Ćwiczenia z języka SQL w oparciu o bazę danych SQLite

SQLite jest biblioteką oprogramowania, która implementuję samodzielną i bezserwerową, nie wymagający konfiguracji **mechanizm bazy danych SQL**. Kod źródłowy dla SQLite znajduję się w domenie publicznej (public domain). Baza danych SQLite jest obecnie bardzo popularna i jest użytkowana w setkach milionów egzemplarzy na całym świecie. SQLite jest używany w systemie Solaris 10 i MAC OS, iPhone i Skype. Biblioteka Qt4 ma wbudowane wsparcie dla SQLite, tak samo jak Python czy PHP. Wiele popularnych aplikacji ma wbudowaną bazę SQLite do obsługi wewnętrznej takie jak Firefox czy Amarok. SQLite ma w większości zaimplementowany standard SQL-92. Silnik SQLite nie tworzy samodzielnego procesu dla bazy. Zamiast tego jest statycznie lub dynamicznie dołączany do aplikacji. Biblioteka SQLite ma niewielkie rozmiary około 470 kB. Baza danych SQLite to pojedynczy plik na dysku, który można umieścić w dowolnym miejscu w hierarchii katalogów. Może być stosowany na różnych systemach operacyjnych, zarówno 32 jak 64 bitowych. SQLite został napisany w języku programowania C i ma powiązania dla wielu języków jak C++, Java, C#, Python, Perl, Ruby, Visual Basic i innych. Biblioteka SQLite pozwala na używanie bazy danych bez konieczności uruchamiania oddzielnego procesu jak to ma miejsce na przykład w przypadku MySQL. Zwartość bazy danych jest przetrzymywana w jednym pliku binarnym. SQlite rozwiązuje odwieczny problem programistów przechowujących dane w plikach płaskich, a mianowicie blokowanie i dostęp współbieżny. Głównym plusem tej bazy wydajność szczególnie przy wykonywaniu zapytań typu INSERT i SELECT, a także wieloplatformowość. Niestety nie jest to rozwiązanie wolne od wad. Podczas procesu dopisywania nowych danych SQLite musi zablokować cały plik, aż do czasu zakończenia operacji. Zatem rozwiązanie to nie jest wydajne w sytuacjach gdy dane podlegają ciągłym zmianom. Drugim poważnym minusem jest brak pojęcia praw dostępu do bazy danych(znanym chociażby z MySQL'a), przez co nie możemy stworzyć „bezpiecznej" tabeli do której dostęp mieliby tylko uprawnieni użytkownicy. W skrócie oznacza to, że musimy sami stworzyć system kontroli dostępu, poprzez nadawanie praw do zapisu i odczytu odpowiednim plikom. Rozwiązanie takie nie jest wygodne. Podsumowując SQLite nie nadaje się do zastosowań na witrynach gdzie dane podlegają ciągłym zmianom, a także tam gdzie jest wymagany system kontroli dostępu. Jednak na małych i średnich stronach rozwiązanie to może okazać się naprawdę wydajne. W końcu do stworzenia tabeli z kilkunastoma rekordami nie potrzeba całego „kombajna" jakim jest niewątpliwie MySQL.

