DATABASES IN BIOLOGY AND MEDICINE

SQL

Biomedical Engineering

Adam Piórkowski

pioro@agh.edu.pl

http://home.agh.edu.pl/~pioro/dyd/

http://home.agh.edu.pl/~pioro/dyd/DBiBaM/

© AP, All rights reserved

Kraków, 2025–2026

SQL

- SQL is a universal, standardized language for communication between a client and a database. Among its many dialects, the most important are the standards, including ANSI SQL and SQL2 (1992).

- SQL is not a procedural language, but a declarative, descriptive language. Its use consists of defining individual commands. These commands can be executed interactively (through terminal-based communication between the user/administrator and the database) or embedded (as an integrated part of an application).

- SQL – [abbreviation?] – SQL is not structural; it is more than just queries.

SQL Syntax

- keywords – a set of more than 200 reserved words used for commands and operators; for clarity, they should preferably be written in uppercase

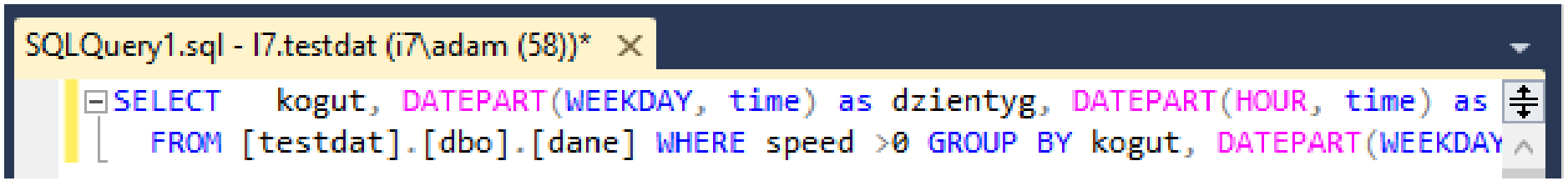
- identifiers – used to distinguish user-created objects in a database,

- operators:
- arithmetic,
- logical,
- comparison,
- text,

- literals – numbers and character strings enclosed in apostrophes (or quotation marks),

```
select kogut, datepart(weekday, time) as
dzientyg, datepart(hour, time) as godzina,
avg(cast (speed as decimal(5,2))), count(*)
from karetkigps where speed > 0 group by
kogut, datepart(weekday, time), datepart(hour,
time) order by kogut, datepart(weekday, time),
datepart(hour, time);
```

```
SELECT kogut, DATEPART(WEEKDAY, time) as
dzientyg, DATEPART(HOUR, time) as godzina,
AVG(CAST (speed AS DECIMAL(5,2))), COUNT(*)
FROM karetkiGPS WHERE speed > 0 GROUP BY
kogut, DATEPART(WEEKDAY, time), DATEPART(HOUR,
time) ORDER BY kogut, DATEPART(WEEKDAY, time),
DATEPART(HOUR, time);
```



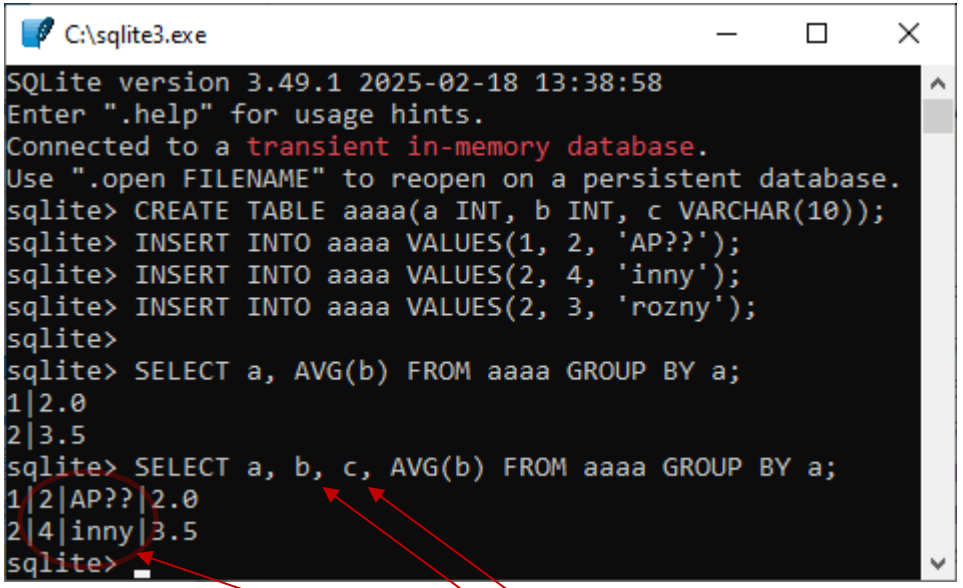
SQL Syntax

- **DDL –** Data Definition Language
  np. `CREATE` (tabele, widoki, itp.), `ALTER` (zmiana), `DROP` (usuwanie)
- **DQL –** Data Query Language
  `SELECT` …
- **DML –** Data Manipulation Language
  `INSERT, UPDATE, DELETE`
  czasem też `SELECT INTO`
- **DCL –** Data Control Language
  `GRANT, REVOKE`

Connecting to a DBMS

```
-- PSQL
psql -h 149.156.XXX.YYY -U student1234 nazwabazy

-- prawa dostępu ustala sie
-- w pliku pg_hba.conf
```




```
-- MySQL
CREATE DATABASE nazwabazy;

CREATE USER student1234 IDENTIFIED BY 'haslo';
GRANT ALL PRIVILEGES ON nazwabazy.* TO 'student1234'@'%';

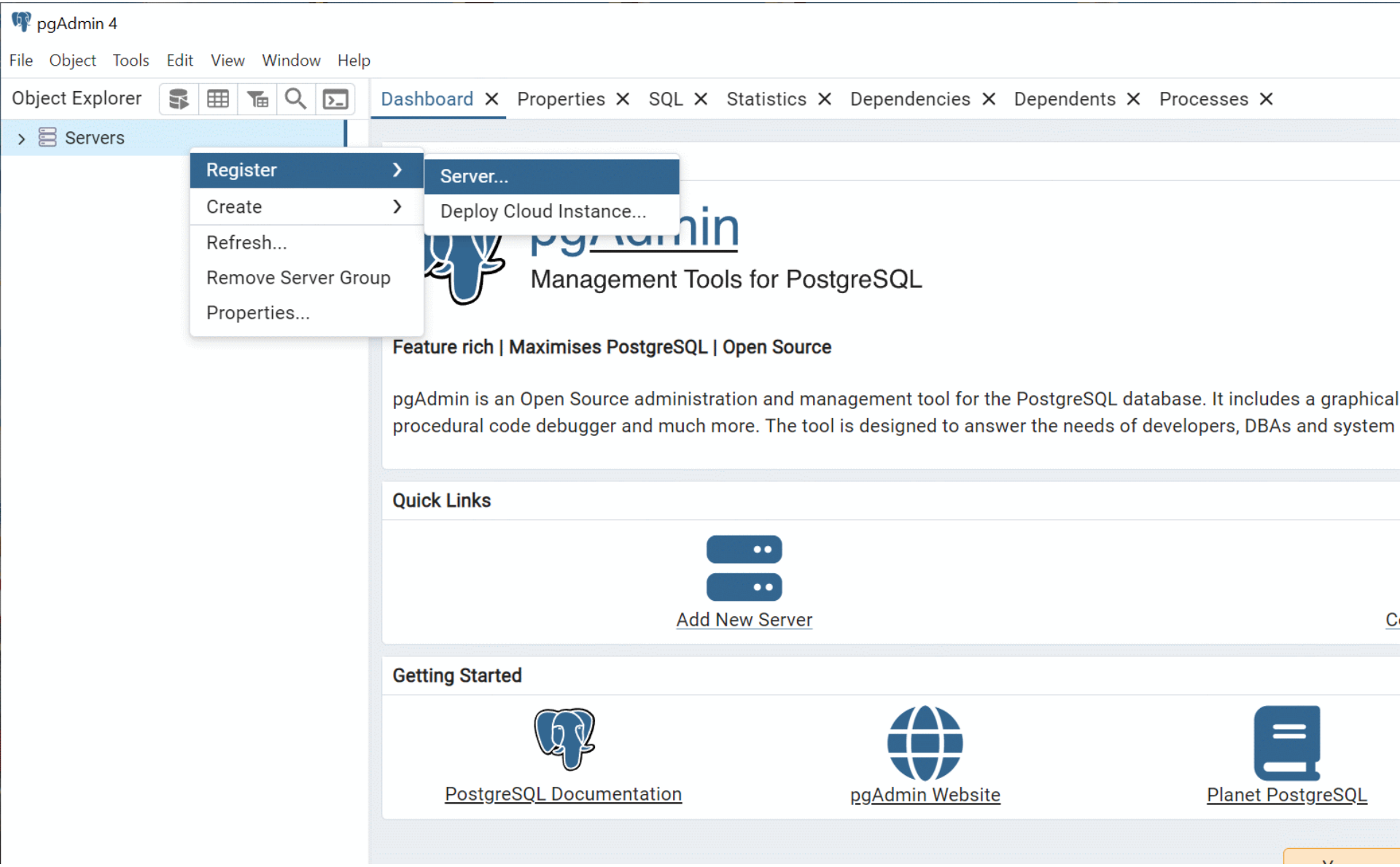
mysql -h 149.156.XXX.YYY -u student1234 -p
```

