
# Materiały szkoleniowe

# Relacyjne bazy danych na przykładzie PostgreSQL (poziom podstawowy)

Ministerstwo Klimatu i Środowiska

Sfinansowano ze środków Narodowego Funduszu Ochrony Środowiska i Gospodarki Wodnej

## Spis treści:

# Wprowadzenie do przedstawianych zagadnień

## Wprowadzenie do baz danych

Definicje bazy danych:

**Techniczna definicja bazy danych** - dane cyfrowe gromadzone zgodnie z zasadami przyjętymi dla danego programu komputerowego specjalizowanego do gromadzenia i przetwarzania tych danych (Wikipedia)

**Prawna definicja bazy danych** - zbiór danych lub jakichkolwiek innych materiałów i elementów zgromadzonych według określonej systematyki lub metody, indywidualnie dostępnych w jakikolwiek sposób, w tym środkami elektronicznymi, wymagający istotnego, co do jakości lub ilości, nakładu inwestycyjnego w celu sporządzenia, weryfikacji lub prezentacji jego zawartości (Ustawa o ochronie baz danych z 27.07.2001)

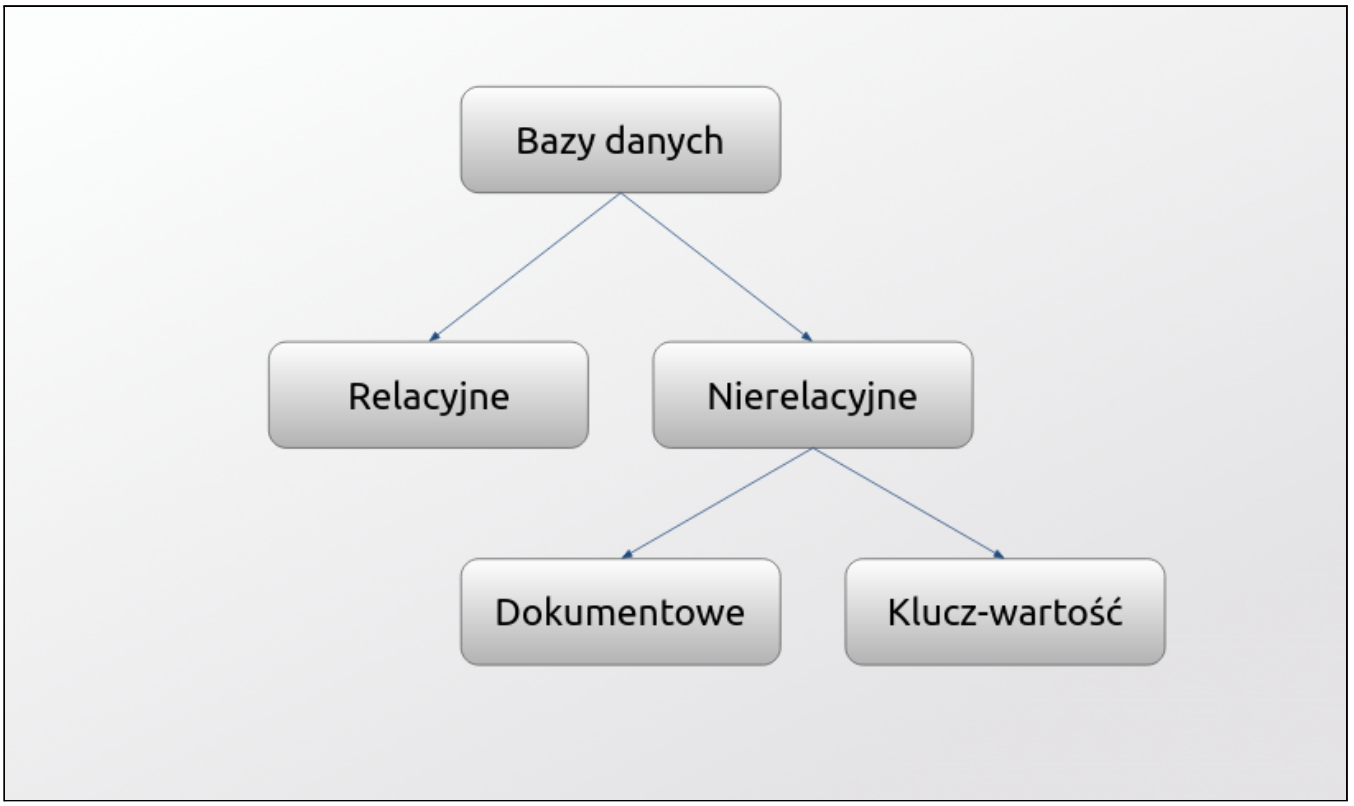
**System zarządzania bazą danych** - Oprogramowanie bądź system informatyczny służący do zarządzania bazą danych. System zarządzania bazą danych może być również serwerem bazy danych (SBD) lub też może udostępniać bazę danych lokalnie – na określonym komputerze.

Zadania systemu zarządzania bazą danych:

- administrowanie danymi,
- zapewnienie integralności danych
- narzędzia dostępu i przeszukiwania danych
- narzędzia usprawniające wyszukiwanie danych (np. indeksy)
- narzędzia autoryzacji użytkowników
- narzędzia umożliwiające równoczesny dostęp wielu użytkowników

# Podział baz danych

Ze względu na model przechowywanych danych bazy dzielimy na relacyjne i nierelacyjne. Nierelacyjne bazy danych dzielimy na dokumentowe oraz klucz-wartość.




## Baza klucz-wartość

Baza klucz-wartość to baza przechowująca dane w formie par klucz (identyfikator tekstowy lub liczbowy) - wartość (dowolne dane, zwykle tekst)

```
{
  "ŚW": "Świerk pospolity",
  "SO": "Sosna pospolita",
  "JD": "Jodła pospolita",
  "BK": "Buk zwyczajny",
  "BRZ": "Brzoza brodawkowata",
  "OL": "Olsza czarna",
  "JB": "Jabłoń domowa",
  "SW": "Sosna wejmutka",
  "CZ": "Czeremcha pospolita",
  "JW": "Klon jawor",
```

```
  "DB": "Dąb szypułkowy"
}
```

Charakterystyka:

- Brak określonego modelu danych dla wartości
- Brak obsługi relacji pomiędzy danymi
- Bardzo szybki dostęp do danych, jeśli znamy klucz (w przeciwnym razie poszukiwanie odbywa się sekwencyjnie)
- Zastosowanie jako pomocnicza struktura danych (cache)

Przykłady systemów:

- Redis,
- Tokyo Cabinet,
- Memcached

## Baza dokumentowa

Baza dokumentowa to baza przechowująca dane w formie dokumentów, najczęściej w formacie JSON.

```
[{
   "kod": "ŚW",
   "nazwa": {
       "polska": "Świerk pospolity",
       "łacińska": "Picea abies"
   },
   "systematyka": {
       "rząd":"sosnowce",
       "rodzina": "sosnowate"
   },
   "wiek rębnosci": 120
},
{"kod":"SO", "nazwa_pl":"Sosna", "wiek_rebny": "120"}
]
```

Charakterystyka:

- Brak określonego modelu danych wewnątrz dokumentu
- Dane mogą być zagnieżdżone
- Brak obsługi relacji pomiędzy danymi

- Szybki dostęp do danych po identyfikatorze, możliwe też przeszukiwanie po wybranych atrybutach
- Zastosowanie - aplikacje Web, przechowywanie danych o dużej zmienności

Przykłady systemów:

- MongoDB,
- CouchDB

Baza relacyjna

Baza relacyjna to baza przechowująca dane w formie tabel o ustalonej strukturze.

| | sub_area | prot_categ | a_year | gat_gl | wiek | udzial | typ_dstan | typ_udz_1 |
|---|---|---|---|---|---|---|---|---|
| 2 | 4.59 | OCH CENNE | 2016 | SO | 46 | 7 | IGLASTE | DOMINATED |
| 3 | 4.18 | OCH CENNE | 2016 | BRZ | 38 | 4 | LISCIASTE | MIXED |
| 4 | 7.97 | OCH CENNE | 2016 | BK | 152 | 8 | LISCIASTE | PURE |
| 5 | 1.42 | OCH CENNE | 2016 | MD | 47 | 7 | IGLASTE | DOMINATED |
| 6 | 4.15 | OCH CENNE | 2016 | SO | 87 | 6 | IGLASTE | DOMINATED |
| 7 | 1.51 | OCH CENNE | 2016 | SO | 97 | 10 | IGLASTE | PURE |
| 8 | 10.81 | OCH CENNE | 2016 | DB | 137 | 7 | LISCIASTE | DOMINATED |
| 9 | 0.81 | OCH CENNE | 2016 | DB | 8 | 4 | LISCIASTE | MIXED |
| 10 | 2.96 | OCH CENNE | 2016 | BK | 20 | 5 | LISCIASTE | DOMINATED |
| 11 | 2.29 | OCH CENNE | 2016 | MD | 50 | 3 | IGLASTE | MIXED |
| 12 | 5.19 | OCH CENNE | 2016 | SO | 97 | 7 | IGLASTE | DOMINATED |
| 13 | 4.56 | OCH CENNE | 2016 | BK | 122 | 5 | LISCIASTE | DOMINATED |
| 14 | 4.78 | OCH CENNE | 2016 | SO | 97 | 9 | IGLASTE | PURE |
| 15 | 4.63 | OCH CENNE | 2016 | DB | 137 | 4 | LISCIASTE | MIXED |
| 16 | 8.85 | OCH CENNE | 2016 | DB | 162 | 5 | LISCIASTE | DOMINATED |
| 17 | 3.96 | OCH CENNE | 2016 | DB | 162 | 7 | LISCIASTE | DOMINATED |

Charakterystyka:

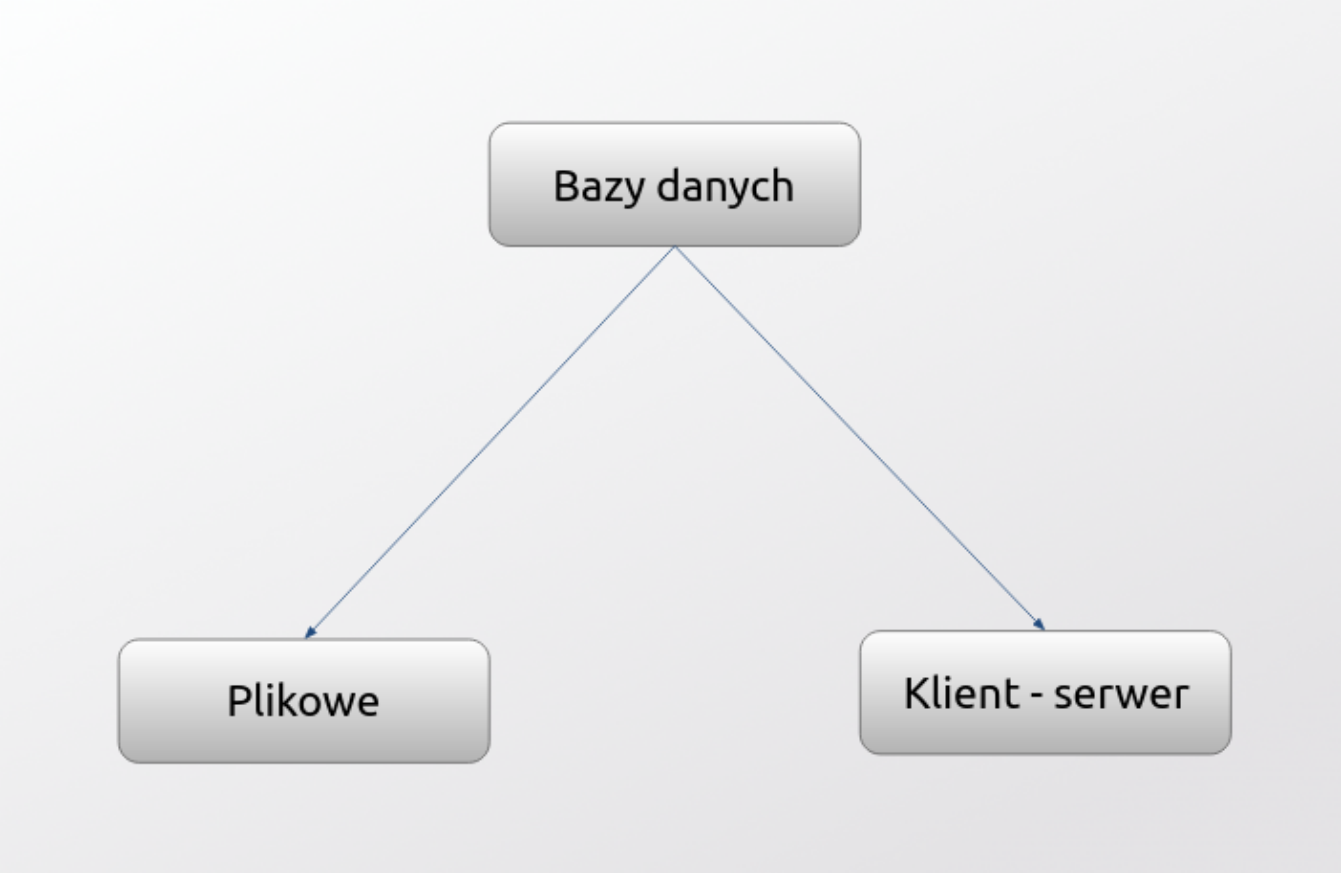
- Zbiór atrybutów dla pojedynczej tabeli jest z góry określony
- Obsługa relacji pomiędzy danymi
- Nie tak szybkie jak bazy klucz-wartość i dokumentowe dla poszukiwania po identyfikatorze, ale bardzo duże możliwości efektywnego przeszukiwania i analizy danych
- Zastosowanie - aplikacje Web, GIS, systemy bankowe, rejestry publiczne, aplikacje mobilne

Przykłady systemów:

- Oracle,
- SQL Server,
- Postgres,
- MySQL

## Bazy plikowe i klient-serwer

Ze względu na sposób przechowywania danych oraz dostępu do nich bazy danych dzielimy na plikowe oraz klient-serwer.




**Baza plikowa** to baza przechowująca dane w pojedynczym pliku na lokalnym dysku urządzenia.

Charakterystyka:

- Baza jest zwykłym plikiem
- Nie wymaga stale działającego programu-serwera bazy
- Nie posiada systemu zarządzania i autoryzacji użytkowników
- Tylko jeden użytkownik może korzystać z bazy jednocześnie

Przykłady systemów:

- SQLite,
- SpatiaLite,
- GeoPackage,
- Geobaza plikowa ESRI

**Baza klient-serwer** to baza przechowująca dane w specjalnej strukturze plików na serwerze.

Charakterystyka:

- Dane są przechowywane na serwerze i udostępniane poprzez sieć
- Pliki źródłowe nie są czytelne dla innego oprogramowania
- Z bazy może korzystać wielu użytkowników jednocześnie
- Wymaga stale działającego programu-serwera bazy danych

Przykłady systemów:

- MySQL,
- Postgres,
- MS SQL,
- MySQL,
- Oracle,
- Geobaza wielodostępna ESRI

## Wprowadzenie do PostgreSQL

Zgodnie z informacjami zawartymi na stronie głównej projektu PostgreSQL to potężny, obiektowo-relacyjny system bazy danych o otwartym kodzie źródłowym, z ponad 30-letnim aktywnym rozwojem, dzięki któremu zyskał dobrą reputację dzięki niezawodności, użyteczności funkcji i wydajności. Sam projekt wyewoluował z innego projektu prowadzonego na uniwersytecie Berkeley o nazwie Ingres.
Nieoficjalne źródła podają, że nazwa powstała jako żart osób tworzących projekt, polegający na zastosowaniu nazewnictwa fragmentów ciągu (prefix, infix, postfix) do nazwy ingres, której naturalnym następstwem będzie postgres.
Zgodnie z dokumentacją projektu w roku 1996 twórcy postanowili zaznaczyć zgodność bazy danych ze standardem SQL dodając odpowiedni postfix do nazwy. W wyniku spłaszczenia podwójnego 's' **nazwę wymawiamy jako jeden wyraz** - **Postgresql**, nie literując dodatkowo SQL (**wymowa postgre-eS-Qu-eL jest błędna**), oraz skracamy do **Postgres** przez wzgląd na poprzednie nazwy, a nie do Postgre jak by wynikało z odcięcia postfixu SQL - takie skracanie jest błędne.

## Używane rozszerzenia

W toku szkolenia będziemy używali trzech rozszerzeń bazy danych:

- **postgis** - rozszerzenie bazy o obsługę danych przestrzennych
- **hstore** - typ danych oparty o strukturę klucz=wartość
- **pg_stat_statements** - rozszerzenie o statystyki pozwalające na analizę planów wykonywanych zapytań

## Instalacja w systemach Linux

Większość dystrybucji systemów Linux ma gotowe prekompilowane paczki zarówno dla bazy danych PostgreSQL jak i dla poszczególnych jej rozszerzeń. W systemach opartych na `apt` bazę instalujemy poleceniem

```
apt install postgresql postgis
```

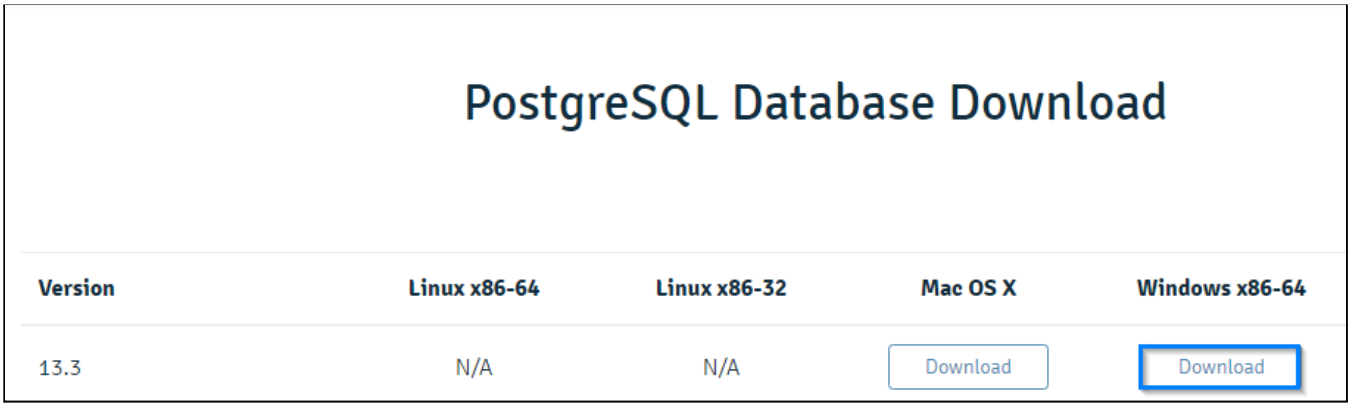
## Instalacja w systemach Windows

W systemach operacyjnych Windows, zgodnie z oficjalną dokumentacją bazę danych wraz z rozszerzeniami należy instalować używając instalatorów dostarczanych przez EnterpriseDB. Instalatory te zawierają:

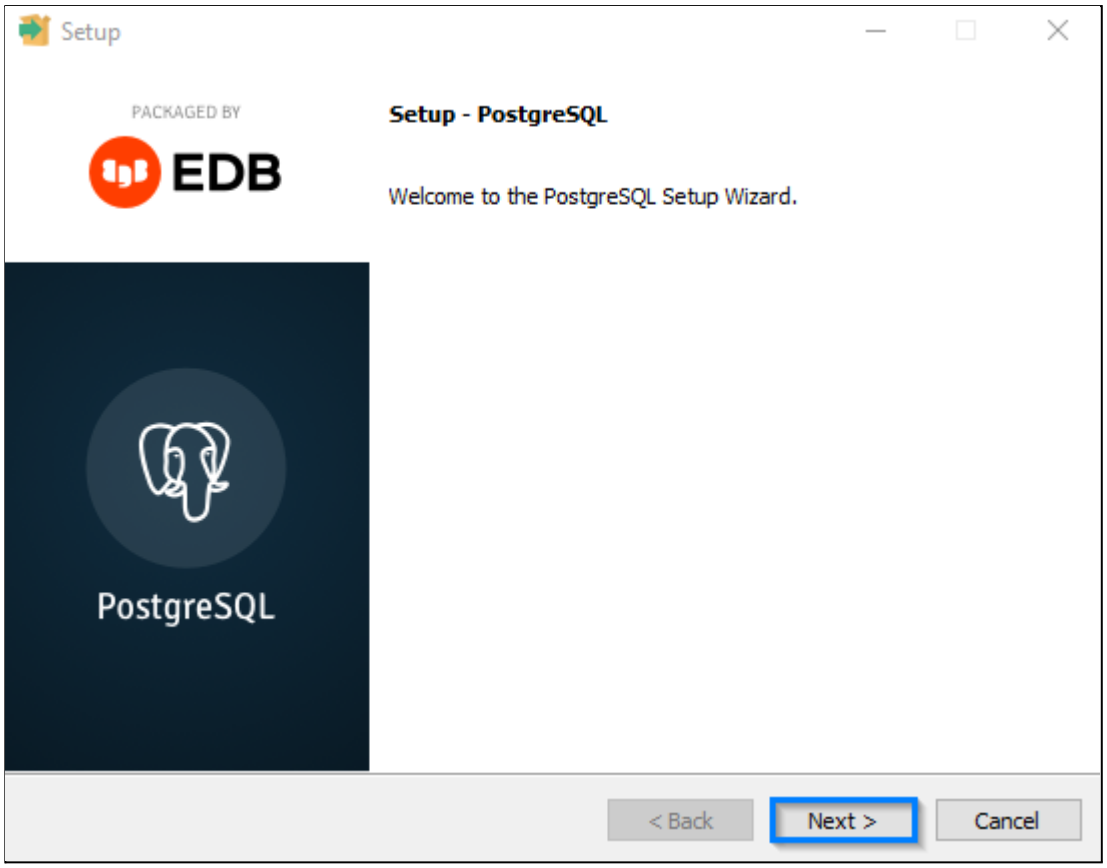
- serwer bazy danych
- pgAdmin - narzędzie do zarządzania oraz pracy na bazie danych
- StackBuilder - manager pakietów pozwalający na pobieranie oraz instalowanie dodatkowych rozszerzeń i sterowników

Instalator może być uruchomiony w trybie interaktywnym oraz cichym.

Instalator możemy pobrać ze strony https://www.enterprisedb.com/downloads/postgres-postgresql-downloads. Szkolenie przeprowadzimy na wersji Postgresql 13.

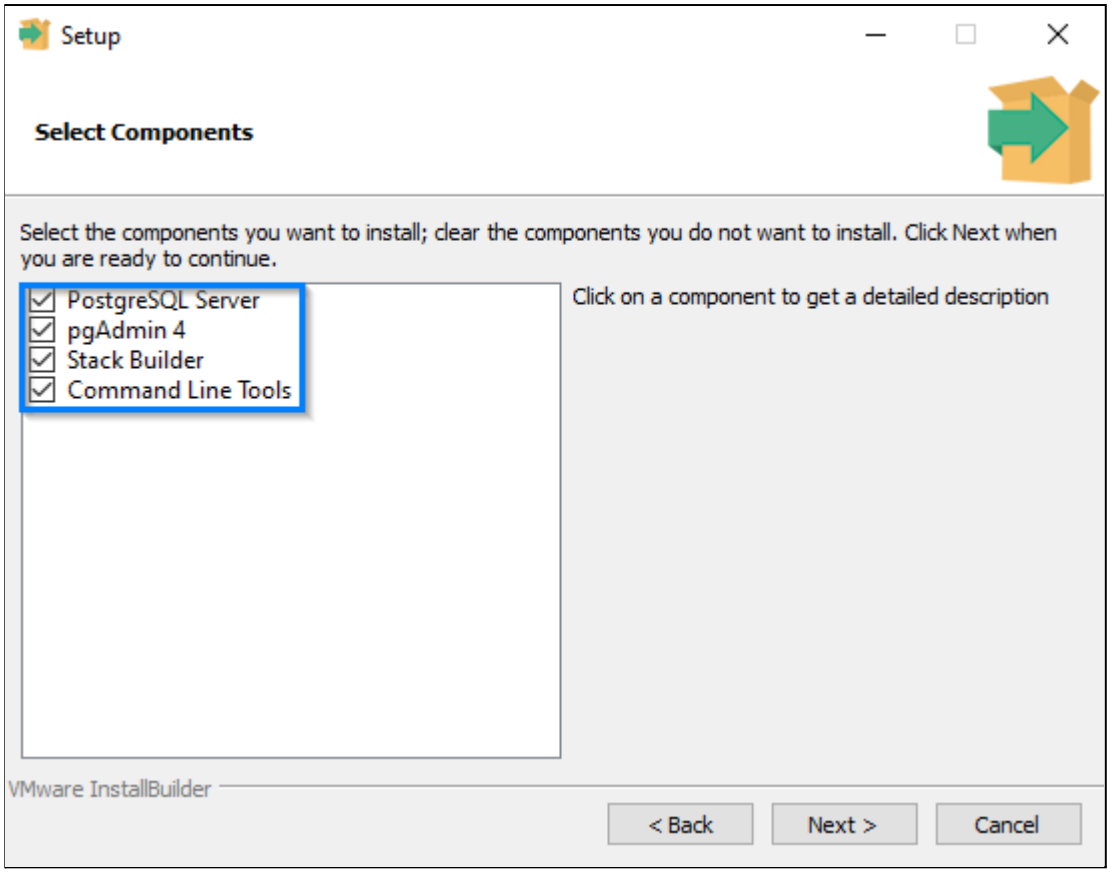


Plik instalatora pobieramy i zapisujemy lokalnie, po czym uruchamiamy instalator. Możliwe że będzie trzeba potwierdzić uprawnienia administratora w kontroli konta użytkownika.



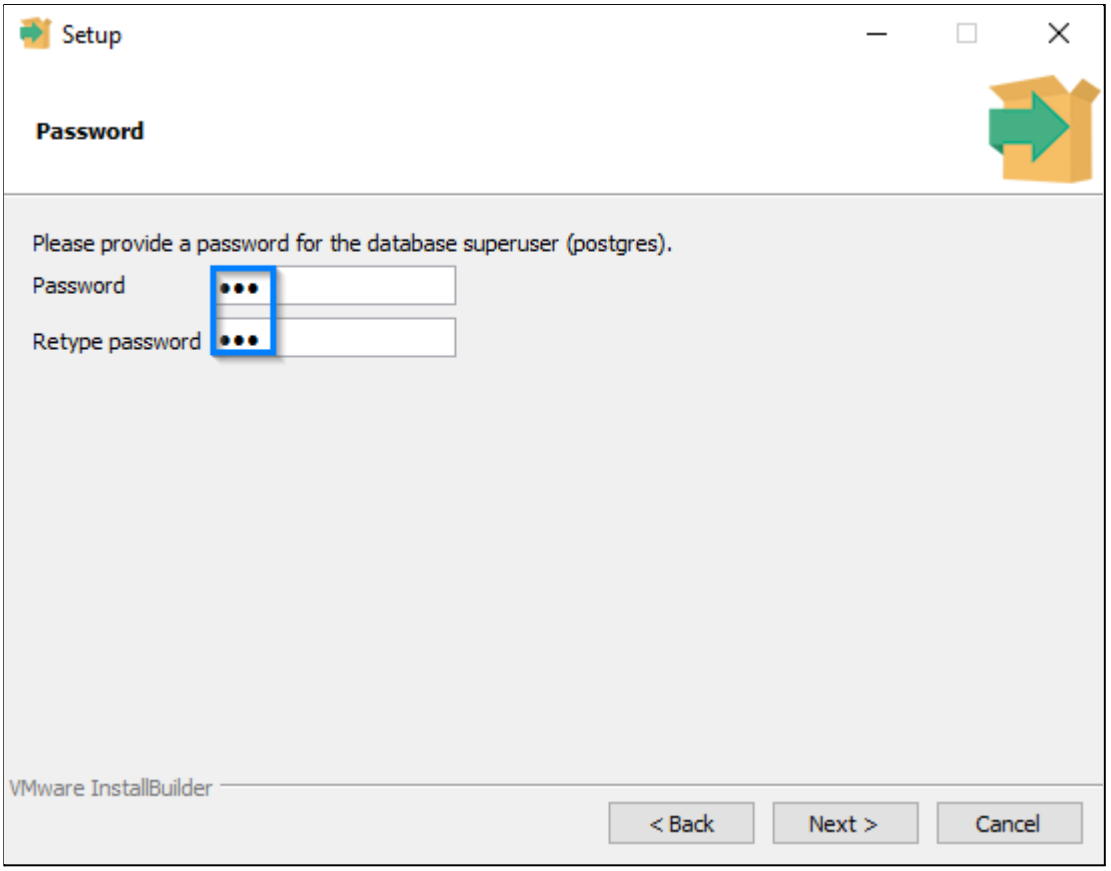

W kolejnym oknie podajemy ścieżkę w której instalator zainstaluje bazę danych - na potrzeby szkolenia sugerujemy pozostawienie wartości domyślnej (C:\Program Files\PostgreSQL\13). W przypadku zmiany tej wartości należy zapamiętać nową ścieżkę ponieważ będzie ona konieczna do podania ponownie później.

W kolejnym kroku wybieramy wszystkie komponenty do instalacji (zaznaczamy wszystkie pola).

