

5. JEZYK SQL

Język SQL jest uniwersalnym, standaryzowanym językiem do komunikacji między klientem a bazą danych. Wśród wielu jego dialektów najważniejsze są standardy, w tym ANSI SQL i SQL2 (1992).

SQL nie jest językiem proceduralnym, jego użycie polega na definiowaniu poszczególnych komend. Komendy te mogą być uruchamiane w sposób interaktywny (w komunikacji terminalowej między użytkownikiem/administratorem a bazą) lub osadzone (jako zaszyta część aplikacji).

SQL – [skrót ?] - SQL nie jest strukturalny, SQL to nie tylko zapytania

Składnia SQL

- słowa kluczowe – zbiór ponad 200 słów zarezerwowanych na komendy i operatory; dla zachowania przejrzystości warto je zapisywać dużymi literami
- identyfikatory – do rozróżniania utworzonych przez użytkownika obiektów w bazie danych,
- operatory:
 - arytmetyczne,
 - logiczne,
 - porównawcze,
 - tekstowe,
- literały – liczby i łańcuchy znaków, zawarte między apostrofami,
- znaki porządkowe – { , ' ; () }.

Fraza – zaczyna się od słowa kluczowego komendy, zawiera odpowiedni zestaw słów kluczowych, identyfikatorów, operatorów, literałów i znaków porządkowych.

Zapytanie - składa się z jednej lub kilku fraz (zagnieżdżenie), kończy się znakiem porządkowym (średnik!).

DDL – Data Definition Language

DQL – Data Query Language

DML – Data Modification Language

DCL – Data Control Language

5.1 Podstawowe typy danych w języku SQL

Łańcuchy tekstowe

CHARACTER (*długość*) – CHAR – łańcuch znaków o długości *długość*, wprowadzone krótsze łańcuchy są dopełniane spacjami.

VARCHAR (*długość*) – CHARACTER VARYING / CHAR VARYING – łańcuch znaków o max. długości *długość*, krótsze łańcuchy nie są dopełniane spacjami.

NATIONAL CHARACTER / NATIONAL CHARACTER VARYING – ciągi tekstowe w standardzie UNICODE

Łańcuchy bitowe

BIT (*długość*) – łańcuch znaków o długości *długość*, wprowadzone krótsze łańcuchy są dopełniane spacjami.

BIT VARYING (*długość*) – łańcuch znaków o maksymalnej długości *długość*.

Liczby dokładne

NUMERIC – liczba dziesiętna

DECIMAL – DEC – liczba dziesiętna

INTEGER – INT – liczba całkowita, zakres ustalony przez system b.d.

SMALLINT – liczba całkowita o mniejszym zakresie.

Liczby zmiennoprzecinkowe

FLOAT (*rozdzielczość*) – liczba zmiennoprzecinkowa o rozdzielczości ustalonej przez użytkownika

REAL (*długość*) – liczba zmiennoprzecinkowa o rozdzielczości ustalonej przez system b.d.

DOUBLE PRECISION (*długość*) – liczba zmiennoprzecinkowa o największej precyzji.

Czas i data

DATE – data zdefiniowana przez trzy liczby całkowite – rok (YEAR), miesiąc (MONTH) i dzień (DAY), YYYY-MM-DD.

TIME (*rozdzielczość*) – czas zdefiniowany przez dwie liczby całkowite – godzina (HOUR), minuty (MINUTE) oraz liczbę dziesiętną sekund (SECOND), HH:MM:SS.SSSS(*rozdzielczość*)

NULL

NULL – wskazanie nieistnienia danej, nieporównywalne, wiele NULL-i może zostać potraktowane jak duplikat, są grupowane, ich obecność w wyrażeniach rzutuje na wynik wyrażenia równy NULL.

Rzutowanie

Konwersje niejawne – dozwolone przez system bazy danych. W przypadku wyrażen 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)`

Operatory

Operatory arytmetyczne

W SQL dostępne są proste operatory arytmetyczne {+, -, +, -, *, /},

Operatory logiczne

W SQL dostępne są operatory logiczne NOT , AND, OR.

Operacje na łańcuchach tekstowych

W SQL operacji na tekstach można dokonać poprzez użycie:

- operatora złączenia łańcuchów tekstowych ||, (w MySQL jest to odpowiednik OR; w MS Access operator +)

- zbioru funkcji do operacji na tekstach:

{ CONCAT(), SUBSTRING(), UPPER(), LOWER(), TRIM(), CHARACTER_LENGTH(), POSITION(), ... }

Kolejność wykonywania działań

Kolejność wykonywania operacji jest wyznaczana w oparciu o priorytet operatora. W przypadku dwóch takich samych operatorów kolejność jest wyznaczana w oparciu o wiązanie (od lewej do prawej).