Relacyjna baza danych jest zbiorem danych zorganizowanych w tablicach. Pomiędzy tabelami mogą istnieć relacje. Tabele są formalnie opisane/ Tabele składają się z wierszy i kolumn. SQL (Structured Query Language) jest językiem zapytań do bazy danych przeznaczony do zarządzania danymi w relacyjnych systemach zarządzania bazami danych. Tabela jest zbiorem wartości, które są zorganizowane za pomocą modelu pionowych kolumn i poziomych wierszy. Kolumny są identyfikowane przez ich nazwy. Schemat systemu bazy danych i jego struktura są opisana w oficjalnym języku. Określa on tabele, pola, relacje, widoki, indeksy, procedury, funkcjie, kolejki, wyzwalacze i inne elementy. Wiersz bazy danych reprezentuje pojedynczy, pośrednio zorganizowany element danych w tabeli. Jest również nazywany krotką lub rekordem. Kolumna jest zestawem wartości danych konkretnego typu, po jednym dla każdego wiersza tabeli. Kolumny zapewniają strukturę, na którą składają się wiersze. Pole jest to pojedynczy element, który istnieje na przecięciu między wierszem i kolumną. Klucz główny jednoznacznie identyfikuje każdy rekord w tabeli. Klucz obcy służy do połączenia więzów między dwiema tabelami. Klucz obcy identyfikuje kolumnę lub zestaw kolumn w jednej (odnośników) tabeli, która odwołuje się do kolumny lub zestawu kolumn w innej (sygnatura) tabeli. Wyzwalany jest proceduralny kod, który jest wykonywany automatycznie w odpowiedzi na określone zdarzenia na konkretnej tabeli w bazie danych. Widok jest to specyficzne spojrzenie na dane w jednej lub kilku tabelach. To zapytanie może zorganizować dane w pewnym określonej kolejności, lub ukryć niektóre dane. Widok składa się z przechowywanego zapytania dostępnego w wirtualnej tabeli składającej się z zestawu wyników kwerendy. W przeciwieństwie do zwykłych tabel widok nie jest częścią fizycznego schematu bazy danych. Jest dynamiczną, wirtualną tabela zawierającą obliczenia lub zestaw z danymi w bazie danych. Transakcja jest autonomiczną jednostką operującą na danych w jednej lub kilku baz danych. Wyniki wszystkich instrukcji SQL w transakcjach mogą być zapisane w bazie danych i mogą być ponownie wykonane. Zestaw wyników SQL jest zbiorem wierszy z bazy danych, zwróconych przez instrukcję SELECT. Zawiera także meta-informacje o zapytaniu, takie jak nazwy kolumn, a także rodzaj i rozmiary każdej kolumny. Indeks jest strukturą danych, która zwiększa szybkość operacji pobierania danych z tabeli bazy danych.

Narzędzie sqlite3

Biblioteka SQLite zawiera proste narzędzie wiersza poleceń o nazwie sqlite3 (sqlite3.exe dla systemów Windows), które pozwala użytkownikowi ręcznie wpisywać i wykonywać polecenia SQL na bazie danych SQLite. Na przykład, aby utworzyć nową bazę danych SQLite o nazwie "testDB" składającą się z jednej tabeli o nazwie "TBL1", możesz to zrobić w następujący sposób:

1. uruchamiamy program sqlite3 poprzez linię poleceń Windows, z parametrem oznaczającym nazwę bazy danych. Jeśli baza nie istnieje to jest tworzona, a gdy już istnieje jest otwierana do edycji.

```
Wiersz polecenia - sqlite3.exe  testDB.db

C:\Users\Robert\Desktop\sqllite3>sqlite3.exe testDB.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite>
```

2. Tworzymy tabelę:

```
Wiersz polecenia - sqlite3.exe  testDB.db

C:\Users\Robert\Desktop\sqllite3>sqlite3.exe testDB.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite> CREATE TABLE tbl1 (one varchar(10), two smallint);
sqlite>
```

3. Dopisujemy przykładowe dane do tabeli:

```
Wiersz polecenia - sqlite3.exe  testDB.db

C:\Users\Robert\Desktop\sqllite3>sqlite3.exe testDB.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite> CREATE TABLE tbl1 (one varchar(10), two smallint);
sqlite> INSERT INTO tbl1 VALUES ('cześć',10);
sqlite> INSERT INTO tbl1 VALUES ('żegnaj',10);
sqlite>
```

4. Wyświetlamy wprowadzone wcześniej dane z tabeli:

```
Wiersz polecenia - sqlite3.exe  testDB.db

C:\Users\Robert\Desktop\sqllite3>sqlite3.exe testDB.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
sqlite> CREATE TABLE tbl1 (one varchar(10), two smallint);
sqlite> INSERT INTO tbl1 VALUES ('cześć',10);
sqlite> INSERT INTO tbl1 VALUES ('żegnaj',10);
sqlite> SELECT * FROM tbl1;
cześć|10
żegnaj|10
sqlite>
```

