które gwarantują bezpieczeństwo przeprowadzenia transakcji. Skrót ACID oznacza:

- Atomicity – atomowość transakcji oznacza, że jest ona nie do rozbicia jak atom (wprawdzie fizycy potrafią rozbić atom, ale w mowie potocznej atom uchodzi za taką drobinkę, że chociaż jest zbudowana z cząstek elementarnych to jest nie do rozbicia, a jak zostanie rozbita to przestaje być atomem). Transakcja może być wykonana albo w całości, albo wcale. Jeżeli transakcja nie została dokończona, to wszystkie dotychczasowe pośrednie czynności uważa się za niebyłe. Tak jak atom transakcja istnieje tylko jako całość.
- Consistency – spójność transakcji oznacza, że transakcja nie może naruszyć integralności bazy danych, zgodności ze schematem bazy danych.
- Isolation – izolacja transakcji oznacza, że dwie lub więcej transakcji przeprowadzanych równolegle jest od siebie odizolowanych, jak i od innych użytkowników bazy danych niekoniecznie przeprowadzających transakcje, przy czym mamy możliwość wyboru jednego z czterech poziomów izolacji (read uncommitted, read committed, repeatable read, serializable).
- Durability – trwałość transakcji oznacza, że w każdej chwili przeprowadzania transakcji jak i oczywiście po zakończeniu transakcji jest zapewniona trwałość danych. System zapewnia aktualne dane nawet po awarii zasilania.

Zilustrujemy transakcję na prostym przykładzie zamiany miejscami dwóch pracowników w dwóch pokojach (aktualnie Kowalski 101, Jarosz 103; mają zamienić się pokojami). Nie dopuścimy do tego, aby chociaż przez ułamek sekundy jeden pracownik już był przypisany do nowego pokoju, a drugi jeszcze nie. W przeciwnym razie godziłoby to w integralność bazy danych, w której jeden pracownik siedzi w jednym pokoju, nie może figurować w dwóch pokojach jednocześnie. Gdyby „przeprowadzkę” zrobić w następujący sposób

```
UPDATE pracownicy SET numer_pokoju=103 WHERE
nazwisko='Kowalski' ;
UPDATE pracownicy SET numer_pokoju=101 WHERE
nazwisko='Jarosz' ;
```

to między wykonaniem pierwszego i drugiego polecenia `UPDATE` upłynie czas. Będzie taki przedział czasu, w którym Kowalski już się przeprowadził, a Jarosz jeszcze nie i siedzą razem w pokoju 103. W skrajnym przypadku między jednym i drugim poleceniem może wydarzyć się awaria sprzętu (zanik napięcia sieciowego) i panowie będą siedzieli razem do usunięcia awarii. Nietrudno wyobrazić sobie, jakie byłyby szkody w przypadku przelewu pieniędzy. Pieniądze pobrane z jednego konta klienta nie wpłynęłyby na konto drugiego klienta,

„rozpłynęłyby” się w systemie. Jeszcze gorzej byłoby w odwrotnej kolejności, gdyby pieniądze wpłacono na konto jednego klienta i nie zdążono ich zdjąć z konta drugiego klienta, gdyż te same pieniądze byłyby jednocześnie na dwóch kontach. Jedynym rozwiązaniem problemu jest tutaj wykonanie *transakcji*, czyli niepodzielnego, *atomowego* bloku poleceń, gdzie mamy gwarancję, że albo zostaną wykonane wszystkie polecenia, albo żadne z nich.

Transakcja jest ciągiem poleceń SQL umieszczonych między poleceniami `BEGIN` i `COMMIT` (`BEGIN` jest tutaj poleceniem, kończy się średnikiem)

```
BEGIN ;
   UPDATE pracownicy SET numer_pokoju=103 WHERE
    nazwisko='Kowalski' ;
   UPDATE pracownicy SET numer_pokoju=101 WHERE
    nazwisko='Jarosz' ;
COMMIT ;
```

Gdybyśmy z pewnych względów chcieli zrezygnować z już rozpoczętej transakcji (np. przypomnieliśmy sobie, że Kowalski już nie pracuje lub chce się przeprowadzić do innego pokoju niż 103), to zamiast `COMMIT` damy `ROLLBACK` (wszystkie polecenia po `BEGIN` do `ROLLBACK` zostaną uznane za niebyłe, tak jakby transakcja w ogóle nie rozpoczęła się)

```
BEGIN ;
   UPDATE pracownicy SET numer_pokoju=103 WHERE
    nazwisko='Kowalski' ;
   UPDATE pracownicy SET numer_pokoju=101 WHERE
    nazwisko='Jarosz' ;
ROLLBACK ;
```

Teraz można rozpocząć nową transakcję od `BEGIN`.

Rozpoczynając transakcję od `BEGIN`, zanim zakończymy tę transakcję dając `COMMIT` (popełnij, spełnij), mamy możliwość cofnięcia się `ROLLBACK TO <nazwa>` do wcześniejszego miejsca, punktu bezpieczeństwa w transakcji oznaczonego jako `SAVEPOINT <nazwa>`. Wszystkie polecenia między `SAVEPOINT <nazwa>` , a `ROLLBACK TO <nazwa>` zostaną uznane za niebyłe. Rozmieszczając dostatecznie gęsto punkty bezpieczeństwa będziemy mogli powrócić do newralgicznych punktów transakcji. Na przykład, gdy w połowie transakcji przypomnieliśmy sobie, że Jarosz chce się przeprowadzić nie do pokoju 101, ale 24, to transakcja będzie przebiegała następująco

```
BEGIN ;
   UPDATE pracownicy SET numer_pokoju=103 WHERE
    nazwisko='Kowalski' ;
   SAVEPOINT punkt_bezpieczenstwa;
   UPDATE pracownicy SET numer_pokoju=101 WHERE
    nazwisko='Jarosz' ;
   ROLLBACK TO punkt_bezpieczenstwa;
   UPDATE pracownicy SET numer_pokoju=24 WHERE
    nazwisko='Jarosz' ;
COMMIT ;
```

Transakcja została popełniona (sprawdź to dając `SELECT*FROM pracownicy;` będzie Kowalski 103, Jarosz 24).

Po zakończeniu powyższej transakcji dokonajmy nowej zmiany pokoju pracownika

```
UPDATE pracownicy SET numer_pokoju=101 WHERE
nazwisko='Jarosz';
```

Obecnie Kowalski siedzi w pokoju 103, a Jarosz w pokoju 101.

Dokonamy następnej transakcji polegającej na wymianie wymienionych powyżej panów pokojami. Spróbujemy przeszkodzić tej transakcji, zakłócić ją symulując drugiego użytkownika operującego w tym samym czasie tą samą tabelą `pracownicy`, tymi samymi nazwiskami. Nie rozłączaj się z bazą danych, to będzie lewe okno. Połącz się drugi raz z bazą danych (użyj `putty`), to będzie prawe okno imitujące drugiego użytkownika. Pracuj jednocześnie w dwóch oknach: lewym i prawym. Wpisuj polecenia w lewym i prawym oknie w kolejności takiej, jak w poniższej tabelce. W lewym oknie jest przeprowadzana transakcja o domyślnym poziomie izolacji `READ COMMITTED`. Podstawową cechą bazy danych jest jej współbieżność. Baza danych jest użytkowana przez wielu użytkowników w tym samym czasie, w tym jeden użytkownik może mieć w jednym czasie otwartą dowolną liczbę połączeń z bazą danych. Transakcja ogranicza współbieżność bazy danych tym bardziej, im wyższy jest poziom izolacji transakcji. Opcjonalnie zamiast `BEGIN, COMMIT` można pisać `BEGIN Nazwa, COMMIT Napis`.

Okno lewe	Okno prawe
BEGIN TRANSACTION; UPDATE pracownicy SET numer_pokoju=101 WHERE nazwisko='Kowalski'; SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Jarosz | 101 Kowalski | 101	
	SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Kowalski | 103 Jarosz | 101 UPDATE pracownicy SET numer_pokoju=105 WHERE nazwisko='Kowalski';
UPDATE pracownicy SET numer_pokoju=103 WHERE nazwisko='Jarosz'; COMMIT TRANSACTION; SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Jarosz | 103 Kowalski | 105 UPDATE pracownicy SET numer_pokoju=101 WHERE nazwisko='Kowalski';	**Użytkownikowi „zawiesił się, zakleszczył system", musi czekać na dokończenie transakcji.** Tuż po zakończeniu transakcji system „odblokował się" i zostało dokończone polecenie UPDATE.

Dopiero dając poza transakcją `UPDATE` osiągnęliśmy zamierzony stan: Kowalski 101, Jarosz 103.

Dokonamy kolejnej transakcji. Znowu wymienimy panów pokojami, ale teraz będzie to transakcja o podwyższonym, najwyższym poziomie izolacji `TRANSACTION ISOLATION LEVEL SERIALIZABLE`.

Okno lewe	Okno prawe
BEGIN TRANSACTION; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; UPDATE pracownicy SET numer_pokoju=103 WHERE nazwisko='Kowalski'; SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Jarosz | 103 Kowalski | 103 UPDATE pracownicy SET numer_pokoju=101 WHERE nazwisko='Jarosz'; **ERROR: could not serialize access due to concurrent update** ROLLBACK; SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ---------- +---------- Kowalski | 101 Jarosz | 105	SELECT nazwisko,numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Kowalski | 101 Jarosz | 103 UPDATE pracownicy SET numer_pokoju=105 WHERE nazwisko='Jarosz'; SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Kowalski | 101 Jarosz | 105

Widzimy, że podniesienie poziomu izolacji transakcji spowodowało zmniejszenie współbieżności bazy danych (poprzednio byliśmy w stanie dokończyć transakcję, teraz nie).

Radykalnie zwiększymy odporność transakcji na zakłócenia zamykając dostęp do tabeli na czas transakcji poleceniem `LOCK` (oglądnij składnię polecenia, `\h LOCK`). Polecenie LOCK działa tylko wewnątrz transakcji, po popełnieniu transakcji tabela automatycznie odblokowuje się. Ponownie zamienimy pokojami panów, aktualnie: Kowalski 101, Jarosz 105.