Zmianę kolejności wykonywania operacji można wykonać przy pomocy nawiasów ().

priorytet operatora	
↑	+, - (znak)
	*, /
	+, -
	<, <=, =, >, >=, <>
	NOT
	AND
	OR

5.2 Tworzenie tabel w języku SQL

CREATE TABLE

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.

Przykładowe tabele:

```
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 ) );
```

Klucz główny prosty:

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

Klucz główny złożony

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

Klucz obcy prosty

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

Klucz obcy złożony

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

ALTER TABLE

Polecenie ALTER TABLE modyfikuje schemat relacji. Składnia jego jest następująca:

```
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;
```

DROP TABLE

Polecenie `DROP TABLE tabela;` usuwa wybraną tabelę z bazy.

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.

5.3 Wstawianie, modyfikowanie i usuwanie danych w tabelach (DML)

INSERT

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

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

Dane można też wpisywać konstrukcją `SELECT INTO`.

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 = 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.

```
TRUNCATE TABLE tabela;
```

5.4 Tworzenie i usuwanie indeksów

Indeksy to dodatkowa struktura w bazie danych, ułatwiająca operacje w tabeli. Nie jest objęta specyfikacją SQL, więc rozwiązania są specyficzne dla danego systemu baz danych.

Tworzenie indeksów dla tabel przeprowadza się komendą `CREATE INDEX` o następującej składni:

```
CREATE INDEX nazwaindexu ON tabela (atrybut 1, atrybut 2, ...);
```

Do usuwania indeksów służy polecenie `DROP INDEX`:

```
DROP INDEX nazwaindex; | DROP INDEKS nazwaindex ON tabela; | DROP INDEX tabela.nazwaindex;
```

5.5 Zapytanie SELECT

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_agrupowania1 [, kolumna_grupowania2, ...] ]
        [HAVING warunek_filtrowania_grupy]
        [ORDER BY kolumna_sortowania1 [, kolumna_sortowania2,...] [ASC DESC] ] ;
```

Wyświetlanie zawartości tabeli:

```
SELECT * FROM tabela;
```

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

DISTINCT – eliminowanie powtarzających się krotek

Tworzenie aliasów kolumn

```
SELECT kolumna1 AS nowa_nazwa_1, wyrażenie+kolumna_3 AS nowa_kolumna FROM tab;
```

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;
```

Zapytanie z warunkiem wybierania

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 kolumna 3 <> 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 * 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__';
```

5.6 Funkcje agregujące

Standard SQL określa następujące funkcje agregujące:

MIN(wyrażenie) - wyznacza minimum danego atrybutu z krotek wg określonego wyrażenia,

MAX(wyrażenie) - wyznacza maksimum danego atrybutu z krotek wg określonego wyrażenia,

SUM(wyrażenie) - wyznacza sumę danego atrybutu z krotek wg określonego wyrażenia,

AVG(wyrażenie) - wyznacza średnią wartość danego atrybutu z krotek wg określonego wyrażenia,

COUNT(wyrażenie) - wyznacza liczbę niepustych dla danych atrybutów krotek, w przypadku '*' dotyczy także wartości NULL,

wyrażenie - jest wyrażeniem (ew. arytmetycznym!) opartym o nazwę kolumny (kolumn).

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).

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 | 4.6666666666666667
(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)
```

=> SELECT COUNT (a+b) FROM test;

```
count
-----
3
(1 row)
```

5.7 Grupowanie danych

Opcja **GROUP BY** atrybut wydziela logiczną grupę krotek o tej samej wartości kolumny atrybut i przeprowadza na tej grupie operacje agregujące.

- 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,
- wyświetlanie w zapytaniu kolumn i wywołań funkcji agregujących jest możliwe, o ile te kolumny będą grupowane,

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

Grupowana (atr1 INT, atr2 INT)

=> SELECT * FROM Grupowana;

atr1	atr2
1	10
1	10
2	10
3	10
1	20
2	20
3	30

(7 rows)

=> SELECT atr1 FROM Grupowana
GROUP BY atr1;

atr1
3
2
1

(3 rows)

=> SELECT atr2 FROM Grupowana
GROUP BY atr2;

atr2
30
20
10

(3 rows)

=> SELECT atr1, atr2 FROM Grupowana
GROUP BY atr1, atr2;

atr1	atr2
1	10
3	10
2	10
2	20
1	20
3	30

(6 rows)

=> SELECT atr1 FROM Grupowana
GROUP BY atr1 HAVING atr1>1;

atr1
3
2

(2 rows)

5.8 Złączenia

```
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 tabA.atr1, tabA.atr2, tabB.atr1, tabB.atr3 FROM tabA, tabB;
albo
```

```
=> SELECT * FROM tabA, 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)

```
=> 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)

```
=> SELECT * FROM tabA, tabB
WHERE 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)