Tworzymy bazę danych

1. Tworzymy bazę filmy.db
2. W bazie filmy.db tworzymy trzy tabele Aktorzy, Filmy, AktorzyWFilmach i wypełniamy je danymi:

```
BEGIN TRANSACTION;
CREATE TABLE Aktorzy(AId integer primary key autoincrement, Name text);
INSERT INTO Aktorzy VALUES(1,'Philip Seymour Hofman');
INSERT INTO Aktorzy VALUES(2,'Kate Shindle');
INSERT INTO Aktorzy VALUES(3,'Kelci Stephenson');
INSERT INTO Aktorzy VALUES(4,'Al Pacino');
INSERT INTO Aktorzy VALUES(5,'Gabrielle Anwar');
INSERT INTO Aktorzy VALUES(6,'Patricia Arquette');
INSERT INTO Aktorzy VALUES(7,'Gabriel Byrne');
INSERT INTO Aktorzy VALUES(8,'Max von Sydow');
INSERT INTO Aktorzy VALUES(9,'Ellen Burstyn');
INSERT INTO Aktorzy VALUES(10,'Jason Miller');
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE Filmy(MId integer primary key autoincrement, Title text);
INSERT INTO Filmy VALUES(1,'Capote');
INSERT INTO Filmy VALUES(2,'Scent of a woman');
INSERT INTO Filmy VALUES(3,'Stigmata');
INSERT INTO Filmy VALUES(4,'Exorcist');
INSERT INTO Filmy VALUES(5,'Hamsun');
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE AktorzyWFilmach(Id integer primary key autoincrement,
                        AId integer, MId integer);
INSERT INTO AktorzyWFilmach VALUES(1,1,1);
INSERT INTO AktorzyWFilmach VALUES(2,2,1);
INSERT INTO AktorzyWFilmach VALUES(3,3,1);
INSERT INTO AktorzyWFilmach VALUES(4,4,2);
INSERT INTO AktorzyWFilmach VALUES(5,5,2);
INSERT INTO AktorzyWFilmach VALUES(6,6,3);
INSERT INTO AktorzyWFilmach VALUES(7,7,3);
INSERT INTO AktorzyWFilmach VALUES(8,8,4);
INSERT INTO AktorzyWFilmach VALUES(9,9,4);
INSERT INTO AktorzyWFilmach VALUES(10,10,4);
INSERT INTO AktorzyWFilmach VALUES(11,8,5);
COMMIT;
```