Okno lewe	Okno prawe
BEGIN TRANSACTION; LOCK TABLE pracownicy;	
	SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz';
SELECT nazwisko, numer_pokoju FROM pracownicy WHERE nazwisko='Kowalski' OR nazwisko='Jarosz'; nazwisko | numer_pokoju ----------+---------- Kowalski | 101 Jarosz | 105	**Użytkownikowi „zawiesił się system”, musi czekać na dokończenie transakcji.**
UPDATE pracownicy SET numer_pokoju=101 WHERE nazwisko='Jarosz'; UPDATE pracownicy SET numer_pokoju=105 WHERE nazwisko='Kowalski'; COMMIT;	nazwisko | numer_pokoju ----------+---------- Jarosz | 101 Kowalski | 105

Przećwicz transakcje na własnych tabelach.

7. Funkcje definiowane przez użytkownika

Język PostgreSQL ma wiele gotowych, wbudowanych funkcji. Na przykład chętnie korzystaliśmy z gotowych funkcji agregujących `count, sum, avg, min, max` (funkcja agregująca to taka funkcja, która zwraca jeden wynik dla wielu wierszy). Oprócz używania gotowych funkcji, użytkownik może pisać własne funkcje na trzy sposoby: w języku SQL, w języku proceduralnym, w języku programowania C. Listę dostępnych języków otrzymamy dając polecenie

```
SELECT*FROM pg_language;
```

```
lanname  | lanowner | lanispl | lanpltrusted | lanplcallfoid | lanvalidator | lanacl
----------+----------+---------+--------------+---------------+--------------+--------
 internal |       10 | f       | f            |             0 |         2246 |
 c        |       10 | f       | f            |             0 |         2247 |
 sql      |       10 | f       | t            |             0 |         2248 |
 plpgsql  |    16392 | t       | t            |         16403 |        16404 |
```

Na liście jest oczywiście język SQL. Użytkownik definiuje własną funkcję w języku SQL poleceniem

```
CREATE [ OR REPLACE ] FUNCTION
name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
[RETURNS retype  | RETURNS TABLE ( colname coltype [, ...] ) ]
{LANGUAGE langname
 AS 'definition'   }
```

Pełna składnia tego polecenia, patrz `\h CREATE FUNCTION`.

Rzadko kiedy uda się nam od razu napisać funkcję w postaci końcowej. Dlatego najlepiej jest dać od samego początku `CREATE OR REPLACE FUNCTION`. Ciało funkcji SQL (`'definition'` lub `$$definition$$` lub `$body$definition$body$`, SQL dopuszcza dolarowe apostrofowanie) jest listą dowolnych poleceń SQL (oprócz `BEGIN, COMMIT, ROLLBACK, SAVEPOINT`). Funkcja zwraca wynik ostatniego polecenia z listy poleceń. Ostatnie polecenie musi być poleceniem `SELECT` lub poleceniem `INSERT, UPDATE, DELETE` zawierającym warunek `RETURNING`, po którym występuje zwracany wynik (wynik musi być takiego typu jaki zdefiniowaliśmy w `RETURNS`). Nie musi jednak tak być w przypadku, gdy funkcja (w istocie procedura) ma przeprowadzić akcję bez zwracania wyniku i jest zdefiniowana jako `RETURNS void` (czyli zwracająca nic). W ciele funkcji są dostępne wszystkie tabele bazy danych.

Napiszemy funkcję obliczającą pole trapezu z użyciem języka SQL

```
CREATE OR REPLACE FUNCTION pole_trapezu(a real,b real,h real)
RETURNS double precision AS
     'SELECT ($1+$2)*$3/2;'
LANGUAGE SQL;
```

Kolejne argumenty wejściowe funkcji `a, b, h,` to w ciele funkcji kolejno `$1, $2, $3,` itd. (nazwy argumentów `a, b, h` mają tylko informacyjne, dokumentacyjne znaczenie, można ich nie podawać, najważniejszy jest typ argumentu, dwie funkcje o takiej samej nazwie różnią się typami argumentów, a nie nazwami argumentów).

Daj `\df` , `\df+` aby zobaczyć, że nowa funkcja znalazła się na liście funkcji. Obliczymy pole trapezu

```
SELECT pole_trapezu(2,5,3);
 pole_trapezu
---------------
         10.5
```

Pisząc dłuższą funkcję można ewentualnie posłużyć się edytorem tekstu. Na przykład powyższą funkcję można było napisać w `mc` w pliku `pole_trapezu.sql` i załadować w `psql` poleceniem

```
\i pole_trapezu.sql
```

Napiszemy funkcję obliczającą obciążenie stropu w pokoju o danym numerze. Obciążenie stropu, to ciężar pracowników danego pokoju plus ciężar rzeczy w danym pokoju

```
CREATE OR REPLACE FUNCTION obciazenie_stropu(numer_pokoju int)
RETURNS real
LANGUAGE SQL
AS
 '(SELECT(
  (SELECT SUM(ciezar_pracownika) FROM pracownicy
   WHERE numer_pokoju=$1)+
  (SELECT SUM(ciezar_rzeczy) FROM pracownicy, rzeczy
   WHERE (numer_pokoju=$1)AND(nawisko=na_stanie_pracownika))
   );'
;
```

Dając `\df` widzimy, że nowa funkcja znalazła się na liście funkcji. Posłużymy się tą funkcją do obliczenia obciążenia stropu w pokoju nr 103.

```
SELECT 'Pokój nr 103 ma obciążenie stropu ',
       (SELECT obciazenie_stropu(103)), ' kg';
```

```
        ?column?                 | ?column? | ?column?
-----------------------------------------+----------+----------
 Pokój nr 103 ma obciążenie stropu |   494.7 | kg
(1 row)
```

Funkcja może zwracać tabelę, `RETURNS TABLE`. Na przykład napiszemy funkcję wyszukującą pracowników o ciężarze powyżej zadanej, progowej liczby kilogramów

```
CREATE OR REPLACE FUNCTION
pracownicy_o_ciezarze_powyzej(ciezar_progowy real)
RETURNS TABLE(nazwisko text,
              numer_pokoju int,
              ciezar_pracownika real)
AS '
  SELECT nazwisko, numer_pokoju, ciezar_pracownika
  FROM pracownicy
  WHERE ciezar_pracownika>$1
  ORDER BY ciezar_pracownika DESC;
'
LANGUAGE sql;
```

Sprawdzamy, czy nowa funkcja znalazła się na liście funkcji (`\df`). Sprawdzamy działanie funkcji

```
SELECT pracownicy_o_ciezarze_powyzej(90);

pracownicy_o_ciezarze_powyzej
------------------------------
 (Nowy,104,99)
 (Krol,103,98)
 (Borek,104,97)
 (Dolasa,108,91.4)
(4 rows)
```

Napisz dowolną, własną funkcję w języku SQL.

Gdybyśmy chcieli usunąć funkcję, to zrobilibyśmy to poleceniem `DROP FUNCTION` (pełna składnia tego polecenia, patrz `\h DROP FUNCTION`).

Szczególnego rodzaju funkcją jest polecenie `PREPARE` o następującej składni (`\h PREPARE`)

```
PREPARE name [( datatype [, ...])]  AS statement
```

Polecenie o wybranej przez nas nazwie powoduje, że polecenie (ang. statement) podlega rozbiorowi, przepisaniu, planowaniu, ale nie zostaje wykonane. Raz przygotowane polecenie może być teraz wielokrotnie wykonywane w krótkim czasie poleceniem `EXECUTE`. W poleceniu `PREPARE` można zadać parametry określonego typu, które będą podstawiane do wykonywanego polecenia jako $1, $2, itd. Polecenie `PREPARE` likwidujemy poleceniem `DEALLOCATE`, chociaż zostanie ono zlikwidowane automatycznie na zakończenie sesji.

Przykładowo napiszemy polecenie `PREPARE` o najprostszej postaci bez parametrów

```
PREPARE wykonaj AS SELECT*FROM pracownicy ORDER BY nazwisko;
```

Teraz aż do końca sesji wykonanie podanego polecenia SELECT sprowadzi się do napisania polecenia

```
EXECUTE wykonaj;
```

Opracuj i sprawdź działanie własnego polecenia `PREPARE`.

Język SQL jest językiem zapytań i nie jest zbyt wygodny do pisania funkcji. Do tego celu lepiej nadaje się język C, a najlepiej specjalne języki proceduralne, jak na przykład język proceduralny PL/pgSQL (celowo opracowany dla ułatwienia pisania funkcji w PostgreSQL).

PL/pgSQL jest językiem o strukturze blokowej, bloki mogą być zagłębione, na początku każdego bloku można deklarować zmienne lub stałe, w ciele funkcji jest dopuszczalne używanie nazw argumentów wejściowych i wyjściowych funkcji, w wyrazach kluczowych nie są rozróżniane małe i duże litery, nazwy są niejawnie zamieniane na pisane małymi literami. Napiszemy funkcję rozwiązującą równanie kwadratowe

```
CREATE OR REPLACE FUNCTION
r_kwadratowe(a real, b real, c real, OUT x1 real, OUT x2 real)
AS
$body$
DECLARE
  delta real;
  sqrtdelta real;
BEGIN
  IF a=0 THEN RAISE EXCEPTION
  'Istnieje tylko jeden pierwiastek x=-c/b, gdyż a=0';
  END IF;
  delta:=b*b-4*a*c;
  IF delta<0 THEN RAISE EXCEPTION
  'Nie ma rzeczywistych pierwiastków, ujemna delta =%',delta;
  END IF;
  sqrtdelta:=SQRT(delta);
  x1:=(-b-sqrtdelta)/2/a;
  x2:=(-b+sqrtdelta)/2/a;
END;
$body$
LANGUAGE plpgsql;
```

Argumenty wyjściowe koniecznie należało oznaczyć jako mające mod OUT. Opcjonalnie dla argumentów wejściowych można było podać mod IN. Zadeklarowano dwie zmienne rzeczywiste `delta` i `sqrtdelta`. Zamiast zwykłego apostrofowania `AS '...'` zastosowano apostrofowanie dolarowe `AS $$...$$` dopisując dodatkowo etykietę (ang. tag) między znakami dolarów wyjaśniającą co znajduje się między apostrofami `AS $body$...$body$`. Etykiety są tak jak łańcuchy czułe na małe/duże litery (nie można było napisać na początku `body` i drugi raz np. `Body`). Dzięki dolarowemu apostrofowaniu, apostrofy wewnątrz ciała funkcji, czy ukośniki nie muszą być podwajane, np. apostrof na początku i końcu łańcucha nie musi być teraz podwójny `''`.