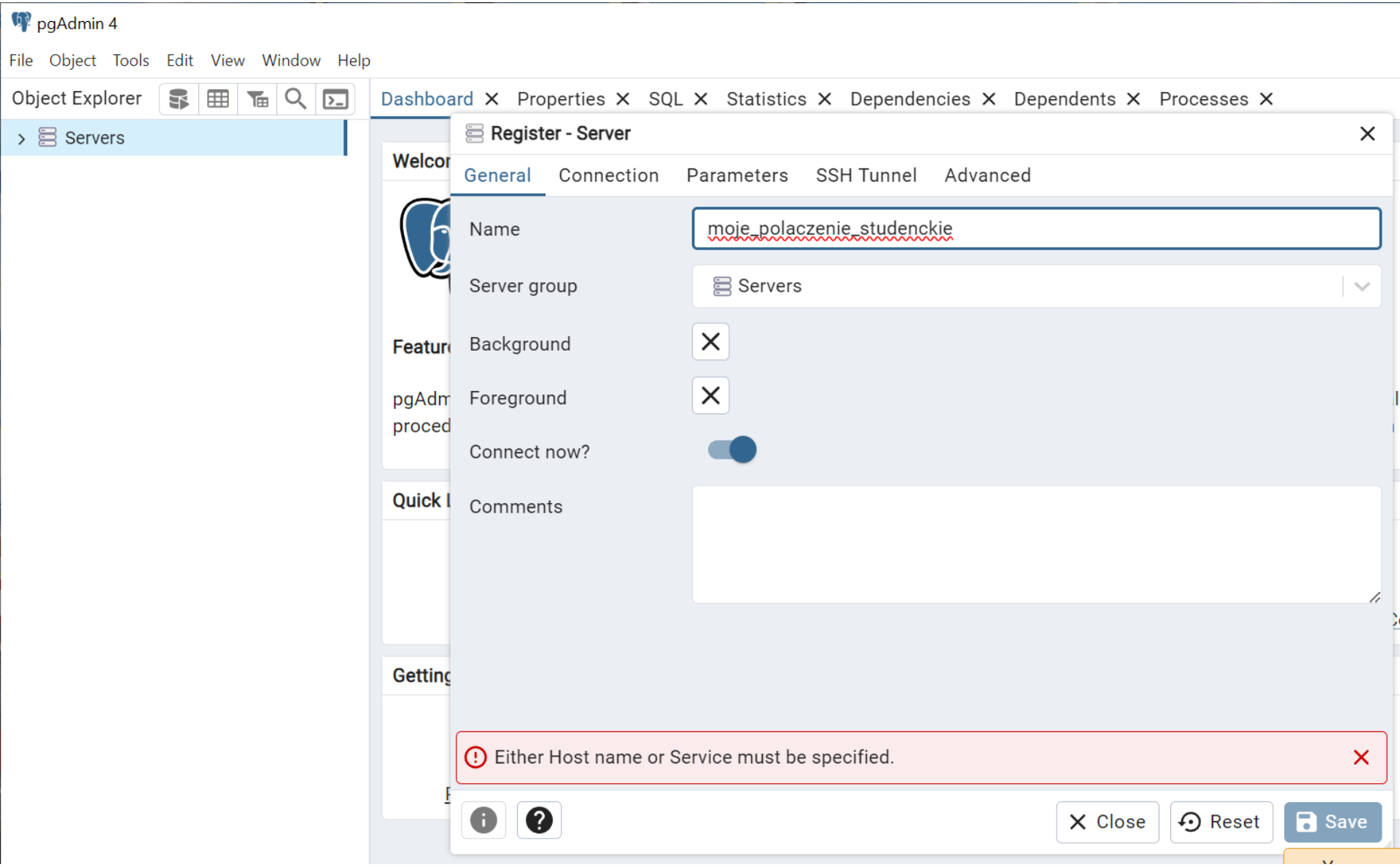
SQL – Practice … psql

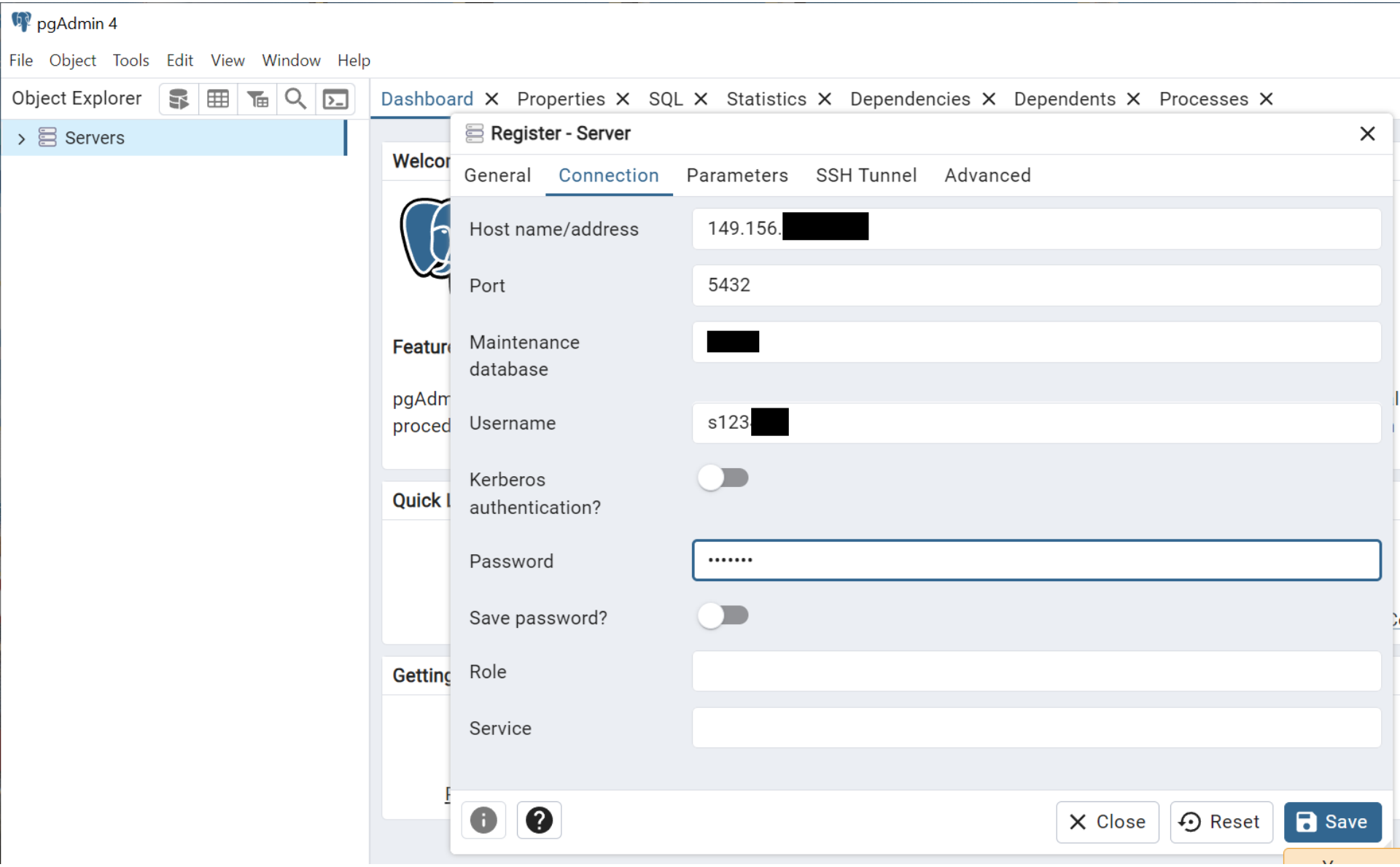
```
nazwabazy=>
nazwabazy=> SELECT * FROM korelacje
nazwabazy-> WHERE ( wymiar > 2 AND
nazwabazy(>
nazwabazy(> normalizacja IS NULL )
nazwabazy->
nazwabazy->
nazwabazy-> ;
 wymiar | normalizacja | korelacja
--------+---------------+-----------
(0 rows)

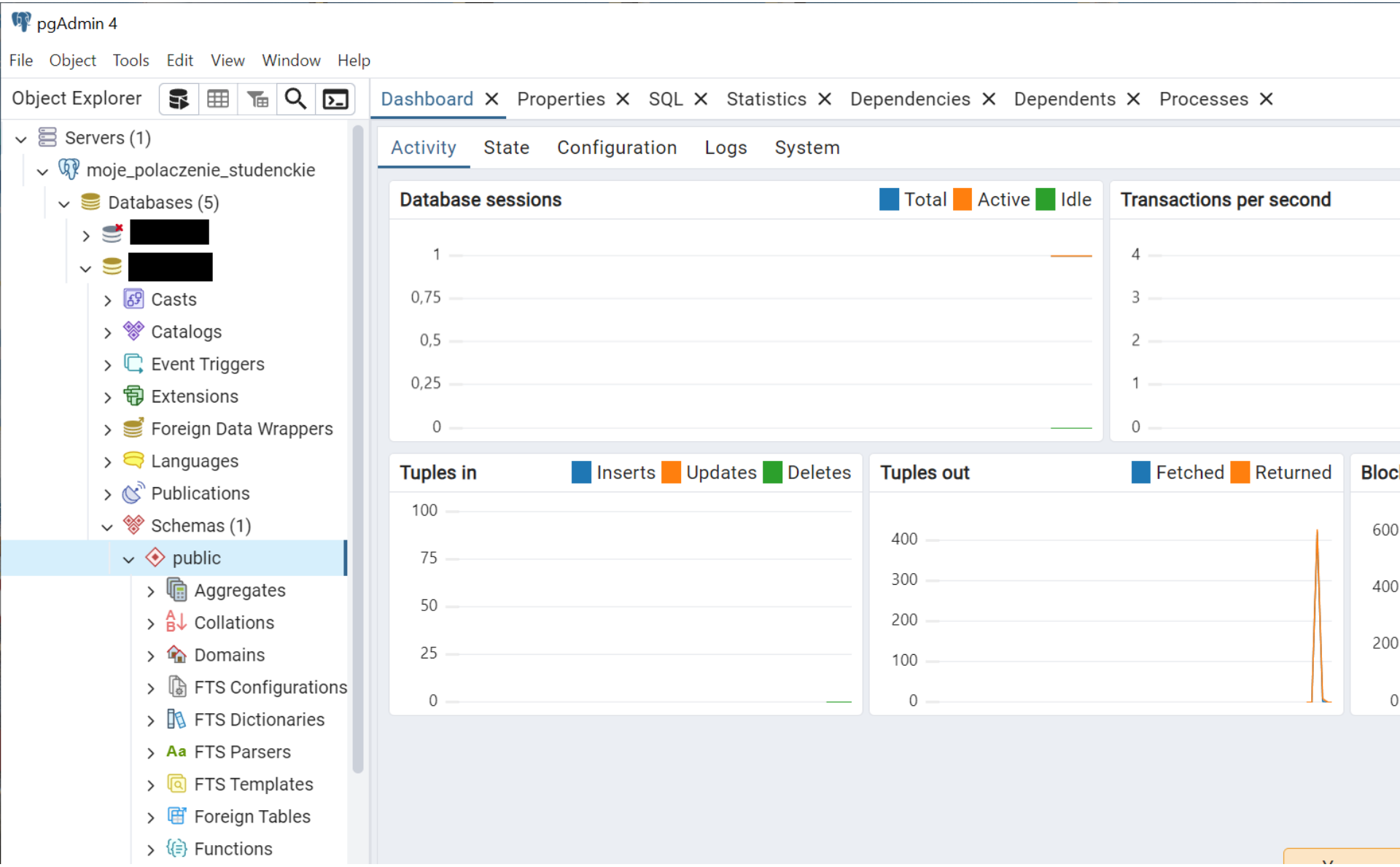
nazwabazy=>
```

WARNING!!! Apostrophes ' ,’ and quotation marks " „” are different characters!!!


pgAdmin 4
File Object Tools Edit View Window Help
Object Explorer
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Servers
Register
Create
Refresh...
Remove Server Group
Properties...
Server...
Deploy Cloud Instance...
pgAdmin
Management Tools for PostgreSQL
Feature rich | Maximises PostgreSQL | Open Source
pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical
procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system
Quick Links
Add New Server
Getting Started
PostgreSQL Documentation
pgAdmin Website
Planet PostgreSQL


pgAdmin 4
File Object Tools Edit View Window Help
Object Explorer
Servers
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Register - Server
General
Connection
Parameters
SSH Tunnel
Advanced
Name
moje_polaczenie_studenckie
Server group
Servers
Background
Foreground
Connect now?
Comments
Either Host name or Service must be specified.
Close
Reset
Save


pgAdmin 4
File Object Tools Edit View Window Help
Object Explorer
Servers
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Register - Server
General
Connection
Parameters
SSH Tunnel
Advanced
Host name/address
149.156.
Port
5432
Maintenance database
Username
s123
Kerberos authentication?
Password
Save password?
Role
Service
Close
Reset
Save


pgAdmin 4
File Object Tools Edit View Window Help
Object Explorer
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Servers (1)
moje_polaczenie_studenckie
Databases (5)
[redacted]
[redacted]
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Activity
State
Configuration
Logs
System
Database sessions
Total
Active
Idle
1
0,75
0,5
0,25
0
Transactions per second
4
3
2
1
0
Tuples in
Inserts
Updates
Deletes
100
75
50
25
0
Tuples out
Fetched
Returned
400
300
200
100
0
600
400
200
0

SQL DATA TYPES

- **Character strings**
- **CHARACTER (length) – CHAR – a character string of length *length*; shorter strings are padded with spaces.**
- **VARCHAR (length) – CHARACTER VARYING / CHAR VARYING – a character string with a maximum length of *length*; shorter strings are not padded with spaces.**
- **NVARCHAR / NATIONAL CHARACTER / NATIONAL CHARACTER VARYING – character strings using the UNICODE standard.**
- **Bit strings**

SQL DATA TYPES

- **Exact numeric types**

- **NUMERIC – decimal number**
- **DECIMAL – DEC – decimal number**
- **INTEGER – INT – integer; range determined by the DBMS**
- **SMALLINT – integer with a smaller range.**

- **Floating-point numbers**

- **FLOAT (precision) – floating-point number with user-defined precision**
- **REAL (length) – floating-point number with system-defined precision (4 B)**
- **DOUBLE PRECISION (length) – floating-point number with the highest precision.**

SQL DATA TYPES

- **Date and time**

- **DATE – a date defined by three integers – year (YEAR), month (MONTH), and day (DAY), YYYY-MM-DD.**
- **TIME (precision) – a time defined by two integer components – hour (HOUR), minutes (MINUTE), and a decimal number of seconds (SECOND), HH:MM:SS.SSSS (precision)**
- **TIMESTAMP – a combination of DATE and TIME (DATETIME – only in MS SQL)**

- **NULL**

- **NULL – indicates the absence of a value; it is not comparable. Multiple NULL values may be treated as duplicates and grouped; their presence in expressions affects the result, which becomes NULL.**

Data Types

TYP	[B]	ZAKRES	OPIS
INTEGER, INT, BIGINT, SMALLINT, TINYINT	4 8,2,1	-2^{31} do 2^{31} -1 -2,147,483,648 - 2,147,483,647	Liczby całkowite
DECIMAL (precyzja, skala) NUMERIC*	~ precyzji	max liczba cyfr ~SZBD	Liczby dziesiętne, np. waluta: 100.15
FLOAT, REAL	Precyzja 1-24 / 25-53 4	-1.79E+308 :-2.23E-308, 2.23E-308 to 1.79E+308 - 3.40E + 38 to -1.18E – 38 1.18E - 38 to 3.40E + 38.	Liczby zmiennoprzecinkowe

In SQL, REAL corresponds to the float type in the Intel processor sense (4 bytes), whereas FLOAT corresponds to double (precisi

NUMERIC ma stałą precyzję, DECIMAL – może mieć większą, niż potrzebuje projektant (związane z wydajnościa szbd, w części szbd te typy są równoważne) - zależy także od S.Z.B.D.

SQL Standard

Data Types

TYP	[B]	ZAKRES	OPIS
CHAR CHAR(20)	1 20	Jeden znak ASCII Tablica 20 znaków ASCII	Jeden znak Dokładnie 20 znaków, dopełniane spacjami
VARCHAR(_X_)	1B - 2GB	Tablica znaków ASCII	Dowolna liczba znaków, ograniczona przez _X_
NVARCHAR(_X_)	2B - 2GB	Tablica znaków UNICODE	Można składować tekst w znakach narodowych
DATE, TIME, DATETIME	10B,..	Np. tekstowo YYYY-MM-DD	czas
BLOB CLOB	Np. do 2GB	Blok binarny Blok tekstowy	Np. na zdjęcia Np. na strony

Floating-Point Numbers

```
CREATE TABLE liczby(i INT, b BIGINT, f FLOAT, r REAL);

INSERT INTO liczby VALUES
     ( 1234567891, 12345678912345678, 123456789123456789, 123456789123456789 );

SELECT * FROM liczby;
     i      |          b          |          f           |      r
------------+---------------------+----------------------+-------------
 1234567891 | 12345678912345678 | 1.23456789123457e+017 | 1.23457e+017

DELETE FROM liczby;

INSERT INTO liczby VALUES( 0, 0,  100000,  100000 );
INSERT INTO liczby VALUES( 0, 0, 1000000, 1000000 );
SELECT * FROM liczby;
 i | b |    f    |   r
---+---+---------+--------
 0 | 0 |  100000 | 100000
 0 | 0 | 1000000 | 1e+006

UPDATE liczby SET f = f + 0.007;
UPDATE liczby SET r = r + 0.007;
SELECT * FROM liczby;
 i | b |      f      |   r
---+---+-------------+--------
 0 | 0 |  100000.007 | 100000
 0 | 0 | 1000000.007 | 1e+006
```

Type Casting

Konwersje niejawne – dozwolone przez system bazy danych.

W przypadku wyrażeń składających się z różnych typów danych, jeżeli możliwa jest konwersja niejawna, to zostanie ona przeprowadzona do najbardziej złożonego typu z użytych typów danych.

Konwersje jawne – konwersje wymuszone przez użytkownika przy pomocy funkcji CAST(). Można dokonywać zmiany liczb (np. dziesiętnych na zmiennoprzecinkowe, zmiennoprzecinkowych na całkowite z utratą części ułamkowej). Podczas operacji zmiany typu danych może zaistnieć błąd – np. nieprawidłowa zawartość tekstu czy przekroczenie zakresu.

Składnia: `CAST ( dana_źródłowa AS typ_wyjściowy )`

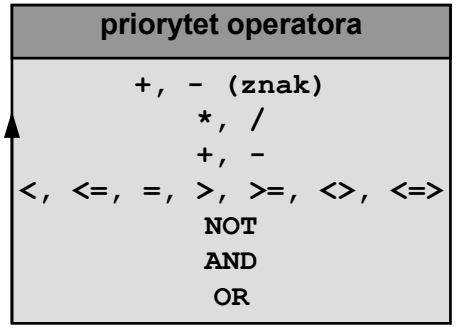
Przykład: [kolumna: `speed INT`]

```
SELECT AVG(speed) FROM TrasyKaretek;

SELECT AVG(CAST (speed AS DECIMAL(5,2))) FROM ..
```

Operators

- **Arithmetic operators**
- **SQL provides basic arithmetic operators {+, -, +, -, *, /}.**



- **Logical operators**
- **SQL provides the logical operators NOT, AND, OR.**

- **Operations on character strings**
- **In SQL, text operations can be performed using:**
- **- the string concatenation operator || (in MySQL, this corresponds to OR; in MS Access, the + operator)**
- **- a set of text-processing functions: { CONCAT(), SUBSTRING(), UPPER(), LOWER(), TRIM(), CHARACTER_LENGTH(), POSITION(), … }**

- **Operator precedence**
- **The order of operations is determined by operator precedence. If two operators have the same precedence, the order is determined by associativity (from left to right). Parentheses () can be used to change the order of operations.**

SQL – Schema Definition

Polecenie CREATE TABLE służy do tworzenia tabel. Jego składnia jest następująca:

```
CREATE TABLE nazwatabeli ( kol1 typdanych1 [atrybuty1],
   kol2 typdanych1 [atrybuty2] ...)
```

Wśród atrybutów mogą się znaleźć:

- `PRIMARY KEY` – atrybut wchodzi w skład klucza podstawowego,
- `FOREIGN KEY ... REFERENCES` – atrybut wchodzi w skład klucza obcego,
- `NOT NULL` – krotki nie mogą w danej kolumnie zawierać wartości NULL
- `UNIQUE` – w danej kolumnie dozwolone są tylko unikalne wartości,
- `CHECK` – pozwala na wprowadzanie danych spełniających zadane warunki,
- `DEFAULT` – określa wartość domyślną dla danej kolumny,
- `CONSTRAINT` – określa nazwę atrybutu – pomocne w tworzeniu kluczy złożonych, unikalności w obrębie kilku kolumn.