3. Tworzymy drugą bazę do ćwiczeń test.db
4. W bazie test.db tworzymy tabele: Samochody, Zamówienia, Znajomi, Klienci, Rezerwacje, Nazwiska, Książki

```
BEGIN TRANSACTION;
CREATE TABLE Samochody(Id integer PRIMARY KEY, Name text, Cost integer);
INSERT INTO Samochody VALUES(1,'Audi',52642);
INSERT INTO Samochody VALUES(2,'Mercedes',57127);
INSERT INTO Samochody VALUES(3,'Skoda',9000);
INSERT INTO Samochody VALUES(4,'Volvo',29000);
INSERT INTO Samochody VALUES(5,'Bentley',350000);
INSERT INTO Samochody VALUES(6,'Citroen',21000);
INSERT INTO Samochody VALUES(7,'Hummer',41400);
INSERT INTO Samochody VALUES(8,'Volkswagen',21600);
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE Zamowienia(Id integer PRIMARY KEY, OrderPrice integer
CHECK(OrderPrice>0),
                    Customer text);
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(1200, "Williamson");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(200, "Robertson");
```

```
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(40, "Robertson");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(1640, "Smith");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(100, "Robertson");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(50, "Williamson");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(150, "Smith");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(250, "Smith");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(840, "Brown");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(440, "Black");
INSERT INTO Zamowienia(OrderPrice, Customer) VALUES(20, "Brown");
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE Znajomi(Id integer PRIMARY KEY, Name text UNIQUE NOT NULL,
                     Sex text CHECK(Sex IN ('M', 'F')));
INSERT INTO Znajomi VALUES(1,'Jane', 'F');
INSERT INTO Znajomi VALUES(2,'Thomas', 'M');
INSERT INTO Znajomi VALUES(3,'Franklin', 'M');
INSERT INTO Znajomi VALUES(4,'Elisabeth', 'F');
INSERT INTO Znajomi VALUES(5,'Mary', 'F');
INSERT INTO Znajomi VALUES(6,'Lucy', 'F');
INSERT INTO Znajomi VALUES(7,'Jack', 'M');
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS Klienci(CustomerId integer PRIMARY KEY, Name text);
INSERT INTO Klienci(Name) VALUES('Paul Novak');
INSERT INTO Klienci(Name) VALUES('Terry Neils');
INSERT INTO Klienci(Name) VALUES('Jack Fonda');
INSERT INTO Klienci(Name) VALUES('Tom Willis');

CREATE TABLE IF NOT EXISTS Rezerwacje(Id integer PRIMARY KEY,
                                      CustomerId integer, Day text);
INSERT INTO Rezerwacje(CustomerId, Day) VALUES(1, '2009-22-11');
INSERT INTO Rezerwacje(CustomerId, Day) VALUES(2, '2009-28-11');
INSERT INTO Rezerwacje(CustomerId, Day) VALUES(2, '2009-29-11');
INSERT INTO Rezerwacje(CustomerId, Day) VALUES(1, '2009-29-11');
INSERT INTO Rezerwacje(CustomerId, Day) VALUES(3, '2009-02-12');
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE Nazwiska(Id integer, Name text);
INSERT INTO Nazwiska VALUES(1,'Tom');
INSERT INTO Nazwiska VALUES(2,'Lucy');
INSERT INTO Nazwiska VALUES(3,'Frank');
INSERT INTO Nazwiska VALUES(4,'Jane');
INSERT INTO Nazwiska VALUES(5,'Robert');
COMMIT;

BEGIN TRANSACTION;
CREATE TABLE Ksiazki(Id integer PRIMARY KEY, Title text, Author text,
                     Isbn text default 'not available');
INSERT INTO Ksiazki VALUES(1,'War and Peace','Leo Tolstoy','978-0345472403');
INSERT INTO Ksiazki VALUES(2,'The Brothers Karamazov',
                           'Fyodor Dostoyevsky','978-0486437910');
INSERT INTO Ksiazki VALUES(3,'Crime and Punishment',
                           'Fyodor Dostoyevsky','978-1840224306');
COMMIT;
```

5. Z utworzonej bazy danych test.db wyświetl tabele poleceniem: .tables (pamiętaj o kropce przed poleceniem)
6. Wyświetl zawartość tabeli nazwiska (Domyślnym trybem wyświetlania jest linia i separator |)

```
sqlite> SELECT * FROM Nazwiska;
```

7. Zmieńmy separator na „:”

```
sqlite> .separator :
sqlite> SELECT * FROM Nazwiska;
```

8. Istnieją jeszcze inne tryby wyświetlania danych. Poniżej dane są wyświetlone w trybie kolumnowym.

```
sqlite> .mode column
sqlite> .headers on
sqlite> SELECT * FROM Nazwiska;
```

9. Polecenie .width dostosowuje rozmiar kolumn:

```
sqlite> .width 23, 19
sqlite> SELECT Title, Author FROM Ksiazki;
```

10. Polecenie .show wyświetla różne ustawienia. Możemy sprawdzić w jakim trybie są wyświetlane dane, szerokość kolumn, jaki jest używany separator, czy jest wyświetlany nagłówek.
11. Polecenie .schema pokazuję strukturę tabeli. Wyświetla zapytanie DDL SQL, którym utworzyliśmy tablę.

```
sqlite> .schema Cars
```