Sprawdzamy dając `\df`, czy funkcja znalazła się na liście funkcji. Sprawdzamy, czy uzyskamy poprawne rozwiązanie dla równania kwadratowego $x^2 - 5x + 6 = 0$

```
SELECT r_kwadratowe(1,-5,6);
```

```
r_kwadratowe
--------------
 (2,3)
(1 row)
```

Napisz dowolną własną funkcję w języku proceduralnym PL/pgSQL.

8. Wyzwalacze

W języku SQL polecenia są wykonywane w narzuconym przez użytkownika porządku. Jeżeli jednak polecenie ma być wykonane nie w z góry przewidzianym porządku, ale jako reakcja na pewne wydarzenie, to należy użyć wyzwalacza (ang. trigger). Wyzwalacz zakładamy poleceniem o następującej składni (`\h CREATE TRIGGER`)

```
CREATE TRIGGER name {BEFORE | AFTER} {event [OR…]}
     ON table [FOR[EACH] {ROW | STATEMENT}]
     EXECUTE PROCEDURE funcname(arguments)
```

Wyzwalacz o nazwie `name` zakładamy dla tabeli o nazwie `table`. Wyzwalacz reaguje tuż przed (`BEFORE`) lub tuż po (`AFTER`) wydarzeniu (`event = INSERT, UPDATE, DELETE, TRUNCATE`), które zaszło dla danej tabeli. Reakcja następuje opcjonalnie raz na całe wydarzenie, polecenie (opcja `STATEMENT`, tzw. wyzwalacz poleceniowy, jest to opcja domyślna) lub raz na wiersz (wydarzenie może dotyczyć wielu wierszy, opcja `ROW`, tzw. wyzwalacz wierszowy). Wydarzenie `TRUNCATE` może wystąpić wyłącznie w wyzwalaczu poleceniowym (nie wolno go użyć w wyzwalaczu wierszowym). Reakcja wyzwalacza, to wykonanie poleceń zapisanych w funkcji wyzwalacza `funcname`. Funkcja wyzwalacza jest wykonywana (wyzwalana) tuż przed wydarzeniem (opcja `BEFORE`) lub tuż po wydarzeniu (opcja `AFTER`). Jeżeli zdefiniowano więcej niż jeden wyzwalacz dla tego samego wydarzenia, dla tej samej tabeli, to wyzwalacze zadziałają w kolejności alfabetycznej swoich nazw.

Z powyższego wynika, że w języku PostgreSQL zanim założymy wyzwalacz, musimy najpierw zdefiniować, napisać funkcję wyzwalacza `funcname` (jest to inaczej niż w standardzie języka SQL-99, gdzie reakcja wyzwalacza jest opisana wewnątrz definicji wyzwalacza). Funkcja wyzwalacza musi być zdefiniowana jako funkcja bez argumentów i zwracająca typ `trigger` . Nie można napisać funkcji wyzwalacza w opcji `LANGUAGE SQL`, gdyż w tym języku jak do tej pory funkcja nie może zwracać typu `trigger`. Napisanie funkcji wyzwalacza dopuszcza większość języków proceduralnych, np. PL/pgSQL, PL/Tcl (Tool Command Language), PL/Perl, PL/Python, C. Funkcja wyzwalacza nie ma argumentów wejściowych jak zwykła funkcja, gdyż dla niej wejściem są dane o specjalnej strukturze `TriggerData`.

Funkcją wyzwalacza można posługiwać się tylko w wyzwalaczach. Jedna funkcja wyzwalacza może być użyta w kilku wyzwalaczach. Funkcja wyzwalacza wykonywana raz na polecenie musi zwracać NULL. Funkcja wyzwalacza wykonywana raz na wiersz `BEFORE` powinna zwracać wartość rekordu/wiersza mającą dokładnie taką strukturę jak struktura tabeli, na którą natknął się wyzwalacz i będzie to nowa wartość wiersza po `INSERT`, `UPDATE`; gdy funkcja zwróci NULL, to dla danego wiersza operacja nie zostanie wykonana (tzw. *wetowanie polecenia*). Zwracana przez funkcję wartość jest ignorowana w wyzwalaczu wierszowym `AFTER`, dlatego może to być NULL. Jeżeli funkcja wyzwalacza wykonuje polecenie SQL, to może ono być poddane działaniu innych wyzwalaczy, takie „kaskadowanie” wyzwalaczy jest dopuszczalne.

Jeżeli funkcja napisana w języku PL/pgSQL jest funkcją wyzwalacza, to przy wywołaniu tej funkcji są automatycznie tworzone specjalne zmienne. Wybrane z nich to:

NEW
Dane typu RECORD. Jest to zmienna przechowująca *nowy* wiersz/rekord bazy danych dla poleceń INSERT, UPDATE w wyzwalaczu wierszowym. Ta zmienna ma wartość NULL dla polecenia DELETE w wyzwalaczu poleceniowym.

OLD
Dane typu RECORD. Jest to zmienna przechowująca *stary* wiersz bazy danych dla poleceń UPDATE, DELETE w wyzwalaczu wierszowym. Ta zmienna ma wartość NULL dla polecenia INSERT w wyzwalaczu poleceniowym.

TG_NAME
Dana typu name, nazwa aktualnego wyzwalacza.

TG_WHEN
Dana typu text, łańcuch BEFORE lub AFTER w zależności od definicji aktualnego wyzwalacza.

TG_LEVEL
Dana typu text, łańcuch ROW lub STATEMENT w zależności od tego czy aktualny wyzwalacz jest wierszowy, czy poleceniowy.

TG_OP
Dana typu text, łańcuch o treści INSERT, UPDATE, DELETE, TRUNCTATE w zależności od tego na jaką operację, polecenie natknął się wyzwalacz.

TG_TABLE_NAME
Dana typu name, nazwa tabeli, na którą powołuje się wyzwalacz.

Napiszemy funkcję wyzwalacza sygnalizującą wykonywanie poleceń (uwaga, pisz w łańcuchach dużymi literami 'INSERT', 'UPDATE' , 'DELETE')

```
CREATE OR REPLACE FUNCTION funkcja_wyzwalacza( )
RETURNS trigger AS $$
BEGIN
   IF (TG_OP='INSERT') THEN
      RAISE NOTICE 'Tabela % % % dnia %',
      TG_TABLE_NAME, TG_WHEN, TG_OP, timeofday();
      RETURN NEW;
   ELSIF (TG_OP='UPDATE') THEN
      RAISE NOTICE 'Tabela % % % dnia %',
      TG_TABLE_NAME, TG_WHEN, TG_OP, timeofday();
      RETURN NEW;
   ELSIF (TG_OP='DELETE') THEN
      RAISE NOTICE 'Tabela % % % dnia %',
      TG_TABLE_NAME, TG_WHEN, TG_OP, timeofday();
      RETURN OLD;
   END IF;
END;
$$ LANGUAGE plpgsql;
```

Dajemy `\df` i widzimy, że na liście pojawiła się nasza nowa funkcja.

Zakładamy wyzwalacz wierszowy o nazwie `trigbefore` sygnalizujący zamiar wykonania (`BEFORE`) poleceń `INSERT, UPDATE, DELETE` na tabeli `pracownicy`

```
CREATE TRIGGER trigbefore
BEFORE INSERT OR UPDATE OR DELETE ON pracownicy
FOR EACH ROW EXECUTE PROCEDURE funkcja_wyzwalacza( );
```

Dla tej samej tabeli załóżmy jeszcze jeden wyzwalacz wierszowy o nazwie `trigafter` sygnalizujący zakończenie wykonania (`AFTER`) poleceń `INSERT, UPDATE, DELETE`

```
CREATE TRIGGER trigafter
AFTER INSERT OR UPDATE OR DELETE ON pracownicy
FOR EACH ROW EXECUTE PROCEDURE funkcja_wyzwalacza( );
```

Dajemy `\d+ pracownicy` i widzimy, że znajdują się tam nasze wyzwalacze. Sprawdzimy jak zadziałają wyzwalacze. Wpisujemy nowy rekord do tabeli `pracownicy`

```
INSERT INTO pracownicy VALUES('Borowski',105,98.5);
```

```
NOTICE:  Tabela pracownicy BEFORE INSERT dnia Tue Jul 13 12:57:38.746250 2010 CEST
NOTICE:  Tabela pracownicy  AFTER INSERT  dnia Tue Jul 13 12:57:38.746476 2010 CEST
```

Polecenie `INSERT` to było jedno z wydarzeń, na które reagują dwa wyzwalacze. Automatycznie została uruchomiona `funkcja_wyzwalacza()`, raz przed i raz po wydarzeniu. Mamy komunikaty podające dla tabeli `pracownicy` datę i czas przed i po wydarzeniu.

Wyzwalacz w fazie `BEFORE` wywołuje dwa dodatkowe efekty:

a) Jeżeli funkcja wyzwalacza zwróci `NULL` dla poleceń `INSERT, UPDATE, DELETE`, to dane polecenie zostaje anulowane i dalsze wyzwalacze nie zostaną wywołane. Oznacza to, że wyzwalacz *wetuje polecenie*.
b) W przypadku poleceń `INSERT` i `UPDATE`, funkcja wyzwalacza może zwrócić wartość rekordu i to właśnie ta wartość (a nie wynikająca z poleceń `INSERT`, `UPDATE`) znajdzie się w bazie danych.

Wyzwalacz usuwamy poleceniem `DROP TRIGGER name ON table` (pełna składnia polecenia, patrz `\h DROP TRIGGER`).

Opracuj wyzwalacz dla własnej tabeli.

9. Indeksy

Znaczenie indeksu w tabeli bazy danych jest takie samo jak znaczenie indeksu w zwykłej książce, służy do przyśpieszenia odnajdowania wybranej treści. Jeżeli czytelnik jest zainteresowany tylko jednym wybranym zagadnieniem, to nie musi kartkować strona po stronie całej książki, aby odnaleźć strony z tym zagadnieniem. Wystarczy, że czytelnik

zaglądnie do alfabetycznego indeksu (skorowidza) zamieszczonego na końcu książki i dzięki niemu szybko odnajdzie numery stron z interesującym go zagadnieniem. Podobnie indeks założony dla wybranych kolumn danej tabeli przyśpieszy odnajdowanie wierszy.