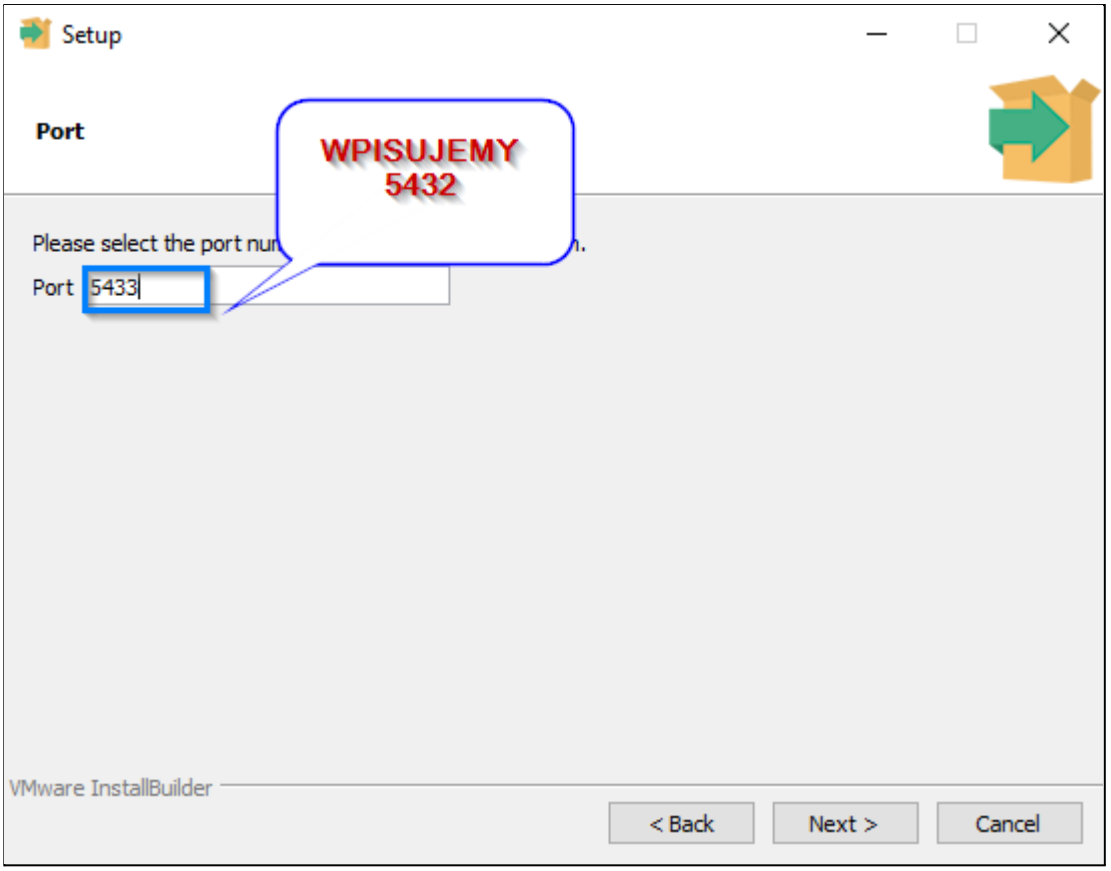



Kolejny krok pozwala na zdefiniowanie folderu w którym przechowywane będą dane. W tym przypadku również można pozostawić wartość domyślną (C:\Program Files\PostgreSQL\13\data).

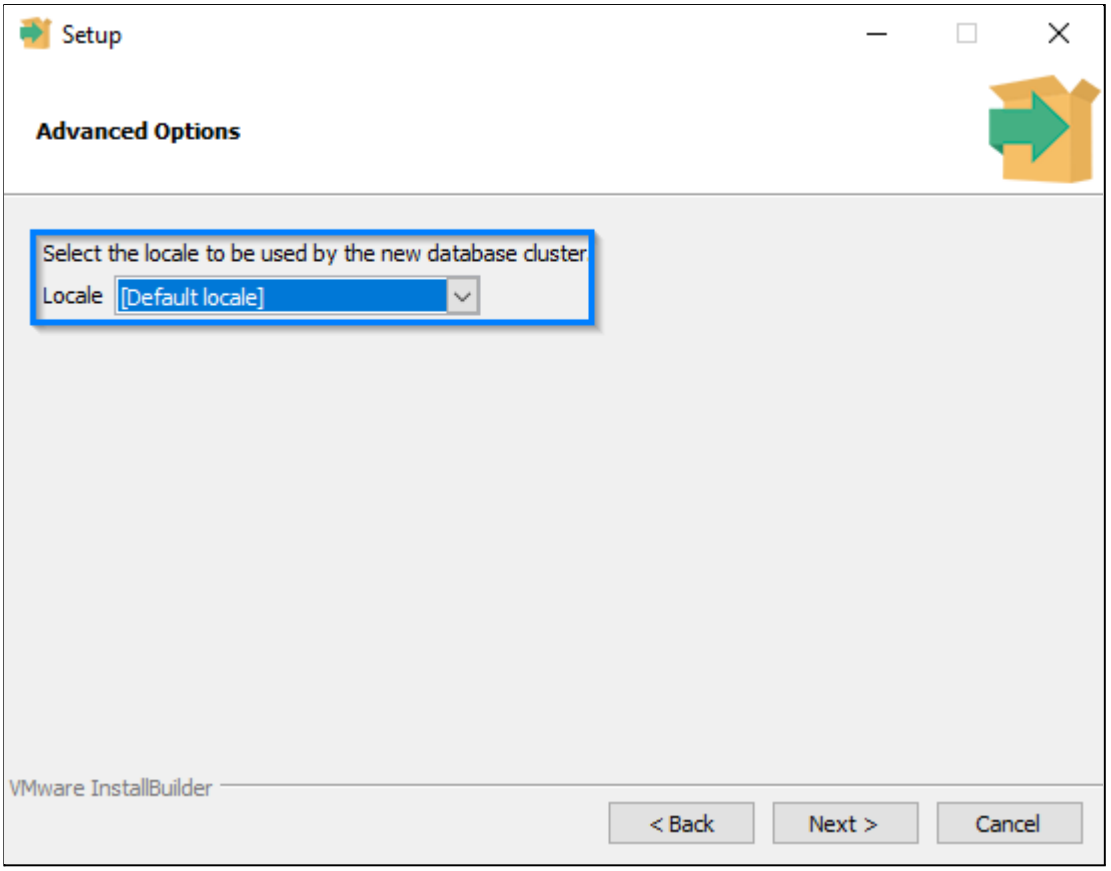
Kolejny krok to ustawienie hasła do bazy danych - na potrzeby szkolenia sugerujemy ustawienie ‘gis’. W przypadku ustawienia innego hasła należy go bezwzględnie zapamiętać - w przeciwnym razie konieczna będzie ponowna instalacja.



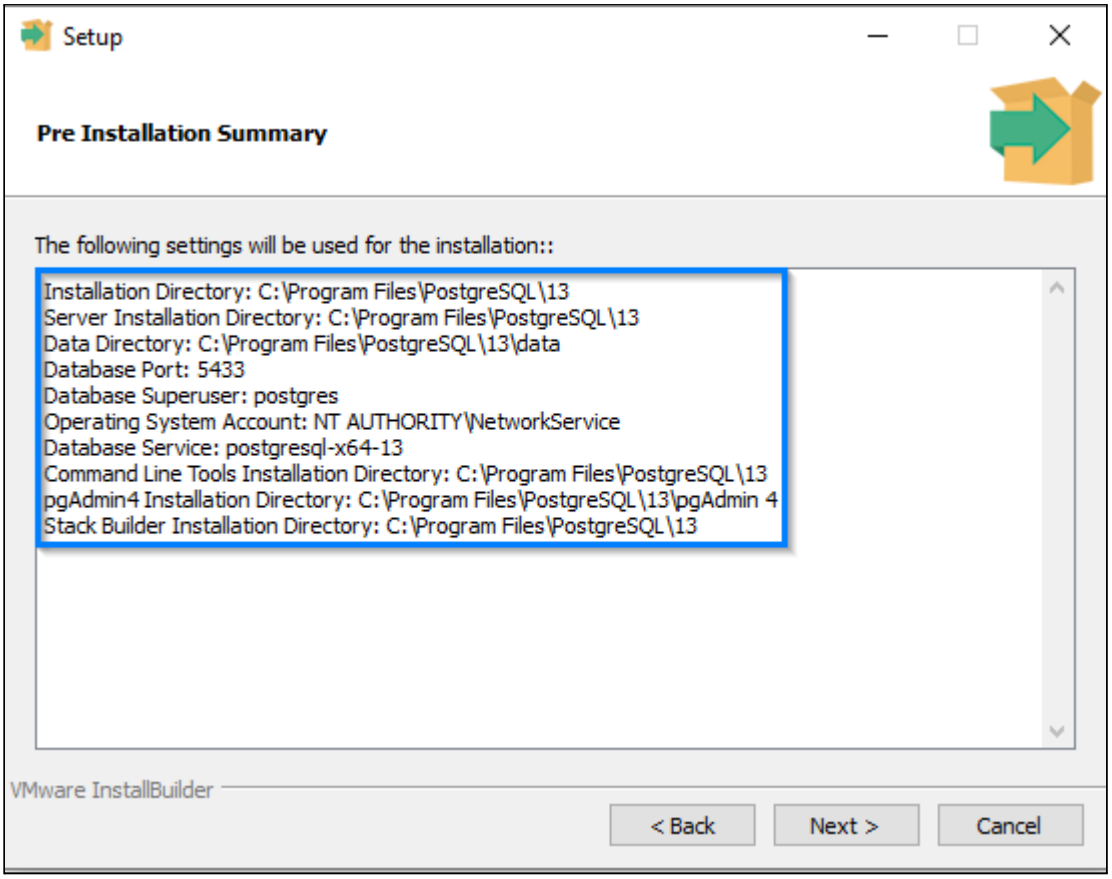

Kolejny krok umożliwia zmianę domyślnego portu, na którym nasłuchuje baza - znów sugerujemy pozostawienie wartości domyślnej 5432.




Ustawienia lokalne systemu również zostawiamy na wartości domyślnej.

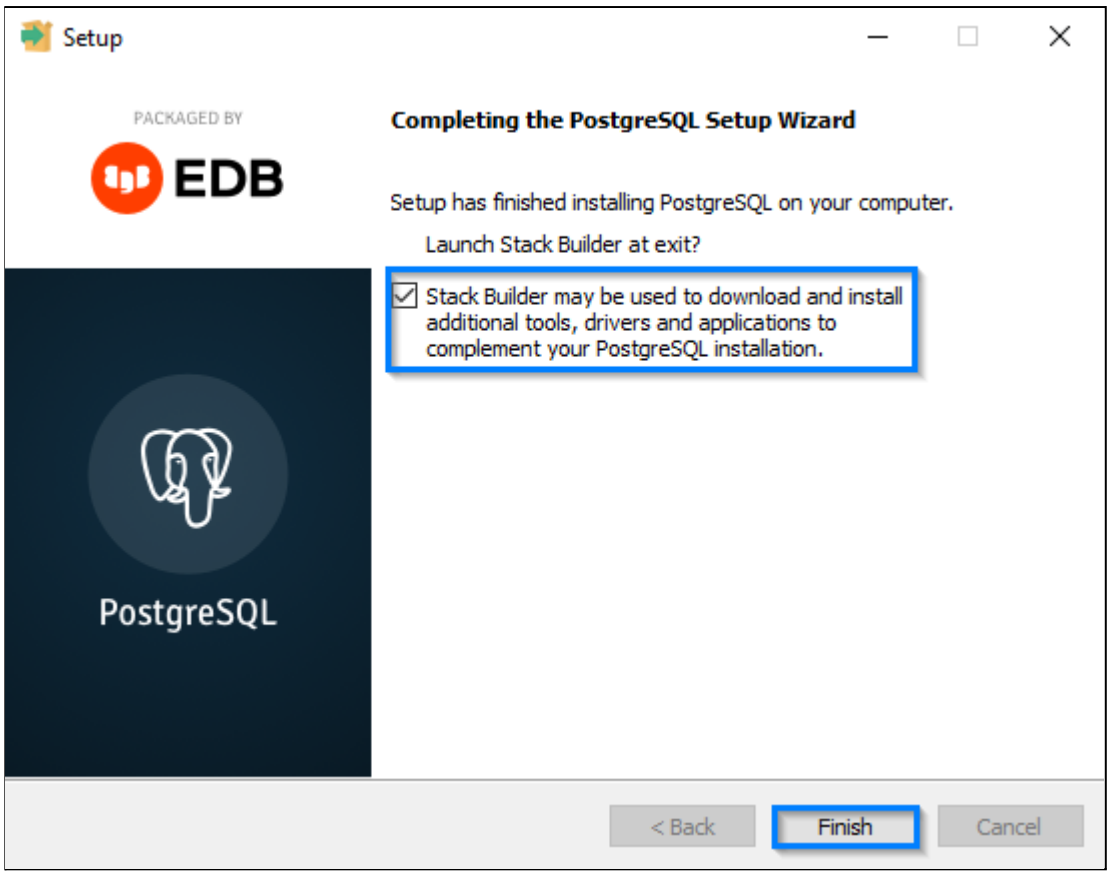



Przeglądamy podsumowanie i jeśli wszystko się zgadza wybieramy opcję dalej.

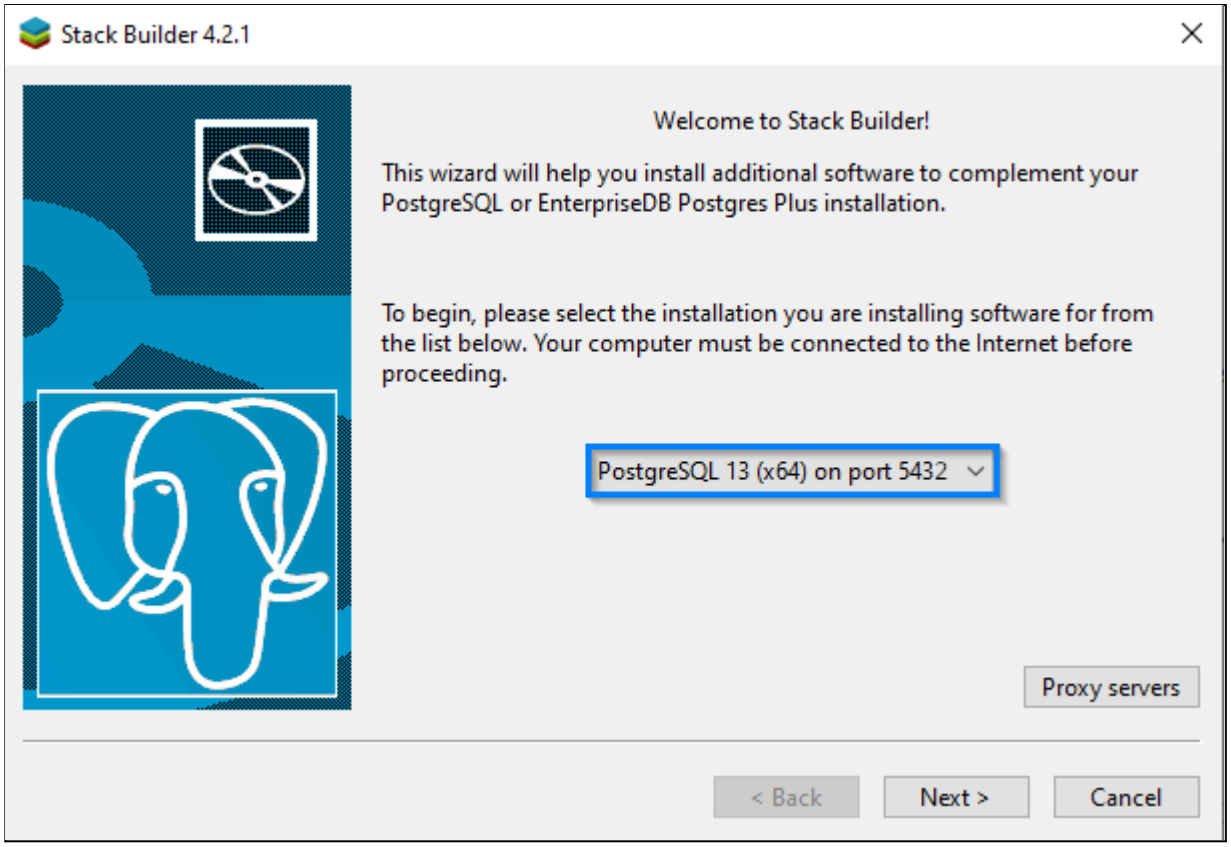



Instalator sprawdzi ustawienia i jeśli wszystko przebiegło prawidłowo wyświetli informację, że może przystąpić do instalacji, wybieramy dalej.

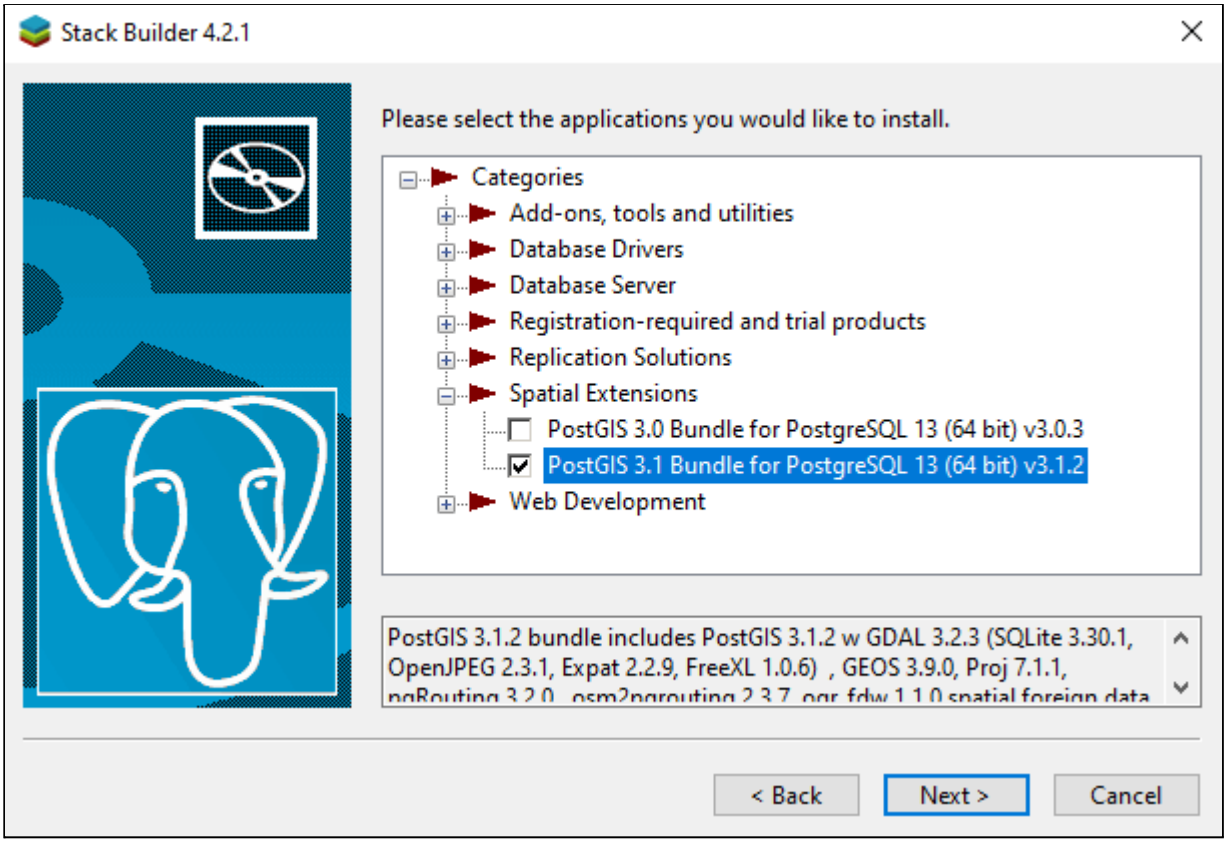
Po zainstalowaniu bazy danych instalator zapyta czy ma uruchomić StackBuilder - narzędzie pozwalające na instalowanie dodatkowych sterowników czy rozszerzeń. Sprawdzamy czy okienko wyboru jest zaznaczone i wybieramy Finish.




W kolejnym kroku wyświetlony zostanie StackBuilder - wybieramy bazę danych do której będziemy instalowali dodatki. W oknie wyboru wybieramy PostgreSQL 13 on port 5432 i klikamy przycisk Next.

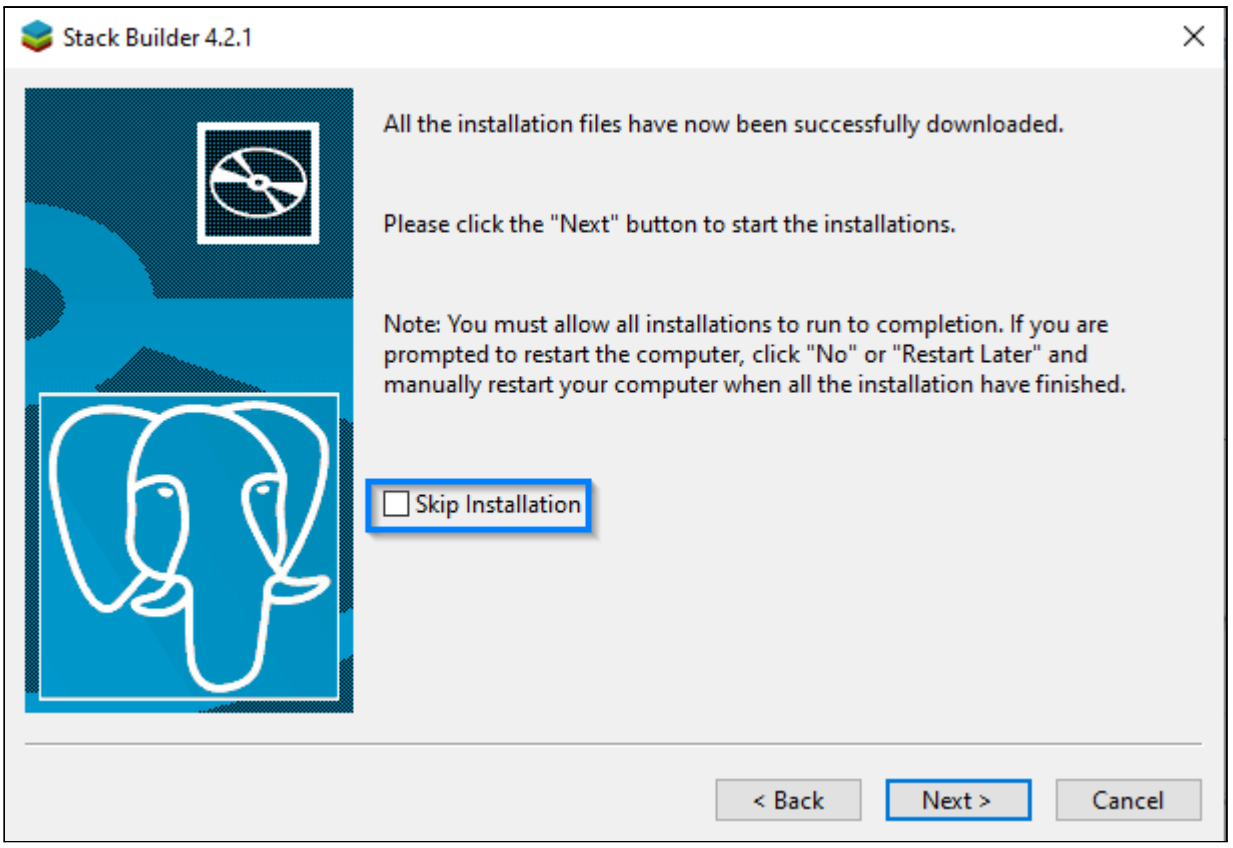



W kolejnym kroku rozwijamy gałąź Spatial extensions i zaznaczamy okno wyboru przy pozycji PostGIS 3.1

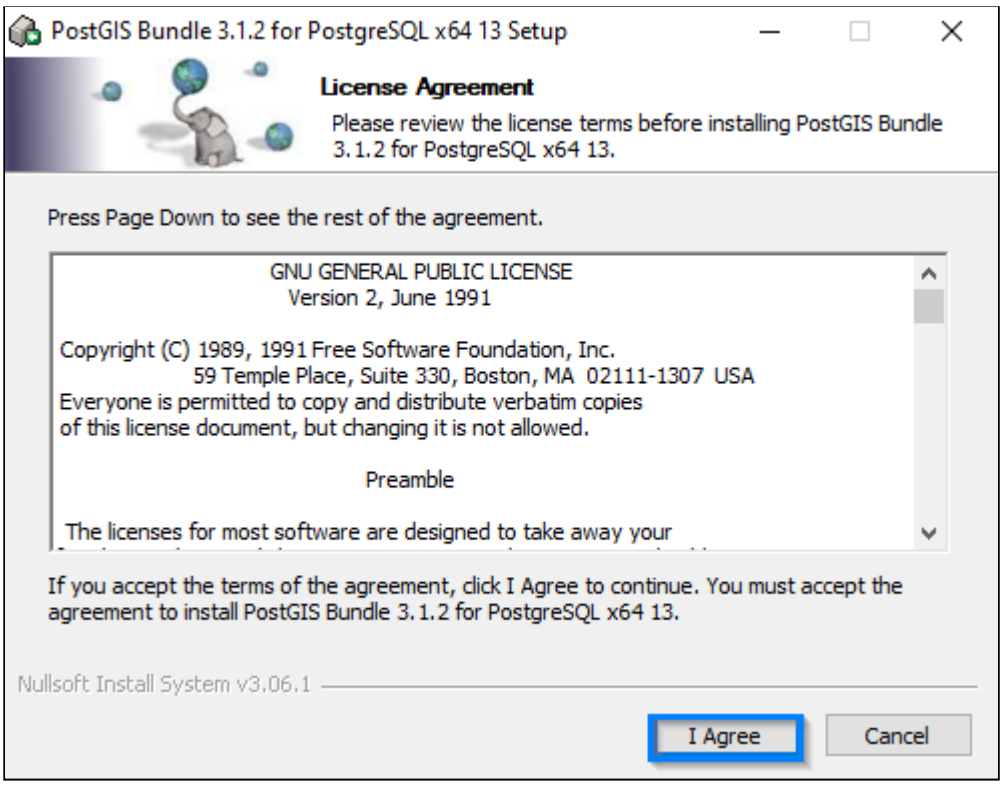



Kolejny krok to podsumowanie i wybór folderu w którym zapisane zostaną pobrane rozszerzenia. Domyślnie jest to C:\Users\nazwa użytkownika.

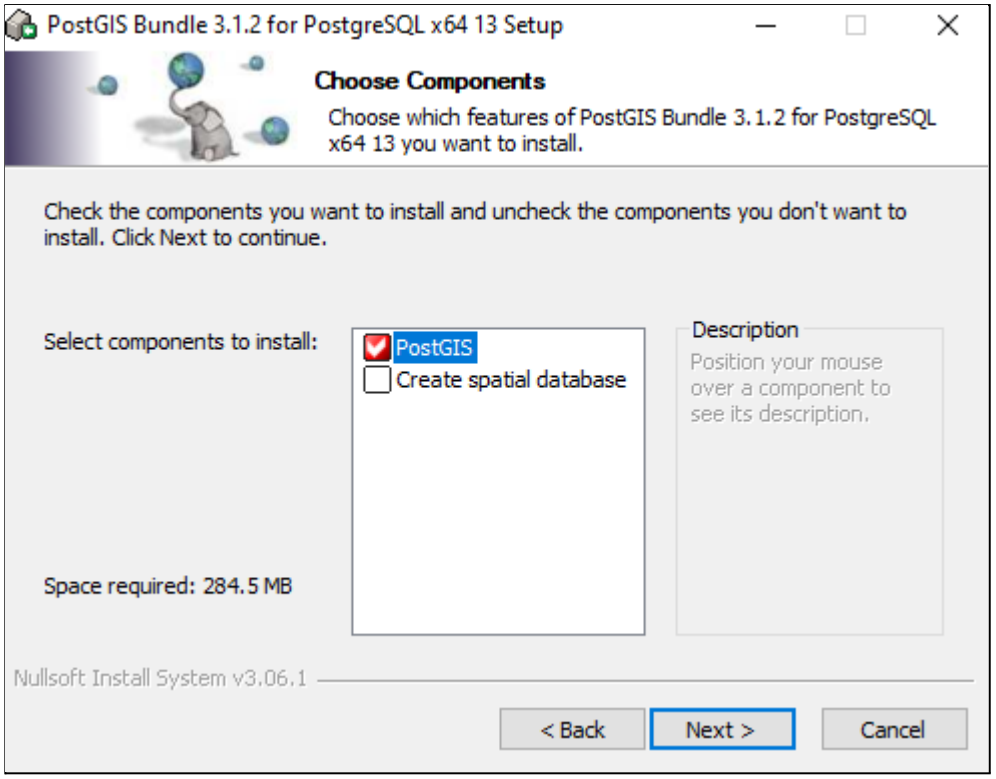
W ostatnim kroku upewniamy się, że okienko wyboru Skip Installation nie jest zaznaczone i wybieramy Next.



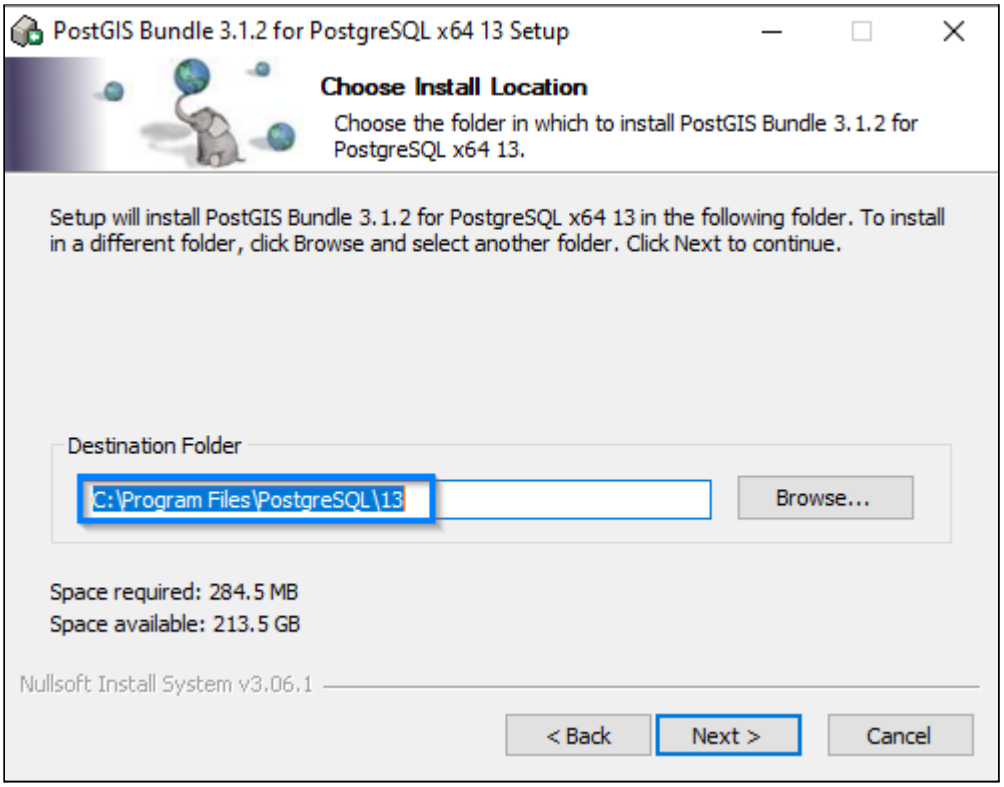

Po pobraniu pakietu instalacyjnego PostGIS zostanie on uruchomiony. W pierwszym kroku potwierdzamy zapoznanie się z licencją.