12. Polecenie CREATE służy do tworzenia tabel. Jest również używane do tworzenia indeksów, widoków i wyzwalaczy. Aby utworzyć tabelę, możemy jej nadać nazwę i nazwę dla kolumn zawartych w tabeli. Każda kolumna może mieć jeden z definiowanych typów danych:

    - NULL - Wartość jest wartością NULL,
    - INTEGER - liczba całkowita,
    - REAL - wartość zmiennoprzecinkowa,
    - TEXT - ciąg tekstowy,
    - BLOB - typ danych, który umożliwia przechowywanie dużych ilości danych binarnych jako pojedynczy obiekt takich jak grafika, muzyka czy filmy.

    Tworzymy prostą tabelę o nazwie Testing korzystając z instrukcji CREATE TABLE. Polecenie **.schema** pokazuję formalną definicję struktury tabeli.

```
sqlite> CREATE TABLE Testing(id integer);
sqlite> .schema Testing
CREATE TABLE Testing(id integer);
```

13. Możemy zweryfikować istnienie tabeli Testing za pomocą polecenia .table Następna instrukcja DROP TABLE usuwa tabelę Testing z bazy danych:

```
sqlite> .table
Nazwiska    Testing  Samochody
sqlite> DROP TABLE Testing;
sqlite> .table
Nazwiska  Samochody
```

14. Polecenie ALTER TABLE w SQLite pozwala użytkownikowi zmieniać nazwę tabeli lub dodanie nowej kolumny do istniejącej tabeli. Nie jest możliwe, aby zmienić nazwę kolumny, usunąć kolumnę, albo dodać lub usunąć ograniczenia z tabeli.

```
sqlite>ALTER TABLE Nazwiska RENAME TO Przyjaciele
```

15. Dodajemy do tabeli przyjaciele nową kolumnę:

```
sqlite> ALTER TABLE Przyjaciele ADD COLUMN Email text;
```

Wyrażenia SQL

Wartości znakowe: wartość znakowa jest to stała pewnego rodzaju. Wartość znakowa może być typu integer, floating, string, BLOB lub NULL.

```
sqlite> SELECT 5, 'Andrzej', 44.5;
5|Andrzej|44.5
```

Powyżej mamy wartości znakowe mianowicie typu integer, string oraz floating.

```
sqlite> .nullvalue NULL
sqlite> SELECT NULL;
NULL
```

Komenda .nullvalue podpowiada SQLite aby pokazać wartość NULL jako NULL. SQLite pokazuje pusty ciąg znakowy domyślnie jako NULL.

```
sqlite> SELECT x'345eda2348587aeb';
4^?#HXz
```

Ciąg znaków typu BLOB zawiera dane w formacie danych szesnastkowych poprzedzonych jednym znakiem "x" lub "X".

Operatory

Operatory są wykorzystywane do budowania wyrażeń. Operatory SQL są bardzo podobne do operatorów matematycznych. Istnieją dwa rodzaje operatorów. Binarne i jednoargumentowe. Operatory binarne mogą pracować z dwoma operandami, operator jednoargumentowy może pracować z jednym.Operator może mieć jeden lub dwa argumenty.Operand jest jednym z wejść (argumenty) od operatora.

Mamy kilka rodzajów operatorów:

- operatory arytmetyczne,
- operatory logiczne,
- operatory relacyjne,
- operatory bitowe,
- inne operatory.

SQLite rozpoznaje następujące operatory binarne.

```
||
*    /    %
+    -
<<   <>   &    |
<    <=   >    >=
=    ==   !=   <>   IS   IN   LIKE   GLOB   BETWEEN
AND
OR
```

Operatory są ułożone według hierarchii ważności Operator || ma najwyższy priorytet, natomiast operator OR ma najniższy priorytet. Poniżej są przedstawione operatory jednoargumentowe:

```
-    +    ~    NOT
```

Operator jednoargumentowy zmienia wartości dodatnie na ujemne i na odwrót.

```
sqlite> SELECT -(3-44);
41
```

Wynikiem działania jest wynik 41. Pozostałe dwa operatory zostaną omówione później.