```
=> 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 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)

```
=> 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)

```
=> 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)

5.9 Działania na zbiorach

SQL umożliwia działania proste na parze tabeli. 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;
```

5.10 Zagnieżdżenia zapytań (podzapytania)

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:

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

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

- 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.

Zagnieżdżenia proste i skorelowane

pracownicy (pesel INT, dochody INT)

```
=> SELECT * FROM pracownicy;
```

pesel	dochody
1	2000
2	3000
3	4000
4	5000

zatrudnienie (regonfirmy INT, peselpracownika INT, pensja INT)

```
=> SELECT * FROM zatrudnienie;
```

regonfirmy	peselpracownika	pensja
111	4	5000
112	2	3000
113	3	3000
113	1	2000

Zagnieżdżenie proste (niezależne) – zapytanie wewnętrzne jest wykonywane tylko raz w czasie wykonywania zapytania zewnętrznego.

Zagnieżdżenie skorelowane – ma miejsce, gdy zapytanie zewnętrzne przekazuje dane do zapytania wewnętrznego – wówczas zapytanie wewnętrzne jest wykonywane dla każdej krotki zapytania zewnętrznego.

```
=> 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

Sortowanie wyników w podzapytaniu nie jest istotne dla wyniku zapytania – wydłuża tylko czas wykonania.

Zapytania zagnieżdżone a złączenia

Dla części zapytań zagnieżdżonych można skonstruować równoważne złączenia tabel:

```
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;
```

5.11 Widoki (perspektywy)

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 między 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>;
```

Przykład:

```
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)
```

5.12 Transakcje

Obsługa transakcji w języku SQL odbywa się przy pomocy komend: BEGIN TRANSACTION, (ew. BEGIN - PostgreSQL, MySQL) , COMMIT, ROLLBACK.

<pre> Tab(a1 INT) ; => BEGIN; => UPDATE Tab Set a1 = a1*2; => DELETE FROM Tab WHERE a1>3; => COMMIT; </pre>	<pre> => 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) </pre>
<pre> Tab(a1 INT) ; => BEGIN; => UPDATE Tab Set a1 = a1*2; => DELETE FROM Tab WHERE a1>3; => ROLLBACK; </pre>	<pre> => 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) </pre>

5.13 Procedury składowane

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 [, ...] ) ]
```

Przykładowe funkcje w języku SQL

```
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);
```

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

```
=> SELECT * FROM dane;
```

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

```
=> 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();
```

Procedury składowane w innych językach (PL/pgSQL, PL/Tcl, PL/Perl) w bazie 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
5.14 Wyzwalacze (Triggers)

Język SQL pozwala na definiowanie akcji użytkownika podczas wybranych operacji w bazie. Takim narzędziem są wyzwalacze. Są one wywoływane podczas zmian zawartości tabel instrukcjami INSERT, UPDATE, DELETE.

Definicja wyzwalacza w SQL jest następująca:

```
CREATE TRIGGER nazwatriggiera { 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)

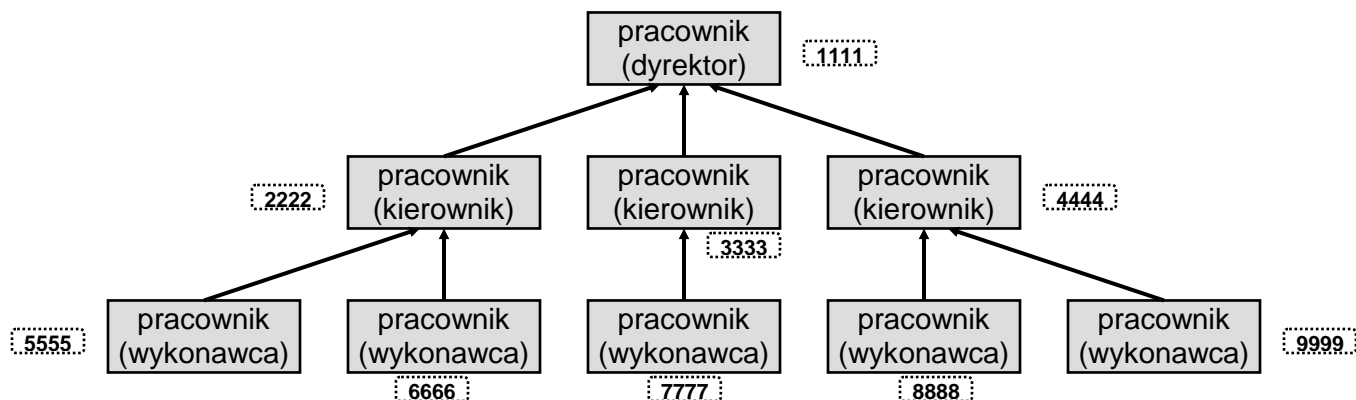
```
SELECT * FROM
    nowiuczniowie;
a
---
1
(1 row)
```

6. REALIZACJA WYBRANYCH STRUKTUR W RELACYJNYCH BAZACH DANYCH

6.1 Reprezentacja drzew w relacyjnych bazach danych

Relacyjne bazy danych opierają się o zbiory -> brak mechanizmów realizacji struktur drzewiastych. Struktury drzewiaste można jednak zaimplementować w r.b.d.

Sposób I – rozszerzenie zbioru atrybutów obiektów-węzłów



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);
```