Indeksowanie jest skuteczne dzięki stosowaniu specjalnych metod zapisu danych na dysku (dane są zapisywane nie w przypadkowy sposób, ale są posortowane). W PostgreSQL mamy do wyboru (USING *method*) cztery metody: B-tree, Hash, GiST, GIN. Domyślnie jest tu stosowana metoda B-tree jako skuteczna w przypadku zapytań z warunkami w postaci porównań (nierówności i równości). W algorytmie poszukiwania liczba porównań zmaleje tutaj logarytmicznie. Na przykład w tabeli z milionem wierszy wyszukanie wiersza będzie wymagało nie miliona porównań, ale wystarczy około dwadzieścia porównań $\log_2 1000000 = 19.3...$.

PostgreSQL automatycznie zakłada indeks dla tabel z kolumnami z ograniczeniem `UNIQUE` lub z kluczem głównym `PRIMARY KEY`. Na przykład tabela `pracownicy` ma klucz główny (kolumna `nazwisko`) i dając `\d pracownicy` widzimy, że został automatycznie założony indeks o nazwie `pracownicy_pkey` na kolumnie `nazwisko` według metody B-tree

Indexes:
"pracownicy_pkey" PRIMARY KEY, btree (nazwisko)

Jeżeli przewidujemy, że będziemy bardzo często używali poleceń z poszukiwaniem według numerów pokojów, to warto założyć indeks o nazwie `indeks_pokojow` na kolumnie `numer_pokoju` dla tabeli `pracownicy` (pełna składnia tego polecenia, patrz `\h CREATE INDEX`)

```
CREATE INDEX indeks_pokojow ON pracownicy(numer_pokoju);
```

Dając `\d pracownicy` widzimy, że powyższy indeks został założony. Teraz system będzie poświęcał nieco czasu na automatyczne uaktualnienie indeksu po każdej modyfikacji tabeli. Ta strata czasu powinna zwrócić się z nawiązką przy częstym korzystaniu z indeksu przy odczytywaniu danych z tabeli.

Planowany czas wykonania polecenia sprawdzamy poleceniem `EXPLAIN` (pełna składnia polecenia, patrz `\h EXPLAIN`), na przykład plan wykonania polecenia `SELECT` (`EXPLAIN` nie wykona polecenia `SELECT`, a tylko poda plan wykonania tego polecenia)

```
EXPLAIN SELECT*FROM pracownicy WHERE numer_pokoju>100 ORDER BY
numer_pokoju;

                    QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1.44..1.47 rows=12 width=40)
   Sort Key: numer_pokoju
   ->  Seq Scan on pracownicy  (cost=0.00..1.23 rows=12 width=40)
         Filter: (numer_pokoju > 100)
(4 rows)
```

W nawiasach są podawane cztery wartości:
- Estymowany koszt startu (tj. czas sortowania przed rozpoczęciem skanowania tabeli)
- Estymowany całkowity koszt
- Estymowana liczba wierszy wyjściowych w wyniku zapytania
- Estymowana średnia szerokość wierszy wyjściowych (w bajtach)

Koszt jest mierzony względem jednostkowego czasu odczytu jednej strony dysku seq_page_cost=1.0 . Koszt odczytu jednego wiersza jest estymowany jako sto razy mniejszy cpu_tuple_cost=0.01 . Nasza tabela ma 18 wierszy i mieści się na jednej stronie dysku

```
SELECT relpages,reltuples FROM pg_class WHERE relname='pracownicy';
```

```
 relpages | reltuples
----------+-----------
        1 |        18
(1 row)
```

Polecenie `ANALYZE` (pełna składnia polecenia, patrz `\h ANALYZE`) pozwala poznać prawdziwy czas wykonania polecenia. Połączymy polecenie `EXPLAIN` z poleceniem `ANALYZE` (`EXPLAIN` poda wyniki estymacji bez wykonania polecenia `SELECT`, `ANALYZE` wykona polecenie `SELECT` i poda prawdziwe czasy)

```
EXPLAIN ANALYZE SELECT*FROM pracownicy WHERE numer_pokoju>100
ORDER BY numer_pokoju;
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Sort  (cost=1.44..1.47 rows=12 width=40) (actual time=0.029..0.030 rows=13 loops=1)
   Sort Key: numer_pokoju
   Sort Method:  quicksort  Memory: 25kB
   ->  Seq Scan on pracownicy  (cost=0.00..1.23 rows=12 width=40) (actual time=0.008..0.015
rows=13 loops=1)
         Filter: (numer_pokoju > 100)
 Total runtime: 0.056 ms
(6 rows)
```

Indeksowanie opłaci się dla dużych tabel o tysiącach i milionach wierszy, tak jak to jest w praktyce (na przykład użytkownicy wyszukiwarek internetowych są bardzo mile zaskoczeni szybkością ich działania). Przy liczbie wierszy mniejszej niż 100 tabela prawdopodobnie zmieści się na jednej stronie dysku i nie należy oczekiwać korzyści z indeksowania (tak jak nie odniesiemy korzyści z indeksowania w „książce”, która ma tylko jedną stronę). Należy ocenić samemu, czy bardziej opłaca się stosować indeks (szybciej wykonywane polecenia `SELECT`), czy lepiej go usunąć (zbyt czasochłonne uaktualnianie indeksu w porównaniu z częstością używania polecenia `SELECT`).

Zlikwidujemy indeks poniższym poleceniem (pełna składnia polecenia, patrz `\h DROP INDEX`)

```
DROP INDEX indeks_pokojow ;
```

Dając `\d pracownicy` widzimy, że nie ma już indeksu przy tabeli `pracownicy`.

Teraz już bez indeksu mamy

```
EXPLAIN ANALYZE SELECT*FROM pracownicy WHERE numer_pokoju>100
ORDER BY numer_pokoju;
```

```
                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Sort  (cost=1.44..1.47 rows=12 width=40) (actual time=0.032..0.035 rows=13 loops=1)
   Sort Key: numer_pokoju
   Sort Method:  quicksort  Memory: 25kB
   ->  Seq Scan on pracownicy  (cost=0.00..1.23 rows=12 width=40) (actual time=0.012..0.018
rows=13 loops=1)
         Filter: (numer_pokoju > 100)
 Total runtime: 0.060 ms
(6 rows)
```

Jak widzimy czas wykonania polecenia nieznacznie wzrósł z 0.056 ms do 0.060 ms, ale system nie traci teraz czasu na porządkowanie indeksu.

Załóż, przetestuj i zlikwiduj indeks dla własnej tabeli.

10. Kursory

Kursor jest wskaźnikiem przesuwającym się po wierszach uzyskanych w wyniku zapytania. Na przykład, jeżeli w wyniku zapytania otrzymaliśmy bardzo dużą liczbę wierszy, trudną do jednorazowego prześledzenia, to dzięki kursorowi możemy przeglądać wyniki zapytania nie w całości, ale po kilka wierszy w jednym czasie. Kursor jest elementem zanurzonego języka SQL i w PostgreSQL używamy go wyłącznie w bloku transakcji lub w bloku funkcji napisanej w języku PL/pgSQL. Kursor zakładamy dla planowanego zapytania poleceniem o składni, z którą zapoznamy się wspomagając się pomocą `\h DECLARE`

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

Przećwiczymy posługiwanie się kursorem na przykładzie transakcji. Rozpoczynamy transakcję

```
BEGIN;
```

Deklarujemy `kursor1` dla polecenia `SELECT*FROM pracownicy`

```
DECLARE kursor1 CURSOR FOR SELECT*FROM pracownicy;
```

Widzimy, że istotnie ten kursor został zadeklarowany

```
SELECT*FROM pg_cursors;
```

```
 name    |                    statement                    | is_holdable | is_binary | is_scrollable |
creation_time
---------+-------------------------------------------------+-------------+-----------+---------------+-----
-------------------------
 kursor1 | DECLARE kursor1 CURSOR FOR SELECT*FROM pracownicy; | f     | f    | t     |
2010-07-15 15:25:20.555664+02
(1 row)
```

Teraz możemy przeglądać wybrane wiersze zapytania (pełna składnia stosownego polecenia, patrz `\h FETCH`), na przykład w przód po trzy wiersze

```
FETCH FORWARD 3 FROM kursor1;
```

```
nazwisko | numer_pokoju | ciezar_pracownika
----------+-----------------+------------------
 Dragan   |          108 |          63.5
 Lenart   |          101 |          75.3
 Szuba    |           24 |          72.6
(3 rows)
```

Kursor możemy przesunąć bez przeglądania danych (pełna składnia stosownego polecenia, patrz `\h MOVE`), na przykład w przód o cztery wiersze

```
MOVE FORWARD 4 IN kursor1;
```

Teraz przeglądanie rozpoczyna się od nowej pozycji kursora

```
FETCH BACKWARD 3 FROM kursor1;
```

```
nazwisko | numer_pokoju | ciezar_pracownika
----------+-----------------+------------------
 Dolasa   |          108 |          91.4
 Wolski   |           55 |          88.7
 Ekiert   |           24 |          66.2
(3 rows)
```

Niepotrzebny już kursor likwidujemy następującym poleceniem (pełna składnia polecenia, patrz `\h CLOSE`),

```
CLOSE kursor1;
```

Dając

```
SELECT*FROM pg_cursors;
```

```
 name | statement | is_holdable | is_binary | is_scrollable | creation_time
-------+-----------+-------------+-----------+---------------+---------------
(0 rows)
```

widzimy, że nie ma już kursora o nazwie `kursor1`. Zamykamy transakcję

```
ROLLBACK;
```

Powtórz powyższe czynności na przykładzie własnego kursora.

W podobny sposób można używać kursorów w bloku funkcji posługując się językiem PL/pgSQL, ale nie będziemy się tym teraz zajmowali.

Na zakończenie zajęć daj `\q` (wyjście z `psql`) , a następnie `exit` (zakończenie połączenia z bazą danych). Nigdy nie wychodź z pokoju pozostawiając otwartą bazę danych.