Operatory arytmetyczne

Operatory arytmetyczne rozumiane przez SQLite są to: mnożenie, dzielenie, dodawanie, odejmowanie i modulo.

```
sqlite> SELECT 3 * 3/9;
1
```

Są to działania mnożenia i dzielenia, które są znane jak wiemy z matematyki.

```
sqlite> SELECT 3 + 4 - 1 + 5;
11
```

Dodawanie i odejmowanie operatorów

```
sqlite> SELECT 11% 3;
2
```

Operator % nazywany jest operatorem modulo. Daje wynik w swoim działaniu, że reszta z dzielenia jednej liczby przez drugą. 11 % 3, 11 % 3 jest modulo 2, ponieważ 3 mieści się 3 trzykrotnie w 11 a reszta wynosi 2.

Operatory logiczne

Z operatorami logicznymi wykonujemy operacje logiczne. SQLite posiada następujące operatory logiczne I , LUB oraz NOT. Operatory logiczne zwracają wartości true lub false. W SQLite, 1 jest to wartość true, 0 jest to false.

I operator zwraca wartość true, jeśli oba operandy są prawdziwe.

```
sqlite> SELECT 0 AND 0, 0 AND 1, 1 AND 0, 1 AND 1;
0|0|0|1
```

Pierwsze trzy operacje dają dają w wyniku wartość false, a ostatni wartość true.

```
sqlite> SELECT 3=3 AND 4=4;
1
```

Oba argumenty są prawdziwe, więc wynik jest prawdziwy (true) (1).

Operator OR jest prawdziwy, jeżeli co najmniej jeden z argumentów jest prawdziwy.

```
sqlite> SELECT 0 OR 0, 0 OR 1, 1 OR 0, 1 OR 1;
0|1|1|1
```

Pierwsza operacja ma wartość false, następne działania maja wartość true.

Operator NOT jest operatorem negacji. To sprawia, że wartość true zmienia się na false, a false na true.

```
sqlite> SELECT NOT 1, NOT 0;
0|1
sqlite> SELECT NOT (3=3);
0
```

Operatory relacyjne

Operatory relacyjne służą do porównywania wartości. Te operatory zawsze skutkują wartością logiczną.

```
sqlite> SELECT 3*3 == 9, 9 = 9;
1|1
```

Zarówno = i == są to operatory równości.

```
sqlite> SELECT 3 < 4, 3 <> 5, 4 >= 4, 5 != 5;
1|1|1|0
```

Użycie operatorów relacyjnych znane jest z matematyki.

Operatory bitowe

Liczby dziesiętne są naturalnymi dla ludzi. Liczby dwójkowe są naturalnymi liczbami dla komputerów. Binarne, ósemkowe, dziesiętne lub szesnastkowe symbole są to tylko zapisy o tym samym numerze. Operatory bitowe pracują z bitami liczby binarnej. Mamy binarne operatory logiczne i operatory przesunięcia. Operator bitowy i(and) wykonuje porównanie bit-by-bit pomiędzy dwoma liczbami. Wynik na pozycji wynosi 1 tylko wtedy jeśli obydwa bity mają wartość 1.

00110 & 00011 = 00010

Pierwsza cyfra to dwójkowo 6. Druga jest 3. Wynik jest 2.

```
sqlite> SELECT 6 & 3;
2
sqlite> SELECT 3 & 6;
2
```

Operator bitowy lub(or) wykonuje porównanie bit-by-bit pomiędzy dwoma liczbami. Wynik na pozycji wynosi 1 jeśli jeden z bitów mają wartość 1.