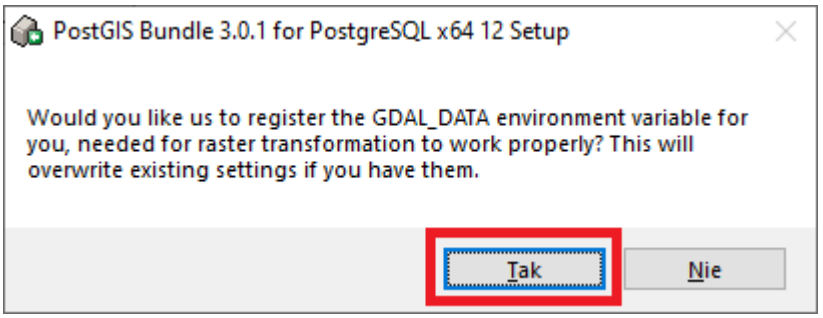
W drugim kroku sprawdzamy czy opcja PostGIS jest zaznaczona i wybieramy Next.



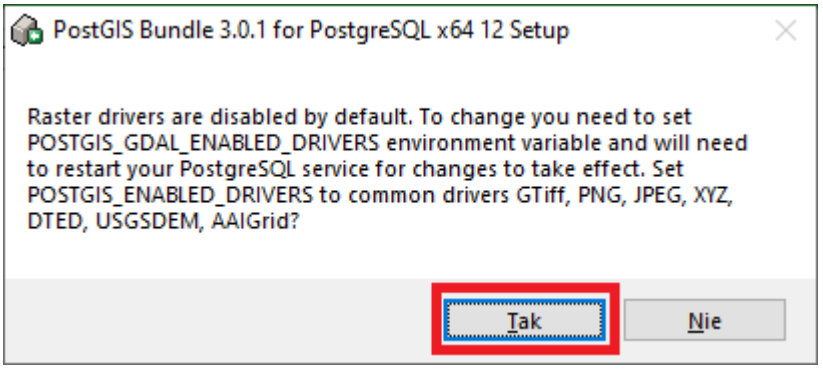

W kolejnym wybieramy lokalizację instalacji. Jeśli domyślna ścieżka instalacji bazy danych została zmieniona tutaj należy również podać zmienioną ścieżkę.



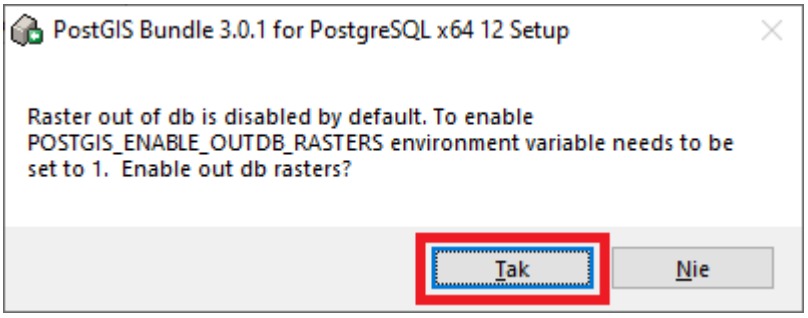
Następnie potwierdzamy zarejestrowanie zmiennej środowiskowej GDAL_DATA



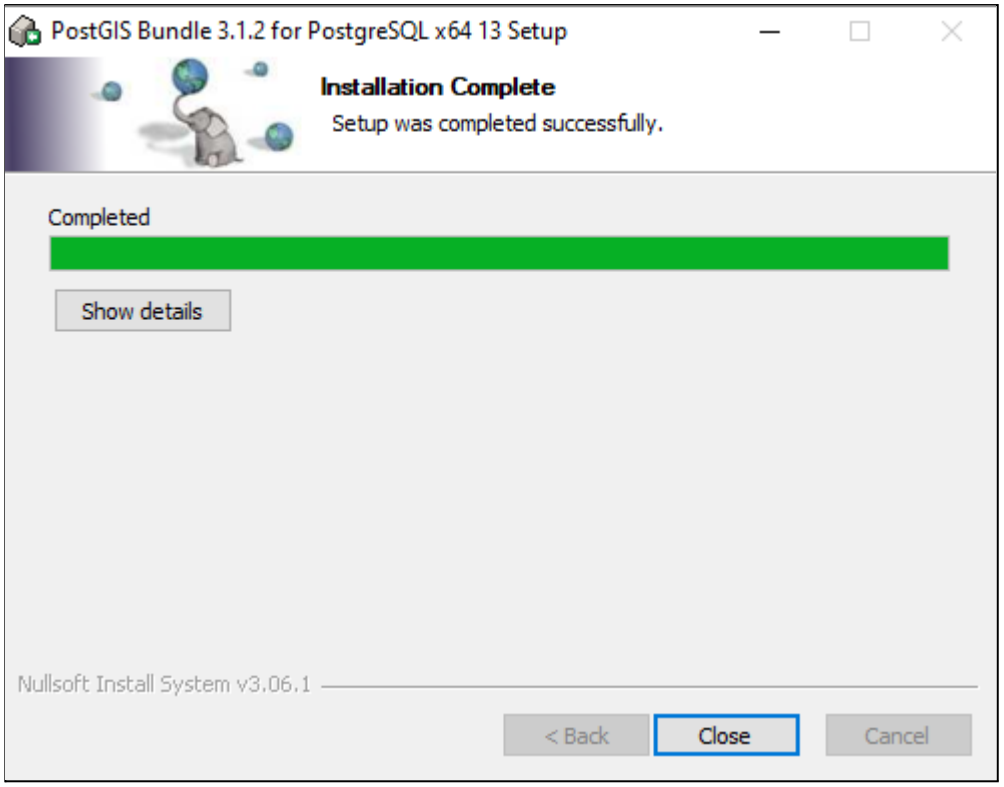

Włączamy sterowniki GDAL do danych rastrowych




i zezwalamy na eksporty rastrów z bazy danych




Jeśli instalacja zakończy się poprawnie powinniśmy zobaczyć okno podsumowania.

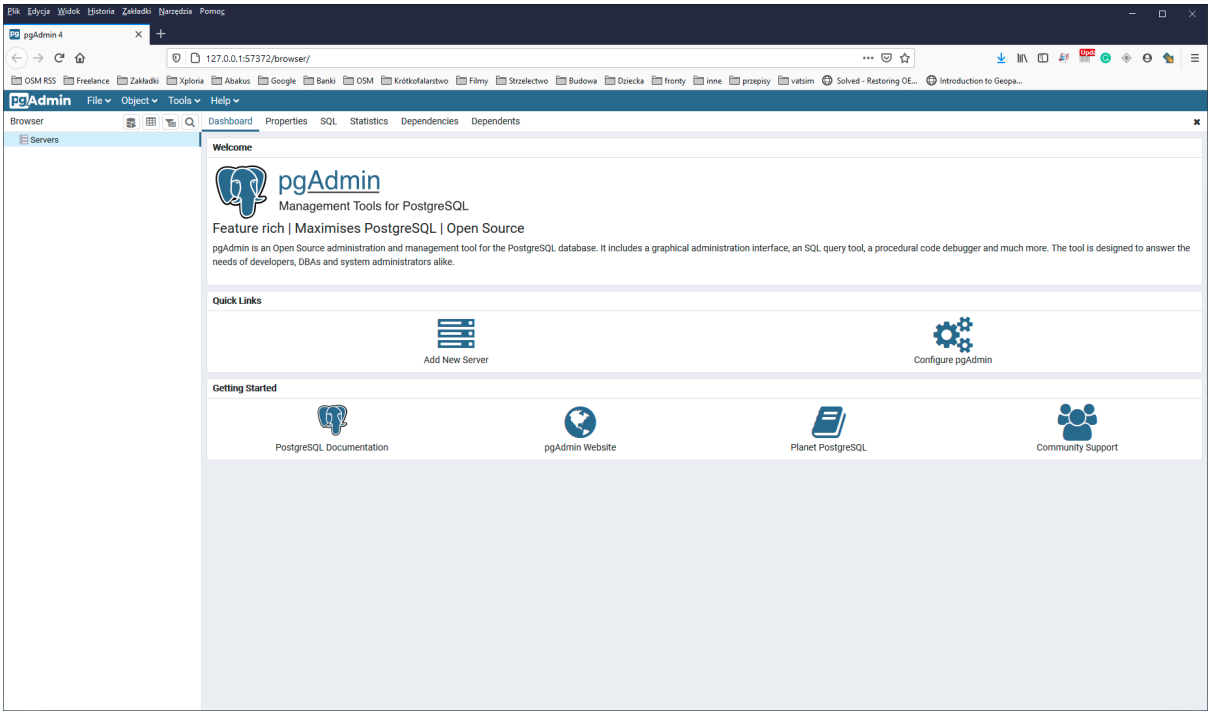

PostGIS Bundle 3.1.2 for PostgreSQL x64 13 Setup
Installation Complete
Setup was completed successfully.
Completed
Show details
Nullsoft Install System v3.06.1
< Back
Close
Cancel

# Konfiguracja aplikacji pgAdmin

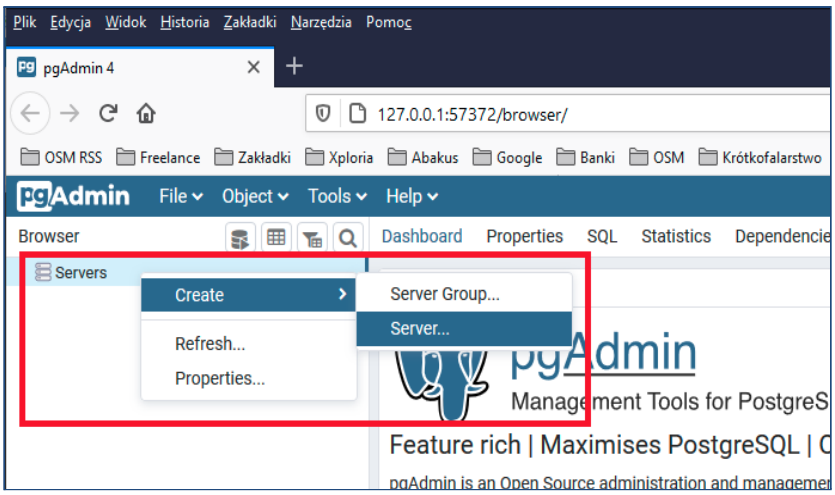
pgAdmin to aplikacja pozwalająca na zarządzanie bazą danych oraz danymi w tej bazie.
Do poprawnego działania konieczne jest wstępne jej skonfigurowanie.

Aplikację odnajdujemy w menu Start systemu Windows i uruchamiamy.

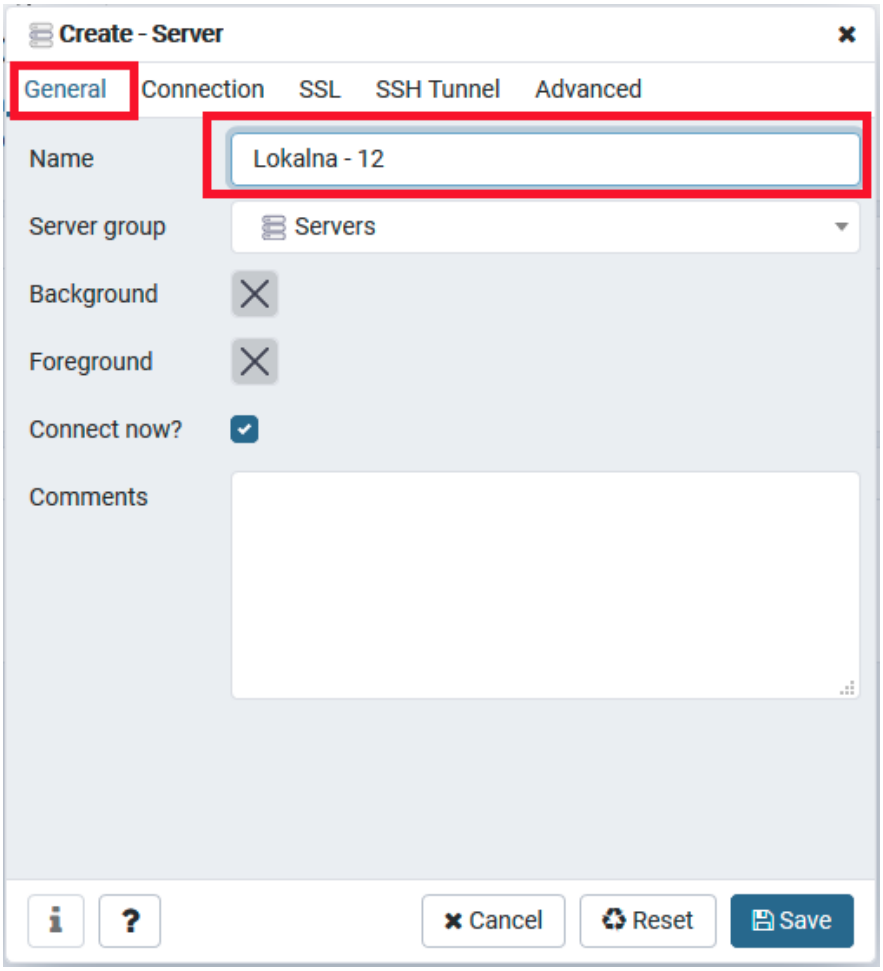
Uruchomiona zostanie przeglądarka internetowa, a przy pierwszym uruchomieniu aplikacja poprosi o wpisanie hasła zabezpieczającego - sugerujemy ponownie użycie hasła `gis`. Wyświetlone zostanie okno aplikacji:



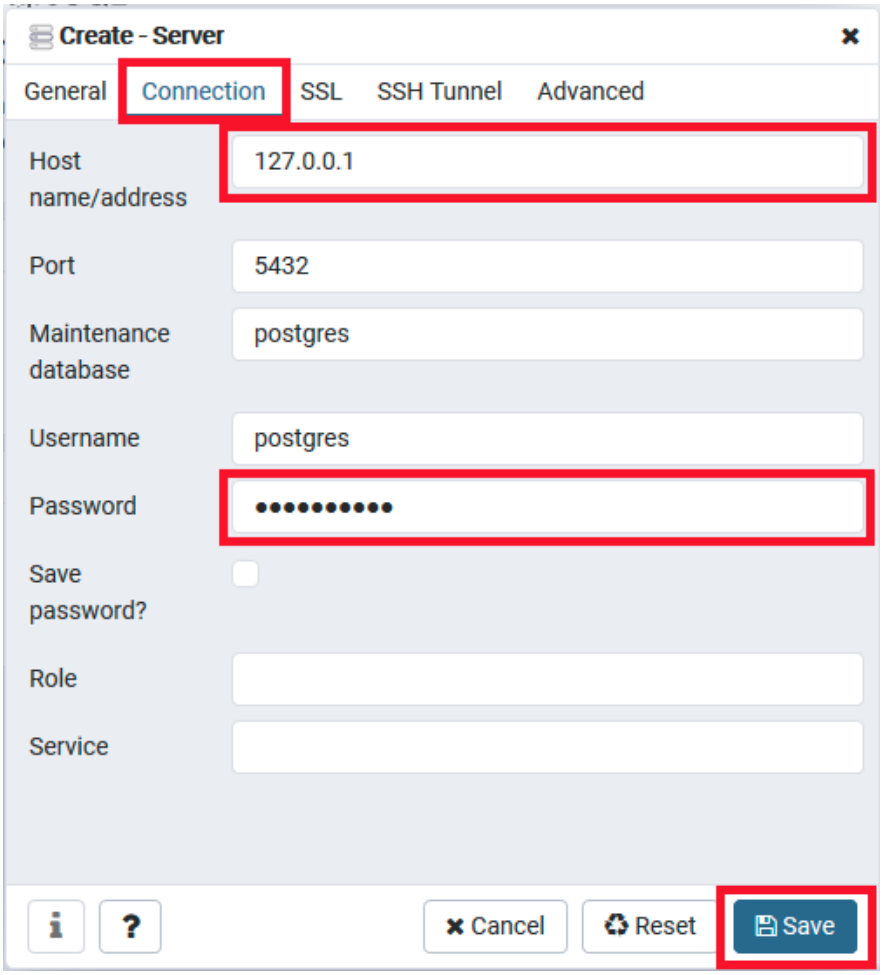

Aplikacja pozwala na zarządzanie równocześnie wieloma bazami danych umieszczonymi na wielu serwerach. W kolejnym kroku połączymy aplikację z bazą danych zainstalowaną w poprzednim kroku szkolenia. W tym celu klikamy prawym przyciskiem myszy na pozycji servers i wybieramy `create > server`




Wyświetlone zostanie okno konfiguracji połączenia z serwerem. W zakładce `General` podajemy nazwę serwera - na potrzeby szkolenia może to być `Lokalna - 13`




Przechodzimy do zakładki `Connection` gdzie podajemy adres serwera `127.0.0.1` oraz hasło użytkownika postgres które zostało utworzone podczas instalacji - zgodnie z zaleceniami powinno to być hasło `gis`.




Jeśli podane dane były prawidłowe aplikacja powinna połączyć się do serwera bazy i wyświetlić w drzewie dostępne na tym serwerze bazy danych.

Jeśli udało nam się wykonać wszystkie powyższe kroki oznacza to, że:

- serwer bazy danych został poprawnie zainstalowany.
- mamy prawidłowo skonfigurowaną aplikację pgAdmin

# Narzędzie PSQL

psql jest narzędziem pozwalającym na dostęp i zarządzanie bazą danych dostępnym z linii komend.

Narzędzie to, wbrew pierwszemu wrażeniu, bardzo często okazuje się przydatne lub wręcz niezbędne. Możemy dzięki niemu zarządzać bazą danych na systemach operacyjnych pozbawionych środowiska graficznego. Pozwala ono również na uruchamianie poleceń SQL, lub nawet całych skryptów zapisanych w plikach, bezpośrednio z wiersza poleceń. Możemy dzięki temu tworzyć skrypty powłoki (.sh) lub wiersza poleceń (.bat) automatyzujące naszą pracę, możemy nawet uruchamiać je z terminarza systemowego (scheduler, cron). Psql potrafi również zapisać wynik zapytania w postaci tabeli HTML co pozwala na tworzenie prostych raportów dostępnych przez przeglądarkę internetową.

Pełny opis narzędzia wraz z przykładami użycia znajdziemy w dokumentacji PostgreSQL.

Uruchomienie:

po otwarciu menu Start wpisać "cmd"
odnaleźć w systemie plik psql.exe (np. C:\PostgreSQL\13\bin) i przeciągnąć go do okna konsoli

Wpisać parametry:

- -h localhost
- -d postgres
- -U postgres

Podczas wpisywania hasła nie pokazują się żadne znaki - jest to normalne.

Komendy psql - zatwierdzane przez wciśnięcie Enter:

- **\l** - wyświetlenie listy baz danych w systemie
- **\l+** - jak wyżej, ale z podaniem rozmiaru na dysku
- **\connect szkolenie_db** - przełączenie się na bazę szkolenie_db
- **\dt** - wyświetlenie listy tabel w bazie
- **\dt+** - jak wyżej, ale z podaniem rozmiaru na dysku
- **\du** - wyświetlenie listy użytkowników
- **\dv** - wyświetlenie listy widoków
- **\di** - wyświetlenie listy indeksów

Zapytanie może być wpisane w wielu wierszach - wiersz kończy się poprzez wciśnięcie Enter.

Zapytanie kończy się znakiem ; i wciśnięcie Enter

```
szkolenie=# select *
szkolenie-# from geometry_columns;
```

Pozostałe przydatne komendy:

- **\timing** - pokazuje czas wykonania zapytania
- **\d <nazwa tabeli>** - pokazuje dostępne kolumny w tabeli
- **\a** - wyłącza / włącza justowanie tabeli
- **\x** - pokazuje kolumny pionowo zamiast poziomo
- **\q** - koniec sesji

# Wprowadzenie do SQL

**SQL (ang. Structured Query Language)** jest to strukturalny język zapytań używany do tworzenia i modyfikowania baz danych oraz danych w nich umieszczonych

Język SQL jest językiem deklaratywnym, oznacza to, że decyzję o sposobie przechowywania i pobrania danych pozostawia się systemowi zarządzania bazą danych.

SQL został opracowany w latach 70. w firmie **IBM** i stał się standardem w komunikacji z serwerami relacyjnych baz danych. Wiele współczesnych systemów relacyjnych baz danych używa do komunikacji z użytkownikiem SQL, dlatego potocznie mówi się, że **korzystanie z relacyjnych baz danych to korzystanie z SQL-a**.

W **1986** SQL stał się oficjalnym standardem, wspieranym przez Międzynarodową Organizację Normalizacyjną (ISO) i jej członka, Amerykański Narodowy Instytut Normalizacji (ANSI).

W 2003 przedstawiono **SQL:2003** – nowy standard języka SQL. Jest to w zasadzie poprawione **SQL:1999** z wyjątkiem części SQL/XML oraz kilku dodatkowych właściwości.

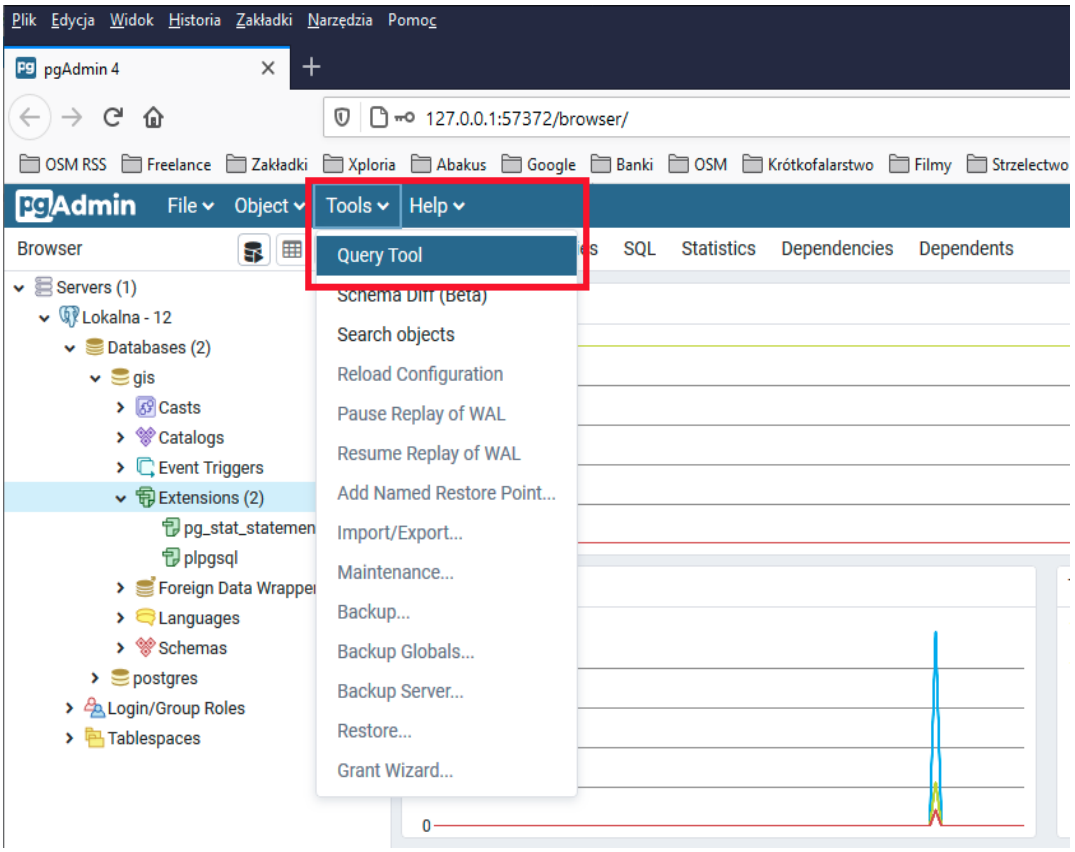
# Typy danych

PostgreSQL udostępnia ponad 50 różnych typów danych - poniżej najważniejsze z nich

- Typy tekstowe
    - Typ **CHAR**: tekst stałej długości (np. suma kontrolna)
    - Typ **VARCHAR**: tekst zmiennej długości, opcjonalny typmod dla ograniczenia maksymalnej długości (alias - CHARACTER VARYING)
    - Typ **TEXT**: tekst dowolnej długości
    - Typ **XML**: dokument XML - musi być well-formed
    - Typ **TSVECTOR**: dla wyszukiwania pełnotekstowego
- Typy liczbowe
    - Typ **INTEGER**: liczba całkowita 32 bit (alias - INT4) (max: 2 147 483 647)
    - Typ **BIGINT**: liczba całkowita 64 bit (alias - INT8) (max: 9 223 372 036 854 775 807)
    - Typ **REAL**: liczba zmiennoprzecinkowa pojedynczej precyzji (6 miejsc po przecinku)
    - Typ **DOUBLE PRECISION**: liczba zmiennoprzecinkowa podwójnej precyzji (alias - **FLOAT**) (15 miejsc po przecinku)
    - Typ **NUMERIC**: liczba dziesiętna o zdefiniowanej precyzji (alias - DECIMAL)
- Typy daty i czasu
    - Typ **DATE**: tylko data
    - Typ **TIME**: tylko czas (może być WITH TIME ZONE i WITHOUT TIME ZONE)
    - Typ **TIMESTAMP**: data i czas - w wariantach WITH TIME ZONE i WITHOUT TIME ZONE
- Inne typy
    - Typ **BYTEA**: dane binarne
    - Typ **BOOLEAN**: wartość prawda/fałsz
    - Typ **HSTORE**: pary klucz-wartość
    - Typy **JSON** i **JSONB**: obiekty JSON (można użyć do wpisania wielu wartości w jedno pole)
    - Typ **OID**: identyfikator dla obiektów systemowych
    - Pseudo-typ **SERIAL**: kolumna INTEGER+sekwencja+wartość domyślna, używany do nadawania identyfikatorów
    - Typy **POINT**, **PATH**, **POLYGON**: typy grafiki wektorowej 2D - **nie używać do danych GIS!**
- Typy PostGIS
    - Typ **GEOMETRY**: najczęściej używany. Dowolny układ współrzędnych, wszystkie obliczenia są wykonywane na współrzędnych płaskich, wyniki zwracane w jednostkach układu. Opcjonalny typmod na typ geometrii (POINT, LINESTRING, POLYGON…) oraz układ współrzędnych
    - Typ **GEOGRAPHY**: dla współrzędnych geograficznych długość-szerokość, obliczenia wykonywane metodami geodezji wyższej, wyniki i parametry podawane są w metrach.

## Wykonywanie zapytań

Wszystkie zapytania uruchamiane w ramach tego szkolenia, jeśli instruktor nie wskaże inaczej, będziemy wykonywali za pomocą narzędzia **Query Tool** dostępnego w menu **tools**.




## Konwersja typów danych

Z uwagi na obecność w systemie bazy danych wielu rodzajów danych o dość zbliżonej charakterystyce, w przypadku przenoszenia, porównywania lub łączenia ze sobą danych przechowywanych w różnych obiektach może zajść potrzeba dokonania ich konwersji z jednego typu na drugi. Zasady dokonywania konwersji pomiędzy różnymi typami danych są mocno zbliżone do zasad znanych z popularnych języków programowania, takich jak C++ czy C#. Choć dokładne kroki podejmowane przez system bazy danych w celu dokonania takiej konwersji są dosyć rozbudowane i często złożone, to w ogólnym przypadku system będzie dążył do stanu, w którym konwersja nie spowoduje żadnej utraty danych lub spowoduje utratę jedynie najmniej znaczącej ich części. Przykładowo jeśli zachodzi konieczność porównania liczby typu INT z liczbą typu BIGINT, wówczas zamiast obcinać większą liczbę w taki sposób, aby dało się ją zareprezentować typem INT, system skonwertuje liczbę typu INT na większy typ BIGINT, wypełniając ją przy tym wiodącymi zerami, co w żaden sposób nie wpłynie na wartość tej liczby.

W PostgreSQL istnieją dwa typy konwersji: jawne (ang. explicit) oraz niejawne (ang. implicit).

Konwersje niejawne wykonywane są przez silnik bazy danych automatycznie, bez udziału i wiedzy użytkownika. Przykładowo gdy wartość typu SMALLINT porównywana jest z wartością typu INT, w celu porównania wartość SMALLINT zostanie uprzednio skonwertowana na typ INT.

Konwersje jawne wykonywane są na wyraźne polecenie użytkownika za pośrednictwem wywołanej przez niego funkcji CAST.

Jedna z podstawowych zasad projektowania baz danych głosi, że konwersji należy wystrzegać się w ogóle, przy czym jeśli już jakaś konwersja jest nieunikniona, warto zadbać, aby była ona konwersją jawną, zdefiniowaną przez nas samych.

W PostgreSQL istnieje dodatkowy operator pozwalający na konwersje danych "::", więc oba poniższe polecenia wykonają dokładnie tę samą operację - przekonwertują tekst '12' do wartości liczbowej

```
SELECT CAST('12' AS integer);
SELECT '12'::integer;
```

# Procedury składowane (funkcje)

Funkcja jest fragmentem kodu - w języku SQL bądź innym obsługiwanym przez bazę - nadającym się do wielokrotnego użytku.

W PostgreSQL funkcje nie muszą być "czyste", tj. przekształcać zbioru argumentów w zbiór wartości bez pozostawiania trwałych zmian w bazie.

Funkcja w PostgreSQL jest definiowana przez nazwę oraz liczbę i typy argumentów - w razie pomyłki w argumentach, komunikat o błędzie będzie brzmiał function does not exist.

Wyróżniamy funkcje skalarne (operujące na pojedynczych wierszach) i agregujące (operujące na grupach wierszy).

Oprócz wbudowanych funkcji, których pełną listę wraz z przykładowymi zastosowaniami można znaleźć w dokumentacji PostgreSQL, użytkownik może tworzyć własne.