Przykłady:

```
CREATE TABLE Prosta (Wymagany INT NOT NULL, Domyslny INT DEFAULT 7,
   Unikalny INT UNIQUE );
CREATE TABLE ZlozonaUnikalnosc (Rh CHAR(1), Oab CHAR(2), CONSTRAINT Grp
   UNIQUE (Rh, Oab));
CREATE TABLE ZWarunkiem (Ograniczony INT CHECK ( Ograniczony > 100 ) );
```

SQL – Schema Definition

Simple primary key

```
CREATE TABLE GlProsty (danap INT PRIMARY KEY);
```

Composite primary key

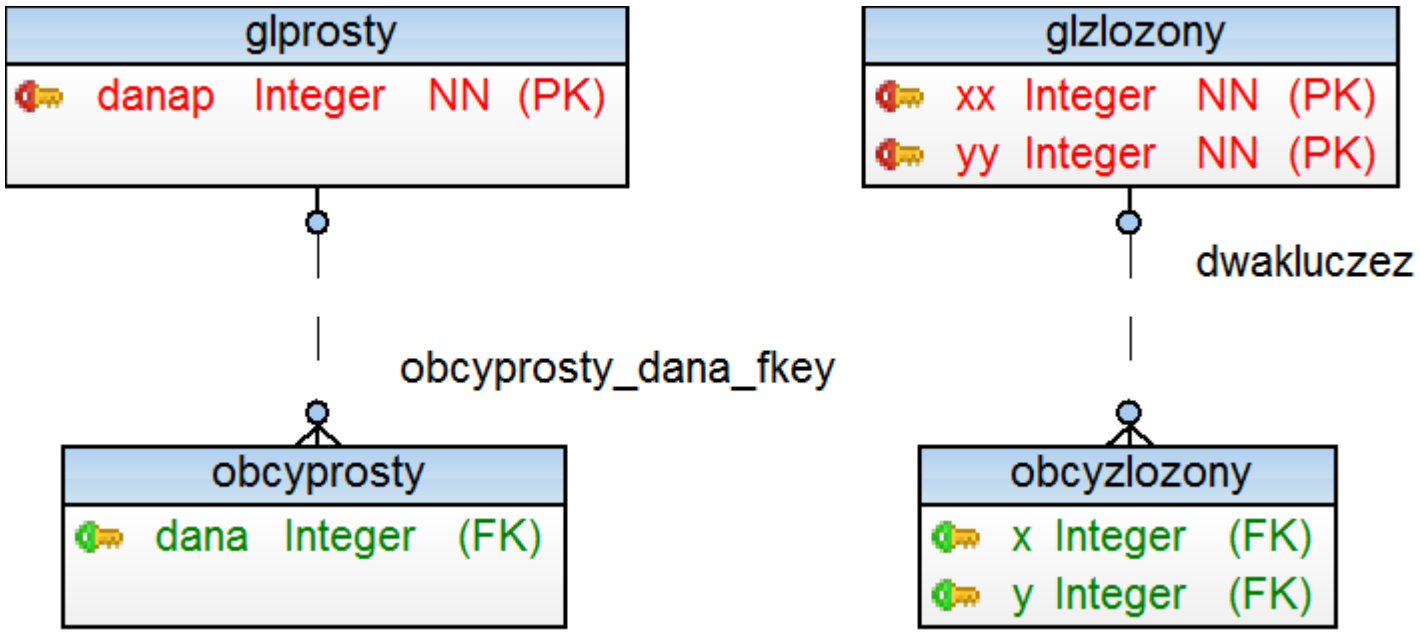
```
CREATE TABLE GlZlozony(xx INT, yy INT,
                       CONSTRAINT klucz_xxyy PRIMARY KEY (xx, yy));
```

Simple foreign key

```
CREATE TABLE ObcyProsty (dana INT REFERENCES GlProsty(danap) );
```

Composite foreign key

```
CREATE TABLE ObcyZlozony ( x INT, y INT, CONSTRAINT dwakluczez
                  FOREIGN KEY (x, y)  REFERENCES GlZlozony (xx,yy) );
```



SQL – Schema Definition

Simple primary key

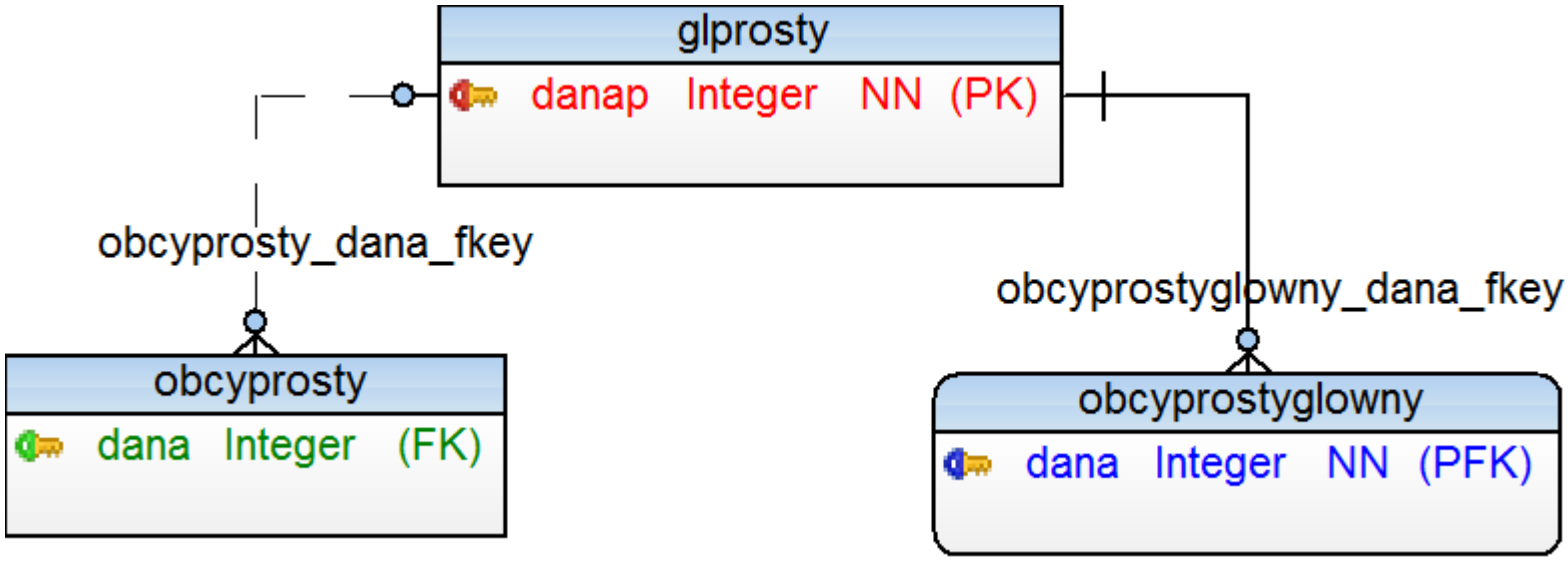
```
CREATE TABLE GlProsty (danap INT PRIMARY KEY);
```

Simple foreign key

```
CREATE TABLE ObcyProsty (dana INT REFERENCES GlProsty(danap) );
```

Simple foreign key used as a primary key (1:1)

```
CREATE TABLE ObcyProstyGlowny (dana INT PRIMARY KEY REFERENCES
                                                GlProsty(danap) );
```



SQL – Schema Definition

Composite primary key

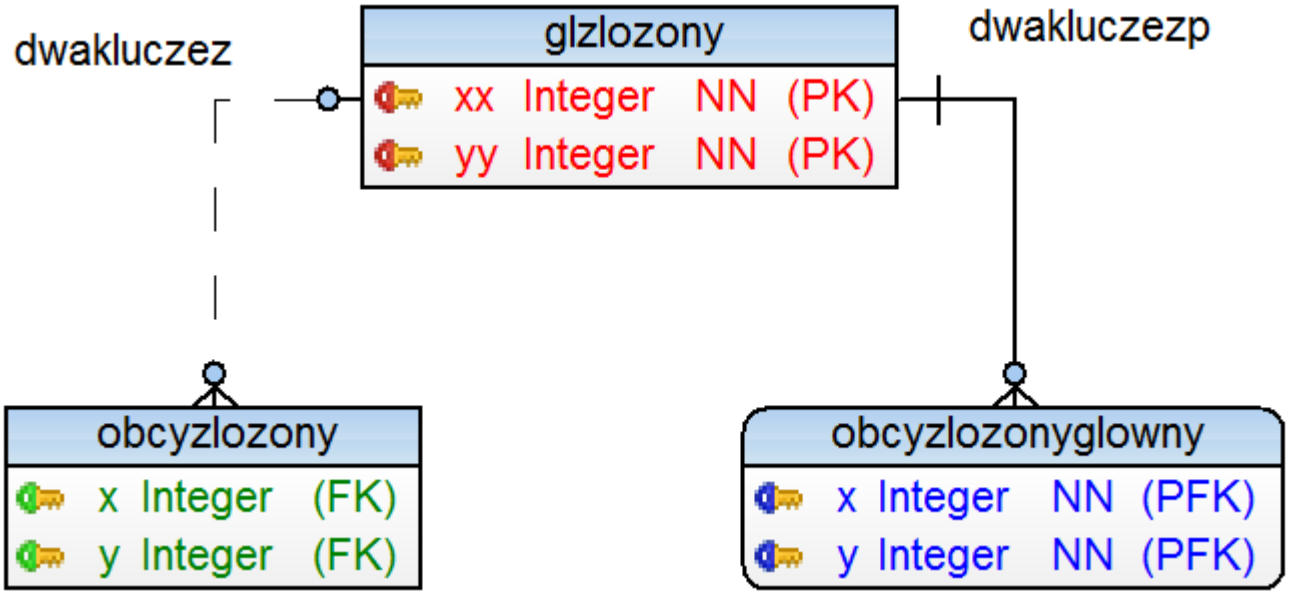
```
CREATE TABLE GlZlozony(xx INT, yy INT,
                       CONSTRAINT klucz_xxyy PRIMARY KEY (xx, yy));
```

Composite foreign key

```
CREATE TABLE ObcyZlozony ( x INT, y INT, CONSTRAINT dwakluczez
                  FOREIGN KEY (x, y)  REFERENCES GlZlozony (xx,yy) );
```

Composite foreign key used as a primary key

```
CREATE TABLE ObcyZlozonyGlowny ( x INT, y INT,
 CONSTRAINT dwakluczezp FOREIGN KEY (x, y) REFERENCES GlZlozony (xx,yy),
                       CONSTRAINT klucz_pk PRIMARY KEY (x, y) );
```



SQL – Schema Definition

Simple primary key

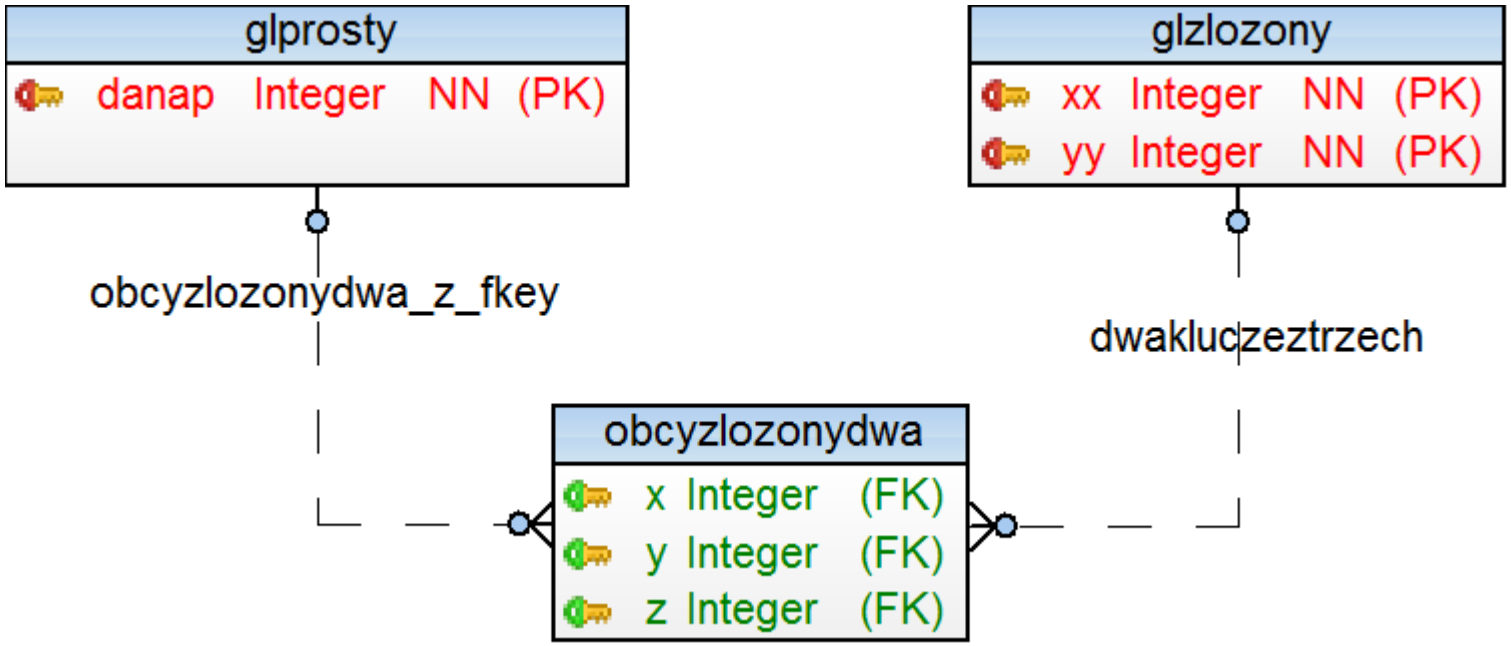
```
CREATE TABLE GlProsty (danap INT PRIMARY KEY);
```

Composite primary key

```
CREATE TABLE GlZlozony(xx INT, yy INT,
                       CONSTRAINT klucz_xxyy PRIMARY KEY (xx, yy));
```

Simple and composite foreign keys

```
CREATE TABLE ObcyZlozonyDwa ( x INT, y INT, z INT REFERENCES
        GlProsty(danap), CONSTRAINT dwakluczeztrzech FOREIGN KEY (x, y)
                                  REFERENCES GlZlozony (xx,yy) );
```



Schema Modification – ALTER/DROP

Polecenie **`ALTER TABLE`** modyfikuje schemat relacji. Składnia:

```
ALTER TABLE nazwatabeli akcja_modyfikująca
```

gdzie akcja modyfikująca to:

- `ADD [COLUMN] nazwakolumny typdanych`
- `DROP COLUMN nazwakolumny`
- `ADD atrybut`
- `DROP CONSTRAINT atrybut`

Przykład:

```
ALTER TABLE GlProsty ADD dodatkowakolumna INT;
ALTER TABLE tabela ADD PRIMARY KEY (id_elementu1, data);
```

`DROP TABLE` usuwa wybraną tabelę z bazy

```
DROP TABLE tabela;
```

`SELECT INTO / CREATE AS`

Przy pomocy konstrukcji SELECT INTO można utworzyć nową tabelę, zawierającą kolumny określone w instrukcji SELECT, wchodzące w skład jednej lub więcej tabel. Nowa tabela od razu jest zapełniana danymi.