00110 | 00011 = 00111

Wynikiem jest 00110 czyli dzisiętnie 7

```
sqlite> SELECT 6 | 3;
7
```

Inne operatory

Istnieją jeszcze niektóre inne operatory. Obejmują one frazy **IN, LIKE, GLOB, BETWEEN.**

```
sqlite> SELECT 'wolf' || 'hound';
wolfhound
```

Operator || jest operatorem konkatenacji ciągów znaków. Łączy po prostu ciągi. Możemy zastosować operator IN w dwóch przypadkach.

```
sqlite> SELECT 'Tom' IN ('Tom', 'Frank', 'Jane');
1
```

Tutaj sprawdzamy, jeśli wartość ciągu ‘Tom’ jest na liście nazw, po operatorze IN. To wartość zwracana jest to wartość logiczna. Na poniższym przykładzie podsumujmy, co mamy w tabeli Samochody.

```
sqlite> SELECT * FROM Samochody;
Id          Name        Cost
----------  ----------  ----------
1           Audi        52642
2           Mercedes    57127
3           Skoda       9000
4           Volvo       29000
5           Bentley     350000
6           Citroen     21000
7           Hummer      41400
8           Volkswagen  21600
```

W drugim przypadku operator IN pozwala na określenie wielu wartości w klauzuli WHERE.

```
sqlite> SELECT * FROM Cars WHERE Name IN ('Audi', 'Hummer');
Id          Name        Cost
----------  ----------  ----------
1           Audi        52642
7           Hummer      41400
```

Z tabeli Cars wybieramy samochody, które są wymienione po operatorze IN.

Operator LIKE jest używany w klauzuli WHERE, aby wyszukać określony wzorzec w kolumnie.

```
sqlite>  SELECT * FROM Samochody WHERE Name LIKE 'Vol%';
Id          Name        Cost
----------  ----------  ----------
4           Volvo       29000
8           Volkswagen  21600
```

W tym miejscu możemy wybrać samochody, których nazwy zaczynają się od "Vol.".

```
sqlite> SELECT * FROM Samochody WHERE Name LIKE '____';
Id          Name        Cost
----------  ----------  ----------
1           Audi        52642
```

W tym miejscu możemy wybrać nazwę samochodów, która składa się z dokładnie czterech znaków. Istnieją cztery znaki podkreślenia. Operator GLOB jest podobny do LIKE, ale używa składni globbing pliku Unix dla jego symboli wieloznacznych. Ponadto w GLOB jest uwzględniana wielkość liter, w przeciwieństwie do LIKE.

```
sqlite> SELECT * FROM Samochody WHERE Name GLOB '*en';
Id          Name        Cost
----------  ----------  ----------
6           Citroen     21000
8           Volkswagen  21600
```

Mamy tutaj samochody, których nazwy kończą się znakami 'en'.

```
sqlite> SELECT * FROM Samochody WHERE Name GLOB '????';
Id          Name        Cost
----------  ----------  ----------
1           Audi        52642
```

W tym miejscu możemy ponownie wybrać nazwę samochodów, która składa się z dokładnie czterech znaków.

```
sqlite> SELECT * FROM Cars WHERE Name GLOB '*EN';
sqlite> SELECT * FROM Cars WHERE Name LIKE '%EN';
Id          Name        Cost
----------  ----------  ----------
6           Citroen     21000
8           Volkswagen  21600
```

Te dwa wyrażenia demonstrują, że operator LIKE rozróżniana wielkość znaków, w GLOB jest uwzględniana wielkość liter.

Operator BETWEEN jest odpowiednikiem pary porównań. BETWEEN b i c jest odpowiednikiem a>=b AND a<=c

```
sqlite> SELECT * FROM Samochody WHERE Cost BETWEEN 20000 AND 55000;
Id          Name        Cost
----------  ----------  ----------
1           Audi        52642
4           Volvo       29000
6           Citroen     21000
7           Hummer      41400
8           Volkswagen  21600
```

W tej instrukcji SQL wybraliśmy samochody, które kosztują pomiędzy 20000 i 55000 jednostkami.