## Funkcje tekstowe

Funkcja lower zmienia wielkość liter w podanym tekście na małe.

```
SELECT lower('Kowalski');
-- kowalski
```

Funkcja upper zmienia wielkość liter w podanym tekście na wielkie.

```
SELECT upper('Kowalski');
-- KOWALSKI
```

Funkcja replace przyjmuje 3 argumenty: tekst do zmiany, fragment podlegający zmianie oraz fragment, który ma zostać wstawiony.

```
SELECT replace('Kowalski', 'ski', 'cze');
-- Kowalcze
```

Funkcja split_part dzieli tekst na fragmenty według podanego separatora, oraz zwraca żądany fragment.

```
SELECT split_part('123-456-789', '-', 2);
-- 456
```

Funkcja trim obcina żądane znaki na początku, końcu lub z obu stron tekstu (najczęstsze zastosowanie: spacje na końcu)

```
SELECT trim(trailing ' ' from 'Murzynowo Kościelne ');
-- 'Murzynowo Kościelne'
```

Funkcja substring ekstrahuje fragment tekstu według położenia pierwszego i ostatniego znaku.

```
SELECT substring('abcdefghijkl' from 1 for 5);
-- abcde
```

## Funkcje liczbowe

Funkcja ceil zaokrągla liczbę dziesiętną w górę, floor - w dół, round - do końcówki 5 w dół, powyżej w górę.

```
SELECT ceil(5.5);
-- 6

SELECT floor(5.5);
-- 5
```

```
SELECT round(5.5);
-- 6
```

Funkcja round na typie numeric z podaniem drugiego argumentu - precyzji umożliwia zaokrąglenie do żądanej precyzji.

```
SELECT round(5.52765, 3);
-- 5.528
```

Funkcja log oblicza logarytm przy podstawie 10, ln - przy podstawie e.

```
SELECT log(3);
-- 0.47712125471966244

SELECT ln(3);
-- 1.0986122886681098
```

Funkcja sqrt oblicza pierwiastek kwadratowy, cbrt - sześcienny, power - podnosi liczbę do potęgi.

```
SELECT sqrt(4);
-- 2

SELECT cbrt(27);
-- 3

SELECT power(2,4);
-- 16
```

Funkcja abs oblicza wartość bezwzględną, sign - znak liczby.

```
SELECT abs(-3);
-- 3

SELECT abs(3);
-- 3

SELECT sign(-3);
-- -1
```

Funkcja radians przelicza stopnie na radiany, degrees - radiany na stopnie.

```
SELECT radians(180);
-- 3.141592653589793

SELECT degrees(pi()/2);
-- 90
```

### Funkcje agregujące

Funkcja max zwraca największą wartość ze zbioru.

```
SELECT max(a)
FROM unnest(ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]) as a;
-- 9
```

Funkcja min zwraca najmniejszą wartość ze zbioru.

```
SELECT min(a)
FROM unnest(ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]) as a;
-- 0
```

Funkcja sum oblicza sumę wartości ze zbioru.

```
SELECT sum(a)
FROM unnest(ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]) as a;
--45
```

Funkcja count zwraca liczbę wartości w zbiorze.

```
SELECT count(*)
FROM unnest(ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]) as a;
--10
```

## Elementy języka SQL

Wyróżniamy 3 główne grupy poleceń SQL:

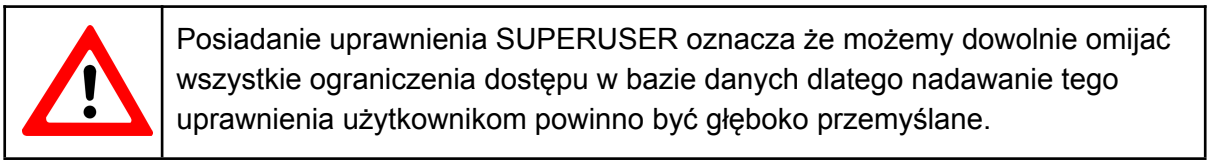
- DML - Data Manipulation Language
  Polecenia wyszukiwania, dodawania, edycji i usuwania danych
- DDL - Data Definition Language
  Polecenia tworzenia, zmiany i usuwania struktur danych (tabel, kolumn…)
- DCL - Data Control Language
  Polecenia kontroli dostępu do danych (tylko bazy klient-serwer)

# Tworzenie szkieletu bazy danych

## Tworzenie użytkowników

W PostgreSQL użytkownika (rolę) może utworzyć inny użytkownik posiadający uprawnienie CREATEROLE lub SUPERUSER.

Nowego SUPERUSER-a może utworzyć tylko inny SUPERUSER.




Użytkowników w bazie danych PostgreSQL możemy utworzyć na dwa sposoby - za pośrednictwem polecenia CREATE USER (CREATE ROLE), lub aplikacji createuser która jest wraperem tego polecenia.

Role tworzone są na poziomie klastra PostgreSQL dlatego są dostępne we wszystkich bazach danych w klastrze.

Rola PostgreSQL może być rozważana zarówno jako użytkownik jak i jako grupa użytkowników równocześnie - w zależności od tego w jaki sposób jest ona używana.

Przykładowego użytkownika (rolę) w klastrze możemy utworzyć za pomocą polecenia:

```
CREATE USER szkolenie_usr WITH LOGIN PASSWORD 'szkolenie_pass';
```

Jedynym wymaganym parametrem polecenia jest nazwa - możemy utworzyć użytkownika bez hasła.
Ważniejsze opcje polecenia:

- SUPERUSER | NOSUPERUSER
  Decyduje czy tworzony użytkownik będzie superuser-em
- CREATEDB | NOCREATEDB
  Decyduje czy tworzony użytkownik będzie mógł tworzyć bazy danych
- CREATEROLE | NOCREATEROLE
  Decyduje czy tworzony użytkownik będzie mógł tworzyć innych użytkowników

- LOGIN | NOLOGIN
  Decyduje czy tworzony użytkownik będzie mógł logować się do baz danych
- [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  Pozwala na podanie hasła

Wszystkie opcje polecenia możemy znaleźć w dokumentacji PostgreSQL. Pozwalają one między innymi na tworzenie hierarchii czy grup użytkowników, określanie limitów ważności konta czy limitów jednoczesnych połączeń.

Analogiczny efekt do poprzedniego zapytania sql możemy uzyskać za pomocą aplikacji ***createuser***:

```
createuser -U postgres -W -l -P szkolenie_usr1
```

Aplikacja po uruchomieniu poprosi o podanie hasła dla nowo tworzonego użytkownika :

```
Enter password for new role: szkolenie_pass
Enter it again: szkolenie_pass
```

Kompletny zestaw parametrów przyjmowanych przez aplikację można uzyskać za pomocą przełącznika ***-?*** lub ***--help***, oraz w dokumentacji PostgreSQL.

## Modyfikacja użytkowników

Użytkowników w klastrze PostgreSQL możemy modyfikować za pomocą polecenia SQL ***ALTER ROLE***. Poniższe polecenie nada użytkownikowi ***szkolenie_usr1*** uprawnienia do tworzenia nowych ról oraz baz danych w klastrze:

```
ALTER ROLE szkolenie_usr1 WITH CREATEDB CREATEROLE;
```

Polecenie to pozwala również na zmianę nazwy roli w bazie danych

```
ALTER ROLE szkolenie_usr1 RENAME TO szkolenie_usr2;
```

## Usuwanie użytkowników

Analogicznie do tworzenia ról w klastrze usuwać role możemy zarówno za pomocą polecenia SQL DROP ROLE

```
DROP ROLE IF EXISTS szkolenie_usr2;
```

jak i za pomocą aplikacji wiersza poleceń ***dropuser*** obudowującej to polecenie SQL

```
dropuser --if-exists szkolenie_usr2;
```

W obu przypadkach możemy użyć dodatkowej klauzuli ***IF EXISTS*** lub przełącznika ***--if-exists***, który spowoduje, że polecenie nie zwróci błędu jeśli usuwany użytkownik nie istnieje.

## Kontrola dostępu do baz danych (pg_hba.conf)

Utworzenie użytkowników w klastrze bazy danych nie determinuje wszystkich opcji zarządzania dostępem użytkowników do klastra lub konkretnych baz danych. Dostępem tym możemy zarządzać za pomocą pliku pg_hba.conf, który znajduje się w folderze instalacji klastra. Przykładowy fragment pliku wygląda następująco:

```
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER        ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host   all          all          127.0.0.1/32            md5
host   all          all          10.210.0.0/24           md5
host   all          ex_rates_usr 0.0.0.0/0               md5
host   all          veo          0.0.0.0/0               md5
```

A poszczególne linie oznaczają, że:

```
local   all             postgres                                peer
```

Dozwolone jest logowanie się użytkownika ***postgres*** do wszystkich baz danych (***all***), jeśli przedstawia się on nazwą użytkownika systemu linux (***peer***) i łączy się za pośrednictwem unix-socket (***local***).

```
local   all             all                                     peer
```

Dozwolone jest logowanie się dowolnego użytkownika (***all***) do wszystkich baz danych (***all***), jeśli przedstawia się on nazwą użytkownika systemu linux (***peer***) i łączy się za pośrednictwem unix-socket (***local***).

```
host   all          all          127.0.0.1/32            md5
```

Dozwolone jest logowanie się dowolnego użytkownika (***all***) do każdej bazy danych (***all***) jeśli łączy się za pomocą połączenia sieciowego (***host***) z adresu pętli lokalnej (***127.0.0.1/32***) i uwierzytelnia się za pomocą loginu i hasła (***md5***)

Modyfikacji ustawień dokonujemy dodając kolejne linie w pliku.

## Tworzenie bazy danych

Analogicznie do tworzenia i usuwania ról bazy danych w klastrze możemy tworzyć zarówno za pomocą polecenia SQL ***CREATE DATABASE*** jak i za pomocą aplikacji ***createdb*** obudowującej to polecenie.

Aby utworzyć bazę danych użytkownik musi być SUPERUSER-em, lub posiadać uprawnienie CREATEDB

Aby utworzyć nową bazę danych uruchamiamy aplikację z następującymi parametrami:

```
$ createdb --username=postgres szkolenie_db_p
```

Jeśli chcemy od razu określić właściciela tej bazy dodajemy parametr --owner

```
$ createdb --username=postgres --owner=szkolenie_usr szkolenie_db
```

lub łączymy się do serwisowej bazy danych o nazwie postgres i wykonujemy polecenie:

```
CREATE DATABASE szkolenie_db1 WITH OWNER szkolenie_usr;
```

Na uwagę zasługuje również opcja TEMPLATE, która pozwala na wskazanie bazy danych na podstawie której będziemy tworzyli nową. Jest to jedna z najszybszych opcji wykonania kopii bazy.

## Modyfikacja definicji bazy danych

Do modyfikacji ustawień głównych bazy danych służy polecenie SQL ALTER DATABASE.

Aby modyfikować ustawienia główne bazy danych użytkownik musi być SUPERUSER-em, lub posiadać uprawnienie CREATEDB

Dzięki poniższemu poleceniu możemy zabronić dostępu do bazy danych

```
ALTER DATABASE szkolenie_db1 WITH ALLOW_CONNECTIONS=false;
```

Ograniczyć ilość jednoczesnych połączeń

```
ALTER DATABASE szkolenie_db1 WITH CONNECTION LIMIT 10;
```

Zmienić nazwę bazy danych

```
ALTER DATABASE szkolenie_db1 RENAME TO szkolenie_db2;
```

Zmienić właściciela bazy

```
ALTER DATABASE szkolenie_db2 OWNER TO postgres;
```

Lub ustawić inne parametry konfiguracyjne.

```
ALTER DATABASE szkolenie_db2 SET log_min_messages='DEBUG5';
```

Więcej informacji na temat samych parametrów konfiguracyjnych zawartych będzie w rozdziale poświęconym ustawieniom i strojeniu bazy danych.

Zmiana nazwy bazy danych do której jesteśmy aktualnie podłączeni nie jest możliwa. Aby to zrobić należy podłączyć się do innej bazy danych. Jeśli w klastrze mamy tylko jedną bazę danych łączymy się do bazy serwisowej ***postgres***.

## Usuwanie bazy danych

Usuwanie bazy danych również możliwe jest za pośrednictwem polecenia DROP DATABASE

```
DROP DATABASE szkolenie_db2;
```

oraz aplikacji dropdb

```
$ dropdb szkolenie_db2
```

Aby modyfikować ustawienia główne bazy danych użytkownik musi być SUPERUSER-em, lub posiadać uprawnienie CREATEDB

UWAGA !!!
Polecenie to bezpowrotnie usuwa bazę danych wraz z całą jej zawartością

## Wprowadzenie do schematów

Klaster PostgreSQL zawiera jedną lub więcej nazwanych baz danych. Jedynie użytkownicy i grupy są definiowane na poziomie klastra i współdzielone we wszystkich bazach danych. Każde połączenie do bazy danych ma dostęp jedynie do danych w tej konkretnej bazie do której połączenie zostało nawiązane.

Każda baza danych zawiera jeden (***public***) lub więcej schematów, które zawierają tabele oraz inne nazwane obiekty bazodanowe takie jak widoki, funkcje, typy danych czy operatory.

Ta sama nazwa obiektu (tabeli, funkcji) może zostać użyta w wielu schematach nie powodując przy tym konfliktów.

W przeciwieństwie do baz danych schematy nie są trwale rozdzielone - każdy użytkownik podłączony do bazy danych ma dostęp równocześnie do wszystkich jej schematów - jeśli uprawnienia na poziomie obiektów nie stanowią inaczej.

Jest przynajmniej kilka powodów dla których warto używać schematów:

- aby umożliwić kilku użytkownikom używanie jednej bazy danych bez przeszkadzania sobie nawzajem
- aby grupować obiekty bazy danych w logiczne grupy, co pozwoli łatwiej nimi zarządzać
- aby umieścić w nich dane zewnętrznych aplikacji wykluczając w ten sposób kolizje nazw

Idea schematów jest analogiczna do idei folderów (katalogów) w systemach plików - z tą jednak różnicą że schematów nie można zagnieżdżać.

Odnosząc się do obiektów w schematach należy używać ich kwalifikowanej nazwy składającej się z nazwy schematu oraz nazwy obiektu rozdzielonych kropką. Na przykład żeby odnieść się do tabeli ***pracownicy*** w schemacie ***kadry*** należy użyć ***kadry.pracownicy***.

## Ścieżka wyszukiwania schematów

Wpisywanie nazwy schematu przed nazwą obiektu może być jednak uciążliwe, a czasami nawet niewskazane, dlatego często odnosząc się do obiektu w bazie używamy jedynie jego nazwy. W takim przypadku baza danych określa który obiekt został wskazany podążając za ścieżką wyszukiwania, która jest listą schematów w których należy szukać. Pierwszy odnaleziony obiekt jest uznawany za właściwy. Jeśli przeszukiwanie ścieżki nie przyniesie spodziewanego rezultatu zwracany jest błąd nawet jeśli obiekt ten istnieje w schemacie, który nie został ujęty w ścieżce wyszukiwania.

Aby wyświetlić aktualną ścieżkę wyszukiwania należy użyć komendy:

```
SHOW search_path;
```

Jeśli chcemy dodać ***nowy_schemat*** do ścieżki wyszukiwania musimy całą ścieżkę zdefiniować od nowa

```
SET search_path TO nowy_schemat,public;
```

Jednocześnie pierwszy schemat podany w ścieżce wyszukiwania staje się schematem domyślnym, więc jeśli w takim przypadku będziemy chcieli utworzyć tabelę i podamy jej nazwę bez wskazywania schematu zostanie ona utworzona właśnie w schemacie ***nowy_schemat.***

## Tworzenie schematu

Schematy w bazie danych tworzymy poleceniem CREATE SCHEMA.

Poniższe polecenie utworzy w bazie schemat o nazwie ***nowy_schemat***

```
CREATE SCHEMA nowy_schemat;
```

Możemy również utworzyć w bazie nowy schemat podając od razu jego właściciela

```
CREATE SCHEMA nowy_schemat AUTHORIZATION szkolenie_usr;
```

Jeśli w powyższym poleceniu nie podamy nazwy schematu zostanie utworzony schemat o nazwie zgodnej z nazwą jego właściciela

```
CREATE SCHEMA AUTHORIZATION szkolenie_usr;
```

Tworząc schemat możemy równocześnie w jednym zapytaniu utworzyć podległe mu elementy, a jeśli dodatkowo użyjemy klauzuli AUTHORIZATION elementy te będą miały ustawionego właściciela na właściciela schematu. Poniższe polecenie utworzy schemat nowy_schemat2, w schemacie tym utworzy tabelę pracownicy oraz widok prac_prawo_jazdy. Właścicielem wszystkich tych elementów będzie szkolenie_usr.

```
CREATE SCHEMA nowy_schemat2 AUTHORIZATION szkolenie_usr
      CREATE TABLE pracownicy (nazwa text, prawo_jazdy bool)
      CREATE VIEW prac_prawo_jazdy AS
      SELECT nazwa FROM pracownicy WHERE prawo_jazdy = true;
```

## Modyfikacja definicji schematu

Zmiany definicji schematu dokonujemy za pomocą polecenia SQL ***ALTER SCHEMA***. Definicję możemy zmieniać w zakresie zmiany nazwy oraz zmiany właściciela.

Żeby używać tego polecenia użytkownik musi być właścicielem schematu, do zmiany nazwy schematu konieczne jest uprawnienie CREATE w bazie danych, w przypadku zmiany właściciela użytkownik musi być bezpośrednim lub pośrednim członkiem roli nowego właściciela (można zmienić właściciela za siebie lub grupę w której jesteśmy)

Poniższe polecenie zmieni nazwę schematu z ***nowy_schemat2*** na ***nowy_schemat3***

```
ALTER SCHEMA nowy_schemat2 RENAME TO nowy_schemat3;
```

A kolejne zmieni właściciela z użytkownika ***szkolenie_usr*** na użytkownika ***postgres***

```
ALTER SCHEMA nowy_schemat3 OWNER TO postgres;
```

## Usuwanie schematu

Schematy usuwamy za pomocą polecenie SQL DROP SCHEMA.

Schemat może zostać usunięty tylko przez jego właściciela lub SUPERUSER-a.

Właściciel schematu może go usunąć wraz ze wszystkimi obiektami które on zawiera nawet jeśli nie jest właścicielem tych podległych obiektów

Polecenie pozwala na użycie dodatkowych słów kluczowych:

- IF EXISTS
  W przypadku jeśli schemat nie istnieje polecenie nie zwróci błędu.
- RESTRICT lub CASCADE
  Pierwsza opcja jest domyślna i powoduje, że polecenie zwróci błąd jeśli próbujemy usunąć schemat który nie jest pusty. Druga opcja wymusza usunięcie wraz ze schematem całej jego zawartości

UWAGA !!!
Opcja CASCADE działa rekurencyjnie - usuwa obiekty zawarte w schemacie oraz wszystkie obiekty od nich zależnie nawet jeśli właściciel schematu nie jest właścicielem tych obiektów, oraz jeśli te obiekty znajdują się poza usuwanym schematem.

Poniższe polecenie usunie schemat ***nowy_schemat3*** wraz ze wszystkimi podległymi obiektami, jeśli schemat by nie istniał nie zwróci błędu.

```
DROP SCHEMA IF EXISTS nowy_schemat3 CASCADE;
```

## Tworzenie tabeli

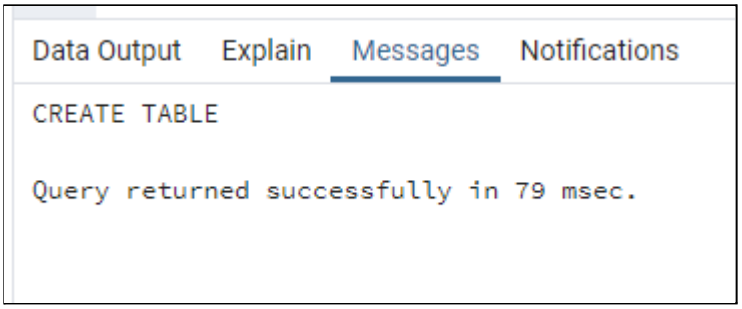
Tabele w bazie danych tworzone są za pomocą polecenia SQL CREATE TABLE.

Polecenie udostępnia wiele opcji i wariantów - w ramach szkolenia omówimy jedynie tę podstawową.

Poniższe polecenie utworzy w bazie danych nową tabelę o nazwie pracownicy zawierającą kolumny imię i nazwisko o typie text, oraz kolumny id o pseudotypie serial. Kolumna id zdefiniowana jest jako klucz główny tabeli.

```
CREATE TABLE pracownicy (
      id serial PRIMARY KEY,
      nazwisko text,
      imie text);
```

W wyniku otrzymamy




Pesudotyp serial użyty przy tworzeniu tabeli powoduje:

- utworzenie kolumny o typie integer
- utworzenie generatora korespondujacego z kolumną
- ustawienie wartości domyślnej na kolejną wartość generatora

Kolumny mogą mieć dowolnie długie nazwy i zawierać dowolne znaki, jeśli jednak:

- zawierają spacje, znaki specjalne
- zawierają słowa kluczowe SQL (np. select, table, insert, order)
- zaczynają się od cyfry

| | |
|---|---|
| | • zawierają wielkie litery<br>ich nazwy muszą być ujęte w "podwójny cudzysłów". |

Pozostałe opcje i warianty pozwalają między innymi na:

- tworzenie tabel tymczasowych
- tworzenie tabel na podobieństwo innych tabel
- partycjonowanie tabel
- określanie kluczy i ograniczeń

Dodatkowo na poziomie każdej kolumny możemy między innymi:

- określić klucz główny
- określić klucz obcy
- przypisać wartość domyślną
- wykluczyć możliwość wpisania wartości pustych
- wymusić unikalność wartości

Z wielu wariantów polecenia wyjątkowo przydatna może być konstrukcja CREATE TABLE AS, która pozwala na utworzenie tabeli na podstawie wyniku zapytania. Poniższe zapytanie utworzy tabelę ***pracownicy1*** która będzie miała taki sam zestaw kolumn co tabela ***pracownicy*** i zawierała dane zawarte w źródłowej tabeli

```
CREATE TABLE pracownicy1 AS SELECT * FROM pracownicy;
```

| | |
|---|---|
|  | Konstrukcja ta jest wyjątkowo przydatna jeśli planujemy większe zmiany w tabeli i chcemy wykonać jej kopię przed przystąpieniem do prac |

## Modyfikacja definicji tabeli

Definicję tabeli modyfikujemy za pomocą polecenia SQL ALTER TABLE.

Podobnie jak polecenie tworzenia tabeli to polecenie również umożliwia wiele opcji i wariantów. W tym przypadku również omówimy tylko te najczęściej używane.

Zmiana nazwy tabeli

```
ALTER TABLE pracownicy RENAME TO pracownicy1;
```

Zmiana schematu tabeli

```
ALTER TABLE pracownicy SET SCHEMA TO nowy_schemat;
```

Zmiana właściciela tabeli

```
ALTER TABLE nowy_schemat.pracownicy OWNER TO postgres;
```

Dodanie nowej kolumny do tabeli

```
ALTER TABLE nowy_schemat.pracownicy
      ADD COLUMN wik text;
```

Zmiana nazwy kolumny

```
ALTER TABLE nowy_schemat.pracownicy
      RENAME COLUMN wik TO wiek;
```

Zmiana typu danych dla kolumny w tabeli

```
ALTER TABLE nowy_schemat.pracownicy
      ALTER COLUMN wiek SET DATA TYPE integer USING cast(wiek as integer);
```

Ustawienie wartości domyślnej dla kolumny

```
ALTER TABLE nowy_schemat.pracownicy
      ALTER COLUMN wiek SET DEFAULT 18;
```

## Usuwanie tabeli

Tabelę możemy usunąć za pomocą polecenia SQL DROP TABLE.

Polecenie może wydać jedynie właściciel tabel, właściciel schematu w którym się znajduje oraz SUPERUSER.

Polecenie usuwa z bazy zarówno tabelę jaki i wszystkie zdefiniowane na niej indeksy, wyzwalacze i ograniczenia.

Jeśli chcemy usunąć tabelę która jest używana w widoku lub kluczu obcym musimy dodatkowo użyć opcji CASCADE. Opcja ta razem z tabelą usunie również widok, a w przypadku klucza obcego usunie jedynie ograniczenie klucza obcego z powiązanej tabeli - nie całą tabelę.

Jeśli chcemy usunąć dane z tabeli ale pozostawić samą tabelę możemy użyć polecenia DELETE lub TRUNCATE.

Poniższe polecenie usunie tabelę ***pracownicy*** znajdującą się w schemacie ***nowy_schemat*** z bazy danych.

```
ALTER TABLE nowy_schemat.pracownicy
```

## Zarządzanie uprawnieniami

Uprawnieniami w bazie danych jak również uprawnieniami do samej bazy zarządzamy za pomocą poleceń GRANT oraz REVOKE.

Wariant polecenia operujący na obiektach bazy danych pozwala na dodanie uprawnień dla jednej lub kilku ról (użytkowników lub grup). Uprawnienia te dodawane są do uprawnień już wcześniej nadanych.

Słowo kluczowe PUBLIC pozwala na wskazanie, że dane uprawnienie ma zostać nadane do wszystkich ról zdefiniowanych w klastrze oraz tych które zostaną utworzone w przyszłości.

Słowo kluczowe WITH GRANT OPTION określa, że użytkownik, który otrzymał wskazane uprawnienie będzie mógł nadawać go innym użytkownikom.

Nie ma potrzeby nadawania uprawnień właścicielowi obiektu, ponieważ z założenia ma on pełne uprawnienia na obiekcie którego jest właścicielem, jednak dla bezpieczeństwa właściciel może sobie odebrać uprawnienia które nie są mu potrzebne.

Uprawnienie do usunięcia obiektu lub do modyfikacji jego definicji nie jest traktowane jako uprawnienie przydzielane - te operacja może wykonywać tylko właściciel obiektu.

Poniższe polecenie SQL nada wszystkie uprawnienia w bazie danych ***szkolenie_db*** użytkownikowi ***szkolenie_usr***, z możliwością nadawania tego uprawnienia innym użytkownikom.

```
GRANT ALL ON DATABASE szkolenie_db TO szkolenie_usr WITH GRANT OPTION;
```

A poniższe zapytanie odbierze użytkownikowi ***szkolenie_usr*** możliwość nadawania uprawnień innym użytkownikom w bazie danych ***szkolenie_db***.

```
REVOKE GRANT OPTION FOR ALL ON DATABASE szkolenie_db FROM szkolenie_usr;
```

Możliwe uprawnienia to:

- SELECT
  Pozwala na wykonywanie polecenia SELECT na wszystkich lub wybranych kolumnach tabeli lub widoku
- INSERT
  Pozwala na wstawianie do tabeli nowych rekordów. Może być nadane na konkretne kolumny - w takim przypadku tylko one mogą wystąpić w poleceniu INSERT
- UPDATE
  Pozwala na modyfikację zawartości wszystkich lub wybranych kolumn w tabeli. W większości przypadków żeby było efektywne potrzebuje również uprawnienia SELECT
- DELETE
  Pozwala na usuwanie rekordów z tabeli
- TRUNCATE
  Pozwala na usuwanie zawartości tabeli
- REFERENCES
  Pozwala na tworzenie kluczy obcych
- TRIGGER
  Pozwala na tworzenie wyzwalaczy
- CREATE
  Pozwala na tworzenie nowych obiektów podległych obiektowi na który zostało nadane (schematów w bazie lub tabel w schemacie)
- CONNECT
  Pozwala na podłączenie się do bazy danych
- TEMPORARY
  Pozwala na tworzenie tabel tymczasowych
- EXECUTE
  Pozwala na wykonywanie funkcji składowanych w bazie
- USAGE
  Pozwala na używanie obiektów na które zostało nadane lub obiektów zawartych w tym obiekcie. Dla schematów pozwala na dostęp do tabel czy widoków, dla generatorów na dostęp do ich wartości.

Uprawnienia szczegółowo są opisane w dokumentacji PostgreSQL.

# Tworzenie danych w bazie

Dane do poszczególnych tabel dodajemy za pomocą polecenia INSERT INTO.