Data Manipulation (DML)

INSERT

Instrukcja INSERT powoduje wpisanie wiersza danych do bazy. Jej składnia jest następująca:

```
INSERT INTO tabela (kol1, kol2, ...) VALUES (w1, w2, ...);
```

UPDATE

Polecenie UPDATE modyfikuje zawartość wybranych atrybutów krotek, które spełniają zadany warunek. Składnia polecenia UPDATE:

```
UPDATE tabela SET atrybut = wartosc  [WHERE warunek];
```

Przykład:

```
UPDATE Wspolrzedne SET x = x + 2, y = 3 WHERE x > 0 AND y > 0;
```

DELETE

Instrukcja DELETE służy do usuwania całych wierszy z bazy w oparciu o warunek (lub wszystkich, jeśli nie jest podany). Składnia jej jest następująca:

```
DELETE FROM tabela [WHERE warunek];
```

TRUNCATE - usuwa całą zawartość tabeli nie sprawdzając warunków i nie angażując zasobów, np.:

```
TRUNCATE TABLE tabela;
```

INSERT INTO …

```
CREATE TABLE dane (id_danych INT PRIMARY KEY, liczba DECIMAL(5,2), tekst VARCHAR(100));

-- wpisanie danych wg zadanej listy atrybutow
INSERT INTO dane (id_danych, liczba, tekst) VALUES (1, 1.0, 'pierwszy');
-- wpisanie danych zgodnie z lista atrybutow
INSERT INTO dane VALUES (2, 2.0, 'drugi');
-- wpisanie danych wg zadanej listy atrybutow (niepelnej)
INSERT INTO dane (id_danych, tekst) VALUES (3, 'nieokreslony');
-- wpisanie grupy wartosci
INSERT INTO dane VALUES (4, 3.3, 'trzeci'), (5, 4.4, 'czwarty'), (6, 5.55, 'piaty');
-- odrzuci wpis ze względu na naruszenie unikalności klucza
INSERT INTO dane VALUES (6, 6.66, 'szosty powt klucza');
-- zaokragli dane
INSERT INTO dane VALUES (7, 7.777, 'siodmy');
-- odrzuci obie wartości, mimo ze tylko jedna narusza klucz
INSERT INTO dane VALUES (6, 6.66, 'szosty powt klucza'),  (8, 8.88, 'osmy');
```

```
bd=> SELECT * FROM dane;
 id_danych | liczba |    tekst
-----------+--------+--------------
         1 |   1.00 | pierwszy
         2 |   2.00 | drugi
         3 |        | nieokreslony
         4 |   3.30 | trzeci
         5 |   4.40 | czwarty
         6 |   5.55 | piaty
         7 |   7.78 | siodmy
(7 rows)
```

UPDATE ....

```
bd=> SELECT * FROM dane;
 id_danych | liczba |    tekst
-----------+--------+--------------
         1 |   1.00 | pierwszy
         2 |   2.00 | drugi
         3 |        | nieokreslony
         4 |   3.30 | trzeci
         5 |   4.40 | czwarty
         6 |   5.55 | piaty
         7 |   7.78 | siodmy
(7 rows)
```

```
-- zmiana wszystkich krotek  (7)
bd=> UPDATE dane SET liczba = liczba*10;
```

UPDATE 7

```
bd=> SELECT * FROM dane;
 id_danych | liczba |    tekst
-----------+--------+--------------
         1 |  10.00 | pierwszy
         2 |  20.00 | drugi
         3 |        | nieokreslony
         4 |  33.00 | trzeci
         5 |  44.00 | czwarty
         6 |  55.50 | piaty
         7 |  77.80 | siodmy
(7 rows)
```

```
-- zmiana wybranego podzbioru krotek (2)
bd=> UPDATE dane SET liczba = liczba + 3 WHERE liczba < 30;
```

UPDATE 2

```
bd=> SELECT * FROM dane;
 id_danych | liczba |    tekst
-----------+--------+--------------
         3 |        | nieokreslony
         4 |  33.00 | trzeci
         5 |  44.00 | czwarty
         6 |  55.50 | piaty
         7 |  77.80 | siodmy
         1 |  13.00 | pierwszy
         2 |  23.00 | drugi
(7 rows)
```

```
-- zmiana dokładnie wskazanej krotki (PK)
bd=> UPDATE dane SET liczba = 100
                 WHERE id_danych = 3;
```

UPDATE 1

```
bd=> SELECT * FROM dane;
 id_danych | liczba |    tekst
-----------+--------+--------------
         4 |  33.00 | trzeci
         5 |  44.00 | czwarty
         6 |  55.50 | piaty
         7 |  77.80 | siodmy
         1 |  13.00 | pierwszy
         2 |  23.00 | drugi
         3 | 100.00 | nieokreslony
(7 rows)
```

SEQUENCE

```
CREATE TABLE tabela (id_wiersza INT, wartosc INT);

CREATE SEQUENCE tabela_s_id_wiersza START 1 INCREMENT 1;
CREATE SEQUENCE tabela_s_wartosc START 10 INCREMENT 10;

INSERT INTO tabela(id_wiersza) VALUES (nextval('tabela_s_id_wiersza'));
INSERT INTO tabela(id_wiersza) VALUES (nextval('tabela_s_id_wiersza'));
INSERT INTO tabela(id_wiersza) VALUES (nextval('tabela_s_id_wiersza'));

bd=> SELECT * FROM tabela;
 id_wiersza | wartosc
------------+---------
          1 |
          2 |
          3 |
(3 rows)

bd=> UPDATE tabela SET wartosc = nextval('tabela_s_wartosc');
UPDATE 3

bd=> SELECT * FROM tabela;
 id_wiersza | wartosc
------------+---------
          1 |      10
          2 |      20
          3 |      30
(3 rows)

bd=> SELECT currval('tabela_s_wartosc'), currval('tabela_s_wartosc'),
                                  nextval('tabela_s_wartosc'), nextval('tabela_s_wartosc');
 currval | currval | nextval | nextval
---------+---------+---------+---------
      30 |      30 |      40 |      50
(1 row)
```

Temporal Data

```
bd=> SELECT CURRENT_DATE;
 current_date
--------------
 2025-03-24
(1 row)


bd=> CREATE TABLE dane_czasowe(data DATE, czas TIME, lacznie TIMESTAMP);
CREATE TABLE

bd=> INSERT INTO dane_czasowe VALUES('2000-01-07', '11:30', '2000-01-07 11:30');
INSERT 0 1

bd=> INSERT INTO dane_czasowe VALUES(CURRENT_DATE, CURRENT_TIME,
                                                          CURRENT_TIMESTAMP);
INSERT 0 1

bd=> SELECT * FROM dane_czasowe;
    data    |      czas       |          lacznie
------------+-----------------+---------------------------
 2000-01-07 | 11:30:00        | 2000-01-07 11:30:00
 2025-03-24 | 11:46:59.95325 | 2025-03-24 11:46:59.95325
(2 rows)
```

SELECT Query

Zapytanie SELECT służy do wybierania danych z jednej lub wielu tabel, spełniających odpowiednie warunki i sformatowanych w odpowiedni sposób.

Składnia polecenia jest następująca:

```
SELECT   wybrane_kolumny
  FROM tabela1 [,tabela2, ...]
  [JOIN [<TYP_ZLACZENIA>]]
  [WHERE warunek_wybierania]
  [GROUP BY kolumna_grupowania1 [,kolumna_grupowania2, ...] ]
  [HAVING warunek_filtrowania_grupy]
  [ORDER BY kol_sortowania1 [ASC DESC] [, kolumna_sortowania2,...] ]
;
```

SELECT Query

Wyświetlanie pełnej zawartości tabeli:

```
SELECT * FROM tabela;
```

Wyświetlanie wybranych kolumn tabeli (projekcja):

```
SELECT kolumna1, kolumna2 FROM tabela;
```

Wyświetlanie krotek bez duplikatów:

```
SELECT DISTINCT * FROM tabela;
```

Sortowanie wyświetlanych wyników:
do sortowania wyników zapytania służy opcja ORDER BY. Wartości NULL są traktowane specyficznie dla danej implementacji bazy danych.

```
SELECT * FROM tabela ORDER BY kolumna1 DESC, kolumna2 ASC;
```

SELECT Query

Warunek wybierania (opcjonalny) określa się przy pomocy słowa kluczowego WHERE. W warunku tym można stosować operatory arytmetyczne, logiczne, tekstowe oraz specjalne konstrukcje, ułatwiające wyszukiwanie.

```
SELECT * FROM tabela WHERE  kolumna1 > 100;
SELECT * FROM tabela WHERE  kolumna1 > 100 OR kolumna2 < 300;
SELECT * FROM tabela WHERE (kolumna1 > 100 OR kolumna2 < 300) AND kolumna3 <> 0;
```

Sprawdzanie obecności wartości atrybutu: **IS NULL / IS NOT NULL**

```
SELECT * FROM tabela WHERE data_obrony IS NULL  AND data_odejscia IS NOT NULL;
```

Sprawdzanie obecności atrybutu w liście wartości:

```
SELECT * FROM tabela WHERE  liczba IN (1,2,3,5,7,11,13,17,19);
```

Sprawdzanie zakresu wartości atrybutu (przedział domknięty):

```
SELECT * FROM tabela WHERE  liczba BETWEEN 100 AND 200;
```

SELECT Query

Tworzenie aliasów kolumn:

```
CREATE TABLE wplaty (id_wplaty INT, kwota DECIMAL(8,2));
INSERT INTO wplaty VALUES(1, 10.0);
INSERT INTO wplaty VALUES(2, 20.0);

SELECT id_wplaty AS numer,
      kwota AS wartosc,
      kwota*0.23 AS "wysokosc podatku",
      0.23 AS podatek
      FROM wplaty WHERE kwota > 15;

 numer | wartosc | wysokosc podatku | podatek
-------+---------+------------------+---------
     2 |   20.00 |           4.6000 |    0.23
(1 row)
```

SELECT Query

Dostęp kropkowy

```
CREATE TABLE osoby(pesel CHAR(11), imie VARCHAR(20), nazwisko VARCHAR(20));
CREATE TABLE dowody(pesel CHAR(11), nr_dowodu CHAR(9));

INSERT INTO osoby VALUES('11111111111', 'Aaa', 'Aaaa');
INSERT INTO osoby VALUES('22222222222', 'Baa', 'Baaa');
INSERT INTO dowody VALUES('11111111111', 'AAA123456');
```

```
=> SELECT * FROM osoby, dowody WHERE pesel = '11111111111';
BLAD:  odnosnik kolumny "pesel" jest niejednoznaczny at character 35
LINE 1: SELECT * FROM osoby, dowody WHERE pesel = '11111111111';
                                          ^
```

```
=> SELECT * FROM osoby, dowody WHERE dowody.pesel = '11111111111';
```

```
    pesel    | imie | nazwisko |    pesel    | nr_dowodu
-------------+------+----------+-------------+-----------
 11111111111 | Aaa  | Aaaa     | 11111111111 | AAA123456
 22222222222 | Baa  | Baaa     | 11111111111 | AAA123456
(2 rows)
```

```
=> SELECT * FROM osoby, dowody WHERE dowody.pesel = osoby.pesel;
    pesel    | imie | nazwisko |    pesel    | nr_dowodu
-------------+------+----------+-------------+-----------
 11111111111 | Aaa  | Aaaa     | 11111111111 | AAA123456
(1 row)
```

SELECT Query

Operacje na tekstach:

```
SELECT * WHERE  marka BETWEEN 'Dacia' AND 'Ford';
```

Operator dopasowywania wzorca:

LIKE:

% - dowolny podciąg znaków, także pusty,

_ - dowolny pojedynczy znak.

```
SELECT * FROM ksiazki WHERE tytul LIKE '%SQL%';
SELECT * FROM spistresci WHERE rozdzial LIKE '_ Postac normalna';
SELECT * FROM linie WHERE nrautobusu LIKE '6__';
```

SELECT Query

```
-- LIMIT
CREATE TABLE wyniki(nr_wartosci INT, wartosc DECIMAL(5,2));
INSERT INTO wyniki VALUES (1, 0.01), (2, 0.02), (3, 0.3), (4, 0.4);

=> SELECT * FROM wyniki LIMIT 2;
 nr_wartosci | wartosc
-------------+---------
           1 |    0.01
           2 |    0.02
(2 rows)

=> SELECT * FROM wyniki ORDER BY wartosc DESC LIMIT 2;
 nr_wartosci | wartosc
-------------+---------
           4 |    0.40
           3 |    0.30
(2 rows)
=> SELECT * FROM wyniki ORDER BY wartosc DESC LIMIT 1;
 nr_wartosci | wartosc
-------------+---------
           4 |    0.40
(1 row)

=> SELECT wartosc FROM wyniki ORDER BY wartosc DESC LIMIT 1;
 wartosc
---------
    0.40
(1 row)
```

AGGREGATE OPERATIONS

Standard SQL określa następujące funkcje agregujące:

`MIN(wyrazenie)` – wyznacza minimum danego atrybutu z krotek wg określonego wyrażenia,
`MAX(wyrazenie)` – wyznacza maksimum danego atrybutu z krotek wg określonego wyrażenia,
`SUM(wyrazenie)` – wyznacza sumę danego atrybutu z krotek wg określonego wyrażenia,
`AVG(wyrazenie)` – wyznacza średnią wartość danego atrybutu z krotek wg określonego wyrażenia,
`COUNT(wyrazenie)` – wyznacza liczbę niepustych dla danych atrybutów krotek,

`wyrazenie` – jest wyrażeniem (np. arytmetycznym!) opartym o nazwę kolumny (kolumn).

`COUNT (*)` - dotyczy także wartości NULL (bezwzględnie wszystkich krotek w tabeli)

Zapytania z funkcjami agregującymi zwracają wartości liczbowe (skalar, wektor). Składnia takich zapytań wymaga, aby owe funkcje występowały jako jedyne cele zapytania SELECT (nie mogą być wyświetlane wraz z atrybutami relacji, chyba, ze z grupowanymi).

```
bd=# SELECT AVG(liczba) FROM roznelicz by;
         avg
--------------------
 22230.000000000000
(1 row)
```

AGGREGATE OPERATIONS

```
bd=#  SELECT AVG(liczba), MIN(liczba) FROM rozneliczby;
        avg          | min
---------------------+-----
 22230.000000000000 |  50
(1 row)
```

```
bd=#  SELECT *, AVG(liczba) FROM rozneliczby;
BLAD:  kolumna "rozneliczby.liczba" musi wystepowac w klauzuli GROUP BY lub
 byc uzyta w funkcji agregujacej at character 8
LINE 1: select *, AVG(liczba) from rozneliczby;
                ^
```

```
bd=#  SELECT COUNT(*) from rozneliczby;
 count
-------
     5
(1 row)
```

AGGREGATE OPERATIONS

```
test(a INT, b INT)

=> SELECT * FROM test;
 a | b
---+---
 1 | 2
 2 | 4
 6 | 8
 2 |
(4 rows)
```

```
=> SELECT MIN (a) FROM test;
 min
-----
   1
(1 row)
```

```
=> SELECT MIN (2*a) FROM test;
 min
-----
   2
(1 row)
```

```
=> SELECT MIN (2*a), AVG(b)
FROM test;
 min |         avg
-----+--------------------
   2 | 4.666666666666667
(1 row)
```

```
=> SELECT COUNT (*) FROM test;
 count
-------
     4
(1 row)
```

```
=> SELECT COUNT (a) FROM test;
 count
-------
     4
(1 row)
```

```
=> SELECT COUNT (b) FROM test;
 count
-------
     3
(1 row)
```

```
=> SELECT COUNT (a*b) FROM test;
 count
-------
     3
(1 row)
```

DATA GROUPING

Opcja **`GROUP BY`** `atrybut` wydziela logiczną grupę krotek o tej samej wartości kolumny `atrybut` i pozwala przeprowadzić na tej grupie operacje agregujące dla innych atrybutów.