Laboratorium Baz Danych, PostgreSQL

Ćwiczenie 3. Aplikacja klienta w języku C

Program PostgreSQL jest rozprowadzany z dwoma interfejsami użytkownika: `libpq` (jest to pierwotny interfejs w języku C będący podstawą dla wielu innych interfejsów, jak ecpg, C++, Perl, Python, Tcl), `ecpg` (ang. embedded SQL in C). Są one opisane w dokumentacji http://www.postgresql.org/docs/, rozdział IV, Client Interfaces. Istnieje też wiele innych interfejsów programowania dla programów klienta (patrz dokumentacja, dodatek G.1. Client Interfaces, Table G-1, Externally Maintained Client Interfaces). W tym ćwiczeniu laboratoryjnym zajmiemy się aplikacjami klienta pisanymi z użyciem interfejsu użytkownika `libpq`. Interfejs użytkownika `libpq`, to biblioteka funkcji pozwalających programom klienta przekazywać zapytania i uzyskiwać odpowiedzi z serwera PostgreSQL (w języku C nie można bezpośrednio pisać poleceń SQL, nie jest to język 4GL). Programy klienta używające interfejsu `libpq` muszą mieć podaną w nagłówku ścieżkę dostępu do biblioteki `libpq-fe.h` i muszą być linkowane z biblioteką `libpq.so`. Przerabiane zagadnienia są dodatkowo opisane w następującej dokumentacji: http://www.redhat.com/docs/manuals/database/ .

1. Kompilowanie prostego programu

Połącz się z serwerem tak jak w poprzednich ćwiczeniach laboratoryjnych (dostęp do komputera klienta jako użytkownik: **`baza`**, hasło: **`danych`**; dostęp do serwera IP 153.19.50.110 poprzez program `putty`, podaj swój login i hasło).

Rozpoczniemy od napisania prostego programu. Będziemy wspomagali się interfejsem Midnight Commander

```
$ mc
```

Użyjemy jego edytora wewnętrznego, sprawdź czy masz ustawioną odpowiednią opcję: Opcje → Konfiguracje → [x] Wewnętrzny edytor. Można posługiwać się dwoma oknami otwartymi poprzez dwukrotne użycie programu `putty`; w jednym oknie dokonujemy edycji, a w drugim oknie wykonujemy obliczenia.

Zakładamy plik o nazwie `program.c`

```
$ touch program.c
```

Równie dobrze mogliśmy założyć ten plik w `mc` dając Shift+F4; tworzy się pusty plik początkowo bez nazwy, któremu nadamy nazwę przy `save`.

Plik `program.c` poddajemy edycji (F4 w `mc`) i niech jego zawartość będzie następująca:

```
#include<stdio.h>
int main(void)
{
```

```
  printf("\n Witaj światku \n");
  return(0);
}
```

Nasz program zawiera (tak jak każdy inny program napisany w języku C) funkcję główną `main( )`. Funkcja nie ma argumentów wejściowych (`void` – nic). Funkcja ma zwracać wartość typu `int`, dlatego daliśmy `return(0)`. Dołączyliśmy w nagłówku bibliotekę `stdio.h` standardowych przyrządów wejścia, wyjścia. Zawiera ona między innymi użytą przez nas funkcję `printf( )`.

Program napisany w języku C należy przetworzyć na program wykonywalny w komputerze, należy go skompilować. Kompilowanie przebiega w czterech etapach: uruchomienie preprocesora dla każdego modułu kodu źródłowego, generowanie kodu assemblera, tłumaczenie z kodu assemblera na kod maszynowy, linkowanie modułów programu w postaci kodu maszynowego i bibliotek. Użyjemy do tego celu kompilatora `gcc` (jest to skrót z ang. GNU Compiler Collection). Bez dodatkowych opcji zostaną wykonane automatycznie cztery etapy kompilacji. Opcja `-o nazwa` pozwala nadać nazwę programu skompilowanego (bez tej opcji program skompilowany miałby domyślnie nazwę `a.out`)

```
$ gcc -o program program.c
```

Po poprawnym kompilowaniu pojawia się w naszym katalogu skompilowany program o nadanej przez nas nazwie `*program`. Uruchamiamy ten program

```
$ ./program
```

Pojawił się napis „Witaj świątku”, czy co tam jeszcze napisaliśmy w `program.c`.

2. Kompilowanie programu zapisanego w kilku plikach

Pisanie długiego, skomplikowanego programu w języku C najlepiej jest rozłożyć na drobniejsze składowe w postaci funkcji pisanych w oddzielnych plikach i kompilować połączone pliki poleceniem `make`. Na przykład program `prog123.c` zawierający część główną i dwie funkcje najlepiej jest rozbić na trzy pliki, plik `prog1.c` definiujący funkcję główną:

```
#include<stdio.h>
int main(void)
{
  printf("\n Witaj świątku \n");
  fun1();
  fun2();
  printf(”\n Koniec programu prog1 z fun1 i fun2 \n”);
  return(0);
}
```

plik `prog2.c` definiujący `fun1( )`:

```
#include<stdio.h>
void fun1(void)
{
  printf("\n To był program z fun1( ) \n");
}
```

i plik `prog3.c` definiujący `fun2( )`:

```
#include<stdio.h>
void fun2(void)
{
  printf("\n To był program z fun2( ) \n");
}
```

Każdorazowa kompilacja wymaga napisania czterech wierszy z plikami obiektowymi z rozszerzeniem `.o` (opcja `-c` oznacza zatrzymanie kompilacji po assemblacji bez linkowania):

```
$ gcc -c prog1.c -o prog1.o
$ gcc -c prog2.c -o prog2.o
$ gcc -c prog3.c -o prog3.o
$ gcc -o prog123 prog1.o prog2.o prog3.o
```

Jeżeli nie było błędów, to w katalogu pojawił się skompilowany program o nadanej przez nas nazwie `*prog123`, który uruchamiamy poleceniem

```
$ ./prog123
```

Pisane przez nas programy często modyfikujemy. Jeżeli ulegnie modyfikacji chociaż jedna część programu, to trzeba od nowa kompilować całość pisząc od nowa powyższe cztery linijki poleceń. Jest to bardzo uciążliwe w praktyce. O wiele wygodniej jest napisać plik wsadowy o zwyczajowej nazwie `makefile` (lub `Makefile` lub `MakeFile`), który zawiera zadania, każde o następującej składni

```
zadanie1: składniki zadania
<TAB>polecenie1
<TAB>polecenie2
      .
      .
      .
```

i dokonać kompilacji jednym krótkim poleceniem `make`. Przed `zadanie1` nie może być spacji. Dla zwiększenia czytelności programu, zamiast jednego razu można dać dwa razy tabulator `<TAB><TAB>`. W razie potrzeby piszemy komentarz po znaku #. Na przykład skompilowanie jednego programu zawartego w trzech powyższych plikach wymaga napisania następującego pliku o nazwie `makefile`, zawierającego cztery zadania:

```
prog123: prog1.o prog2.o prog3.o
          gcc -o prog123 prog1.o  prog2.o prog3.o
prog1.o: prog1.c
          gcc -c prog1.c -o prog1.o
```

```
prog2.o: prog2.c
          gcc -c prog2.c -o prog2.o
prog3.o: prog3.c
          gcc -c prog3.c -o prog3.o
```

Dajemy teraz polecenie

```
$ make
```

i otrzymujemy skompilowany program (jest to `*prog123` w naszym katalogu), który uruchamiamy następująco

```
$ ./prog123
```

Teraz po jakiejkolwiek zmianie w `prog1.c`, `prog2.c` lub `prog3.c`, dokonanie nowej kompilacji sprowadzi się do wydania polecenia `make` (treść `makefile` nie zmienia się). Ten wsadowy sposób kompilowania jest tak użyteczny, że będziemy go używali także w przypadku kompilowania programów mieszczących się w jednym pliku.

3. Program ustanawiający połączenie z bazą danych

Zbiór funkcji (procedur) służących do połączenia z bazą danych, formułowania zapytań, graficznego przedstawienia wyników zapytania jest bardzo obszerny (zwłaszcza ich opis) i znajdziemy go w „PostgreSQL 8.4.3 Documentation” , czy „Red Hat Database Programmers Guide”. Nazwy tych funkcji zestawiono w dodatku 3.1, a opisy wybranych funkcji podano w dodatku 3.2. Nazwa funkcji rozpoczyna się od PQ, a nazwa zwracanej wielkości rozpoczyna się od PG.

Załóż plik `p1.c` i napisz następujący program ustanawiający połączenie z bazą danych (wpisz własną nazwę dla `dbname` i `user`):

```
#include<stdio.h>
#include</usr/include/postgresql/libpq-fe.h>

int main(void)
{
  PGconn *conn;
  conn=PQconnectdb("dbname=student123 user=student123");

  if (PQstatus(conn)==CONNECTION_OK)
  {
    printf("\n Mamy połączenie z bazą danych \n");
    /* Teraz można byłoby posłużyć się bazą danych,
     * ale pozostawmy to do następnego przykładu.
     * Tutaj poprosimy tylko o potwierdzenie nazw
     * bazy danych i użytkownika. */
    printf(”\n Nazwa bazy danych: %s\n”,PQdb(conn));
    printf(”\n Nazwa użytkownika: %s\n”,PQuser(conn));
  }
```

```
  else
  {
    printf("\n Nie uzyskano połączenia z bazą danych:\n");
    printf("%s\n",PQerrorMessage(conn));
  }
  printf("\n Koniec programu p1 \n");
  PQfinish(conn);
  return(0);
}
```

Każdy program C współpracujący z PostgreSQL musi mieć w nagłówku dołączoną bibliotekę `libpq-fe.h`. Jak widzimy w powyższym programie podano ścieżkę dostępu do tej biblioteki (sprawdź używając `mc`, że ta biblioteka znajduje się w podanym katalogu). Zażądamy, aby podczas linkowania została dołączona biblioteka `libpq.so`, w skrócie `lpq`, co zapisano w drugiej linijce poniższego programu (opcja `-lpq`). Napisz plik `makefile` o treści:

```
p1:p1.o
          gcc -o p1 p1.o -lpq
p1.o:p1.c
          gcc -c p1.c -o p1.o
```

Skompiluj program `p1.c` poleceniem

```
$ make
```

Skompilowany program pojawił się w naszym katalogu jako `*p1`. Uruchamiamy ten program

```
$ ./p1
```

Powinniśmy otrzymać komunikat, że uzyskaliśmy połączenie z bazą danych, z potwierdzeniem nazw bazy danych i użytkownika.