W jednym z poprzednich kroków utworzyliśmy w bazie danych tabelę pracownicy. Aby dodać do niej dane użyjemy najprostszej formy polecenia INSERT, gdzie po słowie kluczowym VALUES w nawiasie podaje się kolejne wartości wstawiane do bazy.

```
INSERT INTO pracownicy
     VALUES (nextval('pracownicy_id_seq'), 'Mickiewicz', 'Adam');
```

Jeśli nie wstawiamy wartości do wszystkich kolumn możemy wskazać poszczególne kolumny i przyporządkować im wartości - pozostałym kolumnom zostanie przypisana wartość domyślna lub null

```
INSERT INTO pracownicy (nazwisko, imie)
     VALUES ('Krasicki', 'Ignacy');
```

Możemy również jawnie wymusić wstawienie wartości domyślnej za pomocą słowa kluczowego DEFAULT

```
INSERT INTO pracownicy
     VALUES (DEFAULT, 'Słowacki', 'Juliusz');
```

Polecenie to pozwala również na wstawienie kilku wierszy w jednym zapytaniu

```
INSERT INTO pracownicy VALUES
      (DEFAULT, 'Słodowy', 'Adam'),
      (DEFAULT, 'Bolesław', 'Prus');
```

Bardzo przydatną funkcją jest możliwość połączenia polecenia INSERT z SELECT. Poniższe zapytanie wstawi do tabeli pracownicy jeszcze raz wszystko co znajdzie w tej tabeli.

```
INSERT INTO pracownicy (nazwisko, imie)
     SELECT nazwisko, imie from pracownicy;
```

Słowo kluczowe RETURNING pozwala na zwrócenie z polecenia INSERT wstawionych wartości i jest przydatne na przykład w przypadku generowania identyfikatorów po stronie bazy danych

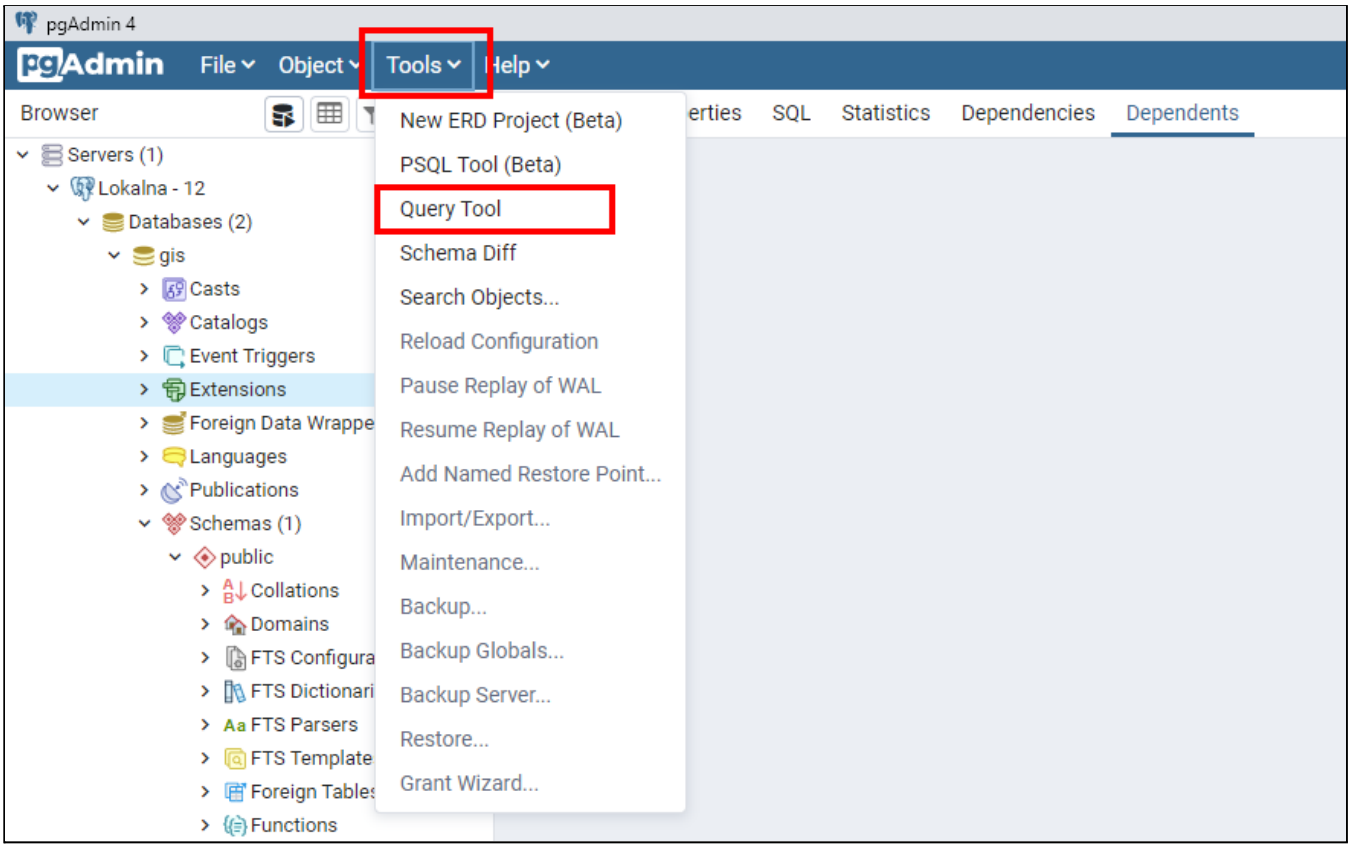
```
INSERT
     INTO pracownicy
     VALUES (DEFAULT, 'Sienkiewicz', 'Henryk')
     RETURNING *;
```

# Pobieranie danych z bazy

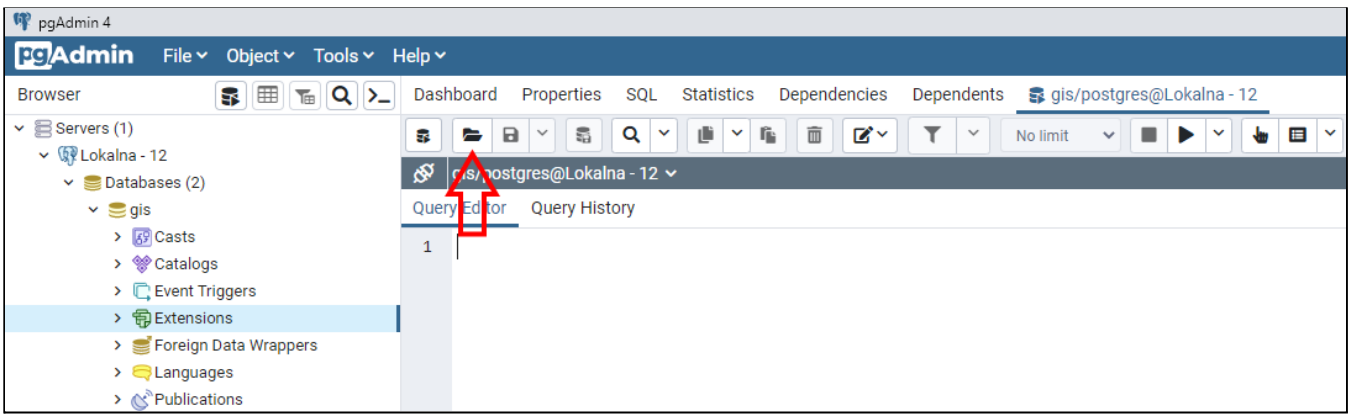
## Import danych szkoleniowych

W kolejnych krokach szkolenia będziemy potrzebowali nieco bardziej złożonej struktury danych oraz większej ilości danych, zaimportujemy je więc do bazy używając wcześniej przygotowanego skryptu.

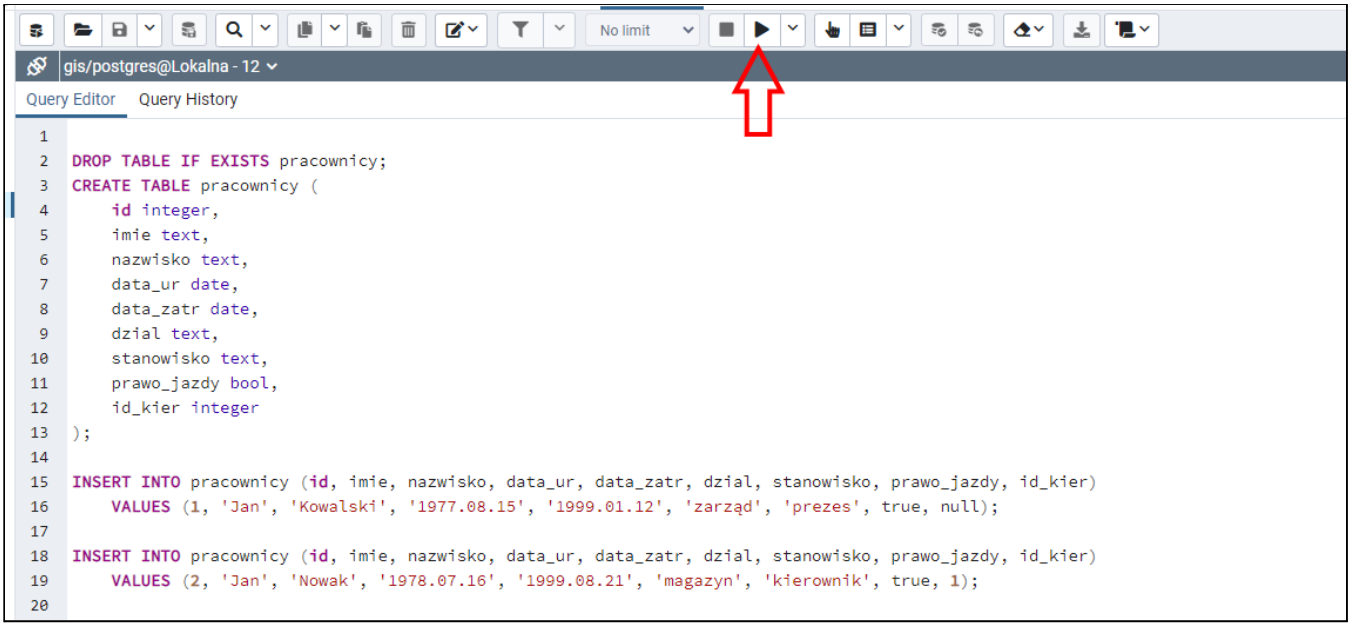
W tym celu uruchamiamy aplikację pgAdmin, a w niej narzędzie do wykonywania zapytań:




Za pomocą ikony folderu otwieramy plik dane.sql dostarczony z materiałami szkoleniowymi




Do okna zostanie wczytany skrypt tworzący strukturę bazy i wypełniający ją danymi - na obecnym etapie powinien on być już w pełni zrozumiały.




Skrypt uruchamiamy za pomocą wskazanego przycisku lub wciskając klawisz F5.

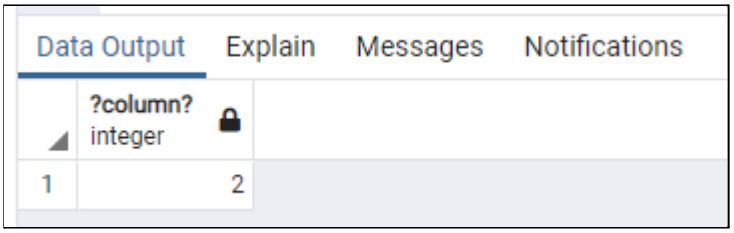
## Instrukcja SELECT

Zapytania SELECT służą przede wszystkim do pobierania danych z bazy danych. Polecenie w najczęściej używanej swojej formie wygląda następująco:

```
SELECT * FROM pracownicy;
```

Samo polecenie służy tutaj za przykład i zostanie omówione w dalszej części szkolenia. Polecenie może operować na 0 lub więcej relacji - poniższe polecenie operujące na 0 relacji zwróci po prostu wynik działania nie sięgając wogóle do bazy danych.

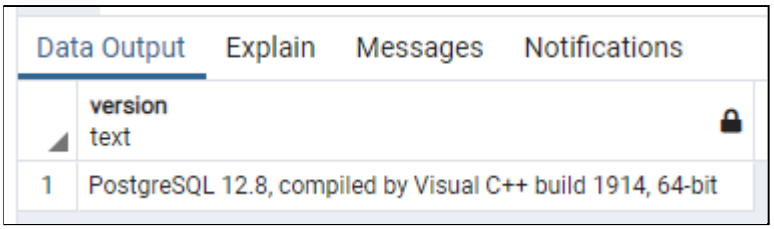
```
SELECT 1+1;
```




Za ich pomocą uruchamiamy również funkcje żądając od nich podania wyniku:

```
SELECT version();
```

To polecenie zwróci nam wersję PostgreSQL




UWAGA !!!
Z powyższej funkcjonalności wynika, że samo polecenie SELECT jak i nadanie uprawnień wyłącznie do uruchamiania SELECT nie gwarantuje, że użytkownik nie będzie miał możliwości zmiany danych w bazie - wystarczy, że poleceniem tym uruchomi funkcje które takie dane zmieni.

## Kolejność przetwarzania instrukcji

Wywołanie polecenia SELECT uruchamia w bazie danych wiele mechanizmów mających na celu możliwie najbardziej wydajne odnalezienie danych w bazie i zwrócenie ich użytkownikowi. Kolejność przetwarzania takiego zapytania jest kluczowa, a poznanie jej i zrozumienie pozwala na tworzenie wydajniejszych zapytań. Mając to na uwadze polecenie to będziemy omawiać właśnie w tej kolejności w jakiej będzie ono przetwarzane:

1. Przygotowywane są tymczasowe zbiory danych określone klauzulą WITH.
2. Sprawdzane są wszystkie relacje określone klauzulą FROM.
3. Eliminowane są wiersze nie pasujące do zadanej klauzuli WHERE.
4. Jeżeli została użyta klauzula GROUP BY, uruchamiane są funkcje agregujące.
5. Jeżeli została użyta klauzula HAVING eliminowane są kolejne wiersze z wyniku po grupowaniu
6. Przetwarzana jest klauzula SELECT odpowiedzialna za wygląd zwracanych danych
7. Jeżeli została użyta klauzula DISTINCT, usuwane są duplikaty.
8. Wykonywane są operacje UNION, INTERSECT lub EXCEPT (operacje na zbiorach wyników kilku zapytań)
9. Jeżeli została użyta klauzula ORDER BY, następuje sortowanie.
10. Jeżeli została użyta klauzula LIMIT lub OFFSET, następuje ograniczenie listy wyników do zadanej liczby.

## Klauzula FROM

W klauzuli FROM, określane są trzy bardzo istotne elementy zapytań:
wyszczególniane są zbiory (źródła) danych (np. tabele, widoki) do których się odnosimy
Jeśli zbiorów jest więcej niż jeden (zapytania do wielu tabel), określany jest typ relacji pomiędzy nimi a także warunki na jakich zasadach te relacje budujemy.

## Jedno źródło i aliasowanie

Dla jednego źródła danych (tabela, widok) podajemy jego nazwę

```
SELECT * FROM pracownicy;
SELECT * FROM place;
```

Jeśli nie znajduje się w domyślnym schemacie musimy również podać nazwę schematu

```
SELECT * FROM public.pracownicy;
```

Do każdego źródła możemy przypisać alias używając słowa kluczowego AS

```
SELECT p.* FROM public.pracownicy as p;
```

Słowo kluczowe AS jest opcjonalne więc możemy je pominąć

```
SELECT p.* FROM pracownicy p;
```

## Złączenie domyślne (iloczyn kartezjański)

Złączenie domyślne wykorzystuje tylko klauzulę WHERE, bez użycia słowa kluczowego JOIN.

```
SELECT * FROM pracownicy p, place pl;
```

Złączenie tego typu tworzy iloczyn kartezjański wszystkich rekordów z obu tabel i mimo, że jest możliwość odfiltrowania niechcianych rekordów klauzulą WHERE i uzyskania prawidłowego wyniku, jak na poniższym przykładzie

```
SELECT *
FROM pracownicy p, place pl
WHERE p.stanowisko = pl.stanowisko;
```

krok pośredni generuje zupełnie niepotrzebnie bardzo obszerne źródło danych, dlatego **nie jest to zalecana metoda łączenia tabel.**

## Złączenie wewnętrzne

Złączenia należy realizować za pomocą klauzuli JOIN oraz warunków podawanych po słowie kluczowym ON.

```
SELECT *
```

```
FROM pracownicy p
  JOIN place pl ON p.stanowisko = pl.stanowisko;
```

Typ złączenia realizowany przez złączenie domyślne jest złączeniem wewnętrznym, oznacza to, że zwracane są tylko takie wyniki, które mają pasującą parę wartości w tabelach "p" i "pl". Zapytanie to można również zapisać w następujący sposób:

```
SELECT *
FROM pracownicy p
  INNER JOIN place pl ON p.stanowisko = pl.stanowisko;
```

## Złączenia zewnętrzne

Złączenie zewnętrzne (OUTER) pozwala na połączenie wielu tabel także wtedy, gdy nie zawsze wystąpią pasujące wiersze. Wówczas dla wiersza bez "pary" zostanie dopasowana wartość pusta (NULL).
Możemy w ten sposób chronić rekordy z tabeli do której dołączamy (lewej) używając słowa kluczowego LEFT

```
SELECT *
FROM pracownicy p
  LEFT OUTER JOIN place pl ON p.stanowisko = pl.stanowisko;
```

Tabeli dołączanej używając słowa kluczowego RIGHT

```
SELECT *
FROM pracownicy p
  RIGHT OUTER JOIN place pl ON p.stanowisko = pl.stanowisko;
```

Lub obu tabel używając słowa kluczowego FULL

```
SELECT *
FROM pracownicy p
  FULL OUTER JOIN place pl ON p.stanowisko = pl.stanowisko;
```

## Samozłączenie

Jeśli istnieje taka potrzeba możemy również złączyć tabelę z samą sobą. W tym celu przy każdym odwołaniu się do tabeli źródłowej należy określić inny jej alias.
Poniższe zapytanie zwróci podległość służbową pracowników

```
SELECT
   kier.nazwisko||' '||kier.imie as zwierzchnik,
   prac.nazwisko||' '||prac.imie as podlegly
FROM pracownicy kier
  JOIN pracownicy prac ON kier.id = prac.id_kier;
```

## Podzapytania

Zapytania SELECT możemy w sobie zagnieżdżać, więc w klauzuli FROM może wystąpić inne zapytanie SELECT

```
SELECT
   row_number() over() as id,
   sub.*
FROM (
    SELECT DISTINCT stanowisko FROM pracownicy ORDER BY 1
        ) as sub;
```

Funkcja **row_number() over()** użyta w poprzednim i kolejnych zapytaniach to funkcja z grupy funkcji okien, których działanie i zastosowanie znacznie wybiega ponad poziom podstawowy szkolenia, dlatego nie będa one omawiane. Na obecnym poziomie szkolenia istotna jest informacja, że funkcja ta użyta w ten sposób w nowej kolumnie ponumeruje rekordy zwracane przez podzapytanie.

## Klauzula WITH

Podzapytania w klauzuli FROM możemy uprościć używając klauzuli WITH (CTE - Common Table Expressions), więc nasze zapytanie z poprzedniego przykładu:

```
SELECT
   row_number() over() as id,
   sub.*
FROM (
    SELECT DISTINCT stanowisko FROM pracownicy ORDER BY 1
        ) as sub;
```

Może również wyglądać tak:

```
WITH
   stanowiska as (SELECT DISTINCT stanowisko FROM pracownicy ORDER BY 1)
SELECT
   row_number() over() as id,
   *
FROM stanowiska;
```

Tabele definiowane w klauzuli WITH mogą być skorelowane.
***Klauzula WITH ma znaczący wpływ na sposób wykonywania zapytania***

Funkcje

Źródłem danych dla klauzuli FROM może być również funkcja

```
SELECT * FROM generate_series(0,5);
```

# Klauzula WHERE

Klauzula WHERE pozwala na filtrowanie rekordów zwróconych przez źródła danych zdefiniowane w klauzuli FROM.

W klauzuli WHERE można podać jeden lub więcej warunków.

Nazwy kolumn podajemy bez cudzysłowu lub w podwójnym cudzysłowie.

Wartości liczbowe podajemy bez cudzysłowu, separatorem dziesiętnym jest zawsze kropka.

Wartości tekstowe podajemy zawsze w pojedynczym cudzysłowie.

## Operatory

Do porównywania wartości w klauzuli WHERE służą operatory porównania - poniżej najczęściej używane:

```
SELECT * FROM pracownicy WHERE stanowisko = 'kierownik';

SELECT * FROM place WHERE stawka > 5;

SELECT * FROM place WHERE stawka >= 7.5;
```

```
SELECT * FROM place WHERE stawka = 10;
```

Operator IN pozwala na sprawdzenie czy wartość jest elementem listy:

```
SELECT * FROM pracownicy WHERE stanowisko IN ('prezes', 'kierownik');
```

Konstrukcja BETWEEN … AND ... sprawdza czy wartość zawiera się między wskazanymi wartościami (zbiór jest obustronnie domknięty)

```
SELECT * FROM place WHERE stawka BETWEEN 4.8 AND 7.5;
```

## Porównywanie z wzorcem

Do porównywania z wzorcem służy między innymi operator **LIKE**. W operatorze tym '_' zastępuje jeden dowolny znak

```
SELECT * FROM pracownicy WHERE stanowisko LIKE 'kierow__';
```

'%' zastępuje dowolny ciąg znaków.

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

Kolejnym operatorem porównania z wzorcem jest **SIMILAR TO**. Jest to fuzja operatora LIKE oraz wyrażeń regularnych RegExp, w jego przypadku

- znaki '**_**' oraz '**%**' działają jak w LIKE
- '**|**' oznacza jedną z dwóch opcji ( a|b = znak a lub b)
- '*****' oznacza opwtórzernie poprzedniego znaku 0 lub więcej razy
- '**+**' oznacza powtórzenie poprzedniego znaku 1 lub więcej razy
- Nawiasy '**()**' mogą być użyte do grupowania obiektów
- Nawiasy kwadratowe '**[...]**' określają klasę znaku jak w POSIX
- **UWAGA** - w przeciwieństwie do POSIX kropka '**.**' nie jest tutaj znakiem specjalnym

Poniższe zapytanie zwróci wszystkie rekordy gdzie nazwisko kończy się na **'ski'** lub **'icz'**

```
SELECT * FROM pracownicy WHERE nazwisko SIMILAR TO '%(ski|icz)';
```

A to zapytanie zwróci rekordy w których pole stanowisko zawiera 'kierowca' lub 'kierownik'

```
SELECT * FROM pracownicy WHERE stanowisko SIMILAR TO 'kierow(ca|nik)';
```

## Porównanie z wartością NULL

Porównywanie z wartością NULL odbywa się na innych zasadach i wymienione wcześniej operatory nie działają prawidłowo w jej przypadku

```
SELECT * FROM pracownicy WHERE prawo_jazdy = null;
```

Do porównania z wartością NULL trzeba stosować specjalne operatory IS NULL oraz IS NOT NULL.

```
SELECT * FROM pracownicy WHERE prawo_jazdy IS NULL;

SELECT * FROM pracownicy WHERE prawo_jazdy IS NOT NULL;
```

## Łączenie warunków

Warunki można łączyć wykorzystując operatory logiczne: AND, OR, NOT

```
SELECT * FROM pracownicy WHERE nazwisko = 'Nowak' OR nazwisko =
'Kowalski';

SELECT * FROM pracownicy WHERE imie = 'Jan' AND nazwisko = 'Nowak';

SELECT * FROM pracownicy WHERE imie = 'Jan' AND NOT nazwisko = 'Nowak';
```

Podczas łączenia warunków należy uważać na kolejność wykonywania działań

```
SELECT * FROM pracownicy
WHERE nazwisko = 'Nowak' AND (imie = 'Jan' OR imie = 'Czesław');
```

Powyższe zapytanie bez nawiasów zwróci nam Jana Nowaka oraz wszystkich Czesławów

```
SELECT * FROM pracownicy
WHERE nazwisko = 'Nowak' AND imie = 'Jan' OR imie = 'Czesław';
```

### Funkcje

W klauzuli WHERE można również używać funkcji

```
SELECT * FROM pracownicy WHERE upper(nazwisko) = 'NOWAK';

SELECT * FROM pracownicy WHERE lower(nazwisko) = 'nowak';

SELECT * FROM pracownicy WHERE length(imie) > 3;
```

### Podzapytania

Podzapytania w klauzuli WHERE realizujemy za pomocą funkcji **exists()**

```
SELECT *
FROM pracownicy pr
WHERE exists(
   SELECT 1 FROM place pl WHERE pl.stanowisko = pr.stanowisko
         );
```

UWAGA !!!
SELECT jest uruchamiany dla każdego rekordu - niska wydajność

## Klauzula GROUP BY

Klauzula GROUP BY pozwala na grupowanie wyniku zapytania.

Najczęściej GROUP BY stosowane jest wraz z funkcjami agregującymi.

Wszystkie kolumny, które mają być zwrócone przez zapytanie grupujące, muszą być:

- albo użyte w funkcji agregującej
- albo użyte w klauzuli GROUP BY.

Poniższe zapytanie zwróci liczbę pracowników zatrudnionych na poszczególnych stanowiskach

```
SELECT stanowisko, count(*)
FROM pracownicy pr
GROUP BY stanowisko;
```

A to zapytanie zwróci średni okres zatrudnienia na poszczególnych stanowiskach

```
SELECT stanowisko, avg(current_date - data_zatr)
FROM pracownicy pr
GROUP BY stanowisko;
```

Do poszczególnych kolumn w klauzuli GROUP BY możemy się odnosić również podając ich indeks

```
SELECT typ, sum(kwota)
FROM faktury
GROUP BY 1;
```

## Klauzula HAVING

Klauzula HAVING pozwala na dodatkowe ograniczenie zwracanego wyniku po wykonaniu grupowania.

Poniższe zapytanie zwróci tylko te stanowiska które są obsadzone więcej niż jedną osobą:

```
SELECT stanowisko, count(*)
FROM pracownicy pr
GROUP BY stanowisko
HAVING count(*) > 1;
```

## Klauzula SELECT

Klauzula SELECT pozwala na określenie kolumn zwracanego wyniku.

Poniższe zapytanie zwróci tylko imiona i nazwiska z tabeli pracowników.

```
SELECT imie, nazwisko FROM pracownicy;
```

Nazwy zwracanych kolumn można zmieniać używając opcjonalnego słowa kluczowego AS

```
SELECT
```

```
   imie,
   nazwisko,
   data_ur as "data urodzenia",
   data_zatr "data zatrudnienia"
FROM pracownicy;
```

Symbol “*” użyty w tej klauzuli spowoduje zwrócenie wszystkich kolumn ze źródła

```
SELECT * FROM pracownicy;
```

W klauzuli SELECT można również używać funkcji i wykonywać dodatkowe operacje

```
SELECT
   imie||' '||nazwisko as "Imię i Nazwisko",
   age(current_date, data_ur) as "wiek"
FROM pracownicy;
```

oraz używać podzapytań

```
SELECT
   imie||' '||nazwisko as "Imię i Nazwisko",
   stanowisko,
   (SELECT stawka
    FROM place pl WHERE pl.stanowisko = pr.stanowisko ) as stawka
FROM pracownicy pr;
```

Podzapytanie użyte w klauzuli SELECT musi zwracać wartość skalarną

Podzapytanie które odnosi się do swojego zapytania nadrzędnego nazywamy podzapytaniem skorelowanym

## Modyfikator DISTINCT

Modyfikator DISTINCT służy do zwrócenia tylko niepowtarzalnych wartości.

```
SELECT DISTINCT stanowisko FROM pracownicy;

SELECT DISTINCT typ FROM faktury;
```

Modyfikator DISTINCT może zostać zastosowany również wspólnie z funkcjami agregującymi, np.

```
SELECT count(DISTINCT stanowisko) FROM pracownicy;
```

DISTINCT ON

Baza PostgreSQL dysponuje również modyfikatorem DISTINCT ON, który nie należy do standardu SQL.
Stosowany jest do zwrócenia pełnych informacji o reprezentatywnym wierszu w każdej kategorii.
Aby DISTINCT ON miał sens, powinien być użyty z klauzulą ORDER BY, która musi zawierać 2 kolumny - jako pierwsza kolumna w której znajdują się kategorie, jako druga kolumna w której znajdują się wartości.

Aby uzyskać informację o najmniejszej fakturze dla każdego typu faktury wykonamy zapytanie:

```
SELECT DISTINCT ON (typ) *
FROM faktury
ORDER BY typ, kwota;
```

Jeśli chcemy uzyskać informację o największej fakturze musimy zmienić kolejność sortowania

```
SELECT DISTINCT ON (typ) *
FROM faktury
ORDER BY typ, kwota DESC;
```

UNION/INTERSECT/EXCEPT

PostgreSQL umożliwia wykonywanie operacji z zakresu algebry zbiorów.

Sumę zbiorów uzyskujemy dzięki klauzuli UNION. Poniższe zapytanie zwróci wszystkie rekordy z tabeli faktury oraz faktury_arch, dbając przy tym żeby nie pojawiły się zdublowane rekordy

```
SELECT * FROM faktury
UNION
SELECT * FROM faktury_arch;
```

Jeśli chcemy otrzymać wszystkie rekordy z obu tabel, wraz z tymi powtarzającymi się użyjemy klauzuli UNION ALL

```
SELECT * FROM faktury
UNION ALL
```

```
SELECT * FROM faktury_arch;
```

Iloczyn uzyskamy za pomocą klauzuli INTERSECT. Poniższe zapytanie zwróci wszystkie rekordy, które występują zarówno w tabeli faktury, jak i w tabeli faktury_arch.

```
SELECT * FROM faktury
INTERSECT
SELECT * FROM faktury_arch;
```

Różnicę uzyskujemy dzięki klauzuli EXCEPT. Poniższe zapytanie zwróci wszystkie rekordy z tabeli faktury_arch, które nie występują w tabeli faktury.

```
SELECT * FROM faktury_arch
EXCEPT
SELECT * FROM faktury;
```

Klauzula ORDER BY

Z założenia dane w bazie danych nie są w żaden sposób posortowane. To samo zapytanie uruchomione wielokrotnie może zwrócić rekordy w innej kolejności.
Jeśli chcemy otrzymać rekordy uporządkowane musimy użyć klauzuli ORDER BY

```
SELECT imie, nazwisko FROM pracownicy ORDER BY nazwisko;
```

Domyślnie sortowanie odbywa się w kierunku rosnącym, aby to zmienić używamy słowa kluczowego DESC

```
SELECT imie, nazwisko FROM pracownicy ORDER BY nazwisko DESC;
```

Oczywiście możliwe jest sortowanie po więcej niż 1 kolumnie

```
SELECT imie, nazwisko FROM pracownicy ORDER BY nazwisko, imie;
```

Domyślnie wartości puste są na początku, możemy to zmienić używając słowa kluczowego NULLS LAST

```
SELECT imie, nazwisko, prawo_jazdy
FROM pracownicy
ORDER BY prawo_jazdy ASC NULLS LAST;
```

## Klauzula LIMIT/OFFSET

Dzięki klauzuli LIMIT mamy możliwość ograniczenia ilości rekordów zwróconych z zapytania. Poniższe zapytanie zwróci 10 pierwszych rekordów z tabeli faktury

```
SELECT * FROM faktury ORDER BY numer LIMIT 10;
```

Jeśli chcemy pominąć jakieś rekordy używamy klauzuli OFFSET

```
SELECT * FROM faktury ORDER BY numer LIMIT 10 OFFSET 10;
```

# Modyfikacja danych w bazie

Modyfikacji danych w bazie danych dokonujemy za pomocą polecenia UPDATE.

Polecenie to zmienia wartości w kolumnach wskazanych w klauzuli SET dla wszystkich rekordów spełniających warunki z klauzuli WHERE.

Poniższe polecenie zmieni imię pracownika o nazwisku ***Kowalski*** na ***Adam***

```
UPDATE pracownicy
SET imie = 'Adam'
WHERE nazwisko = 'Kowalski';
```

Są dwa sposoby modyfikacji danych w tabeli za pomocą wartości pobranych z innej tabeli. Pierwszy z nich to podzapytanie w klauzuli SET

```
UPDATE pracownicy
SET imie = (select imie from pracownicy where nazwisko = 'Nowak')
WHERE nazwisko = 'Kowalski';
```

Drugi sposób to użycie klauzuli FROM

```
UPDATE pracownicy
SET imie = p.imie
FROM pracownicy p
WHERE pracownicy.nazwisko = 'Kowalski'
      AND p.nazwisko = 'Sienkiewicz';
```

Obie formy są poprawne, a wybór której użyć zależy wyłącznie od samego zapytania

Opcjonalna klauzula RETURNING pozwala na zwrócenie poszczególnych kolumn lub całych rekordów które zostały zmodyfikowane

```
UPDATE pracownicy
SET imie = p.imie
FROM pracownicy p
WHERE pracownicy.nazwisko = 'Kowalski'
      AND p.nazwisko = 'Sienkiewicz'
RETURNING *;
```

# Usuwanie danych z bazy

Usuwanie danych z bazy dokonywane jest za pomocą polecenia DELETE.

Polecenie usuwa z tabeli wszystkie rekordy spełniające warunek podany w klauzuli WHERE.

UWAGA !!!
Klauzula WHERE jest opcjonalna. Jeśli jej nie podamy polecenie usunie wszystkie rekordy w tabeli.