- atrybut może być kolumną lub listą kolumn,

- dla każdej wartości kolumny atrybut zwracany jest tylko jeden wiersz,

- jeśli w kolumnie atrybut znajduje się NULL, to w wyniku pojawia się wiersz dla wartości NULL,

- wyświetlane w zapytaniu mogą być tylko te atrybuty, które podlegają grupowaniu albo agregacji (inaczej nie wiadomo, jaka wartość miałaby reprezentować atrybut),

Dodatkowa selekcja grupowania odbywa się poprzez opcję `HAVING`. Polecenie to pozwala określić dodatkowy warunek, który musi spełnić krotka grupy, aby pojawiła się w wyniku zapytania.

Kolejność filtrowania: `FROM / JOIN -> GROUP BY -> HAVING`

DATA GROUPING

```
CREATE TABLE
        pracownicy(id_pr INT, plec CHAR, miasto VARCHAR(20), pensja DECIMAL(10,2));
```

```
INSERT INTO pracownicy VALUES(1, 'M', 'Krakow', 2000.00);
INSERT INTO pracownicy VALUES(2, 'K', 'Krakow', 3000.00);
INSERT INTO pracownicy VALUES(3, 'K', 'Krakow', 2000.00);
INSERT INTO pracownicy VALUES(4, 'K', 'Warszawa', 2800.00);
INSERT INTO pracownicy VALUES(5, 'M', 'Warszawa', 3000.00);
INSERT INTO pracownicy VALUES(6, 'm', 'Warszawa', 3100.00);
```

```
SELECT * FROM pracownicy;
 id_pr | plec |  miasto   | pensja
-------+------+-----------+---------
     1 | M    | Krakow    | 2000.00
     2 | K    | Krakow    | 3000.00
     3 | K    | Krakow    | 2000.00
     4 | K    | Warszawa  | 2800.00
     5 | M    | Warszawa  | 3000.00
     6 | m    | Warszawa  | 3100.00
(6 rows)
```

```
SELECT *, AVG(pensja) FROM pracownicy GROUP BY miasto;
BLAD:  kolumna "pracownicy.id_pr" musi wystepowac w klauzuli GROUP BY lub byc uzyta
w funkcji agregujacej at character 8
LINE 1: SELECT *, AVG(pensja) FROM pracownicy GROUP BY miasto;
                ^
```

DATA GROUPING

```
SELECT miasto, AVG(pensja) FROM pracownicy GROUP BY miasto;
  miasto  |          avg
----------+-----------------------
 Krakow   | 2333.3333333333333
 Warszawa | 2966.6666666666666667
(2 rows)

SELECT plec, AVG(pensja), COUNT(*) FROM pracownicy GROUP BY plec;
 plec |          avg           | count
------+------------------------+-------
 m    | 3100.0000000000000000 |     1
 K    | 2600.0000000000000000 |     3
 M    | 2500.0000000000000000 |     2
(3 rows)

SELECT UPPER(plec), AVG(pensja), COUNT(pensja) FROM pracownicy GROUP BY UPPER(plec);
 upper |          avg           | count
-------+------------------------+-------
 K     | 2600.0000000000000000 |     3
 M     | 2700.0000000000000000 |     3
(2 rows)

SELECT plec,AVG(pensja),COUNT(*) FROM pracownicy GROUP BY plec HAVING COUNT(plec)>1;
 plec |          avg           | count
------+------------------------+-------
 K    | 2600.0000000000000000 |     3
 M    | 2500.0000000000000000 |     2
(2 rows)
```

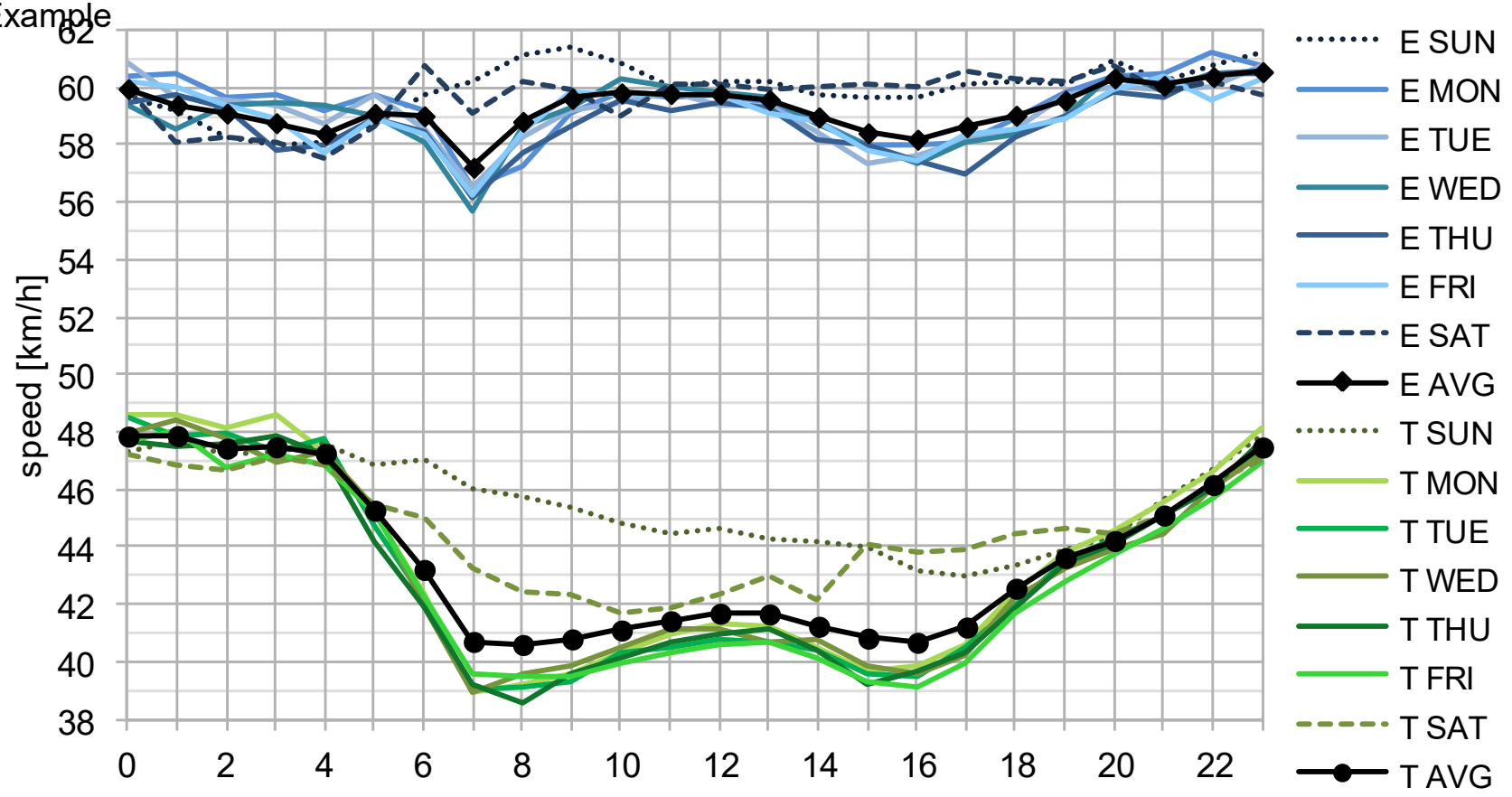
DATA GROUPING

```
SELECT UPPER(plec), miasto, AVG(pensja), COUNT(pensja) FROM pracownicy GROUP BY
UPPER(plec), miasto;
 upper |  miasto  |          avg          | count
-------+----------+-----------------------+-------
 M     | Krakow   | 2000.0000000000000000 |     1
 M     | Warszawa | 3050.0000000000000000 |     2
 K     | Krakow   | 2500.0000000000000000 |     2
 K     | Warszawa | 2800.0000000000000000 |     1
(4 rows)

SELECT UPPER(plec), miasto, AVG(pensja), COUNT(pensja) FROM pracownicy GROUP BY
miasto, UPPER(plec);
 upper |  miasto  |          avg          | count
-------+----------+-----------------------+-------
 M     | Krakow   | 2000.0000000000000000 |     1
 M     | Warszawa | 3050.0000000000000000 |     2
 K     | Warszawa | 2800.0000000000000000 |     1
 K     | Krakow   | 2500.0000000000000000 |     2
(4 rows)

SELECT UPPER(plec), miasto, AVG(pensja), COUNT(pensja) FROM pracownicy GROUP BY
miasto, UPPER(plec) HAVING COUNT(plec) > 1;
 upper |  miasto  |          avg          | count
-------+----------+-----------------------+-------
 M     | Warszawa | 3050.0000000000000000 |     2
 K     | Krakow   | 2500.0000000000000000 |     2
(2 rows)
```

Grouping Example




```
SELECT
  kogut,
  DATEPART(WEEKDAY, time) as dzientyg,
  DATEPART(HOUR, time) as godzina,
  AVG(CAST (speed AS DECIMAL(5,2)))
 FROM karetkiGPS WHERE speed > 0
 GROUP BY kogut, DATEPART(WEEKDAY, time), DATEPART(HOUR, time)
 ORDER BY kogut, DATEPART(WEEKDAY, time), DATEPART(HOUR, time);
```

Piórkowski, A.: Construction of a dynamic arrival time coverage map for emergency medical services. *Open Geosciences*, 2018,*10*(1), 167-173.

Set Operations

SQL umożliwia działania proste na parach tabel. Ciągi atrybutów muszą odpowiadać sobie pod względem typów.

Suma:

```
SELECT  atr1,...,atrN FROM tabA   UNION [ALL]   SELECT  atr1,...,atrN FROM tabB;
```

Różnica:

```
SELECT  atr1,...,atrN FROM tabA   EXCEPT   SELECT  atr1,...,atrN FROM tabB;
```

Iloczyn:

```
SELECT  atr1,...,atrN FROM tabA   INTERSECT   SELECT  atr1,...,atrN FROM tabB;
```

Set Operations

```
CREATE TABLE dane1(l INT, t VARCHAR(20));
INSERT INTO dane1 VALUES(1, 'jeden'), (2, 'dwa'), (2, 'dwa'), (3, 'trzy');
CREATE TABLE dane2(l INT, t VARCHAR(20));
INSERT INTO dane2 VALUES(1, 'jeden'), (2, 'Dwa'), (2, 'dwa'), (4, 'cztery');

SELECT * FROM dane1 UNION SELECT * FROM dane2;
 l |   t
---+--------
 4 | cztery
 3 | trzy
 2 | dwa
 2 | Dwa
 1 | jeden
(5 rows)

SELECT * FROM dane1 UNION ALL SELECT * FROM dane2;
 l |   t
---+--------
 1 | jeden
 2 | dwa
 2 | dwa
 3 | trzy
 1 | jeden
 2 | Dwa
 2 | dwa
 4 | cztery
(8 rows)
```

Set Operations

```
CREATE TABLE dane1(l INT, t VARCHAR(20));
INSERT INTO dane1 VALUES(1, 'jeden'), (2, 'dwa'), (2, 'dwa'), (3, 'trzy');
CREATE TABLE dane2(l INT, t VARCHAR(20));
INSERT INTO dane2 VALUES(1, 'jeden'), (2, 'Dwa'), (2, 'dwa'), (4, 'cztery');

SELECT * FROM dane1 INTERSECT SELECT * FROM dane2;
 l |   t
---+-------
 2 | dwa
 1 | jeden
(2 rows)


SELECT * FROM dane1 EXCEPT SELECT * FROM dane2;
 l |  t
---+------
 3 | trzy
(1 row)

SELECT t FROM dane1 UNION SELECT t FROM dane2;
   t
--------
 dwa
 cztery
 jeden
 trzy
 Dwa
(5 rows)
```

Set Operations

```
CREATE TABLE dane1(l INT, t VARCHAR(20));
INSERT INTO dane1 VALUES(1, 'jeden'), (2, 'dwa'), (2, 'dwa'), (3, 'trzy');
CREATE TABLE dane2(l INT, t VARCHAR(20));
INSERT INTO dane2 VALUES(1, 'jeden'), (2, 'Dwa'), (2, 'dwa'), (4, 'cztery');


SELECT l FROM dane1 UNION SELECT l FROM dane2;
 l
---
 2
 3
 4
 1
(4 rows)


SELECT l FROM dane1 UNION SELECT t FROM dane2;
BLAD: UNION typy integer i character varying nie mogÄ byc dopasowane at character 34
LINE 1: SELECT l FROM dane1 UNION SELECT t FROM dane2;


SELECT * FROM dane1 UNION SELECT l FROM dane2;
BLAD:  kazde zapytanie UNION musi miec te sama liczbe kolumn at character 34
LINE 1: SELECT * FROM dane1 UNION SELECT l FROM dane2;
                                       ^
```

Joins

```
=> SELECT tabA.atr1, tabA.atr2, tabB.atr1, tabB.atr3 FROM tabA, tabB;
```

albo

```
⇒SELECT * FROM tabA, tabB;
```

albo

```
=> SELECT * FROM tabA CROSS JOIN tabB;
```

```
 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   10 |    2 |  200
    1 |   15 |    1 |  100
    1 |   15 |    2 |  200
    2 |   20 |    1 |  100
    2 |   20 |    2 |  200
    3 |   30 |    1 |  100
    3 |   30 |    2 |  200
(8 rows)
```

```
tabA(atr1 INT, atr2 INT);

=> SELECT * FROM tabA;
 atr1 | atr2
------+------
    1 |   10
    1 |   15
    2 |   20
    3 |   30
(4 rows)
```

```
tabB(atr1 INT, atr3 INT);

=> SELECT * FROM tabB;
 atr1 | atr3
------+------
    1 |  100
    2 |  200
(2 rows)
```

Joins

```
=> SELECT * FROM tabA, tabB WHERE tabA.atr1 = tabB.atr1;
albo
=> SELECT * FROM tabA INNER JOIN tabB ON tabA.atr1 = tabB.atr1;

 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   15 |    1 |  100
    2 |   20 |    2 |  200
(3 rows)

=> SELECT * FROM tabA NATURAL JOIN tabB;

 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   15 |    1 |  100
    2 |   20 |    2 |  200
(3 rows)
```

```
tabA(atr1 INT, atr2 INT);

=> SELECT * FROM tabA;
 atr1 | atr2
------+------
    1 |   10
    1 |   15
    2 |   20
    3 |   30
(4 rows)
```

```
tabB(atr1 INT, atr3 INT);

=> SELECT * FROM tabB;
 atr1 | atr3
------+------
    1 |  100
    2 |  200
(2 rows)
```

```
=> SELECT * FROM tabA LEFT OUTER JOIN tabB ON tabA.atr1 = tabB.atr1;
 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   15 |    1 |  100
    2 |   20 |    2 |  200
    3 |   30 |      |
(4 rows)
```

```
tabA(atr1 INT, atr2 INT);

=> SELECT * FROM tabA;
 atr1 | atr2
------+------
    1 |   10
    1 |   15
    2 |   20
    3 |   30
(4 rows)
```

```
=> SELECT * FROM tabA RIGHT OUTER JOIN tabB ON tabA.atr1 = tabB.atr1;
 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   15 |    1 |  100
    2 |   20 |    2 |  200
(3 rows)
```

```
tabB(atr1 INT, atr3 INT);

=> SELECT * FROM tabB;
 atr1 | atr3
------+------
    1 |  100
    2 |  200
(2 rows)
```

```
=> SELECT * FROM tabA FULL OUTER JOIN tabB ON tabA.atr1 = tabB.atr1;
 atr1 | atr2 | atr1 | atr3
------+------+------+------
    1 |   10 |    1 |  100
    1 |   15 |    1 |  100
    2 |   20 |    2 |  200
    3 |   30 |      |
(4 rows)
```