Mierzenie wysokości drzewa jest trudne – można to realizować w oparciu o dodatkowe tabele, aczkolwiek eleganckiego rozwiązania nie ma.

Większość operacji na drzewie wykonuje się przy pomocy autozłączeń.

```
/* wyświetlenie 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)
```

```
/* 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)
```



```
/* 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)
```

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

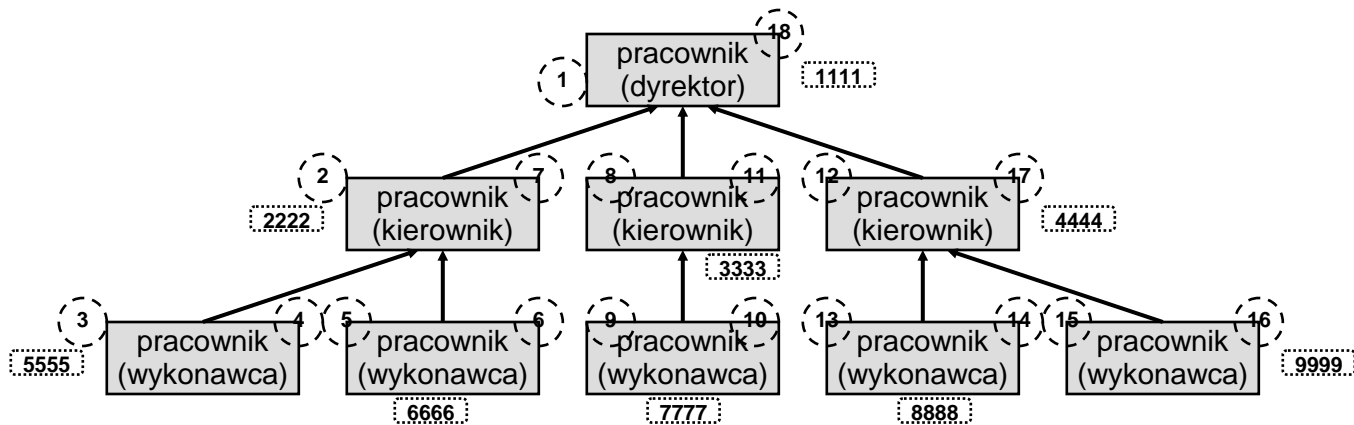
=> SELECT * FROM pracownicy AS pracownicy_zew WHERE
      (SELECT COUNT (*) FROM pracownicy AS pracownicy_wew
      WHERE pracownicy_zew.pesel = pracownicy_wew.peselnad) = 0;

(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)
```

Sposób II – uporządkowany przegląd drzewa

**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);
```

```
/* wyświetlenie korzenia */

=> SELECT * FROM pracownicy WHERE l=1;

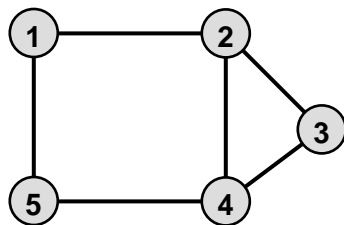
  pesel | imie | nazwisko | l | r
-----+-----+-----+---+---
  1111 | Adam | Nowak    | 1 | 18
(1 row)
```

```
/* wyświetlenie liści */

=> 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)
```

6.2 Reprezentacja grafów w relacyjnych bazach danych

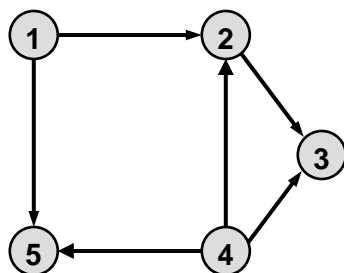


wierzchołki

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

krawędzie

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



wierzchołki

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

6.3 Reprezentacja macierzy w relacyjnych bazach danych

Wektor o nieokreślonej długości:

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

Wektor o określonej długości:

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

Macierz o nieokreślonych wymiarach (i np. rzadka)

```
CREATE TABLE macierz(i INT NOT NULL, j INT NOT NULL, wartosc INT)
```

macierz

i	j	wartosc
1	1	3
1	2	8
1	3	1
2	1	12
2	2	0
2	3	7

M

3	8	1
12	0	7