Jak w przypadku polecenia UPDATE są dwa sposoby na usuwanie danych w jednej tabeli z użyciem wartości w drugiej tabeli - użycie podzapytania w klauzuli WHERE lub użycie opcjonalnej klauzuli USING.

Jak w poprzednich przypadkach klauzula RETURNING zwraca rekordy usuwane.

Co zrobi poniższe zapytanie?

```
DELETE FROM pracownicy
USING place
WHERE place.stanowisko = pracownicy.stanowisko
AND place.stawka < 4.5
RETURNING *;
```

Jeśli jednak chcemy usunąć z tabeli wszystkie rekordy zamiast polecenia DELETE bez klauzuli WHERE zalecane jest użycie polecenia TRUNCATE. Efektem jego działania jest pusta tabela. Polecenie to jest szybsze w działaniu ponieważ nie skanuje tabeli którą czyści, dodatkowo od razu zwraca wolne miejsce bez konieczności wykonywania VACUUM.

# Widoki

Widoki (VIEW) w SQL to nic innego jak tabele wirtualne definiowane za pomocą poleceń SELECT, lub patrząc z drugiej strony sposoby na nazwanie zapytania, które często używamy, żeby uniknąć konieczności częstego wpisywania go.

Widoki dzielimy na niezmaterializowane i zmaterializowane.

## Widoki niezmaterializowane

Widoki niezmaterializowane nie mają w bazie żadnej reprezentacji po stronie danych - każde odwołanie się do widoku powoduje natychmiastowe wywołanie zapytania SELECT które go definiuje i pobranie danych z istniejących tabel.

Niezmaterializowany widok tworzymy za pomocą polecenia CREATE VIEW.

Poniższe zapytanie utworzy widok zwracający imię i nazwisko pracownika jego stanowisko i stawkę

```
CREATE VIEW pracownicy_stawka as
     SELECT pr.imie, pr.nazwisko, pr.stanowisko, pl.stawka
     FROM pracownicy pr
     JOIN place pl ON pr.stanowisko = pl.stanowisko;
```

Do widoku odwołujemy się tak samo jak do tabeli więc żeby teraz wyświetlić dane zwracane przez widok używamy polecenia SELECT

```
SELECT * FROM pracownicy_stawka;
```

Widoki usuwamy za pomocą polecenia DROP VIEW.

```
DROP VIEW pracownicy_stawka;
```

## Widoki zmaterializowane

Widoki zmaterializowane przy tworzeniu generują swoją własną reprezentację kopiując istniejące dane. Każde odwołanie się do takiego widoku zwraca uprzednio przygotowane dane. Ten typ widoku stosuje

się dla długo trwających zapytań operujących na rzadko zmienianych danych w przypadkach kiedy nie zależy nam na dokładnym wyniku.

Poniższe zapytanie utworzy analogiczny do poprzedniego widok zmaterializowany

```
CREATE MATERIALIZED VIEW pracownicy_stawka as
     SELECT pr.imie, pr.nazwisko, pr.stanowisko, pl.stawka
     FROM pracownicy pr
     JOIN place pl ON pr.stanowisko = pl.stanowisko;
```

W przeciwieństwie do poprzedniego typu dane zwracane przez widok zmaterializowany nie zmienią się jeśli dane z których został on utworzony ulegną zmianie. Widok taki należy odświeżyć jawnie stosując polecenie REFRESH MATERIALIZED VIEW.

```
REFRESH MATERIALIZED VIEW pracownicy_stawka;
```

# Indeksy

Indeks jest obiektem bazodanowym niezależnym logicznie i fizycznie od tabeli. Pozwala uzyskać szybszy dostęp do danych. Indeksy zakłada się na kolumnę w tabeli lub kilka kolumn naraz. Oczywiście bez nich wszystko będzie działać, jednak indeksy pozwolą nam szybciej dostać się do danych.

Indeksy przechowują wartości kolumn na które są nakładane oraz ROWID wiersza, dlatego w szczególnych przypadkach pobranie danych może odbyć się bez skanowania samej tabeli a jedynie indeksu.

Korzyść wydajnościowa ze stosowania indeksów jest największa w przypadku dużych tabel (zawierających najwięcej rekordów) oraz zapytań, które wykonywane są najczęściej. W PostgreSQL zaleca się indeksować następujące kolumny:

- kolumny najczęściej padające po słowie WHERE
- kolumny dwóch tabel, które często łączymy JOIN .. ON
- kolumny, według których sortujemy dane w raportach (kolumny padające po słowie ORDER BY i GROUP BY)
- kolumny które często zliczamy (SUM(), AVG(), MIN(), MAX(), COUNT())
- klucze obce i kolumny, których będziemy używać tak jak kluczy obcych
- klucze unikalne UNIQUE_KEY (typu NIP, PESEL itd…)

Można się kierować prostą zasadą, polegającą na tym, że nie tworzymy indeksu jeżeli nie jesteśmy przekonani, że faktycznie będziemy z niego korzystać.

Problemy wynikające z użycia indeksów

- **Konieczność aktualizacji** - Z indeksami wcale nie jest tak różowo jak mogłoby się wydawać. Z jednej strony mogą nam pomóc w przyspieszeniu odczytu danych, ale trzeba wziąć też pod uwagę że takie indeksy trzeba będzie aktualizować przy wykonywaniu operacji UPDATE, DELETE i INSERT. To powoduje wydłużenie tych operacji. Nie możemy więc zakładać więcej indeksów niż jest niezbędne i nie powinniśmy ich stosować tam gdzie korzyść z ich zastosowania jest znikoma.
- **Zajęte miejsce** - Indeksy muszą być przechowywane na dysku podobnie jak tabele. Jeśli więc np. utworzysz na jakiejś tabeli indeksy na każdej kolumnie, musisz liczyć się z tym, że ilość zajmowanego miejsca na potrzeby danej tabeli oraz jej indeksów przynajmniej się podwoi.
- **Blokady podczas tworzenia i odbudowywania** - Podczas budowania lub odbudowywania indeksu zakładana jest blokada na wszystkie wiersze których dotyczy. Wydawać by się mogło że to nic poważnego, tymczasem zakładanie indeksu na tabeli liczącej kilkaset tysięcy rekordów może trwać bardzo długo... a zwłaszcza jeśli mówimy o indeksach przestrzennych. Wszystkie transakcje które będą w tym czasie próbowały założyć swoją blokadę będą czekały (nie zostanie zgłoszony błąd) na zakończenie budowania indeksu. Takie transakcje mogły wcześniej zablokować inne zasoby. Aby ten problem „obejść" możesz zastosować współbieżne tworzenie indeksu.

Indeksy tworzymy za pomocą polecenia SQL CREATE INDEX.

Poniższe polecenie utworzy na tabeli pracownicy indeks bazujący na polu nazwisko

```
CREATE INDEX pracownicy_nazwisko_idx on pracownicy(nazwisko);
```

Indeksy są automatycznie aktualizowane jeśli dane w bazie się zmieniają, nie mniej jeśli na zaindeksowanej tabeli wykonywane jest wiele operacji dodawania i usuwania rekordów indeksy mogą się niepotrzebnie rozrosnąć. W takim przypadku możemy ręcznie wymusić odświeżenie konkretnego indeksu

```
REINDEX INDEX pracownicy_nazwisko_idx;
```

Lub wszystkich indeksów dla tabeli

```
REINDEX TABLE pracownicy;
```

Indeksy usuwamy za pomocą polecenia DROP INDEX

```
DROP INDEX pracownicy_nazwisko_idx;
```

Ogólne informacje o utworzonych indeksach możemy uzyskać uruchamiając polecenie:

```
SELECT
   pg_class.relname,
   pg_size_pretty(pg_class.reltuples::bigint) AS rows_in_bytes,
   pg_class.reltuples AS num_rows,
   count(indexname) AS number_of_indexes,
   CASE WHEN x.is_unique = 1 THEN 'Y'
      ELSE 'N'
   END AS UNIQUE,
   SUM(case WHEN number_of_columns = 1 THEN 1
             ELSE 0
           END) AS single_column,
   SUM(case WHEN number_of_columns IS NULL THEN 0
            WHEN number_of_columns = 1 THEN 0
            ELSE 1
          END) AS multi_column
FROM pg_namespace
LEFT OUTER JOIN pg_class ON pg_namespace.oid = pg_class.relnamespace
LEFT OUTER JOIN
      (SELECT indrelid,
          max(CAST(indisunique AS integer)) AS is_unique
      FROM pg_index
      GROUP BY indrelid) x
      ON pg_class.oid = x.indrelid
LEFT OUTER JOIN
   ( SELECT c.relname AS ctablename, ipg.relname AS indexname, x.indnatts AS
number_of_columns FROM pg_index x
          JOIN pg_class c ON c.oid = x.indrelid
          JOIN pg_class ipg ON ipg.oid = x.indexrelid  )
   AS foo
   ON pg_class.relname = foo.ctablename
WHERE
    pg_namespace.nspname='public'
AND  pg_class.relkind = 'r'
GROUP BY pg_class.relname, pg_class.reltuples, x.is_unique
ORDER BY 2;
```

Poniższe polecenie wyświetli wielkości i statystyki użycia poszczególnych indeksów:

```
SELECT
   t.schemaname,
```

```
    t.tablename,
    indexname,
    c.reltuples AS num_rows,
    pg_size_pretty(pg_relation_size(quote_ident(t.schemaname)::text || '.'
|| quote_ident(t.tablename)::text)) AS table_size,
    pg_size_pretty(pg_relation_size(quote_ident(t.schemaname)::text || '.'
|| quote_ident(indexrelname)::text)) AS index_size,
    CASE WHEN indisunique THEN 'Y'
        ELSE 'N'
    END AS UNIQUE,
    number_of_scans,
    tuples_read,
    tuples_fetched
FROM pg_tables t
LEFT OUTER JOIN pg_class c ON t.tablename = c.relname
LEFT OUTER JOIN (
    SELECT
        c.relname AS ctablename,
        ipg.relname AS indexname,
        x.indnatts AS number_of_columns,
        idx_scan AS number_of_scans,
        idx_tup_read AS tuples_read,
        idx_tup_fetch AS tuples_fetched,
        indexrelname,
        indisunique,
        schemaname
    FROM pg_index x
    JOIN pg_class c ON c.oid = x.indrelid
    JOIN pg_class ipg ON ipg.oid = x.indexrelid
    JOIN pg_stat_all_indexes psai ON x.indexrelid = psai.indexrelid
) AS foo ON t.tablename = foo.ctablename AND t.schemaname = foo.schemaname
WHERE t.schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY 1,2;
```

Zdublowane indeksy możemy odnaleźć uruchamiając polecenie:

```
SELECT pg_size_pretty(sum(pg_relation_size(idx))::bigint) as size,
      (array_agg(idx))[1] as idx1, (array_agg(idx))[2] as idx2,
```

```
      (array_agg(idx))[3] as idx3, (array_agg(idx))[4] as idx4
FROM (
   SELECT indexrelid::regclass as idx, (indrelid::text ||E'\n'||
indclass::text ||E'\n'|| indkey::text ||E'\n'||
                                        coalesce(indexprs::text,'')||E'\n'
|| coalesce(indpred::text,'')) as key
   FROM pg_index) sub
GROUP BY key HAVING count(*)>1
ORDER BY sum(pg_relation_size(idx)) DESC;
```

# Projektowanie baz danych

Relacyjna baza danych opisuje swoim modelem pewien fragment rzeczywistości, niezbędny do realizacji określonych celów dla których została stworzona.

## Wprowadzenie do projektowania bazy danych

Model relacyjnych baz danych, opracowany przez Edgara F. Codd'a i przedstawiony w „A Relational Model of Data for Large Shared Data Banks", definiuje m.in. cechy i strukturę dobrze zaprojektowanej bazy. Najłatwiej streścić sedno i przedyskutować potencjalne zagrożenia (złego projektu) na przykładzie, prezentującym kroki procesu normalizacji.

Po co właściwie wiele tabel, relacji między nimi. Czy nie prościej i łatwiej byłoby wszystko przechowywać w jednym miejscu – np. w jednej tabeli, tak jak w arkuszu Excel ?
Przyjrzyjmy się strukturze tabeli zawierającej informacje o zleceniach i Klientach.

Results | Messages

| | NumerZam | NazwaKlienta | AdresKlienta | DataZamowienia | SzczegolyZamowienia |
|---|---|---|---|---|---|
| 1 | 101 | Jan Kowalski | ul. Jana Pawła 12, 61-600 Poznań, woj. Wielko... | 2012-01-02 00:... | Opony 205 R16 4szt, koszt 1200 PLN |
| 2 | 102 | Anna Dymna | ul. Staszica 1, 30-600 Kraków, Małopolska | 2012-03-22 00:... | Alufelgi Silver 4 szt, koszt 2200 PLN |
| 3 | 103 | Piotr Wawrzyniak | al. Niepodległości 1, 30-600 Kraków, woj. Mało... | 2012-03-22 00:... | Alufelgi Silver 4 szt, koszt 2200 PLN |
| 4 | 104 | Jan Kowalski | ul. Jana Pawła 12, 61-600 Poznań, woj. Wielko... | 2012-10-22 00:... | Komplet żarówek, koszt 80 PLN |
| 5 | 105 | Jan Kowalski | ul. Poznańska 8, 21-120 Wrocław, Dolnośląskie | 2012-05-22 00:... | Płyn do spryskiwacza 1szt, Trójkąt ostrzegawczy 1szt, koszt 15 PLN |

Zwróć uwagę na (najważniejsze) problemy, których dostarczy nam tak zaprojektowana tabela :

- te same informacje przechowywane są wiele razy w wielu wierszach (np. AdresKlienta ). Zajmują one niepotrzebnie zasoby. Nie tylko dyskowe ! Ponieważ każdy wiersz, zawiera komplet informacji, zmniejsza się efektywność składowania danych w tabeli (jest to związane z anatomią pliku danych, sposobem zapisu wierszy). Skutkuje to w dalszej kolejności zwiększeniem ilości operacji odczytu /zapisu (kolejka do I/O to jedno z najczęstszych wąskich gardeł) a także późniejszą alokacją większych zasobów pamięci RAM (dla zdublowanych informacji).

- powtarzające się dane to także problem związany z aktualizacją. Jeśli chcemy poprawić np. adres Klienta, trzeba zmodyfikować wszystkie jego wystąpienia (może to być wiele rekordów). Co jeśli o którymś zapomnimy? Które dane będą prawdziwe? Pojawia się tu problem zachowania integralności danych. Problem aktualizacji eskaluje się gdy te same dane będą przechowywane w kilku tabelach. Nie dość, że trzeba pamiętać o modyfikowaniu wszystkich miejsc, to przecież wpływa to bezpośrednio na czas wykonywanych aktualizacji (transakcji). Zwiększa się prawdopodobieństwo wystąpienia zakleszczeń (deadlocków), następuje degradacja równoległego dostępu do danych. Jest to bardzo namacalny problem, subiektywnego postrzegania szybkości i jakości pracy systemu przez użytkowników.
- kolumny AdresKlienta oraz SzczegolyZamowienia, zwierają kolekcje wartości. Skutkuje to brakiem możliwości wykonania podstawowych operacji na danych – np. podsumowania według wartości zleceń, liczby pozycji etc. Ponadto przeszukiwanie takich kolekcji jest nieefektywne. Sprowadza się do przeszukiwania ciągów tekstowych (stringów). Brak możliwości zapewnienia pełnej integralności danych bo nie ma nad nimi kontroli.
- usunięcie rekordu – pozycji zamówienia, skutkuje utratą informacji o Kliencie, czyli takiej której nie chcielibyśmy tracić.
- analogicznie w drugą stronę – problem z dodaniem informacji – konieczne określenie informacji, których być może jeszcze nie znamy, lub których może w ogóle nigdy nie będzie – np. Klient, który nie złożył żadnych zamówień.

# Postaci bazy danych i normalizacja

Wszystkie powyższe problemy i anomalie, rozwiązują odpowiednie postacie normalne (postulaty Codd'a). Postacie normalne wyższego rzędu, implikują wszystkie niższe. Baza danych jest znormalizowana np. do trzeciej postaci normalnej, jeśli są spełnione 1NF – 3NF.

**Normalizacja to bezstratny proces organizowania danych w tabelach mający na celu zmniejszenie ilości danych składowanych w bazie oraz wyeliminowanie potencjalnych anomalii** opisanych powyżej.

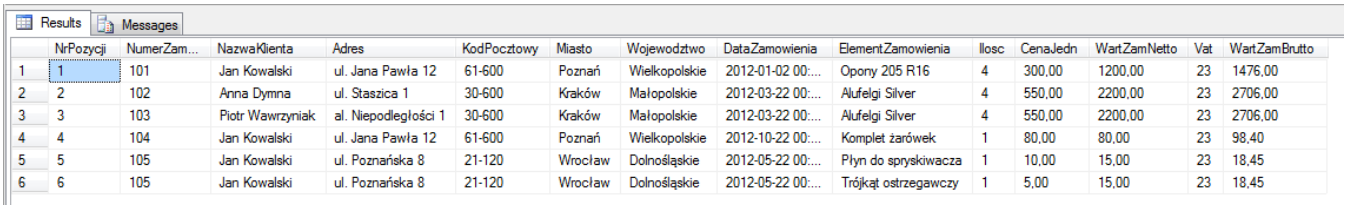
## Pierwsza postać normalna 1NF

Pierwsza postać normalna to podstawa baz – **mówi o atomowości danych**. Czyli tabela (encja) przechowuje dane w sposób atomowy. Każde pole przechowuje jedną informację, dzięki czemu możemy dokonywać efektywnych zapytań. Wprowadza także pojęcie istnienie klucza głównego identyfikującego bezpośrednio każdy wiersz –unikalności. Warto pamiętać również o tym, że w dziedzinie teorii zbiorów kolejność wierszy jest dowolna (chyba że jawnie taki zbiór posortujemy).

Przejście na 1NF, nie może powodować utraty żadnych informacji, nie ma znaczenia kolejność elementów w zbiorze. Ta zasada dotyczy każdej postaci normlanej.

Mówimy, że tabela (encja) jest w **pierwszej postaci normalnej, kiedy wiersz przechowuje informacje o pojedynczym obiekcie, nie zawiera kolekcji, posiada klucz główny (kolumnę lub**

**grupę kolumn jednoznacznie identyfikujących go w zbiorze) a dane są atomowe**. Zobaczmy, jak będzie wyglądała nasza struktura, jeśli spróbujemy doprowadzić ją do 1NF :



Results | Messages

| | NrPozycji | NumerZam... | NazwaKlienta | Adres | KodPocztowy | Miasto | Wojewodztwo | DataZamowienia | ElementZamowienia | Ilosc | CenaJedn | WartZamNetto | Vat | WartZamBrutto |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 101 | Jan Kowalski | ul. Jana Pawła 12 | 61-600 | Poznań | Wielkopolskie | 2012-01-02 00:... | Opony 205 R16 | 4 | 300,00 | 1200,00 | 23 | 1476,00 |
| 2 | 2 | 102 | Anna Dymna | ul. Staszica 1 | 30-600 | Kraków | Małopolskie | 2012-03-22 00:... | Alufelgi Silver | 4 | 550,00 | 2200,00 | 23 | 2706,00 |
| 3 | 3 | 103 | Piotr Wawrzyniak | al. Niepodległości 1 | 30-600 | Kraków | Małopolskie | 2012-03-22 00:... | Alufelgi Silver | 4 | 550,00 | 2200,00 | 23 | 2706,00 |
| 4 | 4 | 104 | Jan Kowalski | ul. Jana Pawła 12 | 61-600 | Poznań | Wielkopolskie | 2012-10-22 00:... | Komplet żarówek | 1 | 80,00 | 80,00 | 23 | 98,40 |
| 5 | 5 | 105 | Jan Kowalski | ul. Poznańska 8 | 21-120 | Wrocław | Dolnośląskie | 2012-05-22 00:... | Płyn do spryskiwacza | 1 | 10,00 | 15,00 | 23 | 18,45 |
| 6 | 6 | 105 | Jan Kowalski | ul. Poznańska 8 | 21-120 | Wrocław | Dolnośląskie | 2012-05-22 00:... | Trójkąt ostrzegawczy | 1 | 5,00 | 15,00 | 23 | 18,45 |

Projekt bazy danych powinien uwzględniać wszystkie wymogi biznesowe, obecne i przyszłe. Nie ma jednej recepty na najlepszy design.

Zwróć uwagę, że kolumna Adres nie jest atomowa. Wszystko tak naprawdę zależy od definicji atomowości, jaka nas satysfakcjonuje. Jeśli atomowa informacja o adresie to dla nas nazwa ulicy z numerem to wszystko jest ok. Co innego jeśli wymogi biznesowe potrzebują rozdzielenia tych informacji – np. system zleceń dla firmy kurierskiej. Obsługa jednej ulicy może być realizowana przez kilku kurierów (np. długa ulica Piotrkowska w Łodzi) – tu z pewnością trzeba by rozdzielić te informacje na dwie kolumny.

W tym miejscu, trzeba dodać jeszcze jedną istotną uwagę. Jeśli normalizujemy tabele, czyli rozdzielamy informacje na dane atomowe, konieczne jest określenie typu danych kolumn.

Stosujemy zawsze najmniejsze typy z możliwych, jakie są konieczne do spełnienia wymogów projektowych. Jeśli składujesz dane np. o dacie urodzin – interesuje nas zazwyczaj tylko data, nic więcej (chyba że w bazie szpitalnej, gdzie również interesująca jest informacja o godzinie narodzin). W większości przypadków, wybrany powinien zostać typ danych date – który zajmuje 3 bajty. Powszechnie popełnianym błędem jest nadużywanie dużych typów. Jeśli zostanie wybrany typ datetime – dla każdego rekordu, dla tej kolumny zostanie skonsumowane 8 bajtów. Niby niewiele, ale czy na pewno ?

- składowanie – różnicę (8-3=5 B) można pomnożyć przez np. N-milionów wystąpień (rekordów) – wtedy zaczyna robić wrażenie.
- pamięć RAM – podczas joinów tabel zamiast 3+3 czyli 6 bajtów na każdy łączony wiersz – zostanie zalokowane 8+8 = 16 B (10 B więcej, znów możemy rozpatrzeć to razy N-milionów wystąpień)
- zwiększony ruch sieciowy, obciążenie interfejsów kart sieciowych,
- zmniejszenie efektywności przechowywania wierszy na stronach – więcej operacji I/O
- a przecież takich kolumn z nadużyciem możemy mieć wiele w tabeli.

Pamiętajmy o tym podczas procesu projektowania czy też normalizacji tabel, że w dobry projekt bazy zaczyna się od fundamentu – czyli typów danych opisujących obiekty.

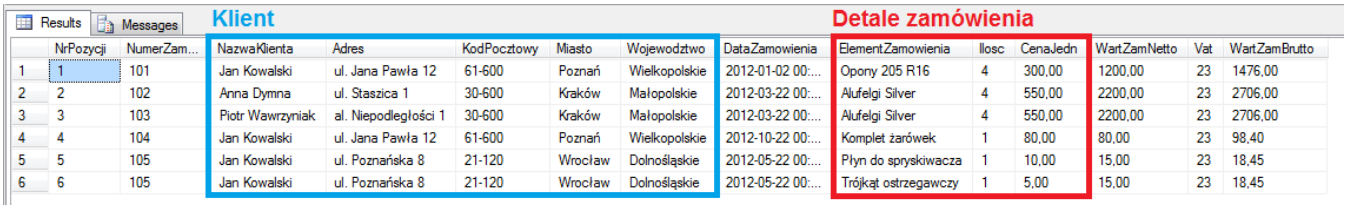
## Druga postać normalna 2NF

Ta postać określa esencję dobrego projektowania bazy. Mówi o tym, że **każda tabela powinna przechowywać dane dotyczące tylko konkretnej klasy obiektów**.

Jeśli mówimy o encji (tabeli) np. Klienci, to wszystkie kolumny opisujące elementy takiej encji, powinny dotyczyć Klientów a nie jakiś innych obiektów (np. ich zamówień).

Zatem normalizując do 2NF, wydzielić należy zbiór atrybutów (kolumn) który jest zależny tylko od klucza głównego. **Wszystkie atrybuty informacyjne (nie należące do klucza), muszą zawierać informacje o elementach tej konkretnej klasy** (encji, tabeli) a nie żadnej innej. Kolumny opisujące inne obiekty, powinny trafić do właściwych encji (tabel) w których te obiekty będziemy przechowywać.

W naszym przykładzie wykonajmy analizę bazy zamówień, która jest już w 1NF. Oznaczmy na początek atrybuty informacyjne które nie należą do klasy Zamówień (nie zależą funkcyjnie od klucza głównego).



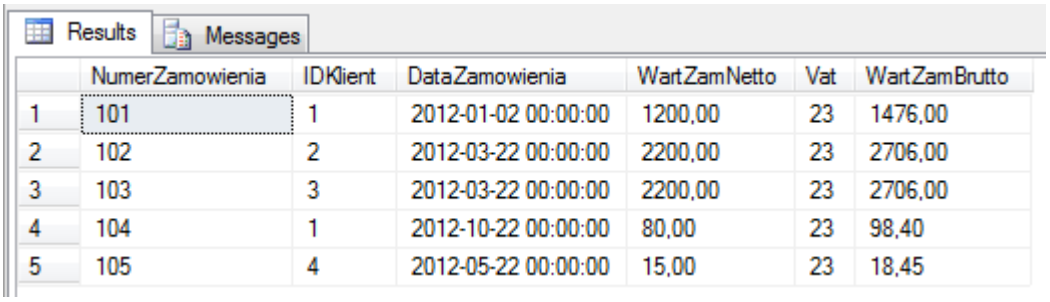

| | NrPozycji | NumerZam... | NazwaKlienta | Adres | KodPocztowy | Miasto | Wojewodztwo | DataZamowienia | ElementZamowienia | Ilosc | CenaJedn | WartZamNetto | Vat | WartZamBrutto |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 101 | Jan Kowalski | ul. Jana Pawła 12 | 61-600 | Poznań | Wielkopolskie | 2012-01-02 00:... | Opony 205 R16 | 4 | 300,00 | 1200,00 | 23 | 1476,00 |
| 2 | 2 | 102 | Anna Dymna | ul. Staszica 1 | 30-600 | Kraków | Małopolskie | 2012-03-22 00:... | Alufelgi Silver | 4 | 550,00 | 2200,00 | 23 | 2706,00 |
| 3 | 3 | 103 | Piotr Wawrzyniak | al. Niepodległości 1 | 30-600 | Kraków | Małopolskie | 2012-03-22 00:... | Alufelgi Silver | 4 | 550,00 | 2200,00 | 23 | 2706,00 |
| 4 | 4 | 104 | Jan Kowalski | ul. Jana Pawła 12 | 61-600 | Poznań | Wielkopolskie | 2012-10-22 00:... | Komplet żarówek | 1 | 80,00 | 80,00 | 23 | 98,40 |
| 5 | 5 | 105 | Jan Kowalski | ul. Poznańska 8 | 21-120 | Wrocław | Dolnośląskie | 2012-05-22 00:... | Płyn do spryskiwacza | 1 | 10,00 | 15,00 | 23 | 18,45 |
| 6 | 6 | 105 | Jan Kowalski | ul. Poznańska 8 | 21-120 | Wrocław | Dolnośląskie | 2012-05-22 00:... | Trójkąt ostrzegawczy | 1 | 5,00 | 15,00 | 23 | 18,45 |

Widać że istnieją w niej atrybuty związane z różnymi obiektami. Najbardziej rzucającymi się w oczy są z pewnością kolumny opisujące klasę Klientów – zaznaczone na niebiesko. Następnie można wydzielić również informacje o detalach zamówienia (ElementZamowienia, Ilosc, CenaJedn) – w czerwonej ramce.

Te wszystkie atrybuty, muszą powędrować do nowych tabeli – właściwych dla obiektów danego typu.

Finalny obraz 2NF będzie składał się więc z 4 tabel : Zamowienia, Klienci, DetaleZamowien i Produkty.

W wyniku tej operacji, tabela z informacjami o zamówieniach będzie wyglądała tak :




| | NumerZamowienia | IDKlient | DataZamowienia | WartZamNetto | Vat | WartZamBrutto |
|---|---|---|---|---|---|---|
| 1 | 101 | 1 | 2012-01-02 00:00:00 | 1200,00 | 23 | 1476,00 |
| 2 | 102 | 2 | 2012-03-22 00:00:00 | 2200,00 | 23 | 2706,00 |
| 3 | 103 | 3 | 2012-03-22 00:00:00 | 2200,00 | 23 | 2706,00 |
| 4 | 104 | 1 | 2012-10-22 00:00:00 | 80,00 | 23 | 98,40 |
| 5 | 105 | 4 | 2012-05-22 00:00:00 | 15,00 | 23 | 18,45 |

Nowe tabele będą wyglądały następująco :

Nowa tabela przechowująca informacje o obiektach typu Klient

Results | Messages

| | IDKlient | NazwaKlienta | Adres | KodPocztowy | Miasto | Wojewodztwo |
|---|---|---|---|---|---|---|
| 1 | 1 | Jan Kowalski | ul. Jana Pawła 12 | 61-600 | Poznań | Wielkopolskie |
| 2 | 2 | Anna Dymna | ul. Staszica 1 | 30-600 | Kraków | Małopolskie |
| 3 | 3 | Piotr Wawrzyniak | al. Niepodległości 1 | 30-600 | Kraków | Małopolskie |
| 4 | 4 | Jan Kowalski | ul. Poznańska 8 | 21-120 | Wrocław | Dolnośląskie |

Nowa tabela przechowująca informacje o detalach zamówień

Results | Messages

| | NumerZamowienia | KodProduktu | CenaJedn | Ilosc |
|---|---|---|---|---|
| 1 | 101 | 1 | 300,00 | 4 |
| 2 | 102 | 2 | 550,00 | 4 |
| 3 | 103 | 2 | 550,00 | 4 |
| 4 | 104 | 3 | 80,00 | 1 |
| 5 | 105 | 4 | 10,00 | 1 |
| 6 | 105 | 5 | 5,00 | 1 |

Nowa tabela przechowująca informacje o produktach

Results | Messages

| | KodProduktu | Nazwa | Producent | CenaJedn |
|---|---|---|---|---|
| 1 | 1 | Opony 205 R16 | Pirelli | 300,00 |
| 2 | 2 | Alufelgi Silver | ENZO | 550,00 |
| 3 | 3 | Opony wymiana | NULL | 80,00 |
| 4 | 4 | Plyn do spryskiwacza | GreenApple | 10,00 |
| 5 | 5 | Trójkat ostrzegawczy | GoSafer | 5,00 |

Jak widać usunięcie redundantnych informacji, skutkuje wzrostem złożoności struktury. Nie dość, że tworzymy nowe tabele, to jeszcze dokładamy kolumny np. IDKlient. Jednak, uwierz mi – nie jest to duża strata, zważywszy na zysk jaki nam daje ten krok normalizacji per saldo.