Dostęp kropkowy

Instancje tabeli

SELECT Query

```
CREATE TABLE osoby(pesel CHAR(11), imie_nazwisko VARCHAR(20), pesel_kierownika CHAR(11));
INSERT INTO osoby VALUES('11111111111', 'Aaa Aaaa', NULL);
INSERT INTO osoby VALUES('22222222222', 'Baa Baaa', '11111111111');
INSERT INTO osoby VALUES('33333333333', 'Caa Caaa', '11111111111');
```

```
=> SELECT * FROM osoby, osoby;
BLAD:  nazwa tabeli "osoby" okreslona wiecej niz raz
```

```
=> SELECT * FROM osoby pracownicy, osoby kierownicy;
```

pesel	imie_nazwisko	pesel_kierownika	pesel	imie_nazwisko	pesel_kierownika
11111111111	Aaa Aaaa		11111111111	Aaa Aaaa	
11111111111	Aaa Aaaa		22222222222	Baa Baaa	11111111111
11111111111	Aaa Aaaa		33333333333	Caa Caaa	11111111111
22222222222	Baa Baaa	11111111111	11111111111	Aaa Aaaa	
22222222222	Baa Baaa	11111111111	22222222222	Baa Baaa	11111111111
22222222222	Baa Baaa	11111111111	33333333333	Caa Caaa	11111111111
33333333333	Caa Caaa	11111111111	11111111111	Aaa Aaaa	
33333333333	Caa Caaa	11111111111	22222222222	Baa Baaa	11111111111
33333333333	Caa Caaa	11111111111	33333333333	Caa Caaa	11111111111

(9 rows)

```
=> SELECT * FROM osoby pracownicy, osoby kierownicy WHERE pracownicy.pesel = kierownicy.pesel;
```

pesel	imie_nazwisko	pesel_kierownika	pesel	imie_nazwisko	pesel_kierownika
11111111111	Aaa Aaaa		11111111111	Aaa Aaaa	
22222222222	Baa Baaa	11111111111	22222222222	Baa Baaa	11111111111
33333333333	Caa Caaa	11111111111	33333333333	Caa Caaa	11111111111

(3 rows)

```
=> SELECT * FROM osoby pracownicy, osoby kierownicy WHERE pracownicy.pesel_kierownika = kierownicy.pesel;
```

pesel	imie_nazwisko	pesel_kierownika	pesel	imie_nazwisko	pesel_kierownika
22222222222	Baa Baaa	11111111111	11111111111	Aaa Aaaa	
33333333333	Caa Caaa	11111111111	11111111111	Aaa Aaaa	

(2 rows)

Joins

```
-- nazwy atrybutow rozne, celem przejrzystosci - bez kluczy

CREATE TABLE Producenci(id_producenta INT, nazwa_producenta VARCHAR(30));
CREATE TABLE Produkty(id_produktu INT, producent INT, nazwa_produktu VARCHAR(30));

INSERT INTO Producenci VALUES(1, 'Predom Polar');
INSERT INTO Producenci VALUES(2, 'Domgos');
INSERT INTO Produkty VALUES(3, 1, 'Pralka PS 663');
INSERT INTO Produkty VALUES(3, 2, 'Latarka');


bd=> SELECT * FROM Producenci INNER JOIN Produkty ON (id_producenta = producent);
 id_producenta | nazwa_producenta | id_produktu | producent | nazwa_produktu
---------------+------------------+-------------+-----------+----------------
             1 | Predom Polar     |           3 |         1 | Pralka PS 663
             2 | Domgos           |           3 |         2 | Latarka
(2 rows)


bd=> SELECT * FROM Producenci NATURAL JOIN Produkty;
 id_producenta | nazwa_producenta | id_produktu | producent | nazwa_produktu
---------------+------------------+-------------+-----------+----------------
             1 | Predom Polar     |           3 |         1 | Pralka PS 663
             1 | Predom Polar     |           3 |         2 | Latarka
             2 | Domgos           |           3 |         1 | Pralka PS 663
             2 | Domgos           |           3 |         2 | Latarka
(4 rows)
```

Joins

```
-- nazwy atrybutow zgodne, celem przejrzystosci - bez kluczy
CREATE TABLE Producenci(id_producenta INT, nazwa_producenta VARCHAR(30));
CREATE TABLE Produkty(id_produktu INT, id_producenta INT, nazwa_produktu
VARCHAR(30));

INSERT INTO Producenci VALUES(1, 'Predom Polar');
INSERT INTO Producenci VALUES(2, 'Domgos');
INSERT INTO Produkty VALUES(3, 1, 'Pralka PS 663');
INSERT INTO Produkty VALUES(3, 2, 'Latarka');

bd=> SELECT * FROM Producenci NATURAL JOIN Produkty;
 id_producenta | nazwa_producenta | id_produktu | nazwa_produktu
---------------+------------------+-------------+----------------
             1 | Predom Polar     |           3 | Pralka PS 663
             2 | Domgos           |           3 | Latarka
(2 rows)

bd=> SELECT * FROM Producenci INNER JOIN Produkty ON (id_producenta =
id_producenta);
BLAD:  odnosnik kolumny "id_producenta" jest niejednoznaczny at character 50
LINE 1: SELECT * FROM Producenci INNER JOIN Produkty ON (id_producen...

-- dostep kropkowy
bd=> SELECT * FROM Producenci INNER JOIN Produkty
                            ON (Producenci.id_producenta = Produkty.id_producenta);
 id_producenta | nazwa_producenta | id_produktu | id_producenta | nazwa_produktu
---------------+------------------+-------------+---------------+----------------
             1 | Predom Polar     |           3 |             1 | Pralka PS 663
             2 | Domgos           |           3 |             2 | Latarka
(2 rows)
```

Self-Join

```
CREATE TABLE Dziesiec(cyfra INT);  -- zlaczenie wielokrotne tej samej tabeli

INSERT INTO Dziesiec VALUES (0);
INSERT INTO Dziesiec VALUES (1);
INSERT INTO Dziesiec VALUES (2);
INSERT INTO Dziesiec VALUES (3);
INSERT INTO Dziesiec VALUES (4);
INSERT INTO Dziesiec VALUES (5);
INSERT INTO Dziesiec VALUES (6);
INSERT INTO Dziesiec VALUES (7);
INSERT INTO Dziesiec VALUES (8);
INSERT INTO Dziesiec VALUES (9);

SELECT dziesiatki.cyfra * 10 + jednostki.cyfra AS liczba
                                    FROM Dziesiec dziesiatki, Dziesiec jednostki;
albo
SELECT dziesiatki.cyfra * 10 + jednostki.cyfra AS "liczba"
                           FROM Dziesiec dziesiatki CROSS JOIN Dziesiec jednostki;

bd=>SELECT dziesiatki.cyfra * 10 + jednostki.cyfra AS liczba FROM Dziesiec
dziesiatki, Dziesiec jednostki;
 liczba
--------
      0
      1
      2
…
     98
     99
(100 rows)
```

Nested Queries

W języku SQL jest możliwe zagnieżdżanie zapytań – umieszczanie zapytań (podzapytań) wewnątrz innych zapytań.

```
SELECT cena FROM tabela WHERE cena > (SELECT AVG(cena) FROM tabela) ;
```

Cechy podzapytania:
- musi być ujęte w nawiasy i nie może być zakończone średnikiem,
- przeważnie występuje we frazie WHERE,
- może być też umieszczone we frazach: SELECT, FROM, HAVING,
- może zawierać odwołania do kolumn wymienionych w zapytaniu zewnętrznym.

```
SELECT peselpracownika FROM zatrudnienie WHERE pensja <
   (SELECT dochody FROM pracownicy WHERE pesel = peselpracownika);
```

tutaj: zatrudnienie (peselpracownika, pensja) , pracownicy (pesel, dochody)

zapytanie zewnętrzne nie może zawierać kolumn tabel wymienionych jedynie w podzapytaniach

```
SELECT peselpracownika FROM zatrudnienie
    WHERE peselpracownika = pesel  AND  pensja <
      (SELECT dochody FROM pracownicy WHERE pesel = peselpracownika);
```

Maksymalna liczba poziomów zagnieżdżeń jest charakterystyczna dla danego systemu baz danych.

Uncorrelated and Correlated Subqueries

Uncorrelated (independent) subquery – the inner query is executed only once during execution of the outer query.

Correlated subquery – occurs when the outer query passes data to the inner query; in this case, the inner query is executed for each tuple of the outer query.

pracownicy (pesel INT, dochody INT)

```
 pesel | dochody
-------+---------
     1 |    2000
     2 |    3000
     3 |    4000
     4 |    5000
```

zatrudnienie (regonfirmy INT, peselpracownika INT, pensja INT)

```
regonfirmy | peselpracownika | pensja
-----------+-----------------+--------
       111 |               4 |   5000
       112 |               2 |   3000
       113 |               3 |   3000
       113 |               1 |   2000
```

```
=> SELECT regonfirmy FROM zatrudnienie
     WHERE peselpracownika IN
        (SELECT pesel FROM pracownicy
              WHERE dochody > 3000);

 regonfirmy
------------
        111
        113
```

```
=> SELECT peselpracownika FROM
     zatrudnienie WHERE pensja <
     (SELECT dochody FROM pracownicy
          WHERE pesel = peselpracownika);

 peselpracownika
-----------------
               3
```

Nested Queries and Joins

For some nested queries, equivalent table joins can be constructed:

```
SELECT DISTINCT * FROM TabA WHERE kolA1 IN (SELECT kolB1 FROM TabB);

SELECT DISTINCT TabA.* FROM  TabA INNER JOIN TabB ON TabA.kolA1 = TabB.kolB1;
```

```
SELECT * FROM TabA WHERE kolA1 NOT IN (SELECT kolB1 FROM TabB);

SELECT kolA1, kolA2 FROM TabA LEFT OUTER JOIN TabB
                    ON TabA.kolA1 = TabB.kolB1 WHERE TabB.kolB1 IS NULL;
```

```
SELECT DISTINCT * FROM dziesiec WHERE dziesiec.cyfra IN (SELECT cyfra FROM sto);


SELECT DISTINCT dziesiec.cyfra, dziesiec.bit, dziesiec.tekst
                    FROM dziesiec INNER JOIN sto ON dziesiec.cyfra = sto.cyfra;
```

```
SELECT * FROM tysiac WHERE tysiac.cyfra =
    (SELECT dziesiectysiecy.cyfra FROM dziesiectysiecy
        WHERE dziesiectysiecy.liczba = tysiac.liczba);
                                                          -- DISTINCT???
SELECT tysiac.* FROM tysiac JOIN dziesiectysiecy
    ON tysiac.cyfra=dziesiectysiecy.cyfra
        WHERE dziesiectysiecy.liczba = tysiac.liczba;
```

UPDATE – złączenia i zagnieżdżenia

```
CREATE TABLE wyniki (nr_wartosci INT, wartosc DECIMAL(5,2));
INSERT INTO wyniki VALUES (1, 0.01), (2, 0.02), (3, 0.3), (4, 0.4);


=> SELECT * FROM wyniki;
 nr_wartosci | wartosc
-------------+---------
           1 |    0.01
           2 |    0.02
           3 |    0.30
           4 |    0.40
(4 rows)

-- modyfikacja w oparciu o agregacje – wyliczona jeden raz na początku zapytania
--       (zagniezdzenie nieskorelowane)

=> UPDATE wyniki
       SET wartosc = wartosc + 0.1 WHERE wartosc < (SELECT AVG(wartosc) FROM wyniki);
UPDATE 2


=> SELECT * FROM wyniki;
 nr_wartosci | wartosc
-------------+---------
           3 |    0.30
           4 |    0.40
           1 |    0.11
           2 |    0.12
(4 rows)
```

UPDATE – złączenia i zagnieżdżenia

```
CREATE TABLE produkty (id_produktu INT, typ INT, nazwa VARCHAR (100),
                                    cena_netto DECIMAL(5,2), cena_brutto DECIMAL(5,2));
INSERT INTO produkty VALUES(1, 1, 'krem', 4.50);
INSERT INTO produkty VALUES(2, 1, 'proszek', 5.50);
INSERT INTO produkty VALUES(3, 2, 'bulka', 1.50);
INSERT INTO produkty VALUES(4, 2, 'rogal', 2.00);
CREATE TABLE podatki (id_typu INT, kategoria VARCHAR(20), podatek DECIMAL(3,2));
INSERT INTO podatki VALUES(1, 'chemia gosp', 0.23);
INSERT INTO podatki VALUES(2, 'zywnosc', 0.05);
```

```
bd=> SELECT * FROM produkty;
 id_produktu | typ |  nazwa   | cena_netto | cena_brutto
-------------+-----+----------+------------+-------------
           1 |   1 | krem     |       4.50 |
           2 |   1 | proszek  |       5.50 |
           3 |   2 | bulka    |       1.50 |
           4 |   2 | rogal    |       2.00 |
(4 rows)
```

```
bd=> SELECT * FROM podatki;
 id_typu |  kategoria   | podatek
---------+--------------+---------
       1 | chemia gosp  |    0.23
       2 | zywnosc      |    0.05
(2 rows)
```

```
-- modyfikacja w oparciu o wartosci z zewnętrznej tabeli (zagniezdzenie skorelowane)
bd=> UPDATE produkty SET cena_brutto
         = cena_netto + cena_netto * (SELECT podatek FROM podatki WHERE id_typu = typ);
UPDATE 4
```

```
bd=> SELECT * FROM produkty;
 id_produktu | typ |  nazwa   | cena_netto | cena_brutto
-------------+-----+----------+------------+-------------
           1 |   1 | krem     |       4.50 |        5.54
           2 |   1 | proszek  |       5.50 |        6.77
           3 |   2 | bulka    |       1.50 |        1.58
           4 |   2 | rogal    |       2.00 |        2.10
(4 rows)
```

```
-- modyfikacja w oparciu o wartosci z zewnętrznej tabeli (zlaczenie)
bd=> UPDATE produkty
     SET cena_brutto = cena_netto + cena_netto * podatek FROM podatki WHERE typ = id_typu;
UPDATE 4
```

```
bd=> SELECT * FROM produkty;
 id_produktu | typ |  nazwa   | cena_netto | cena_brutto
-------------+-----+----------+------------+-------------
           2 |   1 | proszek  |       5.50 |        6.77
           1 |   1 | krem     |       4.50 |        5.54
           4 |   2 | rogal    |       2.00 |        2.10
           3 |   2 | bulka    |       1.50 |        1.58
(4 rows)
```

