O'REILLY®

Wydanie II

MySQL

Jak zaprojektować i wdrożyć wydajną bazę danych

Vinicius M. Grippa
Sergey Kuzmichev

Helion

Tytuł oryginału: Learning MySQL: Get a Handle on Your Data, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-8960-1

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/mysjz2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Modelowanie i projektowanie bazy danych

Podczas implementowania nowej bazy danych bardzo łatwo można wpaść w pułapkę związaną z szybkim przygotowaniem i uruchomieniem rozwiązania, bez poświęcenia odpowiedniej ilości czasu i wysiłku na przemyślenie projektu bazy danych. Taka beztroska bardzo często prowadzi do kosztownych operacji związanych z ponownym projektowaniem i implementowaniem rozwiązania. Projektowanie bazy danych można porównać do przygotowywania projektu domu — za kiepski pomysł uznaje się rozpoczęcie budowy domu, dla którego nie istnieją dokładnie opracowane plany. Dobry projekt pozwala na rozbudowę domu bez konieczności jego rozbierania i rozpoczynania budowy od początku. Jak się wkrótce przekonasz, kiepski projekt ma bezpośrednie przełożenie na niezadowalającą wydajność działania bazy danych.

Jak nie tworzyć projektu bazy danych

Projektowanie bazy danych prawdopodobnie nie zalicza się do najbardziej ekscytujących zadań na świecie, choć zdecydowanie jest jednym z tych najważniejszych. Jednak zanim przejdziemy do omawiania procesu projektowania, warto, żebyś zapoznał się z przykładowym projektem bazy danych.

Załóżmy, że chcemy utworzyć bazę danych przeznaczoną do przechowywania ocen uzyskanych przez studentów informatyki na jednym z uniwersytetów. Można utworzyć tabelę `Student_Grades` do przechowywania ocen uzyskanych przez poszczególnych studentów uczestniczących w zajęciach. Taka tabela będzie miała kolumny przeznaczone dla imion, nazwiska, nazwy przedmiotu i oceny wyrażonej procentowo (tutaj to kolumna `Pctg`). W tabeli znajdują się oddzielne rekordy dla poszczególnych studentów i przedmiotów, na które uczęszczają:

```
+------------+---------+----------------------+------+
| GivenNames | Surname | CourseName           | Pctg |
+------------+---------+----------------------+------+
| John Paul  | Bloggs  | Data Science         |   72 |
| Sarah      | Doe     | Programming 1        |   87 |
| John Paul  | Bloggs  | Computing Mathematics |   43 |
| John Paul  | Bloggs  | Computing Mathematics |   65 |
| Sarah      | Doe     | Data Science         |   65 |
| Susan      | Smith   | Computing Mathematics |   75 |
| Susan      | Smith   | Programming 1        |   55 |
| Susan      | Smith   | Computing Mathematics |   80 |
+------------+---------+----------------------+------+
```

Przedstawiona tutaj lista jest elegancka i zwięzła, zapewnia łatwy dostęp do ocen poszczególnych studentów oraz wyglądem przypomina arkusz kalkulacyjny. Jednak może się zdarzyć, że na uniwersytecie będzie uczył się więcej niż jeden student o danym imieniu i nazwisku. W naszych przykładowych danych mamy dwa rekordy studentek `Susan Smith` uczestniczących w zajęciach `Computing Mathematics`. Jak można ustalić, która z nich uzyskała wynik 75%, a która 80%? Powszechnym sposobem odróżniania powielających się rekordów jest przypisanie im unikatowej wartości liczbowej. W kolejnym fragmencie kodu każdemu studentowi został przypisany unikatowy identyfikator, `StudentID`:

```
+------------+------------+---------+----------------------+------+
| StudentID  | GivenNames | Surname | CourseName           | Pctg |
+------------+------------+---------+----------------------+------+
| 12345678   | John Paul  | Bloggs  | Data Science         |   72 |
| 12345121   | Sarah      | Doe     | Programming 1        |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics|   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics|   65 |
| 12345121   | Sarah      | Doe     | Data Science         |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics|   75 |
| 12345876   | Susan      | Smith   | Programming 1        |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics|   80 |
+------------+------------+---------+----------------------+------+
```

Dzięki tej zmianie wiemy, która Susan Smith uzyskała wynik 80% — to studentka o identyfikatorze 12345303.