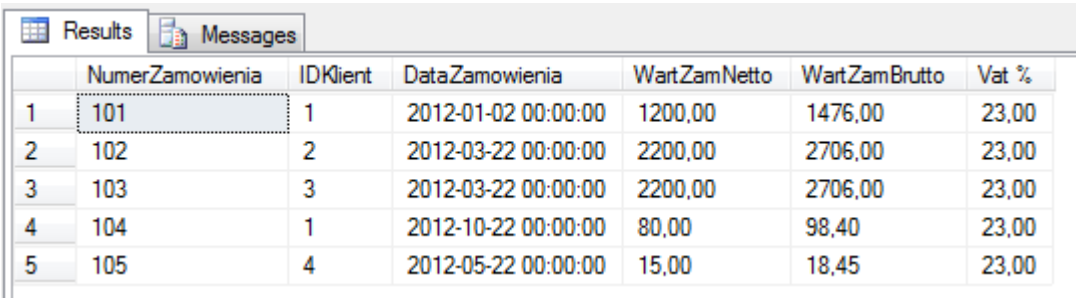
Relacyjne systemy bazy danych są projektowane do działania na zbiorach. Owszem koszt łączenia tabel bywa wysoki, ale zapewniając odpowiednie indeksy na łączonych kolumnach, jesteśmy w stanie zagwarantować równowagę pomiędzy wydajnością a plusami wynikającymi z normalizacji.

Doprowadziliśmy więc do sytuacji, w której każda tabela (encja) przechowuje informacje opisujące tylko obiekty właściwe dla niej. Cały proces, musi być bezstratny !

## Trzecia postać normalna 3NF

Trzecia postać normalna głosi, że **kolumna informacyjna nie należąca do klucza nie zależy też od innej kolumny informacyjnej**, nie należącej do klucza. Czyli każdy niekluczowy argument jest bezpośrednio zależny tylko od klucza głównego a nie od innej kolumny.

W naszym przypadku widać, że kolumny informacyjne związane z kosztami danego zamówienia oraz stawką podatku VAT, są ze sobą skorelowane. Z jednej z nich można śmiało zrezygnować, nie tracąc żadnej informacji (każda z nich zależy od klucza głównego, ale również od pozostałych dotyczących wartości). Możemy wyznaczyć np. wartość brutto na podstawie stawki VAT i wartości netto itd..


Results Messages

| | NumerZamowienia | IDKlient | DataZamowienia | WartZamNetto | WartZamBrutto | Vat % |
|---|---|---|---|---|---|---|
| 1 | 101 | 1 | 2012-01-02 00:00:00 | 1200,00 | 1476,00 | 23,00 |
| 2 | 102 | 2 | 2012-03-22 00:00:00 | 2200,00 | 2706,00 | 23,00 |
| 3 | 103 | 3 | 2012-03-22 00:00:00 | 2200,00 | 2706,00 | 23,00 |
| 4 | 104 | 1 | 2012-10-22 00:00:00 | 80,00 | 98,40 | 23,00 |
| 5 | 105 | 4 | 2012-05-22 00:00:00 | 15,00 | 18,45 | 23,00 |

Innym przykładem, mogą być kolumny wyliczeniowe wartości netto/brutto itp. ). Łatwo jednak w takich przypadkach podać przykłady, świadomego łamania 3NF na rzecz wydajności. Chociażby skomplikowane wyliczenia które muszą być wykonywane dla każdego wiersza, przy każdym zapytaniu.

## Zalety i wady normalizacji

Projektowanie baz danych z pewnością nie jest trywialne. Pierwszym krokiem, powinna być właściwa analiza wymagań biznesowych, uwzględniająca przyszłe potrzebny i możliwości skalowania projektu.

Konieczne jest świadomość istnienia potencjalnych zagrożeń i wyczucie, wynikające z pewnością z doświadczenia – aby odpowiedzieć na pytanie jak mocno normalizować tabele. Najczęściej spotykany poziom to 3 postać normalna. Są sytuacje, zresztą nasuwające się intuicyjnie, w których zależy nam na procesie odwrotnym – denormalizacji celem maksymalnego zwiększenia szybkości wykonywania zapytań.

Poniżej kilka istotnych cech związanych z normalizowaniem baz danych :

- zmniejszamy ogólną liczbę danych przechowywanych w bazach
- rozwiązujemy problemy anomalii dodawania, modyfikacji i usuwania informacji (rekordów) z bazy
- czasem spowalniamy wykonywanie zapytań – cena jaką płacimy za konieczność łączenia tabel.
- ale też przyspieszamy wykonywanie określonych zapytań – tworzymy osobne tabele więc możemy utworzyć więcej indeksów klastrowych, lepsza efektywność przechowywania danych w tabelach.
- wąskie tabele to bardziej efektywne przetwarzanie i składowanie ich na dysku, mniej operacji I/O.
- lepsze zarządzanie transakcjami – szybsze wykonywanie update'ów, mniej blokad.

# Modele relacyjne (ERD)

Z punktu widzenia relacyjnej bazy danych świat rzeczywisty widzimy i analizujemy jako zestaw encji i związków zachodzących między nimi.

- Encja
  Encją jest każdy przedmiot, zjawisko, stan lub pojęcie, czyli każdy obiekt, który potrafimy odróżnić od innych obiektów (na przykład: osoba, samochód, książka, stan pogody). Encje podobne do siebie (opisywane za pomocą podobnych parametrów) grupujemy w zbiory encji. Projektując bazę danych, należy precyzyjnie zdefiniować encje i określić parametry, przy użyciu których będą opisywane.
- Atrybut
  Encje mają określone cechy wynikające z ich natury. Cechy te nazywamy atrybutami. Zestaw atrybutów, które określamy dla encji, zależy od potrzeb bazy danych.
- Dziedzina
  Atrybuty encji mogą przyjmować różne wartości. Projektując bazę danych, możemy określić, jakie wartości może przyjmować dany atrybut. Zbiór wartości atrybutu nazywamy dziedziną (domeną).

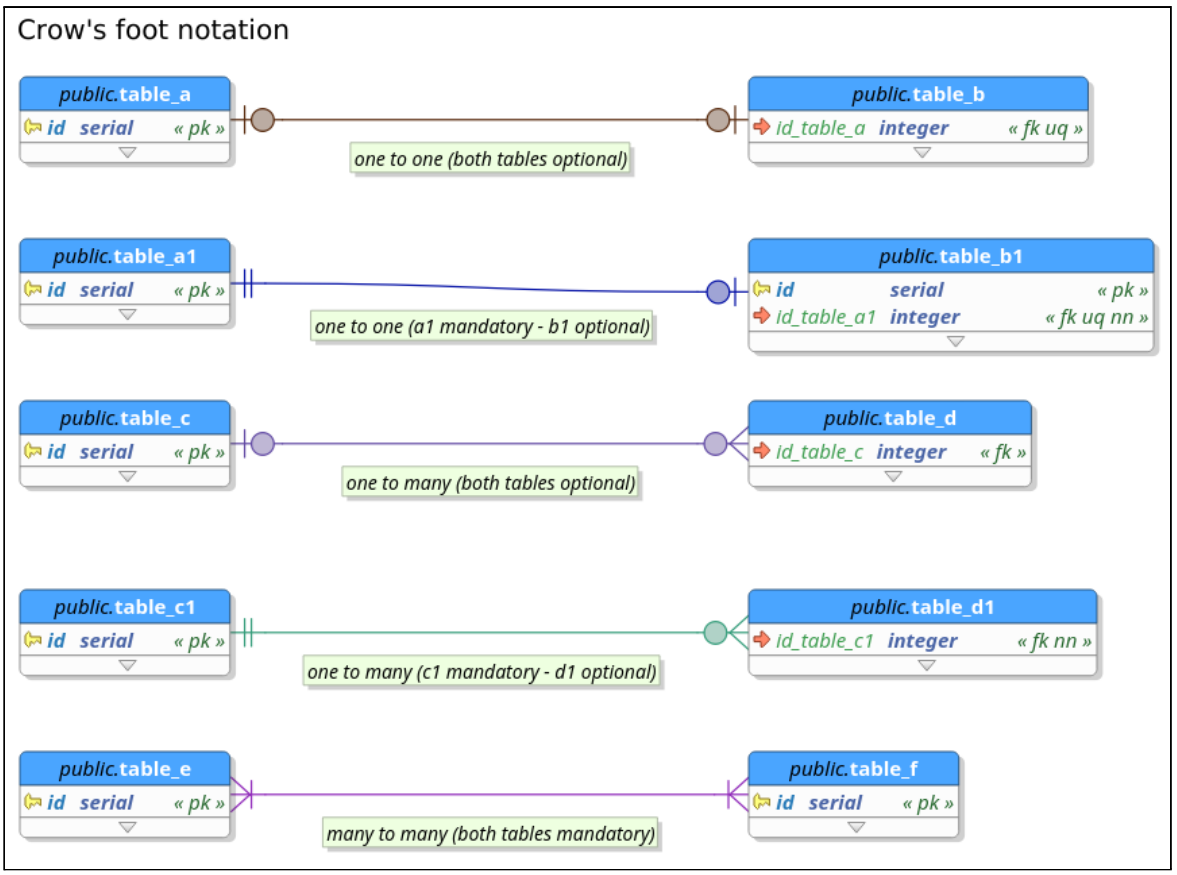
Ponieważ trudno uniknąć błędów podczas wprowadzania danych, należy odpowiednio zabezpieczyć się przed nimi na etapie projektowania bazy danych.
Należy pamiętać, że prawidłowa klasyfikacja danych jest podstawą dobrego projektu bazy danych. Najgorsze efekty otrzymujemy, próbując zaprojektować bazę danych, która będzie gromadziła wszystkie możliwe dane i przetwarzała je na wszystkie możliwe sposoby. Każda encja powinna mieć przynajmniej jeden atrybut lub kombinację kilku atrybutów, które identyfikują ją jednoznacznie. Ten atrybut to **klucz podstawowy encji**.
Jeśli klucz podstawowy składa się z kilku atrybutów, dobrym rozwiązaniem jest zastąpienie go **kluczem sztucznym**. Najczęściej zawiera on unikatowe liczby przypisane kolejnym encjom.

Konceptualne projektowanie bazy danych to konstruowanie schematu danych niezależnego od wybranego modelu danych, docelowego systemu zarządzania bazą danych, programów użytkowych czy języka programowania.

Do tworzenia modelu graficznego schematu bazy danych wykorzystywane są diagramy związków encji, z których najpopularniejsze są diagramy ERD (ang. Entity Relationship Diagram). Pozwalają one na modelowanie struktur danych oraz związków zachodzących między tymi strukturami. Nadają się szczególnie do modelowania relacyjnych baz danych, ponieważ umożliwiają prawie bezpośrednie przekształcenie diagramu w schemat relacyjny.

Diagramy ERD składają się z trzech rodzajów elementów:

- zbiorów encji,
- atrybutów encji,
- związków zachodzących między encjami.




## Klucze główne

Klucz główny (ang. primary key) to nic innego jak unikalny identyfikator każdego rekordu w tabeli. Słówko unikalny mówi o tym, że wartości w tej kolumnie nie mogą się powtarzać. Kolumna która jest kluczem głównym nie może także przechowywać wartości NULL, czyli wartości nieokreślonych/nieznanych. W jednej tabeli może być tylko jeden klucz główny, ale może się on składać z kilku kolumn.

Klucz główny możemy zdefiniować podczas tworzenia tabeli

```
CREATE SCHEMA IF NOT EXISTS model;

CREATE TABLE model.pracownicy (
   id serial PRIMARY KEY,
   nazwisko text,
   imie text,
   stanowisko text
);
```

```
CREATE TABLE model.place (
   stanowisko text PRIMARY KEY,
   stawka double precision
);
```

lub podczas modyfikacji jej definicji

```
ALTER TABLE pracownicy ADD PRIMARY KEY (id);
ALTER TABLE place ADD PRIMARY KEY (stanowisko);
```

## Klucze obce

Klucz obcy służy do definiowania relacji między tabelami. Kolumnę (lub kolumny) którą zdefiniujemy jako klucz obcy w jednej tabeli wiążemy z kolumną (kolumnami) która jest kluczem głównym w drugiej tabeli. Oznacza to, że wartości przechowywane w kolumnie która jest zdefinowana jako klucz obcy w pierwszej tabeli zawsze będą miały swój odpowiednik w kolumnie która jest zdefiniowana jako klucz główny w drugiej tabeli.

Klucz obcy możemy zdefiniować podczas tworzenia tabeli

```
DROP TABLE IF EXISTS model.pracownicy;

CREATE TABLE model.pracownicy (
   id serial PRIMARY KEY,
   nazwisko text,
   imie text,
   stanowisko text REFERENCES model.place (stanowisko)
);
```

Lub podczas modyfikacji jej definicji

```
ALTER TABLE pracownicy
     ADD CONSTRAINT prac_plac FOREIGN KEY (stanowisko)
     REFERENCES place (stanowisko);
```

Powyższe zapytanie się nie uda, ponieważ dane istniejące w bazie naruszają reguły klucza obcego - w tabeli pracownicy istnieje pozycja ze stanowiskiem ‘asystent’ którego nie ma w tabeli place. Jeśli dodamy tą pozycję w tabeli place

```
INSERT INTO place VALUES ('asystent', 4);
```

Ponowne założenie klucza obcego powinno się udać.

# Rozszerzenie PostGIS

PostGIS, jak podaje główna strona projektu to rozszerzenie przestrzennej bazy danych dla obiektowo-relacyjnej bazy danych PostgreSQL. Dodaje obsługę obiektów geograficznych, umożliwiając wykonywanie zapytań o lokalizację w języku SQL.

PostGIS dodaje dodatkowe typy (geometria, geografia, raster i inne) do bazy danych PostgreSQL. Dodaje również funkcje, operatory i rozszerzenia indeksów, które mają zastosowanie do tych typów przestrzennych. Te dodatkowe funkcje, operatory, powiązania indeksów i typy zwiększają moc bazy PostgreSQL, czyniąc z niej szybki, bogaty w funkcje i niezawodny system zarządzania przestrzenną bazą danych.

## Typy danych

Postgis definiuje dwa typy danych przestrzennych:

- geometry - używany do przechowywania danych w płaskich (euklidesowych) układach współrzędnych
- geography - używany do przechowywania danych w kątowych układach współrzędnych

Typy danych możemy definiować ogólnie (np: geometry) i dokładnie (np: geometry(point,4326)). W pierwszym przypadku baza danych dopuści zapisanie w polu każdej geometrii, którą będziemy chcieli wstawić, w drugim przypadku zezwoli tylko na punkty w odwzorowaniu EPSG:4326

## Kiedy użyć geometry a kiedy geography

Typ danych GEOGRAPHY pozwala przechowywać dane używając długości i szerokości geograficznej - bez wchodzenia w zawiłości odwzorowań i układów współrzędnych. Uproszczenie to niesie ze sobą również pewne wady:

- nie wszystkie funkcje obsługują format GEOGRAPHY
- te, które obsługują działają wolniej niż na danych typu GEOMETRY

W takim razie kiedy użyć którego typu?

Typu GEOMETRY lepiej użyć jeśli:

- przetwarzamy dane przestrzenne z niewielkiego obszaru (średniej wielkości państwo)
- wiemy co to odwzorowania i wiemy jak ich używać

- zależy nam na wydajności funkcji przestrzennych

Typu GEOGRAPHY lepiej użyć jeśli:

- przetwarzamy dane na większym obszarze - kontynent lub cała planeta
- nie wiemy co to odwzorowania i nie mamy zamiaru się ich uczyć
- wydajność funkcji przestrzennych nie jest dla nas najistotniejsza

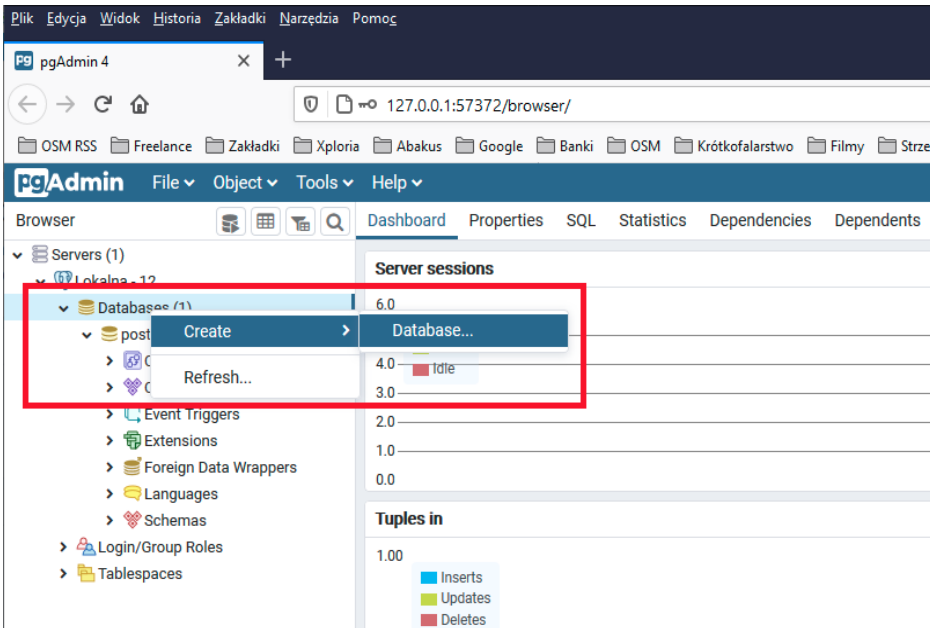
## Omówienie grup funkcji wprowadzanych przez postgis

Postgis rozszerza bazę PostgreSQL o około 300 funkcji, które funkcjonalnie możemy podzielić na kilka grup:

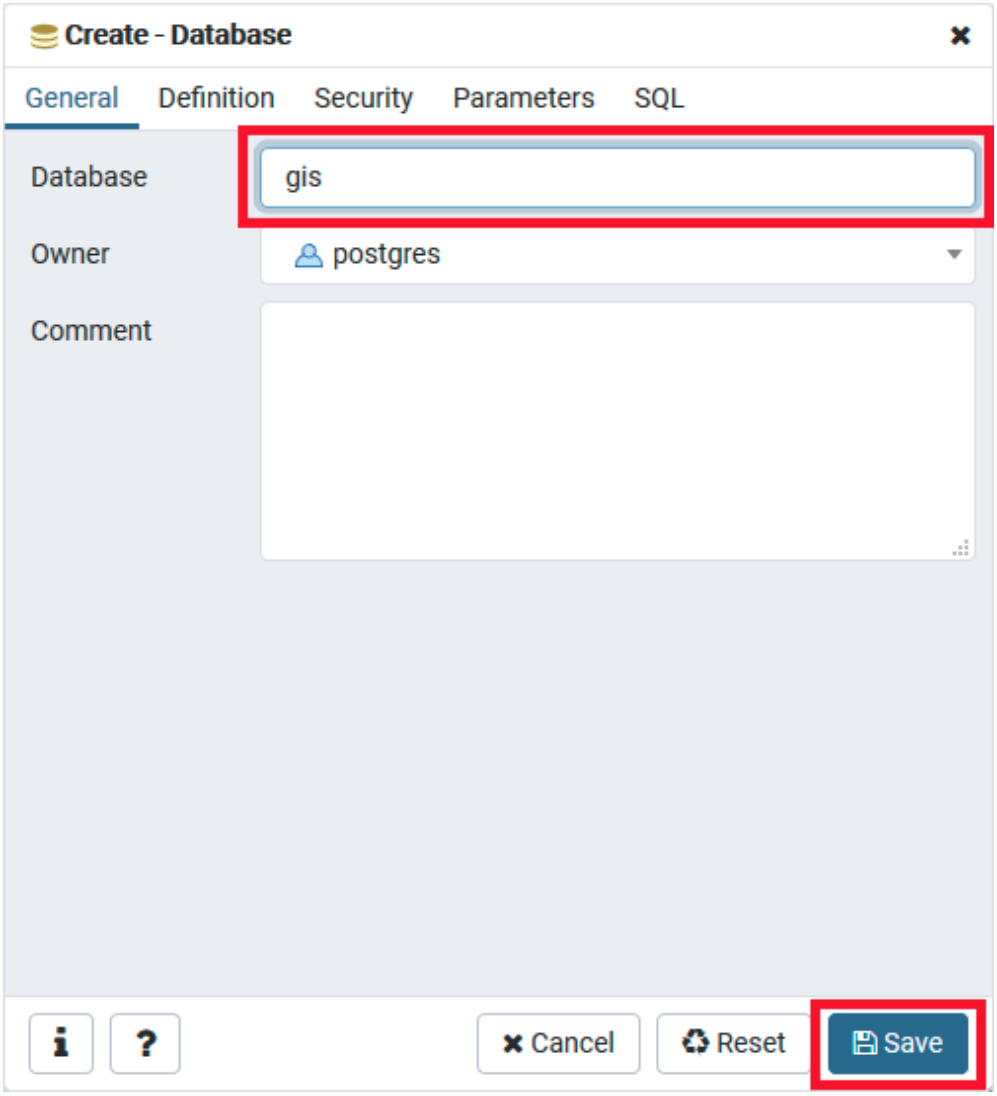
- Funkcje zarządzania tabelami
  Funkcje umożliwiające dodawanie, usuwanie tabel oraz kolumn w tabelach, zwracanie i zmianę zdefiniowanego układu współrzędnych
- Konstruktory
  Funkcje umożliwiające tworzenie geometrii.
- Akcesory
  Zwracają informacje o geometriach
- Edytory
  Funkcje pozwalające na edycje geometrii.
- Walidatory
  Sprawdzanie czy geometria jest poprawna, zwracanie błędów
- Funkcje układów współrzędnych
  Umożliwiają ustawianie, sprawdzanie układu współrzędnych oraz reprojekcje.
- Funkcje wprowadzania geometrii
  Pozwalają na tworzenie geometrii na podstawie znanych reprezentacji.
- Funkcje zwracające geometrie
  Zwracają geometrie w zdefiniowanych reprezentacjach.
- Operatory
  Słowa kluczowe pozwalające na badanie zależności między geometriami.
- Funkcje relacji przestrzennych
  Funkcje badające przecięcia, nakładanie, stykanie, zawieranie się geometrii.
- Funkcje pomiarowe
  Funkcje pozwalające na pomiary geometrii i zależności między nimi.
- Procesory
  Funkcje umożliwiające zmiany geometrii.
- Przekształcenia afiniczne
  Funkcje umożliwiające przekształcenia afiniczne (pokrewne) geometrii.
- Funkcje referencji liniowej
  Funkcje pozwalające na pracę z liniami.

# Utworzenie bazy danych i rozszerzeń przestrzennych

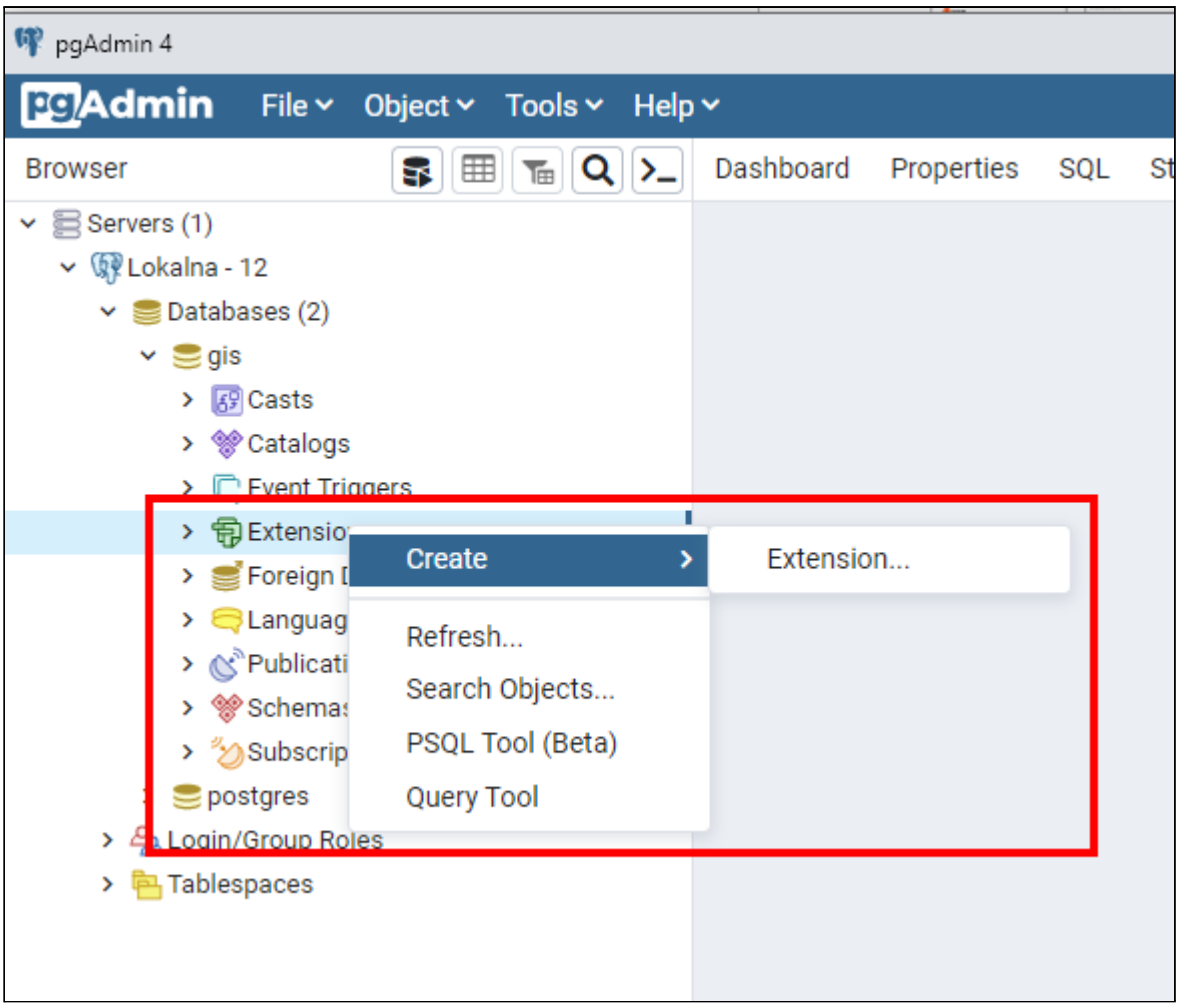
Na potrzeby dalszej części szkolenia utworzymy nową bazę danych. W tym celu w aplikacji PGAdmin klikamy prawym przyciskiem myszy na pozycji `databases` i z menu kontekstowego wybieramy `create > database`



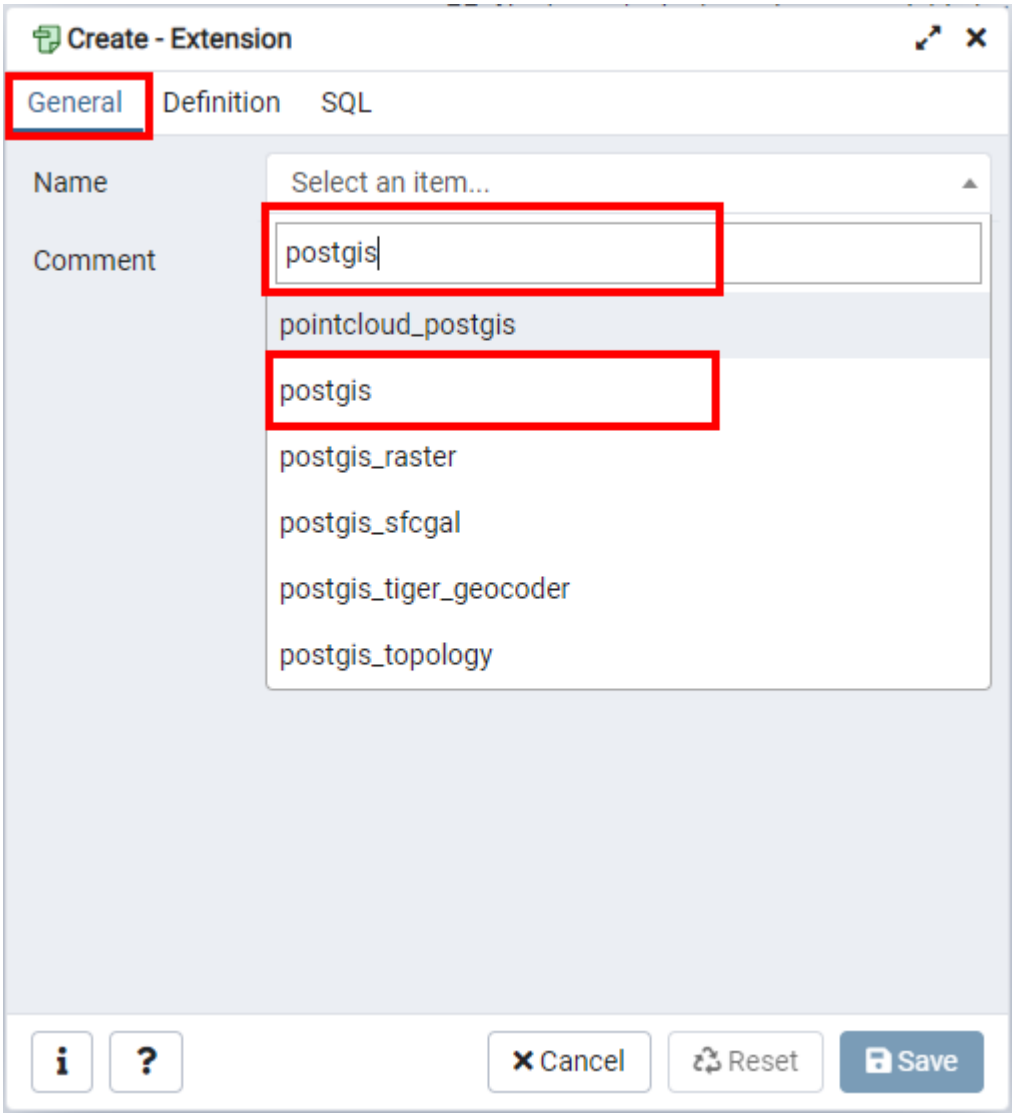

W oknie tworzenia bazy danych w zakładce `General` podajemy jej nazwę - na potrzebę szkolenia sugerujemy nazwę `gis`, i wybieramy `save`




W serwerze powinna zostać utworzona kolejna baza danych - dwuklik w nazwę powinien połączyć aplikację i wyświetlić jej komponenty. Do pracy będziemy potrzebowali jeszcze rozszerzeń, dlatego odnajdujemy na liście `extensions`, klikamy prawym przyciskiem myszy i wybieramy opcję `create > extension`.