Scalar and Vector

```
CREATE TABLE wyniki (nr_wartosci INT, wartosc DECIMAL(5,2));
INSERT INTO wyniki VALUES (1, 0.01), (2, 0.02), (3, 0.3), (4, 0.4);

=> SELECT * FROM wyniki WHERE wartosc * 2 < (SELECT MAX(wartosc) FROM wyniki);
 nr_wartosci | wartosc
-------------+---------
           1 |    0.01
           2 |    0.02
(2 rows)

=> SELECT * FROM wyniki
          WHERE wartosc * 2 < (SELECT wartosc FROM wyniki ORDER BY wartosc DESC LIMIT 1);
 nr_wartosci | wartosc
-------------+---------
           1 |    0.01
           2 |    0.02
(2 rows)

=> SELECT * FROM wyniki WHERE wartosc * 2 < (SELECT wartosc FROM wyniki ORDER BY wartosc DESC);
BLAD:  ponad jeden wiersz zwrocony przez podzapytanie uzyte jako wyrazenie

=> SELECT * FROM wyniki WHERE wartosc IN (SELECT wartosc FROM wyniki ORDER BY wartosc DESC);
 nr_wartosci | wartosc
-------------+---------
           1 |    0.01
           2 |    0.02
           3 |    0.30
           4 |    0.40
(4 rows)
```

Conditional Expressions in SQL

- SQL does not have conditional statements, but since the SQL-92 standard it has provided mechanisms for determining conditional values:
- CASE – the CASE/WHEN/END construct – discussed below,

- COALESCE(value_list) – returns the first NOT NULL value
- COALESCE(w1, w2, w3)
- equivalent to:
- CASE
- WHEN w1 IS NOT NULL THEN w1
- WHEN w2 IS NOT NULL THEN w2
- ELSE w3
- END

- NULLIF(w1, w2) – returns NULL if the values are equal; otherwise, it returns w1
- NULLIF(w1, w2)
- equivalent to:

Using CASE / WHEN

```
CREATE TABLE liczby(id_liczby INT, wartosc INT);
INSERT INTO liczby VALUES(1, 1);
INSERT INTO liczby VALUES(2, 2);
INSERT INTO liczby VALUES(3, -1);
INSERT INTO liczby (id_liczby) VALUES(4);
INSERT INTO liczby VALUES(5, 0);
```

```
bd=> SELECT * FROM liczby;
 id_liczby | wartosc
-----------+---------
         1 |       1
         2 |       2
         3 |      -1
         4 |
         5 |       0
(5 rows)
```

```
bd=> SELECT *,
bd-> CASE
bd->  WHEN wartosc > 0 THEN 'dodatnia'
bd->  WHEN wartosc < 0 THEN 'ujemna'
bd->  WHEN wartosc = 0 THEN 'zero'
bd-> ELSE
bd->  'nieokreslona (NULL)'
bd-> END
bd-> FROM liczby;
 id_liczby | wartosc |        case
-----------+---------+---------------------
         1 |       1 | dodatnia
         2 |       2 | dodatnia
         3 |      -1 | ujemna
         4 |         | nieokreslona (NULL)
         5 |       0 | zero
(5 rows)
```

```
SELECT *,
  CASE
   WHEN wartosc > 0 THEN 'dodatnia'
   WHEN wartosc < 0 THEN 'ujemna'
   WHEN wartosc = 0 THEN 'zero'
  ELSE
   'nieokreslona (NULL)'
  END
 FROM liczby;
```

Views

Język SQL przewiduje tworzenie wirtualnych tabel, opartych o zapytanie SELECT. Tabele te zwane są widokami i są przechowywane w bazie jako definicje zapytań. Ich zbiorem atrybutów są wyszczególnione kolumny z tabeli składowej / tabel składowych, użyte w danym zapytaniu. Widoki posiadają dane – są to krotki wybrane przez instrukcję SELECT. Dane te nie są przechowywane na stałe w bazie. Zestaw krotek jest opracowywany za każdym razem podczas przetwarzania aktualnego zapytania.

Zalety / cele stosowania widoków:
- uproszczony dostęp do danych – łączenie wielu tabel w jedną,
- aktualny stan danych – nie trzeba dokonywać kopiowania danych miedzy tabelami – każde użycie widoku zwraca aktualne dane
- ograniczenie dostępu – możliwość ukrycia wybranych atrybutów przed innymi użytkownikami bez redundancji i problemów aktualizacji,
- możliwość zachowania pewnych funkcjonalności po zmianie logicznej reprezentacji danych,
- widoki mogą enkapsulować skomplikowane zapytania / podzapytania, w szczególności, gdy są używane wielokrotnie,
- widoki mogą być zagnieżdżane.

Składnia tworzenia widoku:

```
CREATE VIEW widok [ (kolumna1, kolumna2, ..) ] AS <Zapytanie_SELECT>;
```

Views

```
TabA( t1 INT, t2 INT, t3 INT);

=> SELECT * FROM TabA;
 t1 | t2 | t3
----+----+----
  1 |  2 |  3
  2 |  4 |  6
(2 rows)

=> CREATE VIEW jedensuma AS SELECT t1, t1+t2+t3 AS "kol suma" FROM TabA;

SELECT * FROM jedensuma;
 t1 | kol suma
----+----------
  1 |         6
  2 |        12
(2 rows)
```

Transactions

```
Tab(a1 INT);
```

```
=> BEGIN;
=> UPDATE Tab SET a1 = a1*2;
```

```
=> DELETE FROM Tab WHERE a1>3;
```

```
=> COMMIT;
```

```
=> SELECT * FROM Tab;
 a1
----
  1
  3
(2 rows)
```

```
=> SELECT * FROM Tab;
 a1
----
  2
  6
(2 rows)
```

```
=> SELECT * FROM Tab;
 a1
----
  2
(1 row)
```

```
=> SELECT * FROM Tab;
 a1
----
  2
(1 row)
```

Transactions

```
Tab(a1 INT);


=> BEGIN;
=> UPDATE Tab SET a1 = a1*2;




=> DELETE FROM Tab WHERE a1>3;




=> ROLLBACK;
```

```
=> SELECT * FROM Tab;
 a1
----
  1
  3
(2 rows)

=> SELECT * FROM Tab;
 a1
----
  2
  6
(2 rows)

=> SELECT * FROM Tab;
 a1
----
  2
(1 row)

=> SELECT * FROM Tab;
 a1
----
  1
  3
(2 rows)
```

Stored Procedures

Systemy baz danych pozwalają na definiowanie w języku SQL funkcji, które są wywoływane przez klienta a wykonywane (interpretowane) po stronie serwera bazy. Kod samej funkcji może być napisany w zależności od danego systemu w:
- SQL,
- języku kompilowanym do kodu wykonywalnego (np. C/C++),
- języku interpretowanym przez serwer, charakterystycznym dla danej platformy (np. **plpgsql**).

Składnia tworzenia funkcji składowanej w SQL dla bazy PostgreSQL jest następująca:

```
CREATE [ OR REPLACE ] FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ RETURNS rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Stored Procedures

```
CREATE FUNCTION kwadratsumy(integer, integer)
  RETURNS integer
    AS 'select $1*$1 + 2*$1*$2 + $2*$2;'
  LANGUAGE SQL;

=>SELECT kwadratsumy(1,2);

 kwadratsumy
-------------
           9
(1 row)


DROP FUNCTION kwadratsumy(integer, integer);
```

Stored Procedures

```
=> CREATE FUNCTION dodatnie_suma()
    RETURNS NUMERIC
     AS
        'UPDATE dane SET a = -a WHERE a <0;
         SELECT AVG (a) FROM dane;'
     LANGUAGE SQL;

⇒SELECT dodatnie_suma();

    dodatnie_suma
--------------------
 2.0000000000000000
(1 row)

=> DROP FUNCTION dodatnie_suma();
```

```
=> CREATE TABLE dane(a INT);

=> SELECT * FROM dane;

 a
----
  1
  2
 -3
(3 rows)
```

Stored Procedures in Other Languages (PL/pgSQL, PL/Tcl, PL/Perl) in PostgreSQL

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int)
AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

www.postgresql.org

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

www.postgresql.org

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

www.postgresql.org

Triggers

Język SQL pozwala na definiowanie akcji użytkownika podczas wybranych operacji w bazie. Takim narzędziem są wyzwalacze. Są one wywoływanie podczas zmian zawartości tabel instrukcjami INSERT, UPDATE, DELETE. Definicja wyzwalacza w SQL jest następująca:

```
CREATE TRIGGER nazwatr { BEFORE | AFTER } { INSERT OR UPDATE OR DELETE }
  tabela FOR EACH { ROW|STATEMENT } EXECUTE PROCEDURE proctriggera
     ( parametry );
```

```
CREATE TABLE uczniowie(a INT);
CREATE TABLE nowiuczniowie(a INT);

CREATE OR REPLACE FUNCTION funkcjawyzwalacza ()
RETURNS TRIGGER AS  '
   BEGIN
      INSERT INTO nowiuczniowie VALUES (1);
      return NEW;
   END;
' LANGUAGE  'plpgsql' ;

INSERT INTO uczniowie VALUES(1);
(1)
CREATE TRIGGER wyzwalacz AFTER INSERT ON uczniowie
   FOR EACH ROW EXECUTE PROCEDURE funkcjawyzwalacza ();

INSERT INTO uczniowie VALUES(2);
(2)
```

```
(1)

 SELECT * FROM
         nowiuczniowie;
 a
---
(0 rows)


(2)

SELECT * FROM
         nowiuczniowie;
 a
---
 1
(1 row)
```

SQL – Practice …

```
CREATE TABLE Wojewodztwa(id_wojewodztwa INT PRIMARY KEY,
                                    nazwa_wojewodztwa VARCHAR(100));
CREATE TABLE Powiaty(id_powiatu INT PRIMARY KEY,
      nazwa_powiatu VARCHAR(100), id_wojewodztwa INT REFERENCES
                                          Wojewodztwa(id_wojewodztwa) );
CREATE TABLE Miejscowosci(id_miejscowosci INT PRIMARY KEY,
     nazwa_miejscowosci VARCHAR(100), id_powiatu INT REFERENCES
                                                Powiaty(id_powiatu));

INSERT INTO Wojewodztwa(id_wojewodztwa, nazwa_wojewodztwa)
                                          VALUES(1,'Malopolskie');
INSERT INTO Wojewodztwa VALUES(2,'Opolskie');
INSERT INTO Wojewodztwa VALUES(3,'Podkarpackie'), (4, 'Slaskie');

INSERT INTO Powiaty VALUES(1, 'Krakowski', 1);
INSERT INTO Powiaty VALUES(2, 'Wielicki', 1);
INSERT INTO Powiaty VALUES(3, 'Nyski', 2);

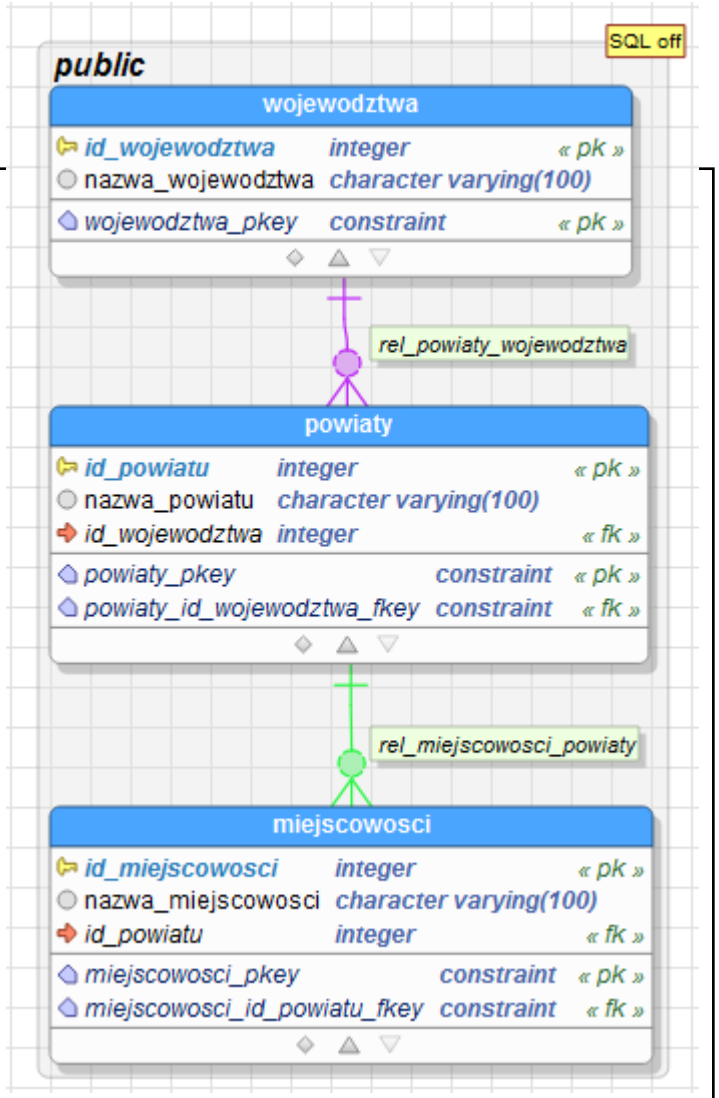
INSERT INTO Miejscowosci VALUES(1, 'Skawina', 1);
INSERT INTO Miejscowosci VALUES(200, 'Wieliczka', 2);
INSERT INTO Miejscowosci VALUES(300, 'Nysa', 3);
```

SQL – Practice …

```
bd=> SELECT * FROM Wojewodztwa;
 id_wojewodztwa | nazwa_wojewodztwa
----------------+-------------------
              1 | Malopolskie
              2 | Opolskie
              3 | Podkarpackie
              4 | Slaskie
(4 rows)

bd=> SELECT * FROM Powiaty;
 id_powiatu | nazwa_powiatu | id_wojewodztwa
------------+---------------+----------------
          1 | Krakowski     |              1
          2 | Wielicki      |              1
          3 | Nyski         |              2
(3 rows)

bd=> SELECT * FROM Miejscowosci;
 id_miejscowosci | nazwa_miejscowosci | id_powiatu
-----------------+--------------------+------------
               1 | Skawina            |          1
             200 | Wieliczka          |          2
             300 | Nysa               |          3
(3 rows)
```



```
bd=> SELECT * FROM Wojewodztwa INNER JOIN Powiaty ON Wojewodztwa.id_wojewodztwa =
Powiaty.id_wojewodztwa;

 id_wojewodztwa | nazwa_wojewodztwa | id_powiatu | nazwa_powiatu | id_wojewodztwa
----------------+-------------------+------------+---------------+----------------
              1 | Malopolskie       |          1 | Krakowski     |              1
              1 | Malopolskie       |          2 | Wielicki      |              1
              2 | Opolskie          |          3 | Nyski         |              2
(3 rows)

bd=> SELECT * FROM Wojewodztwa NATURAL JOIN Powiaty;

 id_wojewodztwa | nazwa_wojewodztwa | id_powiatu | nazwa_powiatu
----------------+-------------------+------------+---------------
              1 | Malopolskie       |          1 | Krakowski
              1 | Malopolskie       |          2 | Wielicki
              2 | Opolskie          |          3 | Nyski
(3 rows)

