W tym odcinku po raz pierwszy potraktujemy tabele jako zbiory danych. Poznasz język SQL i nauczysz się odczytywać, wybierać i sortować dane zapisane w pojedynczych tabelach.

Wprowadzenie

Podstawowe obiekty relacyjnych baz danych — tabele — i główny język programowania tych baz — SQL — wymagają krótkiego przedstawienia.

Tabele jako zbiory danych

Podstawową cechą relacyjnych baz danych jest to, że przechowywane informacje są dostępne jako zbiór dwuwymiarowych tabel. Czyli niezależnie od tego, w jakiej postaci (z reguły jest to postać binarna) serwer baz danych przechowuje dane na dysku i w pamięci, użytkownicy widzą je w postaci tabelarycznej.

Każda tabela musi mieć unikatową w skali schematu (w przypadku MySQL-a schemat i baza są tym samym) nazwę. Nazwę obiektu bazodanowego można poprzedzić nazwą schematu, w którym ten obiekt się znajduje, tak więc pełną nazwą tabeli customer będzie test.customer.

Każda tabela składa się z kolumn i wierszy. Poszczególne kolumny muszą mieć określoną nazwę, a w każdej z nich można przechowywać dane określonego typu. Na przykład w kolumnie fname tabeli customer można zapisać do 32 znaków, w kolumnie title przechowywane są ciągi o długości dokładnie 4 znaków — jeżeli tytuł jakiegoś klienta jest krótszy, zostanie uzupełniony spacjami. Natomiast poszczególne wiersze zawierają dane opisujące określony obiekt. Na przykład jeden wiersz tabeli customer przechowuje imię, nazwisko, tytuł, adres, numer telefonu i identyfikator klienta.

Zauważ, że liczba kolumn tabeli jest stała, natomiast liczba wierszy zmienia się — odpowiada bowiem liczbie przechowywanych w niej obiektów. Zwróć również uwagę, że w poszczególnych kolumnach znajdują się zawsze dane tego samego typu (na przykład w pierwszej kolumnie są tylko nazwiska, a w ostatniej jedynie adresy e-mail). Z kolei w poszczególnych wierszach znajduje się komplet informacji o konkretnych obiektach. (Wiersz nazywany jest również rekordem, a pojedyncza komórka tabeli — polem). Kolejną ważną cechą tabeli jest to, że każdy jej rekord składa się z takiej samej liczby pól. Żaden wiersz tabeli nie może być krótszy lub dłuższy od innych, niedopuszczalne są również przerwy pomiędzy komórkami.

Tabele w relacyjnej bazie danych mają następujące właściwości:

1. Nazwy kolumn muszą być unikatowe w skali tabeli — próba utworzenia tabeli z kilkoma tak samo nazywającymi się kolumnami zakończy się błędem.
2. Kolejność wierszy jest nieokreślona i nieistotna — to, że informacje o kliencie Jenny Stones znajdują się w pierwszym, a nie na przykład trzecim wierszu, nie ma żadnego znaczenia.
3. Kolejność kolumn jest nieokreślona, ale ma wpływ na sposób prezentowania danych — gdyby dane o numerze telefonu znajdowały się w pierwszej, a nie ostatniej kolumnie, dalej byłyby to te same dane, ale zapytania mogłyby uwzględnić zmienioną kolejność kolumn.
4. Wiersze w tabeli muszą być różne — gdybyśmy na przykład chcieli wprowadzić drugi raz informację o tym samym kliencie, musielibyśmy dodać kolejny wiersz. W matematycznym modelu relacyjnych baz danych przyjmuje się, że dwa identyczne elementy zbioru są tak naprawdę jednym elementem, natomiast w tabeli identyczne dane przechowywane są w różnych wierszach.

Standardy SQL i historia ich powstania

Będący logiczną podstawą języka SQL model relacyjny baz danych został formalnie zdefiniowany w 1970 r. przez dr. E.F. Codda, naukowca firmy IBM, w dokumencie zatytułowanym Relacyjny model danych dla dużych współdzielonych banków danych (ang. A Relational Model of Data for Large Shared Data Banks). Firma IBM była wtedy ogromną korporacją prowadzącą badania nad projektami, których praktycznego zastosowania spodziewano się doczekać dopiero po kilkunastu latach.

IBM nie był jedyną firmą prowadzącą w tym czasie badania nad modelem relacyjnym — równolegle powstawał projekt Ingres opracowywany przez naukowców Uniwersytetu Kalifornijskiego w Berkeley, a w 1979 roku pojawił się pierwszy system zarządzania bazami danych wyprodukowany przez mało wtedy znaną firmę Relational Software (która zmieniła potem nazwę na Oracle). Sam IBM przygotował dwa komercyjne systemy bazodanowe, oba będące rozwinięciem projektu System/R — SQL/DS (wprowadzony w 1981 r.) oraz DB2 (rok 1983). Pozostałe firmy, zachęcone wyraźnym znakiem poparcia modelu relacyjnego przez potentata, zaczęły rozwijać własne implementacje modelu. W ciągu kilkunastu lat doczekaliśmy się co najmniej pięciu głównych implementacji języka SQL, które, choć oparte na tym samym modelu logicznym, znacznie się od siebie różnią.

SQL oparty jest na algebrze relacji i rachunku relacyjnym krotki. Rozwinięta przez dr. Codda algebra relacji dostarczyła reguł przetwarzania składni instrukcji języka (odpowiada na pytanie: „Jak wykonać instrukcje?"). Natomiast rachunek relacyjny krotki (ang. Tuple Relational Calculus — TRC), opisując deklaracje języka, odpowiedział na pytanie: „Co należy zrobić?".

Zaletą języka SQL jest przede wszystkim upraszczanie pracy z relacyjnymi bazami danych — zamiast krok po kroku określać, jak serwer bazodanowy ma wykonać dane polecenie, użytkownik deklaruje, w języku przypominającym potoczny angielski, spodziewany wynik. Zadaniem serwera bazodanowego jest zinterpretowanie i wykonanie polecenia. Język SQL umożliwia również wydajne przetwarzanie dużych ilości danych — tak dużych, że w większości przypadków niemieszczących się w pamięci operacyjnej komputera i z tego powodu w razie potrzeby odczytywanych i zapisywanych przez serwer na dyskach twardych.

Oprócz właściwości wynikających z algebry lub rachunku relacyjnego język SQL zawiera dodatkowe funkcje, takie jak:

1. obsługa wstawiania, modyfikacji i usuwania danych,
2. operatory arytmetyczne,
3. wyświetlanie danych,
4. przypisanie (aliasy),
5. funkcje grupujące,
6. i wiele innych.

Pierwszą próbą uporządkowania i ujednolicenia implementacji języka SQL w poszczególnych serwerach bazodanowych był standard opublikowany w roku 1986 przez Amerykański Narodowy Instytut Standardów (ang. American National Standards Institute — ANSI). Kolejną był dokument normalizacyjny wydany rok później przez Międzynarodową Organizację Standardów. Na podstawie tych dokumentów w 1989 r. opublikowano zweryfikowany standard, znany powszechnie jako SQL-89 lub SQL-1.