Niestety mamy kolejny problem. W naszej tabeli John Paul Bloggs ma dwa wyniki uzyskane z przedmiotu Computing Mathematics: pierwszy, 43%, który nie pozwala zaliczyć danego przedmiotu, i drugi, 65%, który daje zaliczenie. W relacyjnej bazie danych rekordy tworzą zbiór i nie ma między nimi żadnej wyraźnie określonej kolejności. Patrząc na tę przykładową tabelę, możemy zgadywać, że zaliczenie przedmiotu odbyło się w trakcie drugiej próby, choć nie mamy pewności. Nie ma żadnej gwarancji, że wynik otrzymany później będzie w bazie danych pojawiał się po otrzymanym wcześniej. Dlatego też musimy zapewnić w bazie danych pewne informacje dotyczące daty zdobycia danej oceny, np. przez dodanie roku (`Year`) i semestru (`Sem`):

```
+------------+------------+---------+----------------------+------+-----+------+
| StudentID  | GivenNames | Surname | CourseName           | Year | Sem | Pctg |
+------------+------------+---------+----------------------+------+-----+------+
| 12345678   | John Paul  | Bloggs  | Data Science         | 2019 |   2 |   72 |
| 12345121   | Sarah      | Doe     | Programming 1        | 2020 |   1 |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics| 2019 |   2 |   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics| 2020 |   1 |   65 |
| 12345121   | Sarah      | Doe     | Data Science         | 2020 |   1 |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics| 2019 |   1 |   75 |
| 12345876   | Susan      | Smith   | Programming 1        | 2019 |   2 |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics| 2020 |   1 |   80 |
+------------+------------+---------+----------------------+------+-----+------+
```

Zauważ, że tabela `Student_Grades` stała się znacznie większa od pierwotnej. Identyfikatory studenta, imiona i nazwiska są powtarzane dla każdego roku. Te dane można wyodrębnić i umieścić w nowej tabeli, np. `Student_Details`:

```
+------------+------------+---------+
| StudentID  | GivenNames | Surname |
+------------+------------+---------+
| 12345121   | Sarah      | Doe     |
| 12345303   | Susan      | Smith   |
| 12345678   | John Paul  | Bloggs  |
| 12345876   | Susan      | Smith   |
+------------+------------+---------+
```

Dzięki temu zmniejszamy ilość informacji przechowywanych w tabeli `Student_Grades`:

```
+------------+----------------------+------+-----+------+
| StudentID  | CourseName           | Year | Sem | Pctg |
+------------+----------------------+------+-----+------+
| 12345678   | Data Science         | 2019 |   2 |   72 |
| 12345121   | Programming 1        | 2020 |   1 |   87 |
| 12345678   | Computing Mathematics | 2019 |   2 |   43 |
| 12345678   | Computing Mathematics | 2020 |   1 |   65 |
| 12345121   | Data Science         | 2020 |   1 |   65 |
| 12345876   | Computing Mathematics | 2019 |   1 |   75 |
| 12345876   | Programming 1        | 2019 |   2 |   55 |
| 12345303   | Computing Mathematics | 2020 |   1 |   80 |
+------------+----------------------+------+-----+------+
```

W celu wyszukania oceny zdobytej przez studenta najpierw trzeba odszukać jego identyfikator w tabeli `Student_Details`, a później dla tego identyfikatora odczytać dane przechowywane w tabeli `Student_Grades`.