-- ZAPYTANIA ŁĄCZĄCE 3 TABELE (lub więcej)

bd=> SELECT * FROM Wojewodztwa NATURAL JOIN Powiaty NATURAL JOIN Miejscowosci;

 id_powiatu | id_wojewodztwa | nazwa_wojewodztwa | nazwa_powiatu | id_miejscowosci | nazwa_miejscowosci
------------+----------------+-------------------+---------------+-----------------+--------------------
          1 |              1 | Malopolskie       | Krakowski     |               1 | Skawina
          2 |              1 | Malopolskie       | Wielicki      |             200 | Wieliczka
          3 |              2 | Opolskie          | Nyski         |             300 | Nysa
(3 rows)

bd=> SELECT * FROM Wojewodztwa INNER JOIN Powiaty ON Wojewodztwa.id_wojewodztwa = Powiaty.id_wojewodztwa
INNER JOIN Miejscowosci ON Powiaty.id_powiatu = Miejscowosci.id_powiatu;
 id_wojewodztwa | nazwa_wojewodztwa | id_powiatu | nazwa_powiatu | id_wojewodztwa | id_miejscowosci | nazwa_miejscowosci | id_powiatu
----------------+-------------------+------------+---------------+----------------+-----------------+--------------------+------------
              1 | Malopolskie       |          1 | Krakowski     |              1 |               1 | Skawina            |          1
              1 | Malopolskie       |          2 | Wielicki      |              1 |             200 | Wieliczka          |          2
              2 | Opolskie          |          3 | Nyski         |              2 |             300 | Nysa               |          3
(3 rows)
```

REALIZACJA WYBRANYCH STRUKTUR DANYCH W RELACYJNYCH BAZACH DANYCH

- **Vector and matrix representation in relational databases**
- **Graph representation in relational databases**
- **Relational databases are based on sets → they do not provide mechanisms for implementing graphs or tree structures. Tree structures can, however, be implemented in relational databases.**
- **Tree representation in relational databases**
- **Method I – extending the attribute set of node objects**
- **Method II – ordered tree traversal**

VECTOR / MATRIX

Wektor o nieokreślonej długości:

```
CREATE TABLE wektor(pozycja INT NOT NULL, wartosc REAL)
```

Wektor o określonej długości:

```
CREATE TABLE wektor(poz INT NOT NULL CHECK(poz >= 1 AND poz <= 10), wartosc REAL)
```

Macierz o nieokreślonych wymiarach (i np. rzadka)

```
CREATE TABLE macierz(w INT NOT NULL, k INT NOT NULL, wartosc REAL);
```

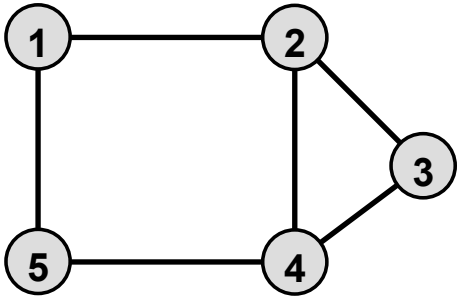
M

3	8	1
12	0	7

macierz

w	k	wartosc
1	1	3
1	2	8
1	3	1
2	1	12
2	2	0
2	3	7

Graph Representation in Relational Databases

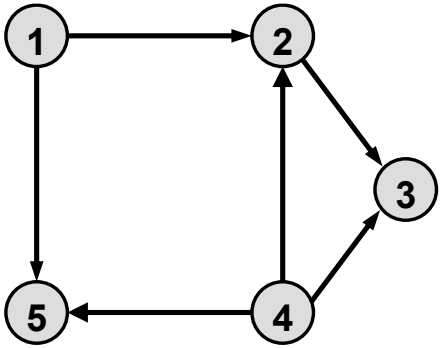


wierzcholki

V	dane
1	aaa
2	bbb
3	ccc
4	ddd
5	eee

krawedzie

V1	V2	dane
1	2	AAAA
2	3	BBBB
2	4	CCCC
3	4	DDDD
4	5	EEEE
5	1	FFFF



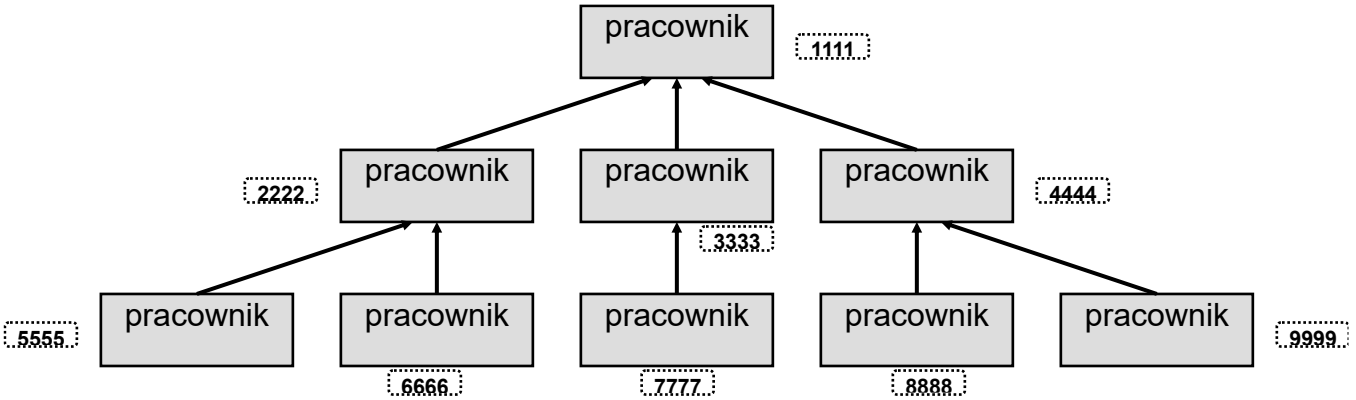
wierzcholki

V	dane
1	aaa
2	bbb
3	ccc
4	ddd
5	eee

luki

Vs	Ve	dane
1	2	AAAA
2	3	BBBB
4	2	CCCC
4	3	DDDD
4	5	EEEE
1	5	FFFF

Trees in Relational Databases – Method I – Extending the Attribute Set of Node Objects



pracownicy

pesel	imię	nazwisko	pesel nad.
1111	Adam	Nowak	
2222	Ewa	Nowak	1111
3333	Jan	Nowak	1111
4444	Anna	Nowak	1111
5555	Marek	Nowak	2222
6666	Olga	Nowak	2222
7777	Piotr	Nowak	3333
8888	Maria	Nowak	4444
9999	Jakub	Nowak	4444

```
CREATE TABLE pracownicy(
  pesel INT,
  imie VARCHAR(20),
  nazwisko VARCHAR(30),
  peselnad INT);
```

Measuring tree height is difficult – it can be implemented using additional tables, although there is no elegant solution.
Most tree operations are performed using self-joins.

Trees in Relational Databases – Method I – Extending the Attribute Set of Node Objects

```
-- wyswietlenie danych osob podrzednych np dla osoby o imieniu EWA

=> SELECT * FROM pracownicy WHERE peselnad =
                                          (SELECT pesel FROM pracownicy WHERE imie = 'Ewa');

 pesel | imie  | nazwisko | peselnad
-------+-------+----------+----------
  5555 | Marek | Nowak    |     2222
  6666 | Olga  | Nowak    |     2222
(2 rows)
```

```
-- wyswietlenie danych osoby nadrzednej dla osoby o imieniu Jakub

=> SELECT * FROM pracownicy WHERE pesel =
                                       (SELECT peselnad FROM pracownicy WHERE imie = 'Jakub');

 pesel | imie | nazwisko | peselnad
-------+------+----------+----------
  4444 | Anna | Nowak    |     1111
(1 row)
```

```
-- wyswietlenie korzenia drzewa

=> SELECT * FROM pracownicy WHERE peselnad IS NULL;

 pesel | imie | nazwisko | peselnad
-------+------+----------+----------
  1111 | Adam | Nowak    |
(1 row)
```

Trees in Relational Databases – Method I – Extending the Attribute Set of Node Objects

```
-- wyswietlenie wezlow posrednich

=> SELECT * FROM pracownicy WHERE pesel  IN  (SELECT peselnad FROM pracownicy);

 pesel | imie | nazwisko | peselnad
-------+------+----------+----------
  1111 | Adam | Nowak    |
  2222 | Ewa  | Nowak    |     1111
  3333 | Jan  | Nowak    |     1111
  4444 | Anna | Nowak    |     1111
(4 rows)
```

```
-- wyswietlenie wezlow posrednich z uzyciem zagniezdzen / aliasow

=> SELECT * FROM pracownicy AS pracownicy_zew WHERE
                 (SELECT COUNT (*) FROM pracownicy AS pracownicy_wew
                           WHERE pracownicy_zew.pesel = pracownicy_wew.peselnad) > 0;
```

(albo)

```
=> SELECT * FROM pracownicy AS pracownicy_zew WHERE EXISTS
                      (SELECT * FROM pracownicy AS pracownicy_wew
                                WHERE pracownicy_zew.pesel = pracownicy_wew.peselnad);

 pesel | imie | nazwisko | peselnad
-------+------+----------+----------
  1111 | Adam | Nowak    |
  2222 | Ewa  | Nowak    |     1111
  3333 | Jan  | Nowak    |     1111
  4444 | Anna | Nowak    |     1111
(4 rows)
```

Trees in Relational Databases – Method I – Extending the Attribute Set of Node Objects

```
-- wyswietlenie wezlow posrednich z uzyciem autozlaczenia


SELECT DISTINCT pracownicy_wew.peselnad FROM pracownicy AS pracownicy_zew
                                INNER JOIN pracownicy AS pracownicy_wew
                                                    ON pracownicy_zew.pesel = pracownicy_wew.peselnad ;
 peselnad
----------
     1111
     2222
     3333
     4444
(4 rows)
```

```
-- wyswietlenie lisci  (w drzewie korzen ma dla peselnad wartosc NULL!)

=> SELECT * FROM pracownicy WHERE pesel NOT IN (SELECT peselnad FROM pracownicy);

 pesel | imie | nazwisko | peselnad
-------+------+----------+----------
(0 rows)


=> SELECT * FROM pracownicy WHERE pesel NOT IN
                                          (SELECT peselnad FROM pracownicy WHERE peselnad IS NOT NULL);

 pesel | imie  | nazwisko | peselnad
-------+-------+----------+----------
  5555 | Marek | Nowak    |     2222
  6666 | Olga  | Nowak    |     2222
  7777 | Piotr | Nowak    |     3333
  8888 | Maria | Nowak    |     4444
  9999 | Jakub | Nowak    |     4444
(5 rows)
```

Trees in Relational Databases – Method I – Extending the Attribute Set of Node Objects

```
-- wyswietlenie lisci z uzyciem zagniezdzen / aliasow

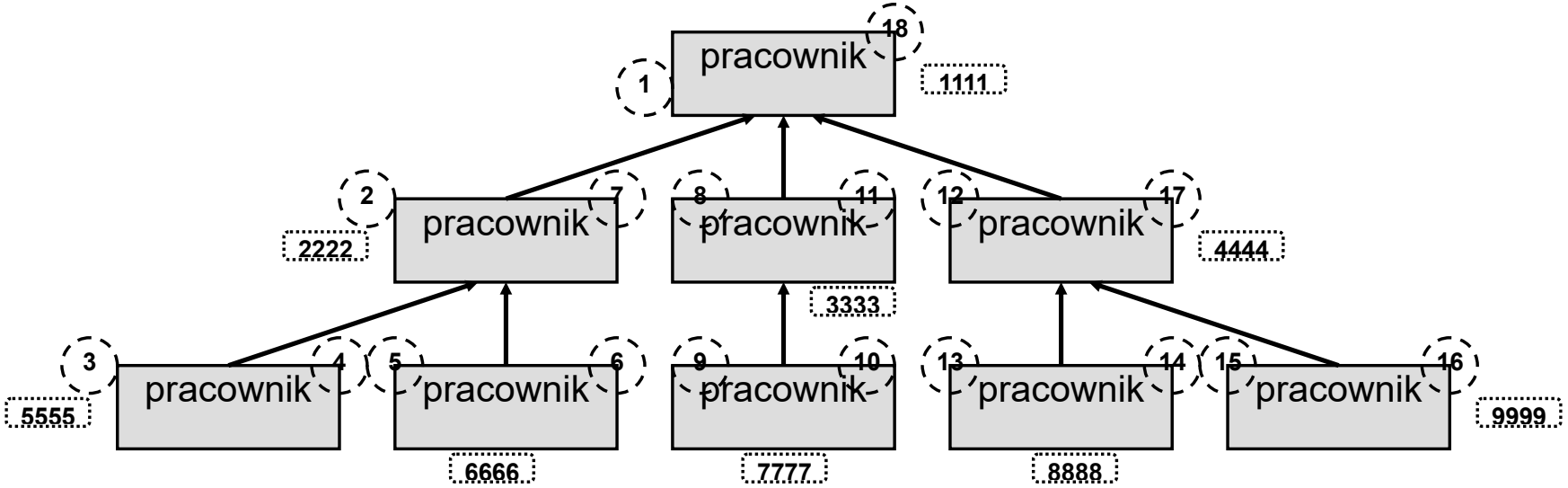

=> SELECT * FROM pracownicy AS pracownicy_zew WHERE
                    (SELECT COUNT (*) FROM pracownicy AS pracownicy_wew
                                    WHERE pracownicy_zew.pesel = pracownicy_wew.peselnad) = 0;
```

(albo)

```
=> SELECT * FROM pracownicy AS pracownicy_zew WHERE NOT EXISTS
                          (SELECT * FROM pracownicy AS pracownicy_wew
                                       WHERE pracownicy_zew.pesel = pracownicy_wew.peselnad);

 pesel | imie  | nazwisko | peselnad
-------+-------+----------+----------
  5555 | Marek | Nowak    |     2222
  6666 | Olga  | Nowak    |     2222
  7777 | Piotr | Nowak    |     3333
  8888 | Maria | Nowak    |     4444
  9999 | Jakub | Nowak    |     4444
(5 rows)
```

Trees in Relational Databases – Method II – Ordered Tree Traversal



pracownicy

pesel	imię	nazwisko	L	R
1111	Adam	Nowak	1	18
2222	Ewa	Nowak	2	7
3333	Jan	Nowak	8	11
4444	Anna	Nowak	12	17
5555	Marek	Nowak	3	4
6666	Olga	Nowak	5	6
7777	Piotr	Nowak	9	10
8888	Maria	Nowak	13	14
9999	Jakub	Nowak	15	16

```
CREATE TABLE pracownicy(
  pesel INT,
  imie VARCHAR(20),
  nazwisko VARCHAR(30),
  l INT, r INT);
```

Trees in Relational Databases – Method II – Ordered Tree Traversal

```
-- wyswietlenie korzenia

=> SELECT * FROM pracownicy WHERE l=1;

 pesel | imie | nazwisko | l | r
-------+------+----------+---+----
  1111 | Adam | Nowak    | 1 | 18
(1 row)
```

```
-- wyswietlenie lisci

=> SELECT * FROM pracownicy WHERE r - l = 1;

 pesel | imie  | nazwisko | l  | r
-------+-------+----------+----+----
  5555 | Marek | Nowak    |  3 |  4
  6666 | Olga  | Nowak    |  5 |  6
  7777 | Piotr | Nowak    |  9 | 10
  8888 | Maria | Nowak    | 13 | 14
  9999 | Jakub | Nowak    | 15 | 16
(5 rows)
```