Niestety, standard ten nie określał wielu podstawowych cech języka, a wiele właściwości zdefiniowano jako zależne od implementacji. Kolejną próbą ujednolicenia była ratyfikacja (w 1992 r.) standardu SQL-92 (SQL-2). Niezbędnym kompromisem okazało się wprowadzenie trzech poziomów zgodności z nowym standardem:

1. Podstawowy poziom zgodności (ang. Entry-level conformance) był właściwie powtórzeniem standardu SQL-1.

2. Pośredni poziom zgodności (ang. Intermediate-level conformance) stanowił ogólnie osiągalny zbiór zasadniczych ujednoliceń języka.

3. Pełny poziom zgodności (ang. Full conformance) — oferował pełną zgodność z właściwościami standardu SQL-2.

Najlepiej znanym i najpowszechniej stosowanym elementem standardu SQL-2 był podział instrukcji na trzy kategorie:

1. Instrukcje DDL (ang. Data Definition Language — język definiowania danych) służące do tworzenia, modyfikowania i usuwania obiektów. Do tej kategorii należały instrukcje CREATE, ALTER i DROP.

2. Instrukcje DML (ang. Data Manipulation Language — język modyfikowania danych) pozwalające odczytywać i modyfikować dane. Do tej kategorii zostały zaklasyfikowane instrukcje SELECT, INSERT, UPDATE i DELETE.

3. Instrukcje DCL (ang. Data Control Language — język kontroli dostępu do danych) umożliwiające nadawanie i odbieranie uprawnień użytkownikom. Do tej kategorii należały instrukcje GRANT i REVOKE.

W 1999 r. organizacje ANSI i ISO opracowały standard SQL-99 (nazywany również SQL-3). Dokument ten po raz pierwszy ujednolica zaawansowane funkcje i obszary zastosowań języka SQL, takie jak modele obiektowo-relacyjnych baz danych, interfejsy poziomu wywołania czy instrukcje umożliwiające zarządzanie spójnością danych. SQL-3 oferuje dwa poziomy zgodności:

1. Podstawowy poziom zgodności (ang. Core SQL-3).

2. Zaawansowany poziom zgodności (ang. Enhanced SQL-3).

Cztery lata później, w roku 2003, przyjęto czwarty standard języka SQL. Był on rozszerzeniem SQL-3 o takie elementy, jak:

1. obsługa danych typu XML,

2. funkcje rankingu,

3. instrukcja MERGE,

4. jednolity mechanizm generowania wartości, np. identyfikatorów wierszy.

W 2006 roku przyjęty został najnowszy standard języka SQL, prawie w całości poświęcony obsłudze dokumentów XML przez serwery bazodanowe. Definiuje on m.in. sposoby przechowywania danych XML w tabelach oraz przeszukiwania i modyfikowania zawartości dokumentów XML za pomocą języka XQuery. **Składnia języka SQL**

W języku SQL występuje pięć głównych kategorii syntaktycznych (pięć zbiorów znaczników języka):

1. Identyfikatory, czyli nazwy obiektów.

2. Literały, czyli stałe. (Wszystkie cyfry, ciągi znaków i daty, jeżeli nie są identyfikatorami, są traktowane jako stałe, czyli literały. W języku SQL ciągi znaków umieszcza się w apostrofach).

3. Operatory, czyli spójniki. (Operatory odgrywają rolę spójników. Niektóre z nich mogą — ale nie powinny, bo instrukcje z nimi są czytelniejsze — być zastąpione odpowiednimi funkcjami. Operatory dzielą się arytmetyczne, znakowe, logiczne, porównania i dodatkowe, charakterystyczne dla języka SQL).

4. Słowa kluczowe, czyli wyrazy interpretowane przez serwer bazodanowy w określony sposób. (Słowa kluczowe to zastrzeżone, mające ściśle określone znaczenie ciągi znaków. Należą do nich instrukcje i klauzule języka SQL, nazwy typów danych, nazwy funkcji systemowych oraz terminy zarezerwowane dla przyszłego użycia w danym serwerze bazodanowym).

5. Ignorowane przez serwery bazodanowe komentarze. (W serwerze MySQL znaki /* oznaczają początek bloku komentarza, a znaki */ jego koniec. Wiersze znajdujące się pomiędzy tymi znakami są traktowane jako komentarz).

Instrukcją języka SQL jest każdy poprawny syntaktycznie (składniowo) zbiór znaczników tego języka. Instrukcje SQL zawsze zaczynają się poleceniem (znacznikiem lub grupą znaczników określających, jaką operację wykona

instrukcja) i zwykle zawierają przynajmniej jedną klauzulę; klauzule są formalnymi modyfikatorami opisującymi bardziej szczegółowo funkcję instrukcji SQL.

W ramach kursu (ze względu na czytelność) przyjąłem zasadę zapisywania poszczególnych klauzul w nowych wierszach. W rzeczywistości w języku SQL znak końca wiersza jest ignorowany i ta sama instrukcja może być zapisana w jednym albo w dziesięciu wierszach.

Zapytaniem jest instrukcja, która zwraca dane (na przykład instrukcja SELECT). Dane zwrócone przez zapytanie są wynikiem działania instrukcji.

Pobieranie danych

Informacje przechowywane w bazach danych mogą być pobrane za pomocą instrukcji języka SQL SELECT. Instrukcja SELECT (zapytanie) określa, jakie dane mają zostać zwrócone w wyniku jej wykonania, natomiast to, w jaki sposób instrukcja będzie wykonana, zależy od serwera baz danych.

Język SQL jest językiem strukturalnym, a nie proceduralnym. W języku tym nie określamy sposobu wykonania zadania (tak jak np. w C), ale jego wynik (a właściwie wymagane do jego otrzymania operacje na pewnych strukturach — zbiorach). Z założenia instrukcje języka SQL przypominają potoczny język angielski. Żeby na przykład odczytać nazwę i cenę produktu, nie napiszemy programu za pomocą jakiegoś algorytmu wyszukiwania znajdującego odpowiednie informacje, ale po prostu powiemy (napiszemy): Odczytaj nazwę i cenę produktu o podanym identyfikatorze.

Instrukcja SELECT służy do pobierania danych z bazy. Instrukcja musi zawierać (z wyjątkiem polecenia SELECT odwołującego się wyłącznie do stałych, zmiennych lub wyrażeń arytmetycznych) co najmniej jedną klauzulę: za pomocą polecenia SELECT określamy interesujące nas kolumny (dokonujemy operacji selekcji pionowej, projekcji), za pomocą klauzuli FROM wskazujemy tabelę, z której pobieramy dane. Z reguły ogranicza się również, za pomocą klauzuli WHERE, liczbę zwracanych wierszy do rekordów spełniających określone kryteria (operacja selekcji poziomej, selekcji).

Odczytujemy wszystkie dane z tabeli

Najprostszy przykład użycia instrukcji SELECT to odczytanie całej zawartości wskazanej tabeli. W SQL-u możemy posługiwać się kilkoma znakami specjalnymi; jednym z nich jest *, oznaczająca „wszystko”. Czyli żeby odczytać wszystko z tabeli customer, należy wykonać instrukcję:

```
SELECT *

FROM customer;
```

albo:

```
SELECT *

FROM test.customer;
```

Odradzam jednak używanie symbolu * — stosując go, narażamy się na otrzymanie błędnych, innych niż się spodziewaliśmy, wyników. Sytuacja taka będzie miała miejsce, gdy ktoś zmieni kolejność lub liczbę kolumn tabeli (np. poprzez dodanie lub usunięcie kolumny), do której odwołujemy się w klauzuli FROM. Ponadto, używając symbolu *, zmuszamy serwer bazodanowy do odczytania wszystkich kolumn tabeli, co może wielokrotnie wydłużyć czas wykonywania zapytania. Jeżeli interesują nas dane tylko z niektórych kolumn, nie powinniśmy odczytywać całej tabeli.