4. Program dokonujący wpisu wiersza do tabeli w bazie danych

Załóż plik `p2.c` i napisz następujący program (wpisz jak zwykle własną nazwę dla `dbname` i `user`):

```
#include<stdio.h>
#include</usr/include/postgresql/libpq-fe.h>

int main(void)
{
  PGconn   *conn;
  PGresult *res;
  conn=PQconnectdb("dbname=student123 user=student123");
  if (PQstatus(conn)==CONNECTION_OK)
  {
    printf("\n Mamy połączenie z bazą danych \n");
```

```
    res=PQexec(conn,
    "INSERT INTO
pracownicy(nazwisko,numer_pokoju,ciezar_pracownika) VALUES
('Walski',103,89.4)");
    printf("\n Wykonano polecenie INSERT INTO pracownicy \n");
    printf("\n Status polecenia:\n");
    printf("%s \n",PQresStatus(PQresultStatus(res)));
    PQclear(res);
  }
  else
  {
    printf("\n Nie uzyskano połączenia z bazą danych:\n");
    printf("%s\n",PQerrorMessage(conn));
  }
  printf("\n Koniec programu p2 \n");
  PQfinish(conn);
  return(0);
}
```

Uwaga: Łańcuchy ”.....” i komentarze // pisz w jednej linii. Powyżej przejścia do nowej linii w łańcuchach zostały wprowadzone niepotrzebnie, samowolnie przez edytor Word.

Napisz plik `makefile` o treści:

```
p2:p2.o
          gcc -o p2 p2.o -lpq
p2.o:p2.c
          gcc -c p2.c -o p2.o
```

Skompiluj program `p2.c` poleceniem

```
$ make
```

Skompilowany program pojawił się w naszym katalogu jako `*p2`. Uruchamiamy ten program

```
$ ./p2
```

Powinniśmy otrzymać komunikat, że wykonano polecenie `INSERT INTO`. Sprawdzamy z użyciem programu `psql`, że dokonano wpisu do tabeli. Powtórne uruchomienie programu `*p2` nie pozwoli dokonać tego samego wpisu, gdyż nie można wpisać do tabeli dwóch takich samych rekordów.

Powtórz powyższe dla własnej tabeli.

5. Program odczytujący dane z tabeli

Załóż plik `p3.c` i napisz następujący program (wpisz własną nazwę dla `dbname` i `user`):

```
#include<stdio.h>
```

```
#include</usr/include/postgresql/libpq-fe.h>

int main(void)
{
  PGconn   *conn;
  PGresult *res;
  int nrows, ncols, r, c;

  conn=PQconnectdb("dbname=student123 user=student123");
  if (PQstatus(conn)==CONNECTION_OK)
  {
    printf("\n Mamy połączenie z bazą danych \n");
    res=PQexec(conn,"SELECT*FROM pracownicy");


    if (PQresultStatus(res)==PGRES_TUPLES_OK)
    {
      printf("\n Odczytano dane z tabeli pracownicy \n");
      nrows=PQntuples(res);
      ncols=PQnfields(res);
      for (c=0; c<ncols; c++)
      {
        printf("%s\t|",PQfname(res,c));
        //printf("%-15s|",PQfname(res,c));
      }
      printf("\n\n");
      for (r=0; r<nrows; r++)
      {
        for (c=0; c<ncols; c++)
        {
          printf("%s\t|", PQgetvalue(res,r,c));
          //printf("%s-15s|", PQgetvalue(res,r,c));
        }
          printf("\n");
      }
    }
    else if (PQresultStatus(res)==PGRES_COMMAND_OK)
    {
      printf("\n Nie ma wyników zapytania: \n");
      printf(”%s\n”,PQerrorMessage(conn));
    }
    else
    {
      printf("\n Zapytanie było niepoprawne: \n");
      printf(”%s\n”,PQerrorMessage(conn));
    }
    PQclear(res);
  }
  else
  {
    printf("\n Nie uzyskano połączenia z bazą danych: \n");
```

```
    printf(”%s\n”,PQerrorMessage(conn));
  }
  printf("\n Koniec programu p3 \n");
  PQfinish(conn);
  return(0);
}
```

Napisz plik `makefile` o treści:

```
P3:p3.o
          gcc -o p3 p3.o -lpq
p3.o:p3.c
          gcc -c p3.c -o p3.o
```

Skompiluj program `p3.c` poleceniem

```
$ make
```

Skompilowany program pojawił się w naszym katalogu jako `*p3`. Uruchamiamy ten program

```
$ ./p3
```

Zostanie podana zawartość tabeli, ale w nieuporządkowany sposób.

Powtórz powyższe dla własnej tabeli.

6. Program porządkujący wydruki

Załóż plik `p4.c` i napisz następujący program (wpisz własną nazwę dla `dbname` i `user`):

```
#include<stdio.h>
#include</usr/include/postgresql/libpq-fe.h>

int main(void)
{
   PGconn *conn;
   PGresult *res;
   PQprintOpt pqp;

   conn=PQconnectdb("dbname=student123 user=student123");
   if (PQstatus(conn)==CONNECTION_OK)
   {
      printf("\n Mamy połączenie z bazą danych \n");
      res=PQexec(conn,"SELECT*FROM pracownicy");
      if (PQresultStatus(res)==PGRES_TUPLES_OK)
      {
         printf("\n Odczytano dane z tabeli pracownicy \n");
         pqp.header=1;
         pqp.align=1;
```

```
            pqp.html3=0;
            pqp.expanded=0;
            pqp.pager=0;
            pqp.fieldSep="|";
            pqp.tableOpt="align-center";
            pqp.caption="pracownicy";
            pqp.fieldName=NULL;
            PQprint(stdout,res,&pqp);
         }
         else if (PQresultStatus(res)==PGRES_COMMAND_OK)
         {
            printf("\n Brak wyników do wyświetlenia: \n");
            printf(”%s\n”,PQerrorMessage(conn));
         }
         else
         {
            printf("\n Zapytanie nie powiodło się: \n");
            printf(”%s\n”,PQerrorMessage(conn));
         }
         PQclear(res);
      }
      else
      {
         printf("\n Nie można połączyć się z bazą danych: \n");
         printf(”%s\n”,PQerrorMessage(conn));
      }
      printf("\n Koniec programu p4 \n");
      PQfinish(conn);
      return(0);
}
```

Napisz plik `makefile` o treści:

```
P4:p4.o
          gcc -o p4 p4.o -lpq
p4.o:p4.c
          gcc -c p4.c -o p4.o
```

Skompiluj program `p4.c` poleceniem

```
$ make
```

Skompilowany program pojawił się w naszym katalogu jako `*p4`. Uruchamiamy ten program

```
$ ./p4
```

Zostanie podana zawartość tabeli, tym razem w uporządkowanej postaci.
Powtórz powyższe dla własnej tabeli.

Na zakończenie zajęć daj `exit` (zakończenie połączenia z bazą danych).

Dodatek 3.1. C biblioteka, libpq

Biblioteka `libpq` zawiera funkcje (procedury) dla programów klienta umożliwiające pisanie zapytań do serwera bazy danych i odbieranie wyników tych zapytań. Najważniejsze z nich zostały wylistowane poniżej.

Procedury ustanawiania i zamykania połączenia z bazą danych:
- PQconnectdb
- PQsetdbLogin
- PQsetdb
- PQconnectStart
- PQconndefaults
- PQfinish
- PQreset
- PQresetStart, PQresetPoll

Funkcje dostępu:
- PQdb
- PQuser
- PQpass
- PQhost
- PQport
- PQtty
- PQoptions
- PQstatus
- PQerrorMessage
- PQbackendPID
- PQgetssl

Główne procedury zapytań:
- PQexec
- PQresultStatus
- PQresStatus
- PQresultErrorMessage
- PQclear
- PQmakeEmptyPGresult

Odzyskiwanie wyników SELECT:
- PQntuples
- PQnfields
- PQfname
- PQfnumber
- PQftype
- PQfmod
- PQfsize
- PQbinaryTuples

Odzyskiwanie wartości SELECT:
- PQgetvalue
- PQgetisnull
- PQgetlength
- PQprint

Odzyskiwanie wyników nie SELECT:
- PQcmdStatus
- PQcmdTuples
- PQoidValue

Przetwarzanie asynchronicznych zapytań:
- PQsetnonblocking
- PQisnonblocking
- PQsendQuery
- PQgetResult
- PQconsumeInput
- PQisBusy
- PQflush
- PQsocket
- PQrequestCancel

Funkcja szybkiej ścieżki:
- PQfn

Funkcja asynchronicznej notyfikacji:
- PQnotifies

Funkcje odnoszące się do COPY :
- PQgetline
- PQgetlineAsync
- PQputline
- PQputnbytes
- PQendcopy

Funkcje śledzenia `libpq`:
- PQtrace
- PQuntrace

Funkcja sterowania `libpq`:
- PQsetNoticeProcessor

Dodatek 3.2. Opis wybranych funkcji z biblioteki `libpq` (alfabetycznie)

Opisy funkcji są ścisłym tłumaczeniem z dokumentacji opracowanej przez autorów biblioteki (nie można dopuścić do odstępstw od dokumentacji, funkcji należy używać tak jak to zakładali ich autorzy). Funkcje opisano w porządku alfabetycznym.

PQclear

Funkcja `PQclear` zwalnia pamięć związaną z `PGresult`. Jeżeli wyniki zapytania nie są dalej potrzebne, to zwalniamy pamięć dając

```
void PQclear(PQresult *res);
```

Wyniki `PGresult` są przechowywane niezależnie od tego czy zostało postawione nowe zapytanie, czy też zamknięto połączenie. Brak wywołanie `PQclear` spowoduje niepotrzebną utratę pamięci po stronie aplikacji.

PQconnectdb

Ta funkcja otwiera nowe połączenie z serwerem bazy danych używając parametrów wziętych z łańcucha `conninfo` i blokuje program, aż próba nawiązania połączenia zostanie zakończona (nie-blokujące odpowiedniki tej funkcji, to `PQconnectStart` i `PQconnectPoll`)

```
PGconn *PQconnectdb(const char *conninfo);
```

Jeden program aplikacji może mieć otwartych kilka połączeń w tym samym czasie. Każde połączenie jest reprezentowane przez jeden obiekt `PGconn`, który jest zwracany przez `PQconnectdb` (lub starszą funkcję `PQsetdbLogin`). W normalnym działaniu `PQconnectdb` zawsze zwraca różny od `NULL` wskaźnik (ang. pointer) obiektu. Przed wysłaniem zapytania poprzez obiekt połączenia `PGconn`, zawsze należy sprawdzić funkcją `PQstatus`, że nawiązywanie połączenia zakończyło się sukcesem.