Pomimo wprowadzonych zmian wciąż mamy pewne problemy, które nie zostały rozwiązane. Na przykład czy powinniśmy przechowywać informacje takie jak data przyjęcia studenta na wydział, adresy pocztowy i e-mail, wysokość wniesionych opłat i dane dotyczące obecności na zajęciach? Czy powinny być przechowywane różne rodzaje adresu pocztowego? W jaki sposób należy przechowywać adres, aby nie pojawiły się problemy, gdy student zmieni adres pocztowy?

Implementacja bazy danych w pokazany sposób jest problematyczna. Będziemy się zajmować kwestiami, o których wcześniej nie myśleliśmy, i będziemy musieli nieustannie zmieniać strukturę bazy danych. Wiele niepotrzebnej pracy można uniknąć przez staranne udokumentowanie wymagań już na samym początku, a następnie przygotowanie na ich podstawie spójnego projektu.

Proces projektowania bazy danych

Podczas projektowania bazy danych można wyróżnić trzy główne etapy, z których każdy pozwala na progresywne przygotowanie opisu na niskim poziomie.

Analiza wymagań

Przede wszystkim trzeba określić i następnie zapisać, jakie informacje z bazy danych będą potrzebne, jakie dane będą w niej przechowywane, a także jak te dane są ze sobą powiązane. W praktyce to może oznaczać dokładną analizę wymagań aplikacji oraz konsultacje z osobami o różnych rolach, które później będą korzystały z bazy danych i aplikacji.

Projekt koncepcyjny

Po określeniu wymagań bazy danych trzeba je przełożyć na postać formalnego opisu projektu bazy danych. W dalszej części rozdziału zobaczysz, jak wykorzystać modelowanie w celu przygotowania projektu koncepcyjnego.

Projekt lokalny

W ostatnim kroku projekt bazy mapujemy na istniejący system zarządzania relacyjnymi bazami danych i tabele bazy danych.

Pod koniec rozdziału pokażemy, jak narzędzie typu open source MySQL Workbench można wykorzystać podczas konwersji projektu koncepcyjnego na schemat bazy danych MySQL.

Model relacji między encjami

Na najbardziej podstawowym poziomie baza danych przechowuje informacje o oddzielnych obiektach, nazywanych *encjami*, i zachodzących między nimi powiązaniach, nazywanych *relacjami*. Na przykład uniwersytecka baza danych może przechowywać informacje o studentach, przedmiotach i datach przyjęcia studentów na wydział. Podobnie baza danych w firmie handlowej może zawierać informacje o oferowanych produktach oraz dane dotyczące klientów i wielkości sprzedaży. W takim przypadku dane o produkcie i kliencie to encje, a dane o wielkości sprzedaży to relacja zachodząca między klientem a produktem. Początkowy użytkownik systemów RDBMS bardzo łatwo może pomylić encje i relacje, czego skutkiem będzie zaprojektowanie relacji jako encji i na odwrót. Najlepszym sposobem na poprawienie własnych umiejętności w zakresie projektowania baz danych jest regularne wykonywanie tego rodzaju zadania.

Popularne podejście dotyczące projektu koncepcyjnego obejmuje użycie modelu *ER* (ang. *entity relationship*), który pomaga w przeniesieniu wymagań na postać formalnego opisu encji i relacji w bazie danych. Zacznij od zapoznania się (w dalszej części rozdziału) ze sposobem działania procesu modelowania ER, a następnie przeanalizuj rozwiązania dla trzech przykładowych baz danych.

Przedstawianie encji

Aby pomóc w wizualizacji projektu, w podejściu modelowania ER rysuje się diagramy. Następnie na diagramie ER encję można przedstawić za pomocą prostokąta zawierającego jej nazwę. W przypadku wspomnianej wcześniej bazy danych w firmie handlowej diagram ER będzie zawierał encje produktu i klienta, jak pokazaliśmy na rysunku 2.1.