Wszystkie instrukcje SQL należy wykonywać w wyniku łączenia się jako administrator testowej bazy za pomocą programu MySQL Query Browser albo tekstowego programu mysql. W drugim przypadku po połączeniu z serwerem trzeba określić bazę danych przez wpisanie USE test;.

Kolejność kolumn wyniku zawsze odpowiada kolejności kolumn tabeli. Nie należy zakładać, że ponowne wykonanie tej samej instrukcji zwróci wiersze w tej samej kolejności. **Wybieramy kolumny z tabel**

Wiemy już, że instrukcja SELECT służy do pobierania danych z bazy i że z reguły zawiera ona co najmniej dwie klauzule: w klauzuli SELECT określamy interesujące nas kolumny, w klauzuli FROM wskazujemy tabelę, z której pobieramy dane (listing 3.1).

Listing 3.1. Zapytanie zwracające imiona i nazwiska klientów

```
SELECT fname, lname

FROM customer;
```

Nazwy kolumn wymienionych w klauzuli SELECT należy oddzielić przecinkami.

W efekcie uzyskamy wycinek (projekcję) tabeli z informacjami o imionach i nazwiskach wszystkich klientów:

```
+-----------+---------+

| fname | lname |

+-----------+---------+

| Jenny | Stones |

| Andrew | Stones |

| Alex | Matthew |

| Adrian | Matthew |

| Simon | Cozens |

| Neil | Matthew |

| Richard | Stones |

| Ann | Stones |

| Christine | Hickman |

| Mike | Howard |

| Dave | Jones |

| Richard | Neill |

| Laura | Hendy |

| Bill | Neill |

| David | Hudson |

| | Wolski |
```

+----------+---------+

Kolejność podawanych nazw kolumn nie jest obojętna. Serwer baz danych zwróci dane, szeregując poszczególne kolumny według kolejności ich występowania w klauzuli SELECT. Czyli wynik wykonania poniższej instrukcji będzie różny od poprzedniego (listing 3.2).

Listing 3.2. Lista nazwisk i imion klientów

```
SELECT lname, fname

FROM customer;
```

+---------+----------+

| lname | fname |

+---------+----------+

| Stones | Jenny |

| Stones | Andrew |

| Matthew | Alex |

| Matthew | Adrian |

| Cozens | Simon |

| Matthew | Neil |

| Stones | Richard |

| Stones | Ann |

| Hickman | Christine |

| Howard | Mike |

| Jones | Dave |

| Neill | Richard |

| Hendy | Laura |

| Neill | Bill |

| Hudson | David |

| Wolski | |

+---------+----------+

Oprócz nazw kolumn język SQL dopuszcza używanie w klauzuli SELECT wyrażeń, aliasów oraz literałów. Dodatkowo możliwe jest łączenie (konkatenacja) kolumn. Kolejne punkty przedstawiają sposoby wykorzystania tych możliwości.

Wyrażenia

W skład wyrażenia mogą wchodzić nazwy kolumn, stałe, wartości liczbowe i ogólnie znane operatory, takie jak: + (dodawania), – (odejmowania), * (mnożenia), / (dzielenia) (listing 3.3).

Listing 3.3. Prosty przykład zastosowania funkcji arytmetycznych — zwróć uwagę, że ta sama kolumna tabeli może być wielokrotnie zwrócona przez zapytanie

```
SELECT cost_price,cost_price*0.22

FROM item;

+------------+-----------------+

| cost_price | cost_price*0.22 |

+------------+-----------------+

| 15.23 | 3.3506 |

| 7.45 | 1.6390 |

| 1.99 | 0.4378 |

| 2.11 | 0.4642 |

| 7.54 | 1.6588 |

| 9.23 | 2.0306 |

| 13.36 | 2.9392 |

| 0.75 | 0.1650 |

| 2.34 | 0.5148 |

| 0.01 | 0.0022 |

| 19.73 | 4.3406 |

| NULL | NULL |

+------------+-----------------+
```

Stosując funkcje arytmetyczne, możemy łatwo sprawdzić, z jaką marżą sprzedawane są poszczególne towary. Ponieważ oprócz informacji o cenie zakupu (umieszczonej w kolumnie cost_price) przechowujemy (w kolumnie sell_price) informacje o cenie sprzedaży, wystarczy odjąć od ceny sprzedaży cenę zakupu każdego towaru. Gotowe rozwiązanie może wyglądać tak jak na listingu 3.4.

Listing 3.4. Lista cen i marż towarów

```
SELECT cost_price, sell_price,sell_price-cost_price

FROM item;
```

```
+------------+------------+----------------------+

| cost_price | sell_price |  sell_price-cost_price |

+------------+------------+----------------------+

| 15.23 | 21.95 | 6.72 |

| 7.45 | 11.49 | 4.04 |

| 1.99 | 2.49 | 0.50 |

| 2.11 | 3.99 | 1.88 |

| 7.54 | 9.95 | 2.41 |

| 9.23 | 15.75 | 6.52 |

| 13.36 | 19.95 | 6.59 |

| 0.75 | 1.45 | 0.70 |

| 2.34 | 2.45 | 0.11 |

| 0.01 | 0.00 | -0.01 |

| 19.73 | 25.32 | 5.59 |

| NULL | NULL | NULL |

+------------+------------+----------------------+
```

Serwer baz danych wykonuje poszczególne operacje arytmetyczne w takiej samej kolejności, jaka obowiązuje w klasycznych działaniach matematycznych, tj. jako pierwsze wykonywane są wyrażenia umieszczone w nawiasach, następnie mnożenie, dzielenie, dodawanie i odejmowanie. Tak więc dwie przedstawione poniżej instrukcje nie są równoważne (listing 3.5).

Listing 3.5. Kolejność wyrażeń w klauzuli SELECT określa kolejność kolumn wyniku zapytania

```
SELECT cost_price,  sell_price,sell_price-cost_price*0.22

FROM item;

+------------+------------+---------------------------+

| cost_price | sell_price |  sell_price-cost_price*0.22 |

+------------+------------+---------------------------+

| 15.23 | 21.95 | 18.5994 |

| 7.45 | 11.49 | 9.8510 |

| 1.99 | 2.49 | 2.0522 |
```

| 2.11 | 3.99 | 3.5258 |

| 7.54 | 9.95 | 8.2912 |

| 9.23 | 15.75 | 13.7194 |

| 13.36 | 19.95 | 17.0108 |

| 0.75 | 1.45 | 1.2850 |

| 2.34 | 2.45 | 1.9352 |

| 0.01 | 0.00 | -0.0022 |

| 19.73 | 25.32 | 20.9794 |

| NULL | NULL | NULL |

+------------+------------+----------------------------+

SELECT cost_price, sell_price,(sell_price-cost_price)*0.22

FROM item;

+------------+------------+-----------------------------+

| cost_price | sell_price | (sell_price-cost_price)*0.22 |

+------------+------------+-----------------------------+

| 15.23 | 21.95 | 1.4784 |

| 7.45 | 11.49 | 0.8888 |

| 1.99 | 2.49 | 0.1100 |

| 2.11 | 3.99 | 0.4136 |

| 7.54 | 9.95 | 0.5302 |

| 9.23 | 15.75 | 1.4344 |

| 13.36 | 19.95 | 1.4498 |

| 0.75 | 1.45 | 0.1540 |

| 2.34 | 2.45 | 0.0242 |

| 0.01 | 0.00 | -0.0022 |

| 19.73 | 25.32 | 1.2298 |

| NULL | NULL | NULL |

+------------+------------+-----------------------------+ **Aliasy**

Język SQL umożliwia zastąpienie nazwami opisowymi domyślnych, utworzonych podczas tworzenia tabel, nagłówków kolumn. Jest to szczególnie przydatne w przypadku używania wyrażeń. Ponieważ wynik wyrażenia jest obliczany w chwili wykonania instrukcji SELECT, w tabeli nie znajduje się kolumna, w której byłyby przechowywane dane będące wynikiem operacji arytmetycznej. W związku z tym serwer baz danych „nie wie", jaka miałaby być domyślna nazwa nowej kolumny. Jak widzieliśmy, MySQL za nagłówek nowej kolumny przyjmie wyrażenie arytmetyczne. Aby utworzyć alias dla kolumny, należy bezpośrednio po nazwie, którą chcemy zastąpić, użyć słowa kluczowego AS, a następnie podać nową nazwę kolumny (listing 3.6).

Aliasy kolumn definiuje się w klauzuli SELECT albo podając je bezpośrednio po oryginalnej nazwie kolumny, albo poprzedzając je słowem kluczowym AS. Radzę zawsze używać słowa kluczowego AS — poprawia ono czytelność zapytań, dzięki czemu łatwiej można zauważyć brak przecinka pomiędzy nazwami dwóch odczytywanych kolumn.

Listing 3.6. Przykład użycia aliasów

```
SELECT cost_price AS 'Cena kupna', sell_price AS 'Cena sprzedaży' ,sell_price-cost_price AS Zysk

FROM item;

+------------+----------------+-------+

| Cena kupna | Cena sprzedaży | Zysk |

+------------+----------------+-------+

| 15.23 | 21.95 | 6.72 |

| 7.45 | 11.49 | 4.04 |

| 1.99 | 2.49 | 0.50 |

| 2.11 | 3.99 | 1.88 |

| 7.54 | 9.95 | 2.41 |

| 9.23 | 15.75 | 6.52 |

| 13.36 | 19.95 | 6.59 |

| 0.75 | 1.45 | 0.70 |

| 2.34 | 2.45 | 0.11 |

| 0.01 | 0.00 | -0.01 |

| 19.73 | 25.32 | 5.59 |

| NULL | NULL | NULL |

+------------+----------------+-------+
```

Słowo kluczowe AS może być opuszczone.

Zwróćmy uwagę, że dwa pierwsze aliasy zostały zapisane w apostrofach — to dlatego, że zawierają spację. Jeżeli nazwa jakiegoś obiektu jest niezgodna z konwencją języka SQL (na przykład zawiera spację), trzeba zapisywać ją w apostrofach.

Literały

Oprócz nazw kolumn, ich aliasów i wyrażeń w klauzuli SELECT można umieszczać literały. Literałem jest dowolny ciąg znaków lub liczb. W wyniku umieszczenia w niej literału serwer baz danych do każdego zwróconego wiersza wpisze do odpowiedniej kolumny treść literału. Na przykład do utworzenia raportu informującego o tym, gdzie mieszka dana osoba, można wykorzystać instrukcję pokazaną na listingu 3.7.

Listing 3.7. SQL jest językiem operowania na zbiorach. Dzięki temu dodanie do każdego wiersza wyniku stałej wartości sprowadza się do wymienienia jej w klauzuli SELECT

```
SELECT fname, lname,' mieszka w ', town

FROM customer;

+-----------+---------+-------------+-----------+

| fname | lname | mieszka w | town |

+-----------+---------+-------------+-----------+

| Jenny | Stones | mieszka w | Hightown |

| Andrew | Stones | mieszka w | Lowtown |

| Alex | Matthew | mieszka w | Nicetown |

| Adrian | Matthew | mieszka w | Yuleville |

| Simon | Cozens | mieszka w | Oahenham |

| Neil | Matthew | mieszka w | Nicetown |

| Richard | Stones | mieszka w | Bingham |

| Ann | Stones | mieszka w | Bingham |

| Christine | Hickman | mieszka w | Histon |

| Mike | Howard | mieszka w | Tibsville |

| Dave | Jones | mieszka w | Bingham |

| Richard | Neill | mieszka w | Winersby |

| Laura | Hendy | mieszka w | Oxbridge |

| Bill | Neill | mieszka w | Welltown |

| David | Hudson | mieszka w | Milltown |

| | Wolski | mieszka w | |
```

+----------+---------+-------------+----------+ **Łączenie ciągów znaków**

Podstawowym założeniem przy projektowaniu i tworzeniu relacyjnych baz danych jest wyodrębnienie i zapisywanie w odrębnych kolumnach danych elementarnych. Na przykład imiona i nazwiska klientów zamiast w jednej kolumnie, zostały zapisane w odrębnych kolumnach. Dokładniej wyjaśnię tę zasadę w odcinku poświęconym projektowaniu baz danych. W każdym razie zapisane w tabelach dane są podzielone na „atomowe” części, co oprócz wielu zalet (dane mogą być na przykład efektywnie wyszukiwane i sortowane na wiele sposobów) ma też jedną wadę — niektóre raporty (tak jak poprzedni) wyglądają dość dziwnie.

Jako generalną zasadę można przyjąć, że w tabeli powinny być przechowywane najmniejsze dane wykorzystywane do wybierania, sortowania lub obliczeń. Jeżeli użytkownika bazy danych może interesować np. zestawienie klientów według numerów ich kodów pocztowych, w tabeli powinna znaleźć się kolumna przechowująca te dane.

W efekcie często chcemy połączyć informacje przechowywane w odrębnych kolumnach i w zestawieniu traktować je jako dane elementarne. Przeprowadzenie takiej operacji w większości serwerów baz danych umożliwia specjalny operator konkatenacji (podwójny znak || albo & lub +). W MySQL-u musimy użyć w tym celu systemowej funkcji CONCAT(). Pierwszy przykład pokazuje sposób wywoływania funkcji systemowych (listing 3.8).

Listing 3.8. Przykład wywołania funkcji CONCAT

```
SELECT CONCAT('Klient ' ,fname )

FROM customer;

+--------------------------+

| CONCAT('Klient ' ,fname ) |

+--------------------------+

| Klient Jenny |

| Klient Andrew |

| Klient Alex |

| Klient Adrian |

| Klient Simon |

| Klient Neil |

| Klient Richard |

| Klient Ann |

| Klient Christine |

| Klient Mike |

| Klient Dave |
```

| Klient Richard |

| Klient Laura |

| Klient Bill |

| Klient David |

| |

+--------------------------+

Parametry wywołania funkcji umieszcza się w nawiasach. Jeżeli funkcja jest wywoływana z większą liczbą parametrów, oddziela się je przecinkami.

W drugim przykładzie wykorzystamy wiadomości o aliasach, literałach i łączeniu ciągów znaków (listing 3.9).

Listing 3.9. Praktyczne wykorzystanie łączenia ciągów znaków

SELECT CONCAT(fname,' ', lname,' mieszka w ', town) AS dane

FROM customer;

+-----------------------------------+

| dane |

+-----------------------------------+

| Jenny Stones mieszka w Hightown |

| Andrew Stones mieszka w Lowtown |

| Alex Matthew mieszka w Nicetown |

| Adrian Matthew mieszka w Yuleville |

| Simon Cozens mieszka w Oahenham |

| Neil Matthew mieszka w Nicetown |

| Richard Stones mieszka w Bingham |

| Ann Stones mieszka w Bingham |

| Christine Hickman mieszka w Histon |

| Mike Howard mieszka w Tibsville |

| Dave Jones mieszka w Bingham |

| Richard Neill mieszka w Winersby |

| Laura Hendy mieszka w Oxbridge |

| Bill Neill mieszka w Welltown |

| David Hudson mieszka w Milltown |

| |

+-------------------------------------+

Niestandardowa funkcja serwera MySQL CONCTAT_WS() pozwala dodatkowo uprościć zapytania, w których łączymy ciągi znaków — jej pierwszym parametrem jest separator (znak lub ciąg znaków, które będą dodane pomiędzy łączonymi kolumnami), pozostałymi — nazwy łączonych kolumn (listing 3.10).

Listing 3.10. Najważniejsze funkcje serwera MySQL są opisane w 5. odcinku kursu

SELECT CONCAT_WS (' - ',fname,lname)

FROM customer;

+---------------------------------+

| CONCAT_WS (' - -',fname,lname) |

+---------------------------------+

| Jenny - Stones |

| Andrew - Stones |

| Alex - Matthew |

| Adrian - Matthew |

| Simon - Cozens |

| Neil - Matthew |

| Richard - Stones |

| Anna - Stones |

| Christine - Hickman |

| Mike - Howard |

| Dave - Jones |

| Richard - Neill |

| Laura - Hendy |

| Bill - Neill |

| David - Hudson |

| Wolski |

+----------+ **Eliminacja duplikatów**

Domyślnie serwery bazodanowe wyświetlają wszystkie wiersze wchodzące w skład wyniku zapytania, nawet jeżeli w kilku wierszach przechowywana jest taka sama wartość. Dlatego, odczytując identyfikatory klientów z tabeli orderinfo, zobaczymy identyfikator tego samego klienta tyle razy, ile razy składał on u nas zamówienia (listing 3.11).

Listing 3.11. Lista identyfikatorów klientów zawierająca duplikaty

```
SELECT customer_id

FROM orderinfo;

+-------------+

| customer_id |

+-------------+

| 3 |

| 8 |

| 8 |

| 13 |

| 15 |

+-------------+
```

Do wyeliminowania z powyższego zestawienia powtarzających się wartości służy słowo kluczowe DISTINCT. Słowo DISTINCT musi pojawić się bezpośrednio po słowie kluczowym SELECT i odnosi się do wszystkich kolumn występujących w klauzuli SELECT. Aby w naszym przypadku wyeliminować powtarzające się identyfikatory, wystarczy napisać (listing 3.12):

Listing 3.12. Ta sama lista bez duplikatów

```
SELECT DISTINCT customer_id

FROM orderinfo;

+-------------+

| customer_id |

+-------------+

| 3 |

| 8 |

| 13 |
```

| 15 |

Ponieważ słowo DISTINCT eliminuje duplikaty kombinacji wszystkich wymienionych w klauzuli SELECT wartości, to jeżeli dodatkowo będziemy chcieli odczytać datę złożenia zamówienia, wynik znowu będzie liczył pięć wierszy. Wynika to z tego, że klient o identyfikatorze 8 złożył zamówienia w różnych dniach (listing 3.13).

Listing 3.13. Słowo kluczowe DISTINCT eliminuje powtarzające się wartości wierszy, a nie pojedynczych wyrażeń

SELECT DISTINCT customer_id, date_placed

FROM orderinfo;

+-------------+-------------+

| customer_id | date_placed |

+-------------+-------------+

| 3 | 2000-03-13 |

| 8 | 2000-06-23 |

| 15 | 2000-09-02 |

| 13 | 2000-09-03 |

| 8 | 2000-07-21 |

+-------------+-------------+

Używanie słowa kluczowego DISTINCT w połączeniu z symbolem wieloznacznym * nie jest błędem składniowym, ale prawie na pewno jest błędem logicznym. **Wartość NULL**

Jednym z podstawowych postulatów twórcy teorii relacyjnych baz danych jest postulat wartości Null. Na jego podstawie w każdym serwerze baz danych jest dostępna specjalna wartość reprezentująca wartość nieokreśloną, brakującą lub nieznaną. Jest to wartość różna od wszelkich konkretnych wartości, w szczególności — od ciągu pustego (" ") i zera (0). Charakterystyczne dla wartości Null jest to, że dowolne wyrażenie, w którym wystąpi ta wartość, również przybierze wartość Null oraz że nie możemy sprawdzać, jaka jest wartości Null (np. sprawdzenie, czy nazwa miasta jest równa Null, jest bezsensowne), a jedynie — czy jakaś wartość jest nieokreślona.

Na przykład podatek VAT od towaru o nieznanej nazwie jest — jak należało się spodziewać — nieznany, a więc będzie reprezentowany przez wartość Null (listing 3.14).

Listing 3.14. Przykład operacji arytmetycznej na wartości Null

SELECT sell_price, sell_price*0.22

FROM item;

+------------+-----------------+

| sell_price | sell_price*0.22 |

+------------+-----------------+

| 21.95 | 4.8290 |

| 11.49 | 2.5278 |

| 2.49 | 0.5478 |

| 3.99 | 0.8778 |

| 9.95 | 2.1890 |

| 15.75 | 3.4650 |

| 19.95 | 4.3890 |

| 1.45 | 0.3190 |

| 2.45 | 0.5390 |

| 0.00 | 0.0000 |

| 25.32 | 5.5704 |

| NULL | NULL |

+------------+-----------------+

To samo dotyczy operacji przeprowadzanych z użyciem funkcji systemowych — to dlatego ostatni wiersz wyniku zapytania SELECT CONCAT('Klient ' ,fname) FROM customer; był nieokreślony.

Porządkowanie danych

Instrukcja SELECT zwraca wiersze w tej kolejności, w jakiej dane są przechowywane w tabeli. Z reguły jest to kolejność, w jakiej były dopisywane następne wiersze z danymi. Do zmiany kolejności, w jakiej zwracane będą wyniki zapytania, służy klauzula ORDER BY. ORDER BY („uporządkuj według") jest opcjonalnym składnikiem instrukcji SELECT. Jeżeli jednak ta klauzula wystąpi, musi być ostatnią.

Kolejność klauzul instrukcji SELECT nie jest dowolna.

Obowiązkowym parametrem klauzuli ORDER BY jest wyrażenie lub nazwa kolumny — według ich wartości należy posortować dane wynikowe. Wykonanie poniższej instrukcji spowoduje wyświetlenie opisów towarów i cen ich zakupu uszeregowanych według cen zakupu (listing 3.15).

Listing 3.15. Posortowana lista towarów

SELECT description, cost_price

FROM item

ORDER BY cost_price;

+-----------------+------------+

| description | cost_price |

+----------------+------------+

| SQL Server 2005 | NULL |

| Carrier Bag | 0.01 |

| Toothbrush | 0.75 |

| Linux CD | 1.99 |

| Tissues | 2.11 |

| Roman Coin | 2.34 |

| Rubik Cube | 7.45 |

| Picture Frame | 7.54 |

| Fan Small | 9.23 |

| Fan Large | 13.36 |

| Wood Puzzle | 15.23 |

| Speakers | 19.73 |

+----------------+------------+

Domyślnie dane szeregowane są w porządku rosnącym, czyli od wartości najmniejszych do największych w przypadku danych liczbowych, od najwcześniejszych do najpóźniejszych w przypadku dat oraz w porządku alfabetycznym w przypadku ciągów znakowych. Aby odwrócić kolejność sortowania, należy bezpośrednio po nazwie kolumny użyć słowa kluczowego DESC (ang. Descending) (listing 3.16).

Listing 3.16. Lista towarów posortowana od najdroższego do najtańszego

```
SELECT description, cost_price
FROM item
ORDER BY cost_price DESC;
```

+----------------+------------+

| description | cost_price |

+----------------+------------+

| Speakers | 19.73 |

| Wood Puzzle | 15.23 |

| Fan Large | 13.36 |

| Fan Small | 9.23 |

| Picture Frame | 7.54 |

| Rubik Cube | 7.45 |

| Roman Coin | 2.34 |

| Tissues | 2.11 |

| Linux CD | 1.99 |

| Toothbrush | 0.75 |

| Carrier Bag | 0.01 |

| SQL Server 2005 | NULL |

+-----------------+------------+

Poszczególne klauzule instrukcji SELECT są od siebie niezależne. To znaczy, że w klauzuli ORDER BY możemy użyć wyrażenia lub nazwy kolumny, która nie występuje w klauzuli SELECT (listing 3.17).

Listing 3.17. Lista nazw towarów posortowana według cen ich zakupu

SELECT description

FROM item

ORDER BY cost_price;

+-----------------+

| description |

+-----------------+

| SQL Server 2005 |

| Carrier Bag |

| Toothbrush |

| Linux CD |

| Tissues |

| Roman Coin |

| Rubik Cube |

| Picture Frame |

| Fan Small |

| Fan Large |

| Wood Puzzle |

| Speakers |

+-----------------+

Oczywiście możemy sortować dane według więcej niż jednego kryterium. Instrukcja z listingu 3.18 zawiera alfabetycznie uporządkowaną listę adresów klientów — najpierw dane sortowane są według nazw miast, a następnie według nazw ulic.

Listing 3.18. Jeżeli kilku klientów mieszka w tym samym mieście, o ich kolejności na liście zadecyduje nazwa i numer ulicy

SELECT town, addressline

FROM customer

ORDER BY town, addressline

+-----------+-------------------+

| town | addressline |

+-----------+-------------------+

| | |

| Bingham | 34 Holly Way |

| Bingham | 34 Holly Way |

| Bingham | 54 Vale Rise |

| Hightown | 27 Rowan Avenue |

| Histon | 36 Queen Street |

| Lowtown | 52 The Willows |

| Milltown | 4 The Square |

| Nicetown | 4 The Street |

| Nicetown | 5 Pasture Lane |

| Oahenham | 7 Shady Lane |

| Oxbridge | 73 Margeritta Way |

| Tibsville | 86 Dysart Street |

| Welltown | 2 Beamer Street |

| Winersby | 42 Thached way |

| Yuleville | The Barn |

+----------+-------------------+

Wybieranie wierszy

Instrukcje SELECT w tej postaci, której używaliśmy do tej pory, zwracały wszystkie wiersze z danej tabeli. Do ograniczenia (wybrania) wierszy w wyniku należy wykorzystać klauzulę WHERE. Odpowiada ona teoriomnogościowemu operatorowi selekcji, czyli wybierania wierszy. Operacja ta najczęściej polega na wyborze grupy wierszy z tabeli na podstawie pewnych kryteriów. Serwer baz danych dla każdego wiersza sprawdzi, czy spełnia on kryteria wyboru, i jeżeli tak — zostanie on dodany do wyniku zapytania.

Klauzula WHERE, o ile była użyta, musi wystąpić bezpośrednio po klauzuli FROM. Kryterium wyboru może być sformułowane za pomocą typowych operatorów porównania lub operatorów charakterystycznych dla języka SQL.

Logika trójwartościowa

Zanim przejdziemy do omówienia klauzuli WHERE, powinniśmy poznać obowiązującą w języku SQL logikę trójwartościową i trzy podstawowe operatory (spójniki) logiczne: NOT, AND i OR.

Podstawą używanej na co dzień klasycznej[1] logiki są trzy zasady uznawane intuicyjnie za prawdziwe:

1.

Zasada tożsamości, na mocy której a=a, czyli każda rzecz jest równa samej sobie.

2.

Zasada sprzeczności, na mocy której ~(a ∧ ~a), czyli z dwóch sprzecznych zdań (predykatów) jedno jest prawdziwe[2].

3.

Zasada wyłączonego środka, na mocy której a ∨ ~a, czyli zdanie (predykat) albo jest prawdziwe, albo fałszywe[3].

W języku SQL żadna z tych zasad nie stosuje się do wartości NULL. Pozostałe wartości są przetwarzane i porównywane zgodnie z trzema zasadami logiki klasycznej. Wynikiem wyrażeń zawierających wartość NULL jest zatem zawsze wartość NULL. Z kolei porównywanie wartości NULL z innymi wartościami, w tym z nią samą, daje w wyniku wartość nieznaną UNKNOWN, a nie prawdę lub fałsz. Jest to sprzeczne z wszystkimi trzema zasadami logiki klasycznej:

1. Wartość NULL nie jest równa samej sobie.
2. Ani wartość NULL, ani jej negacja nie są prawdziwe.
3. Wartość NULL nie jest ani prawdziwa, ani fałszywa.

---

[1] Klasycznej, bo opracowanej przez starożytnych Greków, głównie przez Arystotelesa.

[2] W klasycznej logice symbol ~ oznacza negację (logiczne NIE), a symbol ∨ koniunkcję (logiczne I).

[3] W klasycznej logice symbol oznacza alternatywę (logiczne LUB).

Operatory logiczne

Operatory logiczne można porównać do spójników zdań — możemy dzięki nim połączyć kilka prostych warunków logicznych w jeden złożony.

Klasycznym warunkiem logicznym jest wyrażenie prawdziwe lub fałszywe. W języku SQL warunek logiczny może mieć trzecią wartość — wartość UNKNOWN.

Rolę operatorów logicznych przedstawimy na kilku przykładach:

1. Operator AND (logiczne I, czyli koniunkcja) może być użyty do połączenia następujących warunków logicznych: cena < 500 AND kolor = niebieski. Otrzymany w ten sposób złożony warunek logiczny jest prawdziwy tylko wtedy, gdy cena jest niższa niż 500, a kolor — niebieski.

2. Operator OR (logiczne LUB, czyli alternatywa) może być użyty do połączenia następujących warunków logicznych: kolor = niebieski LUB kolor = czerwony LUB kolor = żółty. Otrzymany w ten sposób złożony warunek logiczny jest prawdziwy, jeżeli kolor jest niebieski, czerwony lub żółty.

3. Operatory AND, OR i NOT (logiczne NIE, czyli negacja) mogą być użyte do połączenia kilku prostych warunków logicznych: cena < 500 AND (kolor NOT czarny OR marka NOT Ford). Otrzymany w ten sposób złożony warunek logiczny jest prawdziwy tylko wtedy, gdy cena jest niższa niż 500 i albo kolor nie jest czarny, albo marką nie jest Ford.

Możliwe wartości (TRUE, FALSE lub UNKNOWN) operatorów logicznych przedstawia się najczęściej w postaci tabel prawdziwości zawierających wszystkie możliwe kombinacje parametrów danego operatora. **Typowe operatory porównania**

Operatory logiczne porównują, czy dany warunek jest spełniony, czyli czy w wyniku porównania argumentów otrzymamy wartość logiczną True (Prawda). Na przykład wszystkie poniższe warunki są prawdziwe: 1<4, 2=2, 5>=5, 'mama'='mama' itd.

Zwróćmy uwagę, że ciągi znaków muszą być umieszczone wewnątrz apostrofów. Domyślnie MySQL przy porównywaniu ciągów znaków nie rozróżnia wielkich i małych liter (listing 3.19).

Listing 3.19. Wynikiem porównania wyrazów kot i KOT jest wartość prawda

```
SELECT 'kot' = 'KOT';

+---------------+

| 'kot' = 'KOT' |

+---------------+

| 1 |

+---------------+
```

Aby na przykład wybrać tylko te towary, których cena zakupu nie przekracza 5 zł, należy wykonać instrukcję z listingu 3.20.

Listing 3.20. Prosty test logiczny wybierający towary o określonej cenie zakupu

```
SELECT *

FROM item
```

```
WHERE cost_price <=5;

+---------+-------------+------------+------------+

| item_id | description | cost_price |  sell_price |

+---------+-------------+------------+------------+

| 3 | Linux CD | 1.99 | 2.49 |

| 4 | Tissues | 2.11 | 3.99 |

| 8 | Toothbrush | 0.75 | 1.45 |

| 9 | Roman Coin | 2.34 | 2.45 |

| 10 | Carrier Bag | 0.01 | 0.00 |

+---------+-------------+------------+------------+
```

Zwróć uwagę, że w wyniku nie znalazł się towar o nieokreślonej nazwie. Pamiętaj, że wartość Null ma specjalne znaczenie i nie można jej sensownie używać z operatorami „mniejszy” czy „równy”.

Aby ograniczyć liczbę informacji o interesujących nas towarach (towarach kupionych za nie więcej niż 5 zł) do ich nazwy i obu cen (zakupu i sprzedaży), należy połączyć w jednej instrukcji operacje projekcji i selekcji. Dodatkowo, wykorzystując klauzulę ORDER BY, możemy posortować wynik zapytania w kolejności od towarów kupionych najtaniej do kupionych najdrożej. Zmodyfikowane polecenie SELECT powinno wyglądać następująco (listing 3.21).

Listing 3.21. Przykład wykorzystania przedstawionych do tej pory wiadomości o instrukcji SELECT

```
SELECT description, cost_price, sell_price

FROM item

WHERE cost_price <=5

ORDER BY cost_price;

+-------------+------------+------------+

| description | cost_price | sell_price |

+-------------+------------+------------+

| Carrier Bag | 0.01 | 0.00 |

| Toothbrush | 0.75 | 1.45 |

| Linux CD | 1.99 | 2.49 |

| Tissues | 2.11 | 3.99 |

| Roman Coin | 2.34 | 2.45 |
```

+----------+----------+----------+

Język SQL pozwala również na użycie nazw kolumn po obu stronach operatora porównania. Aby na przykład wyświetlić informacje o tych towarach, które sprzedajemy poniżej ceny zakupu, napiszemy (listing 3.22):

Listing 3.22. Porównanie danych odczytanych z tabeli. Serwer baz danych sprawdzi ten warunek dla każdego wiersza tabeli

SELECT *

FROM item

WHERE cost_price > sell_price;

+---------+----------+----------+----------+

| item_id | description | cost_price | sell_price |

+---------+----------+----------+----------+

| 10 | Carrier Bag | 0.01 | 0.00 |

+---------+----------+----------+----------+

SQL dopuszcza ponadto tworzenie bardziej skomplikowanych warunków logicznych, wykorzystujących omówione operatory AND (logiczne I, koniunkcja), OR (logiczne LUB, alternatywa) oraz NOT (logiczne NIE, negacja). Wynik operacji porównania obliczany jest na podstawie tabel prawdziwości danego operatora. Tabele prawdziwości dla operatorów NOT, AND i OR przedstawione są na rysunku 3.1.

Rysunek 3.1. Tabele prawdziwości operatorów logicznych

a

NOT a

TRUE

FALSE

FALSE

TRUE

NULL

UNKNOWN

a

b

a OR b

TRUE

TRUE

TRUE

TRUE

FALSE

TRUE

TRUE

NULL

TRUE

FALSE

TRUE

TRUE

FALSE

FALSE

FALSE

FALSE

NULL

UNKNOWN

NULL

TRUE

TRUE

NULL

FALSE

UNKNOWN

NULL

NULL

UNKNOWN

a

b

a AND b

TRUE

TRUE

TRUE

TRUE

FALSE

FALSE

TRUE

NULL

UNKNOWN

FALSE

TRUE

FALSE

FALSE

FALSE

FALSE

FALSE

NULL

FALSE

NULL

TRUE

UNKNOWN

NULL

FALSE

FALSE

NULL

NULL

UNKNOWN

Listing 3.23 pokazuje przykład użycia złożonego warunku logicznego — w wyniku zapytania znajdą się tylko te zamówienia, które zostały złożone przed końcem czerwca 2000 roku przez klienta o identyfikatorze 8.

Listing 3.23. Koniunkcja ogranicza wiersze wyniku

```
SELECT *
FROM orderinfo
WHERE customer_id=8 AND date_placed  <'2000-06-30';
+--------------+-------------+-------------+--------------+----------+
| orderinfo_id | customer_id | date_placed |  date_shipped | shipping |
+--------------+-------------+-------------+--------------+----------+
| 2 | 8 | 2000-06-23 | 2000-06-23 | 0.00 |
+--------------+-------------+-------------+--------------+----------+
```

Specjalne znaczenie wśród operatorów logicznych ma operator negacji. W przeciwieństwie do pozostałych jest on operatorem jednoargumentowym, to znaczy, że w celu obliczenia wyniku wystarczy podać jeden argument. Jak wynika z tabeli prawdziwości, wynik operacji NOT a jest prawdą wtedy i tylko wtedy, gdy argument a jest fałszywy. Operator negacji służy do zaprzeczania warunkom podanym w klauzuli WHERE (listing 3.24).

Listing 3.24. Zapytanie zwracające informacje o mężczyznach spoza Lowtown

```
SELECT title, fname, lname, town

FROM customer

WHERE title = 'Mr' AND town !='Lowtown';

+-------+---------+---------+-----------+

| title | fname | lname | town |

+-------+---------+---------+-----------+

| Mr | Adrian | Matthew | Yuleville |

| Mr | Simon | Cozens | Oahenham |

| Mr | Neil | Matthew | Nicetown |

| Mr | Richard | Stones | Bingham |

| Mr | Mike | Howard | Tibsville |

| Mr | Dave | Jones | Bingham |

| Mr | Richard | Neill | Winersby |

| Mr | Bill | Neill | Welltown |

| Mr | David | Hudson | Milltown |

+-------+---------+---------+-----------+
```

Oczywiście, w klauzuli WHERE, tak jak w klauzulach SELECT czy ORDER BY, możemy używać wyrażeń. W efekcie możemy łatwo wybrać na przykład informacje o tych towarach, które sprzedajemy z ponad 50-procentową marżą (listing 3.25).

Listing 3.25. Przykład wykorzystania wyrażenia w klauzuli WHERE

```
SELECT *

FROM item

WHERE sell_price>=1.5*cost_price;

+---------+-------------+------------+------------+

| item_id | description | cost_price | sell_price |

+---------+-------------+------------+------------+

| 2 | Rubik Cube | 7.45 | 11.49 |

| 4 | Tissues | 2.11 | 3.99 |

| 6 | Fan Small | 9.23 | 15.75 |
```

| 8 | Toothbrush | 0.75 | 1.45 |

+---------+-------------+------------+------------+ **Operatory charakterystyczne dla języka SQL**

Operatorów języka SQL, tak jak pozostałych operatorów, można używać do porównywania różnych typów danych. Do operatorów SQL zalicza się:

1. Operator BETWEEN ... AND (należy do przedziału).
2. Operator IN (lista) (należy do zbioru).
3. Operator LIKE (jest zgodny z wzorcem).
4. Operator IS (jest).

BETWEEN … AND

Operator BETWEEN ... AND służy do sprawdzenia, czy dana wartość znajduje się w podanym przedziale. MySQL traktuje ten przedział jako obustronnie zamknięty, czyli zalicza do niego wartości graniczne. Dodatkowo, jeżeli chcemy, żeby zapytanie zwróciło jakiekolwiek dane, górna granica przedziału nie może być mniejsza niż dolna (listing 3.26).

Listing 3.26. Lista towarów, których cena sprzedaży jest większa od 10, ale mniejsza niż 15

```
SELECT *

FROM item

WHERE sell_price BETWEEN 10 AND 15;
```

+---------+-------------+------------+------------+

| item_id | description | cost_price | sell_price |

+---------+-------------+------------+------------+

| 2 | Rubik Cube | 7.45 | 11.49 |

+---------+-------------+------------+------------+ **IN**

Operator IN służy do sprawdzenia, czy dana wartość należy do podanego zbioru. Przy czym może to być zbiór dowolnych wartości, niekoniecznie liczb (listing 3.27).

Listing 3.27. Dane osób mieszkających w Nicetown lub Welltown

```
SELECT fname, town

FROM customer

WHERE town IN ('Nicetown','Welltown');
```

+-------+----------+

| fname | town |

+-------+----------+

| Alex | Nicetown |

| Neil | Nicetown |

| Bill | Welltown |

+-------+----------+ **LIKE**

Za pomocą operatora LIKE możemy wyszukać dane tekstowe zgodne z podanym wzorcem. Przy tworzeniu wzorca można wykorzystać symbole o specjalnym znaczeniu:

1. Znak % (procent) — zastępuje dowolny ciąg znaków.
2. Znak _ (podkreślenie) — zastępuje dokładnie jeden dowolny znak.

Aby na przykład odszukać wszystkie panie lub panny, których nazwisko zaczyna się od liter ST, napiszemy (listing 3.28):

Listing 3.28. Operator LIKE powinien być używany tylko z danymi tekstowymi

SELECT title, fname, lname

FROM customer

WHERE title LIKE '%s' AND lname LIKE 'St%';

+-------+-------+--------+

| title | fname | lname |

+-------+-------+--------+

| Miss | Jenny | Stones |

| Mrs | Ann | Stones |

+-------+-------+--------+ **IS**

Operator IS służy przede wszystkim do wyszukiwania tych rekordów, w których przechowywana jest wartość nieokreślona (wartość Null), oraz do sprawdzania, czy dana wartość nie jest nieokreślona (listing 3.29).

Listing 3.29. Zapytanie zwracające informacje o towarach, które mają nazwę, ale nie mają określonej ceny sprzedaży

SELECT *

FROM item

WHERE sell_price IS NULL AND description IS NOT NULL;

+---------+-----------------+------------+------------+

| item_id | description | cost_price | sell_price |

+---------+----------------+------------+------------+

| 12 | SQL Server 2005 | NULL | NULL |

+---------+----------------+------------+------------+ **Hierarchia operatorów**

Ostateczny wynik złożonych warunków logicznych zależy od kolejności, w jakiej serwery bazodanowe sprawdzą prawdziwość składających się na nie prostych warunków. Kolejność ta wynika z hierarchii operatorów przyjętej w standardzie języka SQL:

1. Jako pierwsze wykonywane są operacje mnożenia i dzielenia.

2. Następnie dodawania i odejmowania.

3. W trzeciej kolejności wykonywane są standardowe operacje porównania, takie jak „równy” lub „mniejszy niż”.

4. Pierwszym wykonywanym operatorem logicznym jest operator NOT.

5. Następnie wykonywany jest operator AND.

6. Jako ostatnie — operatory SQL (IN, BETWEEN ... AND, LIKE) oraz operator OR.

Jeżeli wyrażenie zawiera kilka operatorów o tym samym priorytecie, wykonywane są one od lewej do prawej.

Ograniczanie liczby wierszy

Język SQL pozwala również ograniczyć liczbę wierszy wyniku zapytania — wystarczy w klauzuli LIMIT wpisać tę liczbę. W efekcie bardzo łatwo możemy np. odczytać nazwy i ceny pięciu najdroższych produktów (listing 3.30).

Listing 3.30. Wynik został ograniczony do pierwszych pięciu wierszy

```
SELECT description, sell_price
FROM item
ORDER BY sell_price DESC
LIMIT 5;
```

+-------------+------------+

| description | sell_price |

+-------------+------------+

| Speakers | 25.32 |

| Wood Puzzle | 21.95 |

| Fan Large | 19.95 |

| Fan Small | 15.75 |

| Rubik Cube | 11.49 |

+-------------+------------+

Klauzulę LIMIT stosuje się wyłącznie razem z klauzulą ORDER BY. W innym wypadku w wyniku znalazłyby się przypadkowe wiersze.

Kolejny przykład pokazuje, jak odczytać informacje o tym, którego towaru jest najmniej w magazynie (listing 3.31).

Listing 3.31. Klauzula LIMIT użyta do wyszukiwania najmniejszych lub największych danych

```
mysql> SELECT item_id, quantity
 -> FROM stock
 -> ORDER BY quantity
 -> LIMIT 1;
+---------+----------+
| item_id | quantity |
+---------+----------+
| 10 | 1 |
+---------+----------+
```

Odczytanie nazw pięciu najdroższych produktów okazało się bardzo proste. A gdybyśmy chcieli poznać nazwy i ceny produktów na miejscach od 3. do 10.? W takim wypadku wystarczy określić pierwszy dodany do wyniku wiersz (listing 3.32).

Listing 3.32. Zapytanie zwracające nazwy siedmiu z dziesięciu najdroższych produktów. W wyniku nie znalazły się dane trzech najdroższych produktów

```
SELECT description, sell_price
FROM item
ORDER BY sell_price DESC
LIMIT 3,7;
+---------------+------------+
| description | sell_price |
+---------------+------------+
| Fan Small | 15.75 |
| Rubik Cube | 11.49 |
```

| Picture Frame | 9.95 |

| Tissues | 3.99 |

| Linux CD | 2.49 |

| Roman Coin | 2.45 |

| Toothbrush | 1.45 |

+---------------+------------+