Łańcuch z parametrami `conninfo` może być pusty (będą użyte parametry domyślne) lub zawierać parametry oddzielone nie przecinkami, ale białymi spacjami (ang. whitespace, są to spacje, tabulatory, znaki nowej linii). Każdy parametr ma następującą postać `keyword = value` (spacje wokół znaku równości są opcjonalne). Jeżeli piszemy wartość `NULL` lub wartość zawierającą spację, to należy ją otoczyć pojedynczymi apostrofami, np. `keyword='a value'`. Pojedyncze apostrofy i lewe ukośniki wewnątrz wartości muszą być poprzedzone lewym ukośnikiem, czyli są pisane jako `\', \\`.

Na dzień dzisiejszy są rozpoznawane następujące wyrazy kluczowe parametrów:

`host`

Nazwa hosta, z którym łączymy się. Jeżeli ta nazwa rozpoczyna się od ukośnika, to specyfikuje raczej połączenie w domenie Unix, a nie połączenie TCP/IP; wartość jest nazwą katalogu, w którym jest zapamiętany plik gniazda. Jeżeli `host` nie jest specyfikowany, to

domyślnie jest to połączenie w domenie Unix z gniazdem w `/tmp` (lub jakimkolwiek katalogiem gniazda specyfikowanym przy instalowaniu PostgreSQL). Na komputerach bez gniazd domeny Unix, jest to domyślnie połączenie z `localhost`.

`hostaddr`

Numeryczny adres IP hosta, z którym łączymy się. Powinien to być standardowy format adresu poczwórnego, kropkowanego IPv4 (np. 153.19.50.110). Jeżeli komputer obsługuje standard IPv6, to on również może być użyty do pisania adresów.

Użycie `hostaddr` zamiast `host` pozwala uniknąć straty czasu na wyszukiwanie nazwy hosta. Jednak potwierdzenie autentyczności klient-serwer poprzez system bezpieczeństwa GSS-API, Kerberos wymaga podania nazwy hosta.

Jeżeli jest specyfikowany `host` bez `hostaddr`, to zostaje wymuszone wyszukiwanie nazwy hosta. Jeżeli jest specyfikowany `hostaddr` bez `host`, to wartość `hostaddr` daje adres serwera; gdy dodatkowo jest używany system Kerberos, to zostanie wywołane zwrotne zapytanie o nazwę hosta dla Kerberos. Jeżeli jest specyfikowany `host` i `hostaddr`, to wartość `hostaddr` daje adres serwera, a wartość `host` jest ignorowana, chyba że dodatkowo jest używany system Kerberos, który wykorzysta wartość `host`. Zauważmy, że najprawdopodobniej potwierdzenie autentyczności nie powiedzie się, gdy biblioteka libpq znajduje się tam, gdzie nazwa hosta nie jest nazwą komputera z `hostaddr`. Także raczej `host`, a nie `hostaddr` jest użyty do zidentyfikowania połączenia w `~/.pgpass`.

Bez nazwy hosta lub adresu hosta, biblioteka libpq zostanie połączona z użyciem gniazda w domenie Unix lub w przypadku komputerów bez gniazd domeny Unix nastąpi próba połączenia z `localhost`.

`port`

Jest to numer portu połączenia z hostem serwera lub rozszerzenie nazwy pliku gniazda dla połączeń w domenie Unix.

`dbname`

Jest to nazwa bazy danych. Domyślnie jest to taka sama nazwa jak nazwa użytkownika.

`user`

Jest to nazwa użytkownika łączącego się z PostgreSQL. Domyślnie jest to nazwa użytkownika systemu operacyjnego obsługującego aplikację.

`password`

Jest to hasło, którego należy użyć, gdy serwer żąda potwierdzenia autentyczności.

`connect_timeout`

Jest to maksymalny czas oczekiwania na połączenie w sekundach (zapisany jako dziesiętna liczba całkowita). Zaleca się, aby było to nie mniej niż 2 sekundy.

`options`

Opcje śledzenia, debuggowania, wysyłane do serwera (szczegóły w Chapter 18. Server Configuration).

`tty`

Dawniej specyfikowano tu dokąd wysłać wyniki debuggowania serwera, obecnie ignorowane.

`sslmode`

Opcja priorytetu połączenia SSL TCP/IP. Domyślnie jest to `sslmode=prefer` (najpierw próbuj połączenie SSL, gdy się nie powiedzie, to próbuj inne połączenie niż SSL).

`requiressl`

Ta opcja jest przestarzała na rzecz opcji `sslmode`. Jeżeli ustawiono wartość 1, to jest wymagane połączenie SSL z serwerem. Biblioteka `libpq` będzie odmawiała dostępu, gdy serwer nie akceptuje połączenia SSL. Jeżeli ustawiono wartość 0 (wartość domyślna), to biblioteka `libpq` będzie negocjowała typ połączenia z serwerem.

`sslcert`

Ten parametr specyfikuje nazwę pliku z certyfikatem SSL klienta.

`sslkey`

Ten parametr specyfikuje lokalizację tajnego klucza używanego do certyfikowania klienta. Domyślnie jest to `~./postgresql/postgresql.key`.

`sslrootcert`

Ten parametr specyfikuje nazwę pliku pierwiastkowego certyfikatu SSL.

`krbsrvname`

Nazwa usługi Kerberos potwierdzenia autentyczności klient-serwer.

`gsslib`

GSS biblioteka do użycia w systemie GSS-API potwierdzania autentyczności. Używana tylko w Windows. Ustaw `gssapi`, aby zmusić bibliotekę libpq do użycia biblioteki GSS-API zamiast domyślnie SSPI.

`service`

Nazwa usługi na użycie dodatkowych parametrów. Specyfikuje nazwę usługi w `pg_service.conf`, gdzie są przechowywane dodatkowe parametry połączenia. To

pozwala, aby specyfikować tylko nazwę usługi, a parametry połączenia będą centralnie obsługiwane.

Jeżeli którykolwiek z powyższych parametrów nie jest specyfikowany, to jest sprawdzana odpowiednia zmienna środowiskowa. Jeśli także ona nie jest ustawiona, to zostaną użyte wbudowane ustawienia domyślne.

PQdb

Funkcja zwraca nazwę bazy danych połączenia

```
char *PQdb(const PGconn *conn);
```

PQerrorMessage

Funkcja zwraca ostatnio wygenerowaną wiadomość o błędzie operacji połączenia

```
char *PQerrorMessage(const PGconn* conn);
```

Prawie wszystkie funkcje biblioteki libpq będą w przypadku niepowodzenia generowały `PQerrorMessage`.

PQexec

Przekazanie polecenia SQL do serwera i oczekiwanie na wyniki

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Funkcja zwraca wskaźnik `PGresult`. Wskaźnik ten jest generalnie różny od `NULL`, za wyjątkiem przypadku braku pamięci lub poważnych błędów, takich jak niemożność wysłania zapytania do serwera. W przypadku zwrócenia wskaźnika `NULL`, powinno to być traktowane jak wynik `PGRES_FATAL_ERROR`. Więcej informacji o takich błędach uzyskamy używając funkcji `PQerrorMessage`.

Dopuszcza się, aby w łańcuchu polecenia `command` występowały wielokrotne polecenia SQL oddzielone średnikami. Takie wielokrotne polecenia są przetwarzane jako jedna transakcja (chyba że użyto jawnie poleceń `BEGIN/COMMIT` wydzielających wielokrotne transakcje). Pamiętajmy jednak, że w `PGresult` są zwracane wyniki tylko ostatniego polecenia z łańcucha poleceń. Jeżeli jedno z poleceń będzie niemożliwe do wykonania, to przetwarzanie łańcucha poleceń zostanie przerwane i zwracany `PGresult` będzie sygnalizował błędne warunki.

PQfinish

Zamknięcie połączenia z serwerem (także zwolnienie pamięci zajmowanej przez obiekt `PGconn`)

```
void PQfinish(PGconn *conn);
```

Zauważmy, że nawet gdy próba połączenia z serwerem nie powiedzie się (co zostanie wykazane przez `PQstatus`), to w aplikacji trzeba wywołać `PQfinish`, aby zwolnić pamięć zajmowaną przez obiekt `PGconn`. Wskaźnik `PGconn` nie może być użyty od nowa zanim nie zostanie wywołane `PQfinish`.

PQgetvalue

Funkcja zwraca wartość pojedynczego pola jednego wiersza z `PGresult` (numerowanie wierszy i kolumn rozpoczyna się od 0)

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Jeżeli dane mają format tekstowy, to wartością zwracaną przez `PQgetvalue` jest wartość pola reprezentowana przez łańcuch znaków ASCII zakończony `NULL`. Jeżeli dane mają format binarny, to zwracana wartość ma binarną reprezentację określoną przez typ danych funkcji `typsend` i `typreceive`. Jeżeli wartością pola jest `NULL`, to jest zwracany pusty łańcuch.

Wskaźnik zwracany przez `PQgetvalue` wskazuje pamięć, która jest częścią struktury `PGresult` i nie należy tego modyfikować. Jeżeli wskaźnik ma być użyty po zmianie struktury `PGresult`, to należy jawnie przekopiować dane do innej pamięci.

PQhost

Funkcja zwraca nazwę hosta serwera połączenia

```
char *PQhost(const PGconn *conn);
```

PQnfields

Funkcja zwraca liczbę kolumn wyniku zapytania

```
int PQncolumns(const PGresult *res)
```

PQntuples

Funkcja zwraca liczbę wierszy wyniku zapytania

```
int PQntuples(const PGresult *res)
```

PQoptions

Funkcja zwraca opcje poleceń w liniach zapytania połączenia

```
char *PQoptions(const PGconn *conn);
```

PQpass

Funkcja zwraca nazwę hasło (password) połączenia

```
char *PQpass(const PGconn *conn);
```

PQport

Funkcja zwraca port połączenia

```
char *PQport(const PGconn *conn);
```

PQprint

Funkcja drukuje w specyfikowanym strumieniu wyjściowym wszystkie wiersze i opcjonalnie nazwy kolumn

```
void PQprint(FILE* fout,      /* output stream */
                  const PGresult *res,
                  const PQprintOpt *po);

struct {
  pqbool  header;   /* print output field headings and row count */
  pqbool  align;    /* fill align the fields */
  pqbool  standard; /* old brain dead format */
  pqbool  html3;    /* output html tables */
  pqbool  expanded; /* expand tables */
  pqbool  pager;    /* use pager for output if needed */
  char    *fieldSep; /* field separator */
  char    *tableOpt; /* insert to HTML table ... */
  char    *caption;  /* HTML caption */
  char    **fieldName;/* NULL terminated array of replacement field
names */
} PQprintOpt;
```

Ta funkcja była używana w `psql` do drukowania wyników zapytania. Funkcja zakłada, że dane mają format `text`.

PQreset

Resetowanie kanału połączenia z serwerem

```
void PQreset(PGconn *conn);
```

Funkcja zamyka połączenie z serwerem i próbuje ustanowić nowe połączenie z tym samym serwerem używając dotychczas obowiązujących parametrów. Jest to użyteczne w sytuacji, gdy bieżące połączenie zostało utracone.

PQresStatus

Funkcja dokonuje konwersji rezultatów z `PQresultStatus` na stałą łańcuchową opisującą kod statusu

```
char *PQresStatus(ExecStatusType status);
```

PQresultStatus

Funkcja zwraca status polecenia

```
ExecStatusType PQresultStatus(const PGresult *res);
```

Zwracana jest jedna z wartości:
PGRES_COMMAND_OK - Pomyślne zakończenie polecenia nie zwracającego danych
PGRES_TUPLES_OK - Pomyślne zakończenie polecenia zwracającego dane (jak SELCT lub SHOW)
PGRES_EMPTY_QUERY - Łańcuch wysłany do serwera był pusty
PGRES_COPY_OUT - Rozpoczęcie transferu danych z serwera (Copy Out)
PGRES_COPY_IN - Rozpoczęcie transferu danych do serwera (Copy In)
PGRES_BAD_RESPONSE - Odpowiedź z serwera nie została zrozumiana
PGRES_NONFATAL_ERROR – Zdarzył się nie-fatalny błąd (notka lub ostrzeżenie)
PGRES_FATAL_ERROR – Zdarzył się fatalny błąd

Jeżeli status jest PGRES_TUPLES_OK, to możemy użyć stosownych funkcji (PQgetvalue, itd.) do przeglądania wierszy zwróconych przez zapytanie. Pamiętajmy przy tym, że polecenie SELECT zwracające zero wierszy będzie miało status PGRES_TUPLES_OK. Status PGRES_COMMAND_OK będą miały polecenia nigdy nie zwracające wierszy (np. INSERT, UPDATE). Status PGRES_EMPTY_QUERY może wskazywać na pluskwę (ang. a bug) w oprogramowaniu klienta.

PQstatus

Funkcja zwraca status połączenia

```
ConnStatusType PQstatus(const PGconn *conn);
```

Status połączenia może mieć wiele wartości. Jednak tylko dwie z nich są widziane na zewnątrz procedury asynchronicznego połączenia: CONNECTION_OK i CONNECTION_BAD. Poprawne połączenie z bazą danych ma status CONNECTION_OK i zasadniczo pozostaje z takim statusem aż do PQfinish. W międzyczasie może jednak nastąpić zmiana statusu na CONNECTION_BAD z powodu utraty połączenia. W takim przypadku aplikacja powinna dokonać odnowy poprzez wywołanie PQreset.

PQuser

Funkcja zwraca nazwę użytkownika połączenia

```
char *PQuser(const PGconn *conn);
```

Dodatek 3.3. GNU – polecenia

GNU Utilities

* Gpg: (gpg). OpenPGP encryption and signing tool (v1).

Individual utilities

* arch: (coreutils)arch invocation. Print machine hardware name.
* base64: (coreutils)base64 invocation. Base64 encode/decode data.
* basename: (coreutils)basename invocation. Strip directory and suffix.
* cat: (coreutils)cat invocation. Concatenate and write files.
* chcon: (coreutils)chcon invocation. Change SELinux CTX of files.
* chgrp: (coreutils)chgrp invocation. Change file groups.
* chmod: (coreutils)chmod invocation. Change file permissions.
* chown: (coreutils)chown invocation. Change file owners/groups.
* chroot: (coreutils)chroot invocation. Specify the root directory.
* cksum: (coreutils)cksum invocation. Print POSIX CRC checksum.
* comm: (coreutils)comm invocation. Compare sorted files by line.
* cp: (coreutils)cp invocation. Copy files.
* csplit: (coreutils)csplit invocation. Split by context.
* cut: (coreutils)cut invocation. Print selected parts of lines.
* date: (coreutils)date invocation. Print/set system date and time.
* dd: (coreutils)dd invocation. Copy and convert a file.
* df: (coreutils)df invocation. Report file system disk usage.
* dircolors: (coreutils)dircolors invocation. Color setup for ls.
* dirname: (coreutils)dirname invocation. Strip non-directory suffix.
* dir: (coreutils)dir invocation. List directories briefly.
* du: (coreutils)du invocation. Report on disk usage.
* echo: (coreutils)echo invocation. Print a line of text.
* env: (coreutils)env invocation. Modify the environment.
* expand: (coreutils)expand invocation. Convert tabs to spaces.
* expr: (coreutils)expr invocation. Evaluate expressions.
* factor: (coreutils)factor invocation. Print prime factors
* false: (coreutils)false invocation. Do nothing, unsuccessfully.
* find: (find)Invoking find. Finding and acting on files.
* fmt: (coreutils)fmt invocation. Reformat paragraph text.
* fold: (coreutils)fold invocation. Wrap long input lines.
* groups: (coreutils)groups invocation. Print group names a user is in.
* gzip: (gzip)Invoking gzip. Compress files.
* head: (coreutils)head invocation. Output the first part of files.
* hostid: (coreutils)hostid invocation. Print numeric host identifier.
* hostname: (coreutils)hostname invocation. Print or set system name.
* id: (coreutils)id invocation. Print user identity.
* install: (coreutils)install invocation. Copy and change attributes.
* join: (coreutils)join invocation. Join lines on a common field.
* kill: (coreutils)kill invocation. Send a signal to processes.
* link: (coreutils)link invocation. Make hard links between files.
* ln: (coreutils)ln invocation. Make links between files.
* locate: (find)Invoking locate. Finding files in a database.
* logname: (coreutils)logname invocation. Print current login name.
* ls: (coreutils)ls invocation. List directory contents.

* md5sum: (coreutils)md5sum invocation. Print or check MD5 digests.
* mkdir: (coreutils)mkdir invocation. Create directories.
* mkfifo: (coreutils)mkfifo invocation. Create FIFOs (named pipes).
* mknod: (coreutils)mknod invocation. Create special files.
* mv: (coreutils)mv invocation. Rename files.
* nice: (coreutils)nice invocation. Modify niceness.
* nl: (coreutils)nl invocation. Number lines and write files.
* nohup: (coreutils)nohup invocation. Immunize to hangups.
* od: (coreutils)od invocation. Dump files in octal, etc.
* paste: (coreutils)paste invocation. Merge lines of files.
* pathchk: (coreutils)pathchk invocation. Check file name portability.
* printenv: (coreutils)printenv invocation. Print environment variables.
* printf: (coreutils)printf invocation. Format and print data.
* pr: (coreutils)pr invocation. Paginate or columnate files.
* ptx: (coreutils)ptx invocation. Produce permuted indexes.
* pwd: (coreutils)pwd invocation. Print working directory.
* readlink: (coreutils)readlink invocation. Print referent of a symlink.
* rmdir: (coreutils)rmdir invocation. Remove empty directories.
* rm: (coreutils)rm invocation. Remove files.
* runcon: (coreutils)runcon invocation. Run in specified SELinux CTX.
* seq: (coreutils)seq invocation. Print numeric sequences
* sha1sum: (coreutils)sha1sum invocation. Print or check SHA-1 digests.
* sha2: (coreutils)sha2 utilities. Print or check SHA-2 digests.
* shred: (coreutils)shred invocation. Remove files more securely.
* shuf: (coreutils)shuf invocation. Shuffling text files.
* sleep: (coreutils)sleep invocation. Delay for a specified time.
* sort: (coreutils)sort invocation. Sort text files.
* split: (coreutils)split invocation. Split into fixed-size pieces.
* stat: (coreutils)stat invocation. Report file(system) status.
* stty: (coreutils)stty invocation. Print/change terminal settings.
* sum: (coreutils)sum invocation. Print traditional checksum.
* su: (coreutils)su invocation. Modify user and group ID.
* sync: (coreutils)sync invocation. Synchronize memory and disk.
* tac: (coreutils)tac invocation. Reverse files.
* tail: (coreutils)tail invocation. Output the last part of files.
* tee: (coreutils)tee invocation. Redirect to multiple files.
* test: (coreutils)test invocation. File/string tests.
* time: (time). Run programs and summarize system resource usage.
* timeout: (coreutils)timeout invocation. Run with time limit.
* touch: (coreutils)touch invocation. Change file timestamps.
* true: (coreutils)true invocation. Do nothing, successfully.
* truncate: (coreutils)truncate invocation. Shrink/extend size of a file.
* tr: (coreutils)tr invocation. Translate characters.
* tsort: (coreutils)tsort invocation. Topological sort.
* tty: (coreutils)tty invocation. Print terminal name.
* uname: (coreutils)uname invocation. Print system information.
* unexpand: (coreutils)unexpand invocation. Convert spaces to tabs.
* uniq: (coreutils)uniq invocation. Uniquify files.
* unlink: (coreutils)unlink invocation. Removal via unlink(2).
* updatedb: (find)Invoking updatedb. Building the locate database.

* uptime: (coreutils)uptime invocation. Print uptime and load.
* users: (coreutils)users invocation. Print current user names.
* vdir: (coreutils)vdir invocation. List directories verbosely.
* wc: (coreutils)wc invocation. Line, word, and byte counts.
* whoami: (coreutils)whoami invocation. Print effective user ID.
* who: (coreutils)who invocation. Print who is logged in.
* xargs: (find)Invoking xargs. Operating on many files.
* yes: (coreutils)yes invocation. Print a string indefinitely.

Kernel

* grub-install: (grub)Invoking grub-install. Install GRUB on your drive
* grub-terminfo: (grub)Invoking grub-terminfo.
Generate a terminfo command from a terminfo name
* GRUB: (grub). The GRand Unified Bootloader

Libraries

* RLuserman: (rluserman). The GNU readline library User's Manual.

Miscellaneous

* Ipc: (ipc). System V style inter process communication

Network Applications

* Wget: (wget). The non-interactive network downloader.

Text creation and manipulation

* Grep: (grep). Print lines matching a pattern.
* Sed: (sed). Stream EDitor.

Utilities

* Gzip: (gzip). The gzip command for compressing files.