W zakładce `General`, polu `name` wyszukujemy `postgis` i wybieramy z listy.

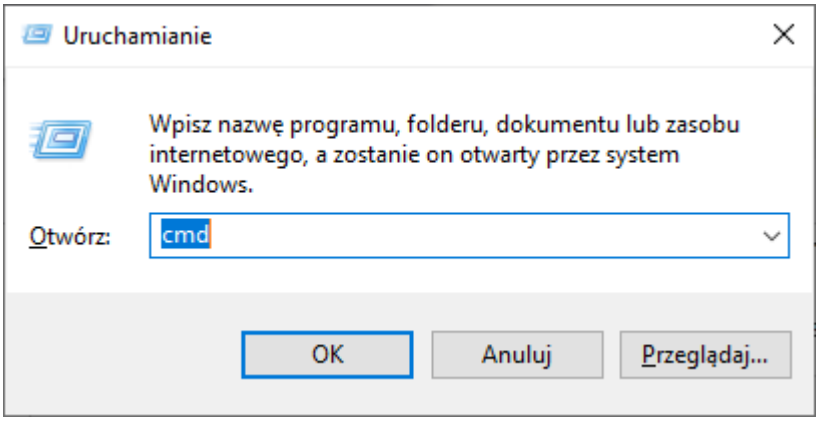



Powyższe kroki powtarzamy w celu instalacji rozszerzenia postgis_raster.

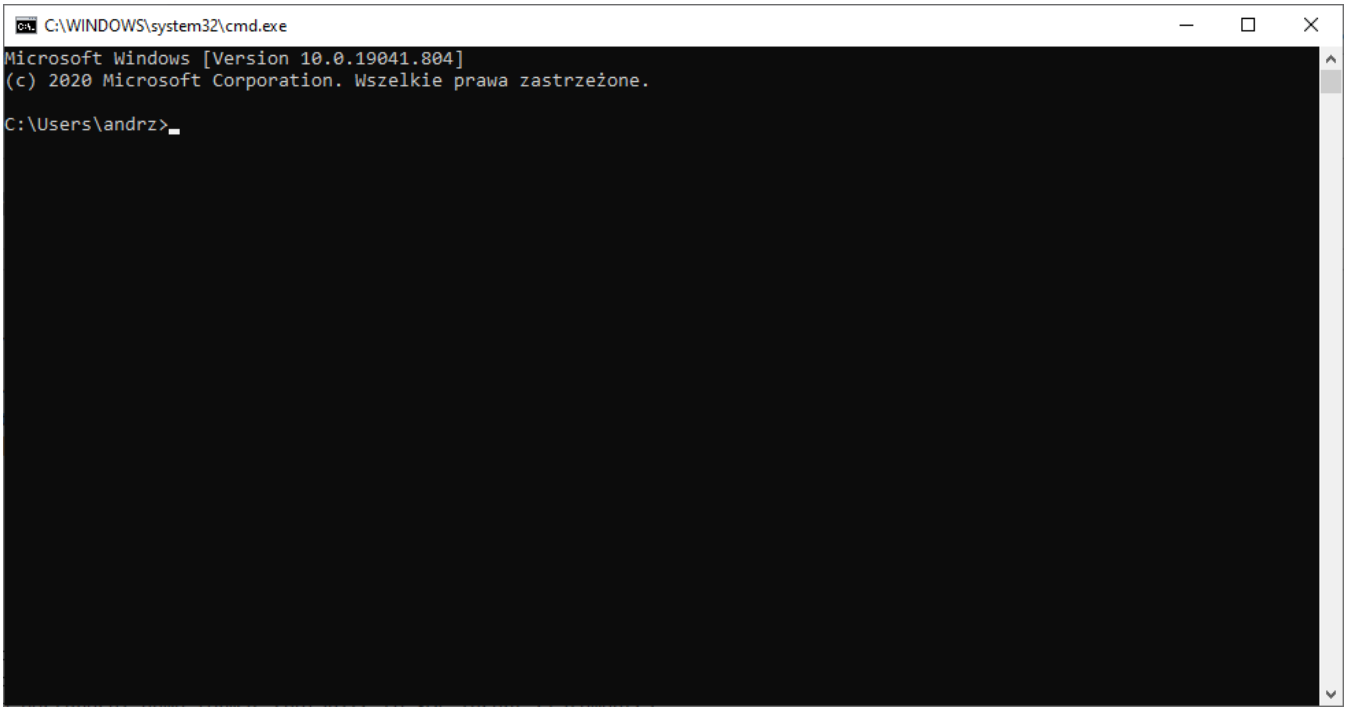
Rozszerzenie postgis dodaję do bazy typy danych i funkcje potrzebne do przechowywania danych wektorowych, rozszerzenie postgis_raster do danych rastrowych.

## Import danych wektorowych do bazy danych

Rozszerzenie PostGIS udostępnia aplikację pozwalającą na import danych wektorowych do bazy danych - shp2pgsql. Aby uruchomić aplikację musimy najpierw uruchomić wiersz polecenia - w tym celu wybieramy skrót klawiaturowy Windows + r i w pojawiającym się oknie wpisujemy cmd, po czym zatwierdzamy przyciskiem OK.

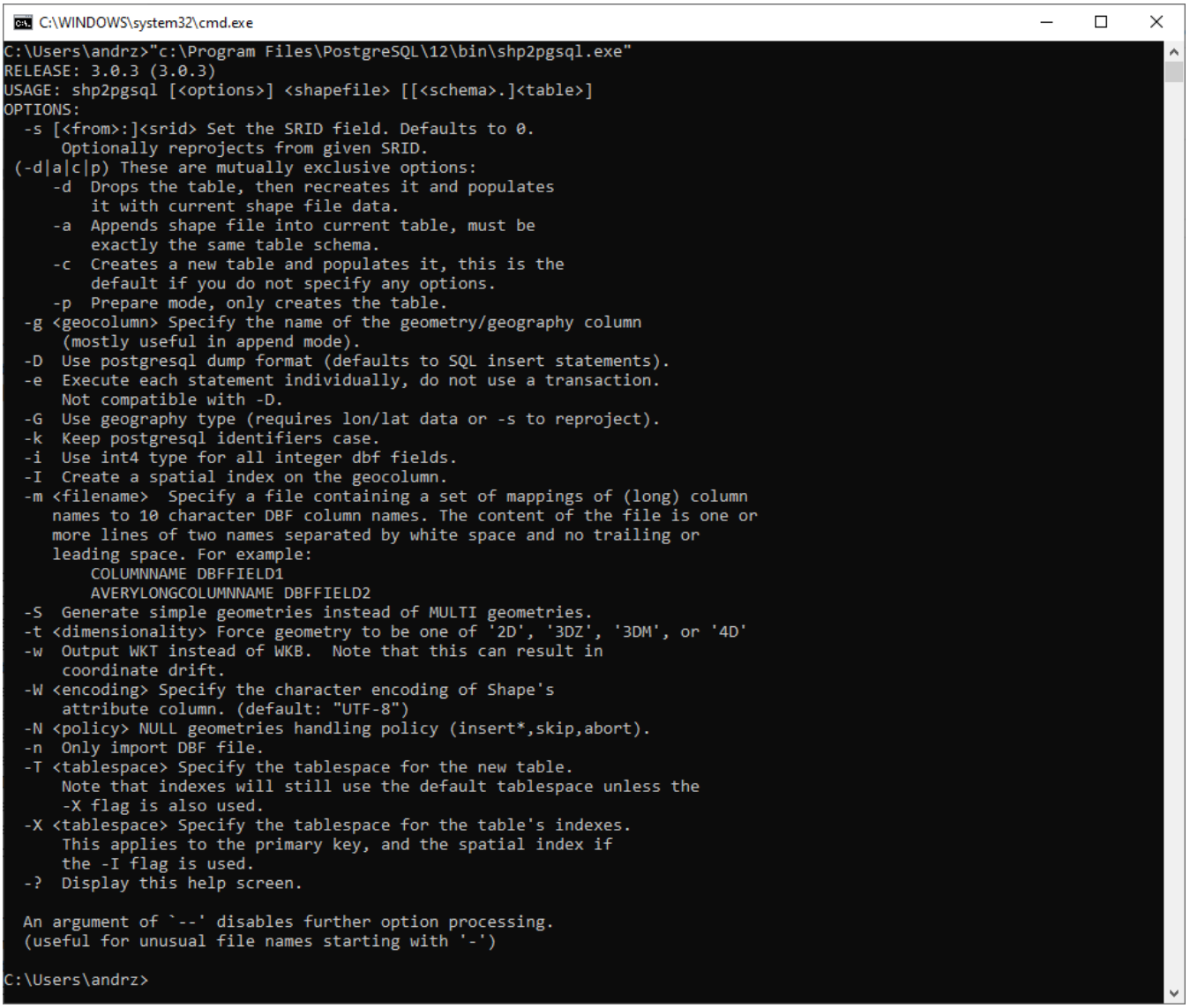



Po zatwierdzeniu powinna pojawić się konsola tekstowa:




Opcje aplikacji możemy uzyskać uruchamiając ją bez żadnych parametrów.

```
"c:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe"
```



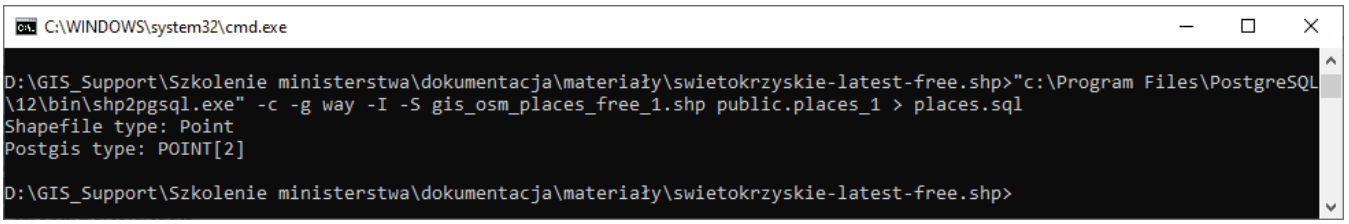

Dane konwertujemy za pomocą polecenia:

```
"c:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe" -c -g way -I -S
gis_osm_places_free_1.shp public.places_1 > places.sql
```

Analiza polecenia:

- **c:\Program Files\PostgreSQL\13\bin\shp2pgsql.exe** - pełna ścieżka do programu shp2pgsql.exe
- **-c** - tworzy nową tabelę w bazie
- **-g way** - kolumna z geometrią będzie miała nazwę way
- **-I** - tworzy indeks przestrzenny
- **-S** - wymusza tworzenie geometrii prostych zamiast złożonych
- **gis_osm_places_free_1.shp** - nazwa pliku, który będziemy przetwarzać
- **public.places_1** - nazwa schematu i tabeli w bazie do której będziemy importować
- **> places.sql** - przekierowanie wyjścia aplikacji do pliku places.sql

Jeśli wszystko przebiegło prawidłowo polecenie powinno wyświetlić na konsoli następujący komunikat:

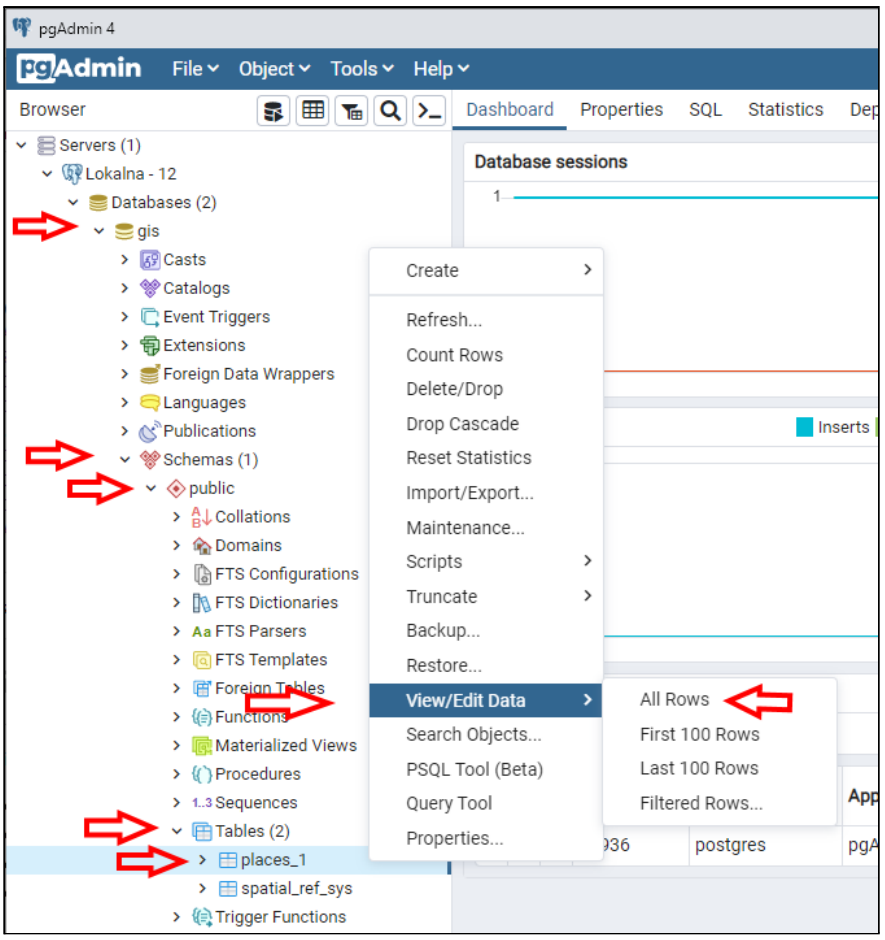



W bieżącym folderze powinien zostać utworzony plik o nazwie **places.sql**, który importujemy za pomocą aplikacji psql poleceniem

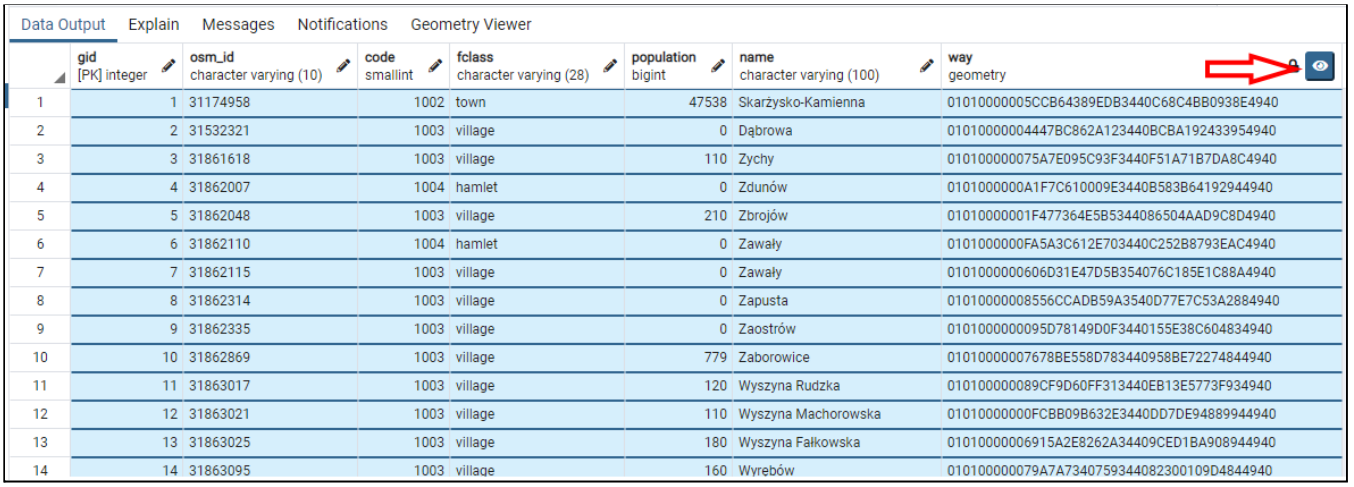
```
"c:\Program Files\PostgreSQL\13\bin\psql.exe" -f places.sql -U postgres gis
```

Jeśli wszystko przebiegło prawidłowo po wielu liniach zawierających informację o importowaniu kolejnego rekordu konsola powinna wyświetlić następujące informacje:


Zaimportowane dane wektorowe możemy wyświetlić za pomocą aplikacji pgAdmin, w tym celu w drzewie przeglądarki rozwijamy kolejno gis > Schemas > public > Tables, odnajdujemy tabelę places_1, klikamy na niej prawym przyciskiem myszy i z menu kontekstowego wybieramy opcję View/Edit Data > All Rows

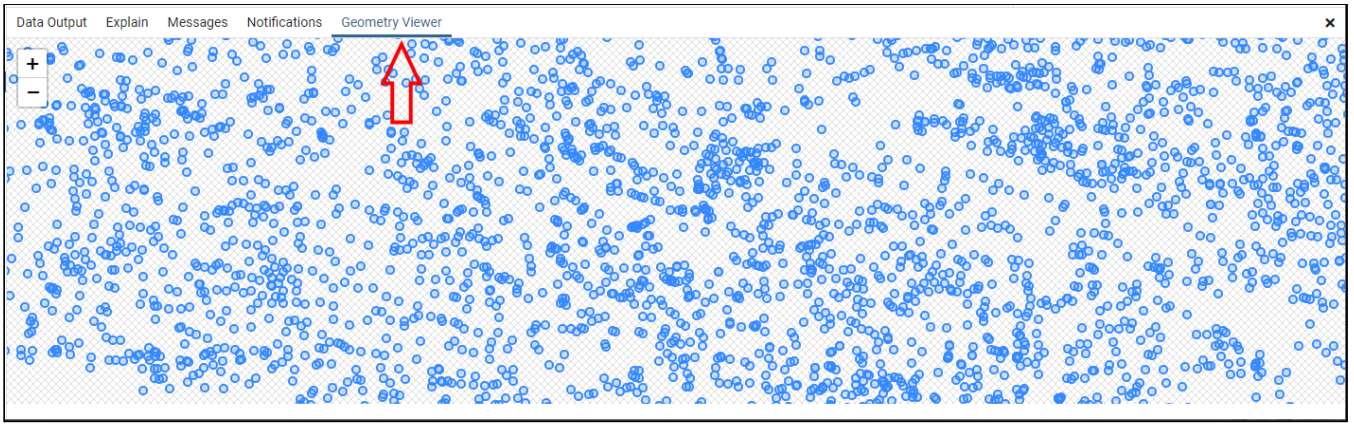



W oknie przeglądania danych wybieramy ikonę oka znajdującą się w nagłówku kolumny z geometrią:




| | gid [PK] integer | osm_id character varying (10) | code smallint | fclass character varying (28) | population bigint | name character varying (100) | way geometry |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 31174958 | 1002 | town | 47538 | Skarżysko-Kamienna | 01010000005CCB64389EDB3440C68C4BB0938E4940 |
| 2 | 2 | 31532321 | 1003 | village | 0 | Dąbrowa | 01010000004447BC862A123440BCBA192433954940 |
| 3 | 3 | 31861618 | 1003 | village | 110 | Zychy | 010100000075A7E095C93F3440F51A71B7DA8C4940 |
| 4 | 4 | 31862007 | 1004 | hamlet | 0 | Zdunów | 0101000000A1F7C610009E3440B583B64192944940 |
| 5 | 5 | 31862048 | 1003 | village | 210 | Zbrojów | 01010000001F477364E5B5344086504AAD9C8D4940 |
| 6 | 6 | 31862110 | 1004 | hamlet | 0 | Zawały | 0101000000FA5A3C612E703440C252B8793EAC4940 |
| 7 | 7 | 31862115 | 1003 | village | 0 | Zawały | 0101000000606D31E47D5B354076C185E1C88A4940 |
| 8 | 8 | 31862314 | 1003 | village | 0 | Zapusta | 01010000008556CCADB59A3540D77E7C53A2884940 |
| 9 | 9 | 31862335 | 1003 | village | 0 | Zaostrów | 0101000000095D78149D0F3440155E38C604834940 |
| 10 | 10 | 31862869 | 1003 | village | 779 | Zaborowice | 01010000007678BE558D783440958BE72274844940 |
| 11 | 11 | 31863017 | 1003 | village | 120 | Wyszyna Rudzka | 010100000089CF9D60FF313440EB13E5773F934940 |
| 12 | 12 | 31863021 | 1003 | village | 110 | Wyszyna Machorowska | 01010000000FCBB09B632E3440DD7DE94889944940 |
| 13 | 13 | 31863025 | 1003 | village | 180 | Wyszyna Fałkowska | 01010000006915A2E8262A34409CED1BA908944940 |
| 14 | 14 | 31863095 | 1003 | village | 160 | Wyrebów | 010100000079A7A73407593440823001 09D4844940 |

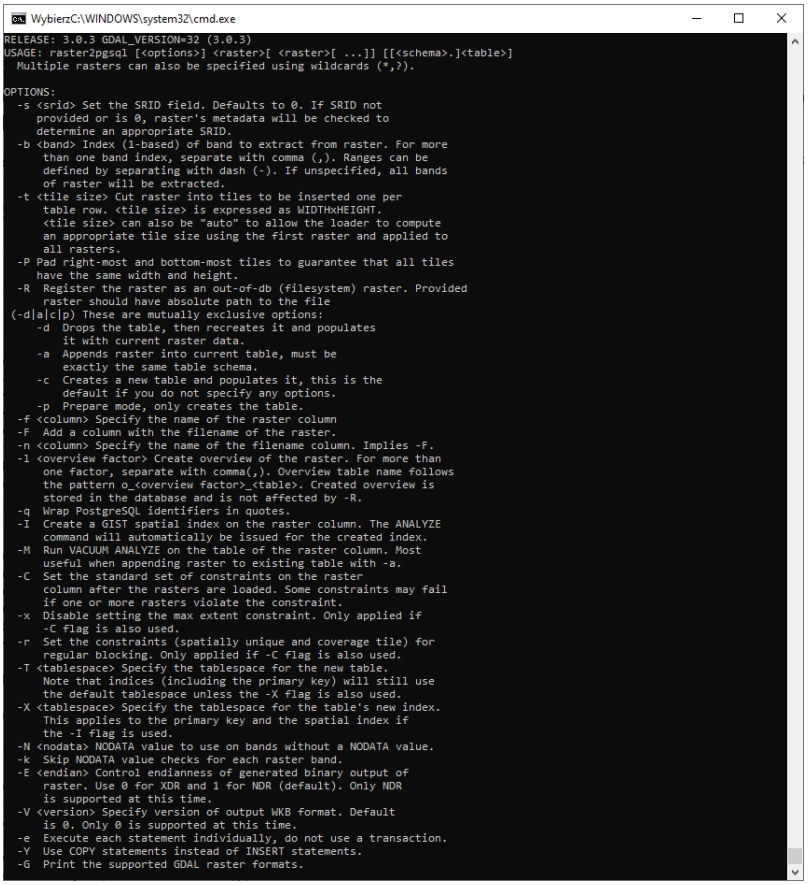
Nasza warstwa zostanie wyświetlona w zakładce Geometry Viewer




# Import danych rastrowych do bazy danych

Importy danych rastrowych do bazy wykonywane są za pomocą narzędzia raster2pgsql. Krótką pomoc na temat programu uzyskamy uruchamiając go bez żadnych parametrów.

```
"c:\Program Files\PostgreSQL\13\bin\raster2pgsql.exe"
```



Do celu szkolenia użyjemy rastra z wysokością terenu powstałego przez reprojekcję i przycięcie zasobu EU_DEM utworzonego w ramach projektu Copernicus. Plik dostępny jest w plikach szkolenia pod nazwą EUDEM_swietokrzyskie_2180.tif

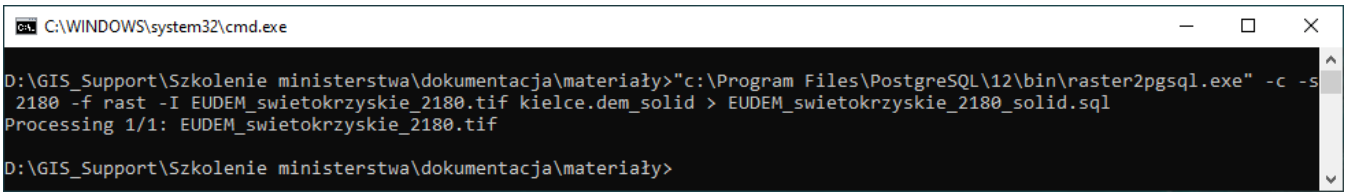
Aby zaimportować raster do bazy uruchamiamy komendę:

```
"c:\Program Files\PostgreSQL\13\bin\raster2pgsql.exe" -c -s 2180 -f rast
-I EUDEM_swietokrzyskie_2180.tif public.dem_solid >
EUDEM_swietokrzyskie_2180_solid.sql
```

Analiza polecenia:

- **"c:\Program Files\PostgreSQL\13\bin\raster2pgsql.exe"** - pełna ścieżka do aplikacji
- **-c** - tworzymy nową tabelę w bazie danych
- **-s 2180** - kod EPSG odwzorowania rastra
- **-f rast** - kolumna z rastrem będzie sie nazywała rast
- **-I** - tworzymy indeks przestrzenny
- **EUDEM_swietokrzyskie_2180.tif** - nazwa konwertowanego pliku
- **public.dem_solid** - nazwa tabeli w bazie danych
- **> EUDEM_swietokrzyskie_2180_solid.sql** - przekieruj wyjście do pliku

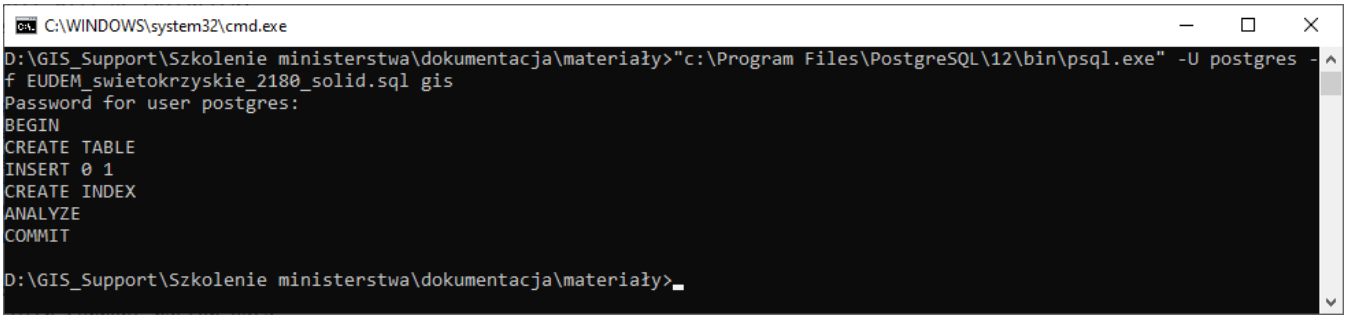
Konsola powinna wyświetlić następującą informację



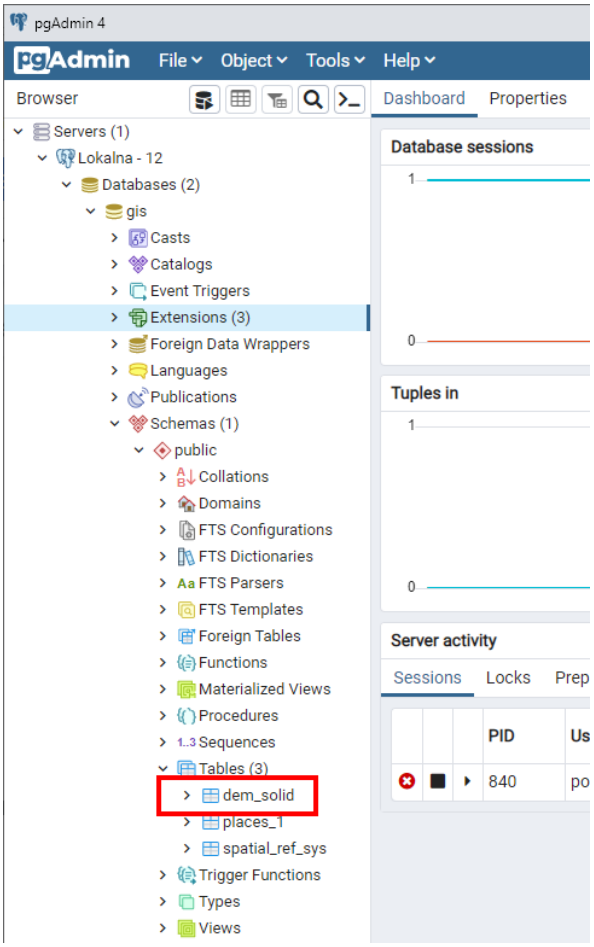

Wynikowy plik sql importujemy za pomocą aplikacji psql:

```
"c:\Program Files\PostgreSQL\13\bin\psql.exe" -U postgres -f
EUDEM_swietokrzyskie_2180_solid.sql gis
```

Jeśli operacja przebiegnie pomyślnie konsola powinna wyświetlić poniższe komunikaty



Obecność tabeli z danymi rastrowymi w bazie danych możemy potwierdzić za pomocą aplikacji pgAdmin




Niestety w przeciwieństwie do danych wektorowych nie pozwala ona na wyświetlenie przechowywanych w bazie danych rastrowych.

| | |
|---|---|
|  | Do efektywnego zarządzania zarówno danymi rastrowymi jak i wektorowymi w bazie danych zalecamy używanie aplikacji QGIS. |

Źródła:

1. Szkolenie to oparte jest w głównej mierze na materiałach: Materialy_szkoleniowe_-_SQL_w_PostgreSQL_poziom_podstawowy.pptx.pdf (ekoportal.gov.pl)
2. Rozdział Projektowanie baz danych wzorowany jest na artykule “Projektowanie i normalizacja baz danych” dostępnym na portalu sqlpedia.pl.
3. Rozdział Modele Relacyjne wzorowany jest na artykule “Projektowanie bazy danych” dostępnym na portalu informatyka.orawskie.pl
4. Rozdział Konwersja typów danych wzorowany jest na artykule “Typy i konwersje danych” dostępnym na portalu centrumxp.pl.