Rysunek 2.1. Encja na diagramie ER jest przedstawiana za pomocą nazwanego prostokąta

Baza danych jest zwykle używana do przechowywania konkretnych cech charakterystycznych, czyli *atrybutów*, encji. W rekordzie każdego klienta w bazie danych firmy handlowej mogą znajdować się informacje takie jak imię i nazwisko, adres e-mail, adres pocztowy i numer telefonu. W znacznie

bardziej rozbudowanej aplikacji CRM (ang. *customer relationship management*) mogą być przechowywane także imiona współmałżonka i dzieci klienta, język używany przez klienta, historia kontaktów klienta z firmą itd. Atrybuty opisują encję, do której należą.

Atrybuty można tworzyć na podstawie mniejszych fragmentów danych — np. adres pocztowy można przygotować z wykorzystaniem nazwy ulicy, numeru, kodu pocztowego oraz nazw miejscowości i państwa. Atrybut jest określany mianem *złożonego*, jeśli składa się z mniejszych fragmentów danych. W przeciwnym razie to będzie atrybut *prosty*.

Dla danej encji część atrybutów może mieć wiele wartości. Na przykład klient może podać więcej niż tylko jeden numer telefonu, stąd możliwość przechowywania *wielu wartości* dla tego atrybutu.

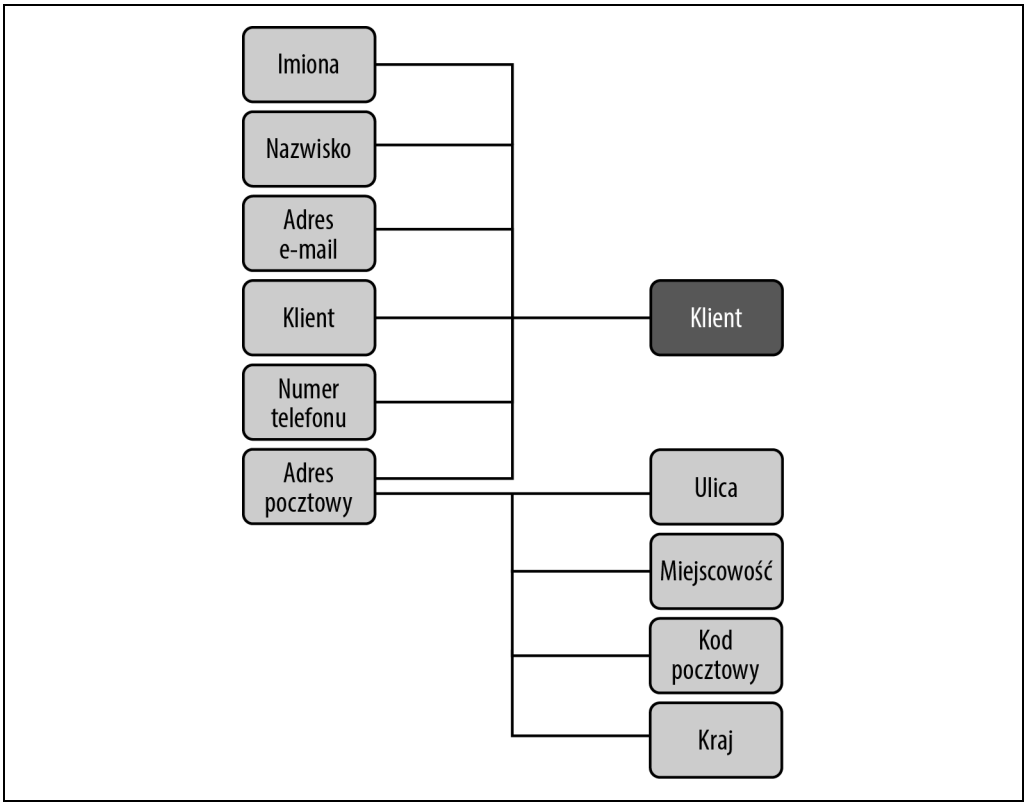
Atrybuty pomagają w odróżnianiu encji tego samego typu. Wprawdzie można wykorzystać atrybut do odróżniania klientów, ale takie rozwiązanie okazuje się nieodpowiednie, ponieważ wielu klientów może mieć to samo imię i/lub nazwisko. Aby można było odróżniać tych klientów, potrzebny jest atrybut (bądź minimalne połączenie atrybutów), który będzie unikatowy dla każdego klienta. Atrybut lub atrybuty identyfikujące tworzą unikatowy klucz, który w bazie danych jest nazywany *kluczem podstawowym* (ang. *primary key*).

W omawianym przykładzie można przyjąć założenie, że dwoje klientów nie będzie miało tego samego adresu e-mail, więc tym samym może on być kluczem podstawowym. Jednak podczas projektowania bazy danych trzeba dokładnie zastanowić się nad implikacjami dokonywanych wyborów. Na przykład, jeśli postanowisz identyfikować klienta na podstawie jego adresu e-mail, to w jaki sposób zapewnisz obsługę klienta używającego wielu adresów e-mail? Aplikacja wykorzystująca bazę danych utworzoną z przyjętym wcześniej założeniem może traktować każdy adres e-mail jako należący do innej osoby. Umożliwienie klientowi podania wielu adresów e-mail może się wiązać z koniecznością modyfikacji trudnej do wprowadzenia. Ponadto używanie adresu e-mail jako klucza podstawowego oznacza, że każdy klient musi mieć taki adres. W przeciwnym razie nie będzie można zapewnić prawidłowej obsługi klientów, którzy nie mają adresu poczty elektronicznej.

Szukając innych atrybutów możliwych do wykorzystania jako klucz alternatywny, dostrzegamy, że wprawdzie dwóch klientów może mieć ten sam numer telefonu (więc nie można go użyć jako klucza podstawowego), ale prawdopodobnie będą oni mieli inne imię i nazwisko. Dlatego też połączenie imienia, nazwiska i numeru telefonu można wykorzystać jako klucz złożony.

Nie ulega wątpliwości, że może istnieje wiele potencjalnych kluczy, które następnie mogą być używane do identyfikowania encji. Jedna z alternatywnych opcji, inaczej jeden z kluczy *kandydackich*, stanie się kluczem głównym, inaczej kluczem *podstawowym*. Podczas dokonywania wyboru zwykle uwzględnia się to, czy atrybut będzie miał wartość, czy zachowa unikatowość dla poszczególnych encji, a także jaka jest jego wielkość (im klucz jest mniejszy, tym jest łatwiejszy w obsłudze oraz tym szybsze są operacje wyszukiwania przeprowadzane za jego pomocą).

Na diagramie ER atrybuty są przedstawiane w postaci opatrzonych etykietami owalnych kształtów, połączonych z encją, jak pokazaliśmy na rysunku 2.2. Nazwy atrybutów tworzących klucz podstawowy zostały podkreślone. Elementy każdego atrybutu złożonego zostały ze sobą połączone. Natomiast jeśli atrybut ma więcej niż tylko jedną wartość, wówczas jest połączony z więcej niż tylko jednym owalnym kształtem.



Rysunek 2.2. Diagram ER przedstawiający encję klienta

Wartości atrybutów zostały wybrane ze zbioru wartości dozwolonych. Na przykład można określić, że atrybuty imion i nazwisk mogą być ciągami tekstowymi składającymi się z maksymalnie 100 znaków, a numer telefonu może być ciągiem tekstowym składającym się z maksymalnie 40 znaków. Podobnie atrybut ceny mógłby być liczbą całkowitą.

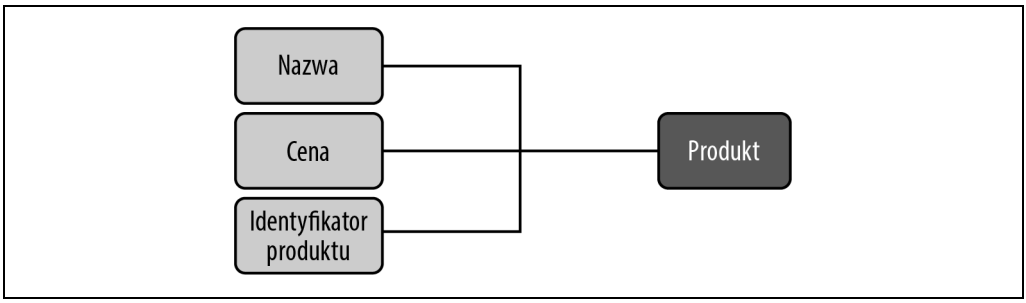
Atrybut może nie mieć przypisanej wartości. Na przykład może się zdarzyć, że klient nie poda numeru telefonu. Jednak klucz podstawowy encji (obejmujący komponenty klucza podstawowego składającego się z wielu atrybutów) zawsze musi istnieć (formalnie rzecz biorąc, musi być wartością typu `NOT NULL`). Dlatego też w przypadku prawdopodobieństwa, że klient nie poda adresu e-mail, tego adresu nie można użyć jako klucza.

Należy zachować ostrożność podczas rozważania atrybutu jako składającego się z wielu wartości: Czy wszystkie wartości są odpowiednikami, czy też przedstawiają zupełnie co innego? Na przykład, jeśli przechowujemy wiele numerów telefonu klienta, czy nie będzie lepszym rozwiązaniem, jeśli zostaną one opisane oddzielnie jako numer telefonu firmowy, domowy, komórkowy itd.?

Przejdźmy do następnego przykładu. Wymagania bazy danych firmy handlowej mogą określać, że produkt ma nazwę i cenę. Dostrzegamy produkt jako encję, ponieważ to jest zupełnie oddzielny obiekt. Jednak nazwa produktu i jego cena to nie są oddzielne obiekty, ale raczej atrybuty opisujące encję produktu. Trzeba w tym miejscu zwrócić uwagę, że jeśli dla poszczególnych rynków

mają być zdefiniowane odmienne ceny, wówczas cena nie będzie już powiązana jedynie z encją produktu i trzeba będzie modelować ją odmiennie.

W przypadku niektórych aplikacji żadne połączenie atrybutów nie pozwala na unikatową identyfikację encji (ewentualnie użycie ogromnego klucza złożonego byłoby zbyt niewygodne). Dlatego też jest tworzony sztuczny atrybut, zdefiniowany jako unikatowy i tym samym możliwy do użycia w charakterze klucza — identyfikator studenta, numer ubezpieczenia społecznego, numer i seria prawa jazdy, numer karty bibliotecznej itd. to wszystko są przykłady unikatowych atrybutów tworzonych dla różnych aplikacji. W przypadku naszej aplikacji dla firmy handlowej może się zdarzyć, że różne produkty będą przechowywane razem z taką samą nazwą i ceną. Na przykład mogą istnieć dwa modele produktu „Hub czteroportowy USB 2.0" w cenie 4,95 zł każdy. Aby móc rozróżniać poszczególne produkty oferowane przez sklep, każdemu z nich można przypisać unikatowy numer identyfikacyjny, który następnie może być wykorzystany jako klucz podstawowy. W takim przypadku encja produktu będzie zawierała atrybuty nazwy, ceny i identyfikatora, jak pokazaliśmy na diagramie ER, który możesz zobaczyć na rysunku 2.3.



Rysunek 2.3. Diagram ER przedstawiający encję produktu

Przedstawianie relacji

Encje mogą wchodzić w reakcje z innymi encjami. Na przykład klient może kupić produkt, student może zapisać się na zajęcia, pracownik może podać adres itd.

Podobnie jak encja, także relacja może mieć atrybuty. Istnieje możliwość zdefiniowania sprzedaży jako relacji między encją klienta (identyfikowanego za pomocą unikatowego adresu e-mail) a dowolną liczbą encji produktu (każda z nich identyfikowana za pomocą unikatowego numeru identyfikacyjnego produktu) istniejących w określonym dniu i o określonej godzinie (znacznik czasu).

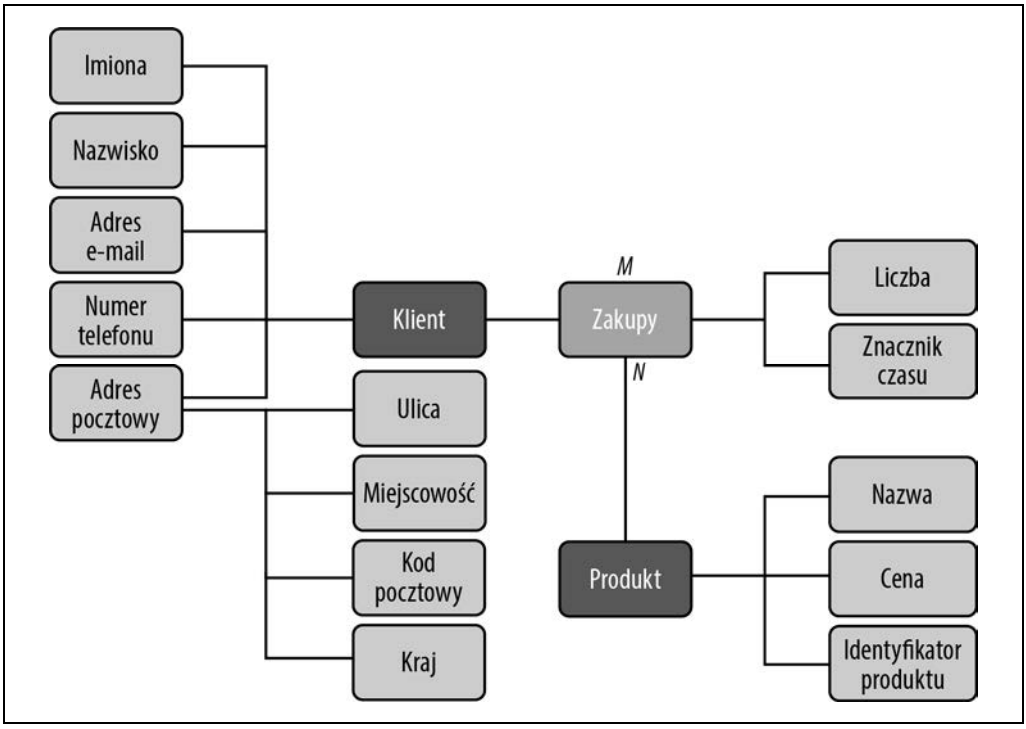
Każda operacja sprzedaży może być rejestrowana w bazie danych. Dzięki temu będzie wiadomo, że np. o godzinie 15:13 w środę 22 marca Marcon Albe kupił jedną sztukę produktu „Raspberry Pi 4", jedną sztukę „napęd 500 GB SSD M.2 NVMe" i dwie sztuki „głośniki 2000 W 5.1 Channel Sub-Woofer".

Po każdej stronie relacji mogą pojawić się różne liczby encji. Dlatego też klient może kupić dowolną liczbę produktów, produkt zaś może być kupiony przez dowolną liczbę klientów. Taki rodzaj relacji jest nazywany *wiele do wielu*. Mamy również relacje typu *jeden do wielu*. Na przykład dana osoba może mieć wiele kart kredytowych, przy czym każda z tych kart należy tylko do

jednej osoby. Patrząc na to w jeszcze inny sposób, relacja typu *jeden do wielu* staje się relacją *wiele do jednego* — np. wiele kart kredytowych należy do jednej osoby. Z kolei numer silnika przedstawia relację typu *jeden do jednego*, ponieważ każdy silnik ma tylko jeden numer seryjny, który z kolei jest używany w tylko jednym silniku. Można stosować skróty *1:1*, *1:N* i *M:N*, określające poszczególne typy relacji: „jeden do jednego", „jeden do wielu" i „wiele do wielu".

Liczba encji znajdujących się po jednej ze stron relacji (*liczebