

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

100 sposobów na SQL

Autor: Andrew Cumming, Gordon Russell

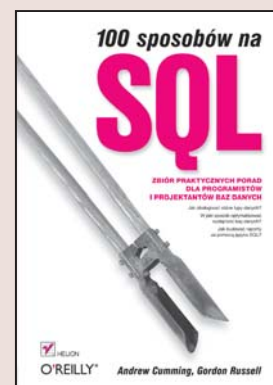
Tłumaczenie: Marcin Karbowski

ISBN: 978-83-246-0985-7

Tytuł oryginału: [SQL Hacks](#)

Format: B5, stron: 400

[Przykłady na ftp: 135 kB](#)



Zbiór praktycznych porad dla programistów i projektantów baz danych

- Jak obsługiwać różne typy danych?
- W jaki sposób optymalizować wydajność baz danych?
- Jak budować raporty za pomocą języka SQL?

Jesteś programistą, któremu przypadło w udziale opracowanie aplikacji bazodanowej? Szukasz sposobu na zoptymalizowanie działania zapytań SQL? A może zastanawiasz się, w jaki sposób sprawnie zarządzać kontami użytkowników bazy danych? Język SQL to potężne narzędzie, którego opanowanie pozwoli Ci na sprawne poruszanie się w gąszczu tabel każdej bazy danych. Za pomocą odpowiednio sformułowanych instrukcji można manipulować danymi, zarządzać kontami użytkowników i generować raporty. Jednak, pomimo stosunkowo niewielkiej ilości poleceń, język SQL kryje w sobie wiele zawichości.

Dzięki książce „100 sposobów na SQL” nie będziesz musiał odkrywać każdej z nich samodzielnie. W każdym z rozdziałów znajdziesz praktyczne porady i sposoby rozwiązywania typowych zadań programistycznych związanych z bazami danych i językiem SQL. Poznasz podstawy języka, dowiesz się, w jaki sposób przetwarzać różne typy danych i korzystać z symboli zastępczych. Nauczysz się tworzyć aplikacje sieciowe, optymalizować wydajność tabel i zapytań SQL oraz tworzyć raporty. Przeczytasz ponadto o administrowaniu serwerem bazy danych i udostępnianiu tabel użytkownikom.

- Podstawowe elementy języka SQL
- Łączenie tabel
- Przetwarzanie danych tekstowych
- Operacje na liczbach i datach
- Bazy danych w aplikacjach sieciowych
- Zapytania niezależne od tabel
- Maksymalizowanie wydajności zapytań
- Tworzenie raportów
- Administrowanie kontami użytkowników
- Udostępnianie informacji

Zostań ekspertem w programowaniu baz danych!



Spis treści

Twórcy książki	9
Wstęp	11
Rozdział 1. Podstawy SQL	19
1. Uruchamianie SQL za pomocą wiersza poleceń	19
2. Nawiązywanie połączenia między aplikacją a bazą danych SQL	27
3. Warunkowe polecenia INSERT	32
4. Polecenie UPDATE	36
5. Rozwiązywanie krzyżówek w SQL	39
6. Nie powtarzajcie bez przerwy tych samych obliczeń	41
Rozdział 2. Polecenia JOIN, UNION oraz VIEW	45
7. Modyfikowanie schematu bez naruszania kwerend	45
8. Filtrowanie wierszy i kolumn	47
9. Filtrowanie kolumn indeksowanych	50
10. Konwertowanie podzapytań na wyrażenia JOIN	52
11. Konwertowanie zagregowanych podzapytań na wyrażenia JOIN	55
12. Upraszczenie skomplikowanych poleceń UPDATE	56
13. Dopasowywanie złączeń do relacji	58
14. Tworzenie kombinacji	61
Rozdział 3. Obsługa tekstu	65
15. Wyszukiwanie słów kluczowych bez użycia operatora LIKE	65
16. Wyszukiwanie ciągów tekstowych w kolumnach	69
17. Rozwiązywanie anagramów	71
18. Sortowanie poczty elektronicznej	76
Rozdział 4. Daty	81
19. Konwertowanie ciągów tekstowych na daty	81
20. Wyszukiwanie trendów	85
21. Tworzenie raportów w oparciu o dowolne przedziały czasowe	89
22. Raporty kwartalne	94
23. Drugi wtorek miesiąca	97

Rozdział 5. Dane liczbowe	103
24. Mnożenie wartości	103
25. Sumy pośrednie	105
26. Dołączanie wierszy pominiętych przez wyrażenie JOIN	109
27. Identyfikowanie nakładających się zakresów	111
28. Unikanie dzielenia przez zero	114
29. Wyrażenie COUNT	115
30. Wyznaczanie wartości maksymalnej z dwóch pól	118
31. Dzielenie rezultatów funkcji COUNT	120
32. Błędy podczas zaokrąglania	121
33. Jednoczesne pobieranie wartości i sum częściowych	123
34. Obliczanie mediany	126
35. Przedstawianie wyników w formie wykresu	130
36. Obliczanie odległości między lokalizacjami wskazywanymi przez system GPS	132
37. Porównywanie faktur i wpłat	136
38. Wyszukiwanie błędów transpozycji	139
39. Naliczanie podatku progresywnego	143
40. Rangi	146
Rozdział 6. Aplikacje sieciowe	149
41. Kopiowanie stron internetowych do tabeli	149
42. Prezentowanie danych z wykorzystaniem skalowalnej grafiki wektorowej	157
43. Wzbogacanie aplikacji sieciowych o narzędzia do nawigacji	164
44. Definiowanie połączenia między systemem MySQL a programem Access	170
45. Przetwarzanie dzienników serwerów sieciowych	174
46. Przechowywanie obrazów w bazie danych	181
47. Atak SQL injection	185
48. Zapobieganie atakowi SQL injection	191
Rozdział 7. Porządkowanie danych	197
49. Śledzenie rzadko zmieniających się wartości	197
50. Łączenie tabel zawierających różne dane	200
51. Wyświetlanie wierszy w charakterze kolumn	202
52. Wyświetlanie kolumn w charakterze wierszy	205
53. Usuwanie niespójnych rekordów	207
54. Denormalizowanie tabel	209
55. Importowanie danych innych użytkowników	211
56. Zabawa w swatanie	213
57. Generowanie unikalnych numerów sekwencyjnych	215

Rozdział 8. Przechowywanie małych ilości danych	221
58. Przechowywanie parametrów w bazie danych	221
59. Definiowanie osobnych parametrów dla poszczególnych użytkowników	227
60. Lista parametrów	231
61. Bezpieczeństwo oparte na wierszach	232
62. Wykonywanie kwerend bez wykorzystywania tabel	235
63. Tworzenie wierszy bez udziału tabel	237
Rozdział 9. Blokowanie i wydajność	241
64. Określanie poziomu izolacji	241
65. Blokowanie pesymistyczne	246
66. Blokowanie optymistyczne	248
67. Niejawne blokowanie wewnątrz transakcji	251
68. Obsługa powtarzanych operacji	252
69. Wykonywanie funkcji w bazie danych	257
70. Łączenie kwerend	259
71. Pobieranie dużej liczby wierszy	261
72. Pobieranie podzbioru uzyskanych rezultatów	264
73. Przechowywanie plików w bazie danych	266
74. Porównywanie i synchronizowanie tabel	270
75. Minimalizowanie obciążenia łącz dla zbyt wielu złączeń	274
76. Kompresowanie w celu uniknięcia typu danych LOB	278
Rozdział 10. Raporty	281
77. Uzupełnianie brakujących wartości w tabeli przestawnej	281
78. Podział na zakresy	286
79. Jednoznaczne identyfikowanie aktualizacji	290
80. Sześć stopni od Kevina Bacona	295
81. Tabele decyzyjne	298
82. Generowanie sekwencyjnych lub brakujących danych	302
83. Wyszukiwanie n pierwszych wierszy w grupach	309
84. Przechowywanie list wartości oddzielonych przecinkami w kolumnach	312
85. Analizowanie prostych drzewek	314
86. Definiowanie kolejek w bazie danych	318
87. Tworzenie kalendarza	319
88. Testowanie dwóch wartości za pomocą podzapytania	322
89. Wybieranie trzech możliwości spośród pięciu	324

Rozdział 11. Użytkownicy i administracja	329
90. Implementowanie kont na poziomie aplikacji	329
91. Eksportowanie i importowanie definicji tabel	336
92. Wdrażanie aplikacji	345
93. Automatyczne tworzenie kont użytkowników	350
94. Tworzenie kont użytkowników i administratorów	352
95. Automatyczne aktualizacje	355
96. Tworzenie dziennika zdarzeń	358
Rozdział 12. Szerszy dostęp	363
97. Anonimowe konta	364
98. Wyszukiwanie i przerywanie długo wykonywanych kwerend	366
99. Zarządzanie przestrzenią dyskową	370
100. Uruchamianie kwerend za pośrednictwem stron internetowych	374
Skorowidz	381

Polecenia JOIN, UNION oraz VIEW

Sposoby 7. – 14.

W celu powiązania ze sobą dwóch tabel wykorzystać możemy *złączenie*. Często wynika ono z zastosowania *odwołania do klucza zewnętrznego*. Załóżmy na przykład, iż dysponujemy tabelą `pracownik`, zawierającą kolumnę z numerami `id` poszczególnych wydziałów dla każdego z pracowników. Aby wyświetlić nazwy wydziałów przypisanych pracownikom, możemy posłużyć się poleceniem `JOIN`:

```
SELECT pracownik.nazwa, wydzial.nazwa
FROM pracownik JOIN wydzial ON (pracownik.wydzial=wydzial.id)
```

Domyślnie wybierane jest złączenie `INNER JOIN`. Istnieją również inne rodzaje złączeń: `LEFT OUTER JOIN`, `FULL OUTER JOIN` oraz `CROSS JOIN`. Wszystkie opisano w niniejszym rozdziale.

Dwie tabele można połączyć również za pomocą polecenia `UNION`. W przeciwieństwie do polecenia `JOIN` powoduje ono połączenie wierszy w obu tabelach — otrzymujemy jeden wynik. Tabele muszą mieć taką samą liczbę kolumn, a odpowiadające sobie kolumny muszą być tego samego typu.

Za pomocą polecenia `VIEW` możemy nadawać kwerendom nazwy. Wyrażenie `SELECT` (wykorzystujące polecenie `JOIN` lub `UNION`) da się zapisać jako widok danych. W miarę możliwości system będzie traktował widok jak tabelę podstawową, umożliwiając przeprowadzanie na niej operacji `SELECT`, `JOIN`, `UPDATE`, `DELETE` oraz `INSERT` (z pewnymi ograniczeniami).



SPOSÓB

7.

Modyfikowanie schematu bez naruszania kwerend

Jeśli zmieniają się wymagania dotyczące oprogramowania i konieczne jest zaprojektowanie nowej bazy danych, nie trzeba od razu pozbywać się całego wcześniej napisanego kodu. Dzięki zastąpieniu nieistniejących tabel widokami danych istniejące kwerendy nadal będą działać.

Prędzej czy później będziecie musieli wprowadzić gruntowne zmiany w strukturze bazy danych. Dzięki zastosowaniu odpowiednich rozwiązań można to osiągnąć, nie pozbywając się oryginalnego kodu.

Przykładowo powiedzmy, że firma prowadzi rejestr wyposażenia w formie przedstawionej w tabeli 2.1.

Tabela 2.1. Tabela „sprzet”

Numer	Opis	DataZakupu
50430	Komputer PC	2004-07-02
50431	Monitor 19 cali	2004-07-02

Załóżmy teraz, że firma otwiera nowe biuro i konieczne jest prowadzenie rejestru dla obu filii z osobna. Czy należy zrobić kopię aplikacji i bazy danych, czy też zmienić jej strukturę?

Kopiowanie bazy danych

Skopiowanie bazy danych oraz aplikacji wydaje się kuszącym rozwiązaniem. Niestety, mimo iż pomaga to uporać się z doraźnymi problemami, na dłuższą metę przyniesie więcej szkody niż pożytku. Konieczne będzie obsługiwanie dwóch aplikacji, zakupienie dodatkowego sprzętu i tworzenie dwóch baz danych, których ewentualne połączenie będzie trudnym zadaniem. Sytuacja pogorszy się, jeśli otwarta zostanie kolejna filia.

Zmiana tabeli

SQL daje nam do dyspozycji polecenie pozwalające na dodanie kolumny przy jednoczesnym zachowaniu istniejących danych (możemy również zmieniać nazwy pól i usuwać zbędne elementy tabel):

```
ALTER TABLE sprzet ADD COLUMN biuro VARCHAR(20);  
UPDATE sprzet SET biuro = 'SiedzibaGłówna';
```

W ten sposób dodajemy nową kolumnę i przypisujemy wszystkie wiersze do istniejącego biura (w praktyce sytuacja wyglądałaby nieco inaczej — część wyposażenia na ogół przewożona jest do nowej placówki). Nazwa pierwszego biura zmieniona zostaje na „Siedzibę Główną”. Możemy teraz rozpocząć tworzenie listy wyposażenia dla nowego biura. Wcześniej jednak należy sprawdzić poprawność kwerend opartych na zmienionej tabeli. Wyrażenia `INSERT`, które nie wskazują określonych kolumn, nie zadziałają poprawnie. Jeśli zatem polecenie `INSERT` miało postać:

```
INSERT INTO sprzet VALUES (50322, 'Drukarka Laserowa',DATE '2004-07-02');
```

pojawi się komunikat o błędzie. Jeśli jednak kwerenda była następująca:

```
INSERT INTO sprzet (Numer,Opis,DataZakupu)  
VALUES (50322,'Drukarka Laserowa',DATE '2004-07-02');
```

zostanie ona przeprowadzona poprawnie, a w kolumnie `biuro` znajdzie się wartość `NULL`.

Istnieje spora szansa, iż kwerendy będą działać pomimo zmian wprowadzonych w tabeli, ale zwrócone wyniki odnosić się będą do obu filii, nawet jeśli przetwarzane dane dotyczyć mają tylko jednej z nich.

Zastępowanie tabeli widokiem danych

Alternatywnym rozwiązaniem jest skopiowanie danych do nowej tabeli i zastąpienie istniejącej widokiem danych:

```
CREATE TABLE SprzetOgolem
(Numer          INTEGER PRIMARY KEY
, Biuro         VARCHAR(20) DEFAULT 'SiedzibaGłówna'
, Opis         VARCHAR(100)
, DataZakupu   DATE
);
INSERT INTO SprzetOgolem
  SELECT Numer, 'SiedzibaGłówna', Opis, DataZakupu FROM sprzet;
```

Nowa tabela zawiera takie same dane — cały sprzęt po raz kolejny przypisaliśmy do starego biura. W razie potrzeby (przeniesienia części wyposażenia do nowej placówki) należy zmienić odpowiednie wiersze.

Możemy teraz usunąć starą tabelę i zastąpić ją widokiem danych:

```
DROP TABLE sprzet;
CREATE VIEW sprzet AS
  SELECT Numer, Opis, DataZakupu
  FROM SprzetOgolem WHERE biuro='SiedzibaGłówna';
```

Dzięki temu wszystkie wcześniej utworzone kwerendy będą działać poprawnie — ponieważ odwołują się one do nazwy `sprzet` i nie ma znaczenia, czy określa ona tabelę, czy widok danych. Kierownik pierwszego biura będzie mógł nadal korzystać z tej samej aplikacji i danych dotyczących sprzętu pozostawionego w jego placówce, z możliwością wykonywania na nich operacji `INSERT` oraz `UPDATE`. Możemy nawet przyznać mu uprawnień ograniczające wykonywane operacje do danych związanych z jego filią.

Czeka nas jednak jeszcze nieco dodatkowej pracy. Widok danych `sprzet` przechowywać możemy lokalnie dla każdego konta z osobna. Dzięki temu jego zawartość będzie inna dla każdego z użytkowników. Więcej informacji na ten temat znaleźć można w sposobie 59.



SPOSÓB
8.

Filtrowanie wierszy i kolumn

Nie warto pobierać od razu całej tabeli. Można korzystać z filtrowania wierszy i kolumn w celu zmniejszenia przepływu danych w systemie,

Niektórzy programiści starają się za wszelką cenę unikać baz danych. Opanowują pojedyncze wyrażenie SQL i wykorzystują je na okrągło, bez względu na okoliczności. Do szczęścia potrzeba im jedynie polecenia `SELECT * FROM t`. Po prostu wczytują całą tabelę i traktują ją jak gigantyczną tablicę. Nie ma potrzeby opanowywania całego języka SQL prawda? Problem w tym, że takie podejście jest nieefektywne.

Załóżmy, że obsługujemy stronę internetową, której poszczególne fragmenty przechowywane są w bazie danych. Ułatwia to zarządzanie jej zawartością i kontrolę wersji, ale obejrzenie każdej ze stron wymaga pobrania danych z bazy. Sama tabela ma dwa pola: `nazwaStrony` i `zawartosc`. Jak najefektywniej rozwiązać ten problem w języku Perl? Nazwa poszukiwanej strony przechowywana jest w zmiennej `$p`:

```
my $sql = "SELECT nazwaStrony, zawartosc FROM strona";
my $sth = $dbh->prepare($sql);
my $rsh = $sth->execute();
while (my $row = $sth->fetchrow_hashref() ) {
    print $row->{zawartosc} if ($row->{nazwaStrony} eq $p);
}
```

Przedstawiony kod cechuje liniowy spadek wydajności. W miarę dodawania kolejnych stron zwiększa się przesył danych między umieszczoną na serwerze bazą danych a programem. Konieczne jest przeprowadzenie filtrowania.

Podczas pracy w SQL należy filtrować dane w celu uzyskania wymaganych informacji. Poniższy kod jest znacznie lepszy, choć nadal ma pewne wady:

```
my $sql = "SELECT nazwaStrony, zawartosc FROM strona WHERE nazwaStrony =
'".$p."";
my $sth = $dbh->prepare($sql);
my $rsh = $sth->execute();
my $row = $sth->fetchrow_hashref();
print $row->{zawartosc} if $row;
```

Możliwe, iż zmienna `$p` przybierze nieoczekiwaną wartość. Przykładowo zamiast `index.html` wpisana zostanie nazwa `index'html`. W takim przypadku kwerenda nie zadziałałaby z powodu błędu składni.

Zignorowanie tego problemu grozi nie tylko błędami składni. Umożliwia również włamanie do bazy danych za pomocą ataku *SQL injection* [Sposób 48].

Przed wspomnianym atakiem można się zabezpieczyć, stosując symbole zastępcze, określane również jako *zmiennie wiązane* lub *parametry kwerendy*. Zwykle wprowadza się je, umieszczając znak `?` w miejscu, w którym pojawić się ma zawartość zmiennej, a następnie przesyłając zmienną jako parametr wywołania `execute` API. Przedstawiony kod przybiera zatem postać:

```
my $sql = "SELECT nazwaStrony, zawartosc FROM strona WHERE nazwaStrony = ? ";
my $sth = $dbh->prepare($sql);
my $rsh = $sth->execute($p);
my $row = $sth->fetchrow_hashref();
print $row->{zawartosc} if $row;
```

W ten sposób możemy obsługiwać większą liczbę parametrów; wystarczy wprowadzać je w kolejności, w jakiej `?` pojawia się w ciągu znaków `$sql`. Dodatkową zaletą tego rozwiązania jest *buforowanie kwerendy*. Za każdym razem, kiedy uruchamiamy nasz kod, w bazie danych wykonywane jest to samo polecenie — bez względu na wyszukiwaną stronę. Strona ta przekazywana jest oddzielnie jako zmienna odpowiadająca symbolowi zastępczemu.

Filtrowanie danych na serwerze zapewnia szybszy czas odpowiedzi i zmniejsza wymagania dotyczące przepustowości pasma łączącego bazę danych z programem. Jest to efektywniejsze rozwiązanie również z innych powodów. Jednym z nich jest indeksowanie [Sposób 9].

Wiele serwerów baz danych przechowuje ostatnio wykonywane kwerendy w analizatorach składni. W buforze może nawet zostać umieszczony plan kwerendy. Jeśli kwerenda jest zawsze taka sama, mechanizm SQL nie musi za każdym razem przygotowywać jej do uruchomienia.

Symbole zastępcze nie są wyłącznie domeną języka Perl. Języki opisane w podrozdziale „Nawiązywanie połączenia między aplikacją a bazą danych SQL” [Sposób 2.] posiadają podobne rozwiązania. Oto przykłady umieszczania zmiennej `mojParametr` w ramach symbolu zastępczego.

Perl

```
my $sql = "SELECT kolumna FROM tabela WHERE kolumna = ? ";
my $sth = $dbh->prepare($sql,$mojParametr);
```

Java

```
PreparedStatement sql =
    con.prepareStatement("SELECT kolumna FROM tabela WHERE kolumna = ? ");
sql.setString(1, mojParametr);
ResultSet cursor = sql.executeQuery();
```

Ruby

```
sql = db.prepare("SELECT kolumna FROM tabela WHERE kolumna = ? ");
sql.bind_param(1,mojParametr);
```

C#

W C# w charakterze symbolu zastępczego nie stosujemy znaku ?. W zamian nadajemy mu nazwę i umieszczamy przed nim znak @:

```
SqlCommand cmd = new SqlCommand(
    "SELECT kolumna FROM tabela WHERE kolumna = @param1");
cmd.Parameters.Add("@param1", mojParametr);
```

PHP

W PHP stosowane symbole zastępcze zależą od bibliotek wykorzystywanych przy nawiązywaniu połączenia z bazą danych. Do dyspozycji mamy biblioteki owijające, w tym ADOdb (<http://adodb.sourceforge.net>), które mogą nam znacznie ułatwić życie. Wyrażenie wykorzystujące ADOdb ma następującą postać:

```
$DB->Execute("SELECT kolumna FROM tabela WHERE kolumna = ?",
    array(mojParametr));
```

Ten sam efekt bez posługiwania się ADOdb można osiągnąć za pomocą specjalistycznych funkcji, takich jak `mysql_stmt_bind_param` lub `oci_bind_by_name`.

SPOSÓB
9.

Filtrowanie kolumn indeksowanych

Filtrowanie kwerend zwiększa wydajność. Dodatkową poprawę w tym zakresie zapewnia wykorzystanie kolumn indeksowanych.

Kwerendy mogą zwracać wszystkie wiersze i wszystkie kolumny zawarte w tabelach. Ale co zrobić, jeśli chcemy uzyskać dostęp jedynie do kilku kolumn lub wierszy? Przesyłanie zbędnych danych jest marnotrawstwem zasobów systemowych. Tworząc zapytania uwzględniające jedynie potrzebne nam informacje, *odfiltrowujemy* zbędne kolumny i wiersze. Aby przeprowadzić filtrowanie kolumn, należy upewnić się, czy zapytanie nie obejmuje niepotrzebnych fragmentów tabeli (na przykład nie zawiera znaku `*` w wyrażeniu `SELECT`). Wiersze filtrować można za pomocą warunku `WHERE`, jak również innych wyrażen (na przykład `HAVING`).

Zdefiniowanie klucza głównego tworzy indeks powiązanych z nim kolumn. Dzięki temu wyszukiwanie danych jest znacznie szybsze niż w przypadku braku indeksu. Indeks jest wykorzystywany do zapewnienia unikalności klucza, a unikalność ta jest warunkiem jego ważności. Indeks daje również dodatkowe korzyści przy stosowaniu poleceń `JOIN`.

Filtrowanie danych nieposiadających indeksu może powodować znaczne problemy związane z wydajnością. Indeks skraca czas wyszukiwania i może być wykorzystany przez optymalizatory zapytań podczas wstępnego filtrowania danych (w przypadku braku indeksu jest ono oparte na przeszukiwaniu całej tabeli danych). W zależności od przetwarzanej kwerendy program optymalizujący może nawet wykorzystać indeks do całej operacji.



Jeśli do uzyskania wyników potrzebny jest jedynie indeks (nie ma konieczności analizowania bazy danych), indeks taki nazywamy *indeksem pokrywającym* — „pokrywa” on całość kwerendy.

Rozważmy teraz bazę danych zawierającą strony. Załóżmy, że w tabeli przechowywane są nazwy i zawartość stron, jak również ich poprzednie wersje, zapisywane w celu kontroli wersji (patrz tabela 2.2).

Tabela 2.2. Tabela „strona”

zawartosc	nazwaStrony	nazwaUzytkownika	ostatniaModyfikacja	numerWersji
<code>hello</code>	index.html	gordon	2006-03-01	1
<code><h1>Hia</h1></code>	index.html	gordon	2006-10-10	2
<code><p>page2</p></code>	p2.html	andrew	2006-02-05	1
<code><h1>Indeks</h1></code>	contents.html	gordon	2006-02-05	1

Możemy teraz obsługiwać zmiany wprowadzane na stronie przez różnych użytkowników i prowadzić ich rejestr. Poniższa kwerenda umożliwia pobranie z bazy bieżącej wersji strony *index.html*:

```
SELECT nazwaStrony, zawartosc
FROM strona x
WHERE nazwaStrony = 'index.html'
AND numerWersji = (
    SELECT MAX (y.numerWersji) FROM strona y
    WHERE y.nazwaStrony = 'index.html'
);
```

Zaprezentowana kwerenda jest dość efektywna. Indeks przypisany kolumnie `nazwaStrony` umożliwi szybkie wyszukanie nazwy `index.html` — bez konieczności analizowania wszystkich wierszy. Tabela posiada klucz główny (`nazwaStrony`, `numerWersji`), który — mimo iż nie opiera się wyłącznie na kolumnie `nazwaStrony` — powinien działać jeszcze lepiej, ponieważ zawiera wszystkie dane objęte kwerendą. System może wykorzystywać tego typu złożone indeksy, o ile jest w stanie wyszukać indeksowane dane przez odczytywanie zawartości tabeli od lewej do prawej, bez korzystania z niepotrzebnych kolumn. Proces ten określany jest nazwą *częściowego dopasowywania indeksów*.

Jeśli tabela posiada indeks (`numerWersji`, `nazwaStrony`, `ostatniaModyfikacja`), uzyskanie indeksu dla kolumny `numerWersji` nie stanowi problemu. Jeśli jednak potrzebny jest indeks kolumny `nazwaStrony`, rozwiązanie to okaże się nieefektywne, ponieważ na początku listy wymieniona jest kolumna `numerWersji`. Warto zatem dokładnie przemyśleć kolejność elementów tworzonego indeksu złożonego lub złożonego klucza podstawowego. Każda z często wykorzystywanych kolumn powinna pojawić się jako pierwsza przynajmniej raz.

Załóżmy, że do wyboru mamy indeksy (`nazwaStrony`, `numerWersji`) i (`numerWersji`, `nazwaStrony`). Wiemy, że w kolumnie `nazwaStrony` znajdują się tysiące różnych rekordów, a w kolumnie `numerWersji` — jedynie kilka wersji strony. W takiej sytuacji zdecydowanie należy wybrać pierwszą możliwość. Umieszczanie dyskryminatora niskiej rangi na pierwszym miejscu w indeksie nie jest najlepszym pomysłem. Jeszcze gorszym rozwiązaniem jest tworzenie osobnego indeksu dla kolumny `numerWersji`.

Zastosowanie indeksów przyspiesza wykonywanie kwerend wykorzystujących warunki złączeń i wyrażenia `WHERE` zawierające znaki `=` oraz `>`. Warto zatem rozważyć dodanie indeksów do kolumn uwzględnionych w tego typu zapytaniach. Przeanalizujemy poniższą kwerendę dotyczącą tabeli `t`:

```
SELECT z
FROM t
WHERE x = 6 AND y > 7;
```

Przedstawiona kwerenda przeprowadza filtrowanie w oparciu o kolumny `x` i `y`. Jeśli z uwagi na jej częste wykorzystywanie chcielibyśmy poprawić jej wydajność, najlepszym rozwiązaniem byłoby opracowanie odpowiednich indeksów. Indeksy `x` i `y` stworzyć można w następujący sposób:

```
CREATE INDEX ind_1 ON t (x);
CREATE INDEX ind_2 ON t (y);
```

Utworzenie dwóch indeksów nie jest optymalne, jeśli w całej aplikacji wykorzystywać będziemy jedynie powyższe zapytanie wykorzystujące jednocześnie i kolumnę x i kolumnę y . W takim przypadku idealnym rozwiązaniem jest przypisanie x roli pierwszego dyskryminatora i umieszczenie y po nim:

```
CREATE INDEX ind_1 ON t (x,y);
```

Oczywiście optymalizator może zignorować utworzone przez nas indeksy, jeśli uzna, że takie rozwiązanie przyspieszy wykonywanie kwerendy. Na ogół jednak indeksy zwiększają znacznie efektywność tworzonego kodu.


**SPOSÓB
10.**

Konwertowanie podzapytań na wyrażenia JOIN

Czasami chcemy pobrać dane z jednej tabeli, wykorzystać je do przetworzenia danych w innej, a uzyskane rezultaty wykorzystać w jeszcze innej tabeli. Kuszącym rozwiązaniem wydaje się utworzenie trzech osobnych zapytań, jednak najlepiej wykonać wszystkie wymienione operacje w ramach jednego wyrażenia SQL.

Rozważmy bazę danych zawierającą stanowiska pracowników. Do każdego stanowiska przypisana jest ranga, od której zależy miesięczna płaca. Całość przedstawiają tabele 2.3, 2.4 i 2.5.

Tabela 2.3. Tabela „etaty”

Pracownik	Stanowisko
Grzegorz Nowak	Wykładowca
Andrzej Kowalski	Nauczyciel
Marcin Maliniak	Technik

Tabela 2.4. Tabela „rangy”

Stanowisko	Ranga
Wykładowca	WYK1
Nauczyciel	WYK2
Technik	TECH1

Tabela 2.5. Tabela „pensje”

Ranga	Płaca
WYK1	2000,00
WYK2	3000,00
TECH1	5000,00
TECH2	6000,00

Wyznaczenie wysokości pensji Andrzeja Kowalskiego wymaga wykonania trzech czynności. Najpierw należy określić jego stanowisko:

```
mysql> SELECT stanowisko FROM etaty WHERE pracownik = 'Andrzej Kowalski';
+-----+
| stanowisko |
+-----+
| Nauczyciel |
+-----+
```

Następnie musimy wyznaczyć rangę przypisaną do danego stanowiska:

```
mysql> SELECT ranga FROM rangi WHERE stanowisko = 'Nauczyciel';
+-----+
| ranga |
+-----+
| WYK2  |
+-----+
```

Na koniec należy odszukać wysokość pensji odpowiadającą randze WYK2:

```
mysql> SELECT placa FROM pensje WHERE ranga = 'WYK2';
+-----+
| placa |
+-----+
| 3000,00 |
+-----+
```

Nie jest to zbyt efektywne rozwiązanie, ponieważ wymaga przesłania do bazy trzech osobnych zapytań i przetwarzania uzyskanych w międzyczasie wyników. Jeśli w trakcie całego procesu zawartość bazy ulegnie zmianie, moglibyśmy otrzymać niepoprawną odpowiedź lub nawet komunikat o błędzie. Łączenie kwerend bywa stresujące. Zestresowani programiści często używają podzapytań:

```
mysql> SELECT placa FROM pensje WHERE ranga =
-> (SELECT ranga FROM rangi WHERE stanowisko =
-> (SELECT stanowisko FROM etaty WHERE pracownik = 'Andrzej
Kowalski')));
+-----+
| placa |
+-----+
| 3000,00 |
+-----+
```

Przedstawiony kod eliminuje pewne problemy dzięki sprowadzeniu całej operacji do jednej kwerendy, podzapytania zmniejszają jednak szybkość wykonania polecenia. Jeśli wyrażenia zawarte w podzapytaniach nie zawierają funkcji zagregowanych (takich jak MAX()), najprawdopodobniej w ogóle nie ma konieczności używania podzapytań. W zamian wystarczy zastosować polecenie JOIN. Aby przekształcić kwerendę opartą na podzapytaniach w wyrażenie JOIN, należy wykonać następujące czynności:

1. Oznaczamy wszystkie kolumny nazwą tabeli, w której są one zawarte.
2. Jeśli odwołanie do tej samej tabeli znajduje się w dwóch różnych wyrażeniach FROM, należy zastosować nazwy zastępcze (w tym przykładzie nie jest to konieczne).
3. Wszystkie warunki FROM łączymy razem, tworząc pojedyncze wyrażenie FROM.
4. Usuwamy wszystkie wystąpienia (SELECT.
5. Zamieniamy drugie słowo kluczowe WHERE na AND.

Oto etap pośredni:

```
SELECT pensje.placa FROM pensje, rangi, etaty WHERE pensje.ranga=  
(SELECT rangi.ranga from rangi AND rangi.stanowisko=  
(SELECT etaty.stanowisko from etaty AND etaty.pracownik = 'Andrzej  
Kowalski' )>>
```

Końcowa postać przedstawia się następująco:

```
SELECT placa FROM pensje, rangi, etaty  
WHERE pensje.ranga=rangi.ranga  
AND rangi.stanowisko=etaty.stanowisko  
AND etaty.pracownik = 'Andrzej Kowalski';
```

Innym rozwiązaniem jest zamiana warunków zdefiniowanych wewnątrz podzapytań na warunki JOIN ON:

```
SELECT placa  
FROM pensje JOIN rangi ON (pensje.ranga=rangi.ranga)  
JOIN etaty ON (rangi.stanowisko=etaty.stanowisko)  
WHERE etaty.pracownik = 'Andrzej Kowalski';
```

Wyszukiwanie danych spoza bazy

Wielu programistów potrafi zastępować podzapytania wyrażeniami JOIN, prawdziwym wyzwaniem są natomiast operacje polegające na wyszukiwaniu nieistniejących danych. W jaki sposób można na przykład stwierdzić, czy w bazie znajdują się rangi nieprzypisane do żadnego stanowiska? Kosztowne obliczeniowo rozwiązanie zakłada wyszukanie wszystkich rang w tabeli pensje, a następnie sprawdzenie każdej z nich w tabeli rangi. Nie trzeba chyba rozwodzić się nad wydajnością tej techniki. Alternatywną metodą jest wykorzystanie podzapytania zawierającego warunek NOT IN, to jednak również nie oszczędza zasobów systemowych:

```
mysql> SELECT pensje.ranga FROM pensje  
-> WHERE ranga NOT IN (SELECT ranga FROM rangi);  
+-----+  
| ranga |  
+-----+  
| TECH2 |  
+-----+
```

Spadki wydajności wynikać mogą z konieczności utworzenia tymczasowej tabeli pośredniej, koniecznej do wykonania podzapytania. Wspomniana tabela wykorzystywana jest następnie do przeprowadzenia kwerendy zewnętrznej. Podczas tworzenia tabeli tymczasowej nie zostaną wykorzystane indeksy przypisane do tabeli pensje. W rezultacie podczas wykonywania operacji przeszukana będzie cała tabela tymczasowa.

Jest to odwrotność zaprezentowanego wcześniej problemu osadzonych podzapytań. W tym przypadku poszukujemy niepasujących wierszy w tabelach. Zastosowanie wcześniejszej techniki z użyciem operatora != zamiast = spowoduje jedynie wielki bałagan i nie przybliży nas do rozwiązania. W zamian należy wykorzystać wyrażenie OUTER JOIN. Za

jego pomocą łączymy wszystkie tabele zawarte we frazie FROM. Poszukujemy elementów tabeli pensje niewymienionych w tabeli rangi. Dzięki zastosowaniu wyrażenia OUTER JOIN niepasujące wiersze będą miały wartość NULL w polu rangi.ranga:

```
mysql> SELECT pensje.ranga
-> FROM pensje LEFT OUTER JOIN rangi ON (pensje.ranga = rangi.ranga)
-> WHERE rangi.ranga IS NULL;
+-----+
| ranga |
+-----+
| TECH2 |
+-----+
```

Technikę tę wykorzystać można do eliminowania wyrażeń EXISTS i NOT EXISTS. Wyeliminowanie podzapytań ułatwia optymalizatorowi wykorzystywanie indeksów.



SPOSÓB

11.

Konwertowanie zagregowanych podzapytań na wyrażenia JOIN

Podzapytania niezawierające funkcji zagregowanych można zastępować wyrażeniami JOIN oraz OUTER JOIN. A co, jeśli podzapytania zawierają wspomniane funkcje?

Niektóre podzapytania łatwo wyeliminować [Sposób 10.], inne przysparzają pod tym względem nieco trudności. Przeanalizujemy tabelę 2.6 zawierającą dane dotyczące zamówień.

Tabela 2.6. Tabela „zamowienia”

Klient	Kiedy	Ilosc_towaru
Krzysiek	2006-10-10	5
Krzysiek	2006-10-11	3
Krzysiek	2006-10-12	1
Wojtek	2006-10-10	7

Załóżmy, że musimy znaleźć dni, w których poszczególni klienci zakupili najwięcej towaru:

```
SELECT klient,kiedy,ilosc_towaru
FROM zamowienia o1
WHERE o1.kiedy = (
  SELECT MAX(kiedy)
  FROM zamowienia o2
  WHERE o1.klient = o2.klient
);
```

Wykonanie zaprezentowanego powyżej kodu będzie dość powolne, ponieważ wymaga przeszukania wszystkich wierszy tabeli zamowienia. Ponadto stare wersje MySQL nie obsługują podzapytań. W celu ich wyeliminowania możemy posłużyć się warunkiem HAVING oraz złączeniem tabeli z nią samą:

```
SELECT o1.klient,o1.kiedy,o1.ilosc_towaru
FROM zamowienia o1 JOIN zamowienia o2 on (o1.klient = o2.klient)
GROUP BY o1.klient,o1.kiedy,o1.ilosc_towaru
HAVING o1.kiedy = MAX(o2.kiedy)
```


Oto otrzymany wynik:

```

+-----+-----+-----+
| klient | kiedy      | ilosc_towaru |
+-----+-----+-----+
| Wojtek | 2006-10-10 |              7 |
| Krzysiek | 2006-10-12 |              1 |
+-----+-----+-----+
2 rows in set (0.00 sec)

```

Technika ta sprawdza się dla wszystkich funkcji zagregowanych.

SPOSÓB
12.

Upraszczenie skomplikowanych poleceń UPDATE

Polecenie UPDATE daje możliwość przeprowadzania skomplikowanych obliczeń. Dzięki temu można uniknąć konieczności stosowania kursora lub wykonywania wspomnianych obliczeń poza bazą danych.

Przykład polecenia UPDATE przytaczany w wielu książkach opisujących podstawy SQL prezentuje prostą operację, polegającą na podniesieniu pensji wszystkich pracowników o 100 zł.

```

UPDATE pracownik
SET placa = placa +100;

```

Jest to z pewnością proste wyrażenie — jednak może się ono okazać zbyt proste, by miało jakkolwiek wartość praktyczną. Załóżmy, iż negocjacje w sprawie płac zakończyły się bardziej złożonymi warunkami, których wprowadzenie wymaga dostępu do innych tabel w bazie.

Pracownicy posiadający czyste konto wykroczeń dyscyplinarnych otrzymają 100 zł podwyżki, natomiast ci, którzy dopuścili się tylko jednego wykroczenia, będą otrzymywać wypłatę taką samą jak dotąd. Pensja pracowników z dwoma i więcej wykroczeniami na koncie zostanie obniżona o 100 zł. Dane dotyczące pracowników oraz zachowywanej przez nich dyscypliny przechowywane są w tabelach `pracownik` oraz `dyscyplina`:

```

mysql> SELECT * FROM pracownik;
+-----+-----+-----+
| id | imie   | placa  |
+-----+-----+-----+
| 1  | Janusz | 5000.00 |
| 2  | Marcin | 5000.00 |
| 3  | Marian | 5000.00 |
+-----+-----+-----+
mysql> SELECT * FROM dyscyplina;
+-----+-----+
| kiedy      | prac |
+-----+-----+
| 2006-05-20 | 1 |
| 2006-05-21 | 1 |
| 2006-05-22 | 3 |
+-----+-----+

```

Moglibyśmy napisać skomplikowane polecenie UPDATE aktualizujące tabelę `pracownik` na podstawie odwołań do tabeli `dyscyplina`, łatwiej jednak będzie podzielić cały proces na dwa etapy. Najpierw przygotujemy widok danych obliczający nowe wartości płac, a następnie wprowadzimy je do bazy za pomocą polecenia UPDATE.

Widok danych `nowePlace` posiada dwie kolumny: klucz podstawowy aktualizowanej tabeli (`pracownik`) oraz nowe wartości `plac`. Ich zawartość możemy obejrzeć przed wykonaniem kwerendy `UPDATE`.

Widok zawierający nowe płace dla każdego z pracowników definiujemy w następujący sposób:

```
mysql> CREATE VIEW nowePlace AS
-> SELECT id, CASE WHEN COUNT (prac) = 0 THEN placa +100
->                WHEN COUNT (prac) > 1 THEN placa -100
->                ELSE placa
->            END AS v
-> FROM pracownik LEFT JOIN dyscyplina ON (id=prac)
-> GROUP BY id,placa;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM nowePlace;
+----+-----+
| id | v      |
+----+-----+
| 1  | 4900.00 |
| 2  | 5100.00 |
| 3  | 5000.00 |
+----+-----+
```

Widok danych można obejrzeć w celu przeanalizowania nowych płac, jednak tabela `pracownik` nie została jeszcze zaktualizowana. Dobrym pomysłem byłoby zlecenie innemu pracownikowi sprawdzenia nowych wartości przed wprowadzeniem ostatecznych zmian.

Nowe wartości `plac` umieszczamy w bazie za pomocą pojedynczego wyrażenia `UPDATE`. Jest to ważne, ponieważ musimy mieć pewność, że sprawdzone dane zawarte w widoku zgadzają się z danymi wprowadzanymi do bazy.

```
mysql> UPDATE pracownik
-> SET placa = (SELECT v FROM nowePlace
->             WHERE nowePlace.id=pracownik.id)
-> WHERE id IN (SELECT id FROM nowePlace);
Query OK, 2 rows affected (0.01 sec)
Rows matched: 3 Changed: 2 Warnings: 0
```

```
mysql> SELECT * FROM pracownik;
+----+-----+-----+
| id | imie  | placa |
+----+-----+-----+
| 1  | Janusz | 4900.00 |
| 2  | Marcin | 5100.00 |
| 3  | Marian | 5000.00 |
+----+-----+-----+
3 rows in set (0.00 sec)
```

Wykorzystywanie kursora

Podczas wykonywania kilku operacji podobnych do przedstawionej powyżej kuszące wydaje się wykorzystanie kursora opisanego w sposobie 2. Zaletą polecenia `UPDATE` jest

jego zwięzłość, szybkość i niepodzielność. Przejrzystość kodu zależy od stylu programowania, do którego jesteście przyzwyczajeni. Język SQL stosowany bez kursorów znakomicie pasuje do stylu deklaratywnego.

Korzystanie z widoków danych

Operację aktualizacji można przeprowadzić bez wcześniejszego tworzenia widoku danych, utrudnia to jednak wyświetlenie podglądu końcowych wyników. Inną wadą skomplikowanych wyrażeń UPDATE jest niewygodny proces debugowania: każdy test polecenia powoduje zmianę zawartości bazy danych. Przed przeprowadzeniem kolejnej próby konieczne jest przywrócenie jej do poprzedniego stanu. Dzięki zastosowaniu widoku danych możemy sprawdzić rezultaty kwerendy bez wprowadzania zmian do bazy.

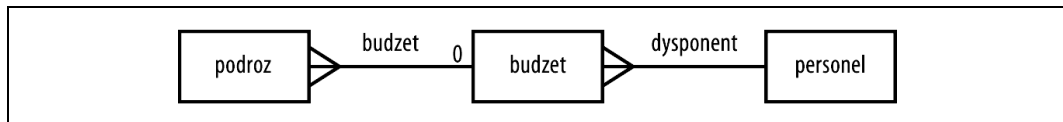


SPOSÓB 13.

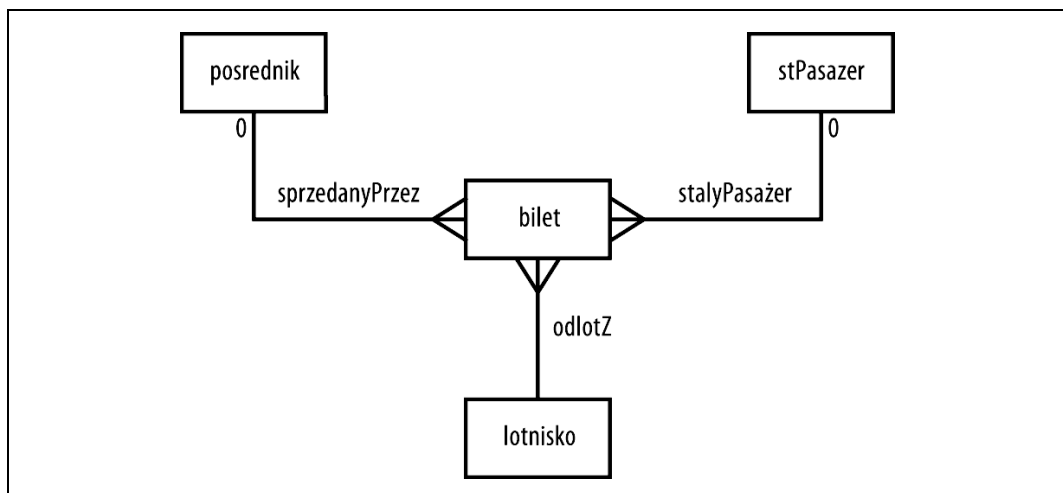
Dopasowywanie złączeń do relacji

Kiedy relacja między tabelami jest opcjonalna, najlepiej zastosować polecenie OUTER JOIN. Jeśli musimy je zastosować w trakcie wprowadzania wielu zmian, czasem konieczna jest zamiana wszystkich wyrażeń INNER JOIN na wyrażenia OUTER JOIN.

Wyróżniamy dwa wzorce złączeń: łańcuch i gwiazdę (patrz rysunki 2.1 i 2.2). Ich opis przedstawiony jest poniżej.



Rysunek 2.1. Wzorzec łańcucha



Rysunek 2.2. Wzorzec gwiazdy

Łańcuch

W przedstawionym przykładzie łańcucha znajdują się dwa odwołania. Odwołanie z tabeli `podroz` (patrz tabela 2.7) do tabeli `budzet` (patrz tabela 2.8) jest opcjonalne — użytkownicy mogą wstawiać w polu `budzet` w tabeli `podroz` wartość `NULL`. Niezbędne jest połączenie między tabelami `budzet` i `personel` (patrz tabela 2.9) — każdy wiersz w pierwszej z nich musi posiadać odpowiadającą mu wartość w polu `dysponent`. Kwerendy oparte na tabelach `podroz` oraz `budzet` wykorzystują zatem wyrażenie `OUTER JOIN`, natomiast kwerendy związane z tabelami `budzet` i `personel` — wyrażenie `INNER JOIN`.

Tabela 2.7. Tabela „podroz”

podrozID	opis	budzet
POD01	Sycylia	NULL
POD02	Egipt	CT22

Tabela 2.8. Tabela „budzet”

budzetID	opis	dysponent (NOT NULL)
CT22	Mesa oficerska	ST02

Tabela 2.9. Tabela „personel”

personelID	imie	ranga
ST01	Grzegorz	Kapitan
ST02	Janusz	Porucznik

Aby wyświetlić listę wszystkich podróży wraz ze szczegółami dotyczącymi ich budżetu, musimy zastosować wyrażenie `LEFT OUTER JOIN`. W ten sposób uwzględnione zostaną również podróże nieposiadające przypisanego budżetu:

```
mysql> SELECT podrozID, podroz.opis, budzet.opis
-> FROM podroz LEFT OUTER JOIN budzet ON
(podroz.budzet=budzet.budzetID);
+-----+-----+-----+
| podrozID | opis   | opis   |
+-----+-----+-----+
| POD01    | Sycylia | NULL   |
| POD02    | Egipt  | Mesa oficerska |
+-----+-----+-----+
```



Możemy również zmienić frazę `FROM`, na przykład: `FROM budzet RIGHT OUTER JOIN podroz ON podroz.budzet=budzet.budzetId`.

Aby uwzględnić imię dysponenta budżetu, musimy dołączyć również tabelę `personel`. Wartość `NULL` dla pola `dysponent` jest niedozwolona, co może prowadzić do wniosku, iż wyrażenie `INNER JOIN` da poprawne rezultaty. Niestety, jest to błędny wniosek:

```
mysql> SELECT podrozID, podroz.opis,budzet.opis, imie
-> FROM podroz LEFT OUTER JOIN budzet ON (podroz.budzet=budzet.budzetID)
-> INNER JOIN personel ON (dysponent=personelID);
+-----+-----+-----+-----+
| podrozID | opis   | opis           | imie   |
+-----+-----+-----+-----+
| POD02    | Egipt  | Mesa oficerska | Janusz |
+-----+-----+-----+-----+
```

Łańcuch złączeń obliczany jest od lewej do prawej, przez co rezultat polecenia LEFT JOIN z pierwszej kwerendy jest dołączany do tabeli budzet za pomocą wyrażenia INNER JOIN. Wiersz zawierający dane podróży POD01 nie jest wyświetlany, ponieważ w polu budzet posiada wartość NULL. Można rzecz jasna posłużyć się nawiasami lub zmienić kolejność wyrażeń JOIN tak, aby INNER JOIN było przetwarzane jako pierwsze, jednak z punktu widzenia optymalizatora najlepszym rozwiązaniem jest dalsze stosowanie polecenia LEFT OUTER JOIN:

```
mysql> SELECT podrozID, podroz.opis,budzet.opis, imie
-> FROM podroz LEFT OUTER JOIN budzet ON (podroz.budzet=budzet.budzetID)
-> LEFT OUTER JOIN personel ON (dysponent=personelID);
+-----+-----+-----+-----+
| podrozID | opis   | opis           | imie   |
+-----+-----+-----+-----+
| POD01    | Sycylia | NULL           | NULL   |
| POD02    | Egipt  | Mesa oficerska | Janusz |
+-----+-----+-----+-----+
```

Gwiazda

Cechą charakterystyczną dla wzorca gwiazdy jest jedna, centralna tabela. Wszystkie pozostałe tabele są z nią połączone za pomocą odpowiednich relacji. Relacje te mogą być opcjonalne lub obowiązkowe.

W poniższym przykładzie rolę tabeli centralnej spełnia bilet. Wszystkie bilety przypisane są do jednego lotniska, ale tylko niektóre sprzedane zostały przez pośrednika i jedynie część z nich zakupiona została przez osoby posiadające konto stałego pasażera (stPasazer):

```
CREATE TABLE bilet
(biletid CHAR(4) PRIMARY KEY
, posrednik CHAR(4) NULL
, odlotZ CHAR(3) NOT NULL
, stPasazer CHAR(4) NULL
, FOREIGN KEY (posrednik) REFERENCES posrednik(id)
, FOREIGN KEY (odlotZ) REFERENCES lotnisko(id)
, FOREIGN KEY (stPasazer) REFERENCES stPasazer(id)
);
```

W przypadku schematu gwiazdy wyrażeń LEFT OUTER JOIN należy używać jedynie w odniesieniu do tabel, które tego wymagają. Kolejność złączeń nie ma znaczenia.

```
mysql> SELECT lotnisko.nazwa AS lotnisko,
-> posrednik.nazwa AS posrednik,
-> stPasazer.imieNazwisko stPasazer
-> FROM bilet LEFT OUTER JOIN posrednik ON (posrednik = posrednik.id)
```

```
->          INNER JOIN lotnisko ON (odlotZ = lotnisko.id)
->          LEFT OUTER JOIN stPasazer ON (stPasazer = stPasazer.id);

-----+-----+-----+
| lotnisko | posrednik | stPasazer |
-----+-----+-----+
| Warszawa | NULL      | NULL      |
| Warszawa | Sigma Travel | NULL      |
| Katowice | Sigma Travel | Marcin Karbowski |
| Katowice | NULL      | Wojciech Pajak |
-----+-----+-----+
```

Ponieważ wszystkie tabele łączą się ze sobą poprzez tabelę centralną, to ona dyktuje wymagania dotyczące zastosowanych wyrażeń JOIN. Wartość NULL w polu posrednik nie wpływa na relacje między tabelami stPasazer i lotnisko a tabelą bilet. Podobnie wartość NULL w polu stPasazer nie ma wpływu na relację łączącą tabele lotnisko i posrednik z tabelą bilet.



SPOSÓB 14.

Tworzenie kombinacji

Zastosowanie polecenia JOIN bez dodatkowych warunków powoduje połączenie każdego wiersza jednej tabeli z każdym wierszem innej. Utworzone zatem zostają wszystkie możliwe kombinacje wierszy. Bardzo często jest to skutek pomyłki, ale bywa również użyteczne.

Kwerendy wykorzystujące polecenie CROSS JOIN pojawiają się rzadko, warto jednak wiedzieć, jak się nimi posługiwać — na wypadek, gdyby ich zastosowanie okazało się niezbędne. Jeśli tabela wykorzystywana jest więcej niż raz, mówimy o *złączeniu tabeli z nią samą* (ang. *self-join*). Jeśli złączenie dwóch wystąpień tej samej tabeli nie jest obwarowane żadnymi warunkami, otrzymamy w rezultacie wszystkie możliwe kombinacje jej wierszy. Przeprowadzając złączenie na tabeli zawierającej dane A w pierwszym wierszu i dane B w wierszu drugim, otrzymamy: ('A', 'A'), ('A', 'B'), ('B', 'A') oraz ('B', 'B'). Lista uwzględnia zatem każdą kombinację wierszy.

Załóżmy, że w lokalnej lidze grają cztery drużyny piłkarskie. Każda z nich rozegrać ma dwa mecze z wszystkimi pozostałymi — raz u siebie i raz na wyjeździe. Całość przedstawiona została w tabelach 2.10 i 2.11.

Tabela 2.10. Tabela „drużyny”

NazwaDrużyny

Lwy
Tygrysy
Pumy
Ropuchy

Tabela 2.11. Tabela „wyniki”

Gospodarze	Goscie	bramkiGospodarzy	bramkiGosci
Lwy	Pumy	1	4
Ropuchy	Tygrysy	3	5
Pumy	Tygrysy	0	0

Musimy napisać kwerendę prezentującą wyniki wszystkich meczy. W celu uzyskania wszystkich możliwych kombinacji posługujemy się poleceniem `CROSS JOIN`:

```
mysql> SELECT gospodarze.nazwaDrużyny Gospodarze, goscie.nazwaDrużyny Goscie
-> FROM drużyny gospodarze CROSS JOIN drużyny goscie
-> ;
```

Gospodarze	Goscie
Lwy	Lwy
Tygrysy	Lwy
Pumy	Lwy
Ropuchy	Lwy
Lwy	Tygrysy
Tygrysy	Tygrysy
Pumy	Tygrysy
Ropuchy	Tygrysy
Lwy	Pumy
Tygrysy	Pumy
Pumy	Pumy
Ropuchy	Pumy
Lwy	Ropuchy
Tygrysy	Ropuchy
Pumy	Ropuchy
Ropuchy	Ropuchy

```
16 rows in set (0.00 sec)
```

Teraz trzeba wprowadzić warunek uniemożliwiający łączenie drużyny z nią samą:

```
mysql> SELECT gospodarze.nazwaDrużyny Gospodarze, goscie.nazwaDrużyny Goscie
-> FROM drużyny gospodarze CROSS JOIN drużyny goscie
-> WHERE gospodarze.nazwaDrużyny != goscie.nazwaDrużyny
-> ;
```

Gospodarze	Goscie
Tygrysy	Lwy
Pumy	Lwy
Ropuchy	Lwy
Lwy	Tygrysy
Pumy	Tygrysy
Ropuchy	Tygrysy
Lwy	Pumy
Tygrysy	Pumy
Ropuchy	Pumy
Lwy	Ropuchy
Tygrysy	Ropuchy
Pumy	Ropuchy

```
12 rows in set (0.00 sec)
```

Aby wyświetlić wyniki rozegranych meczy, pozostawiając puste miejsca w przypadku meczy jeszcze nierozegranych, należy posłużyć się poleceniem `LEFT OUTER JOIN` ([Sposób 13.] i [Sposób 26.]) w celu połączenia iloczynu zbiorów z tabelą wyniki.

```
mysql> SELECT gospodarze.nazwaDrużyny Gospodarze, goscie.nazwaDrużyny Goscie,
-> wyniki.bramkiGospodarzy, wyniki.bramkiGosci
-> FROM drużyny gospodarze CROSS JOIN drużyny goscie LEFT JOIN wyniki on
-> (gospodarze.nazwaDrużyny = wyniki.gospodarze)
-> AND wyniki.goscie = goscie.nazwaDrużyny)
```

```
-> WHERE gospodarze.nazwaDruzyny != goscie.nazwaDruzyny;
+-----+-----+-----+-----+
| Gospodarze | Goscie | bramkiGospodarzy | BramkiGosci |
+-----+-----+-----+-----+
| Tygrysy    | Lwy     | NULL              | NULL        |
| Pумы       | Lwy     | NULL              | NULL        |
| Ropuchy    | Lwy     | NULL              | NULL        |
| Lwy        | Tygrysy | NULL              | NULL        |
| Pумы       | Tygrysy | 0                  | 0           |
| Ropuchy    | Tygrysy | 3                  | 5           |
| Lwy        | Pумы    | 1                  | 4           |
| Tygrysy    | Pумы    | NULL              | NULL        |
| Ropuchy    | Pумы    | NULL              | NULL        |
| Lwy        | Ropuchy | NULL              | NULL        |
| Tygrysy    | Ropuchy | NULL              | NULL        |
| Pумы       | Ropuchy | NULL              | NULL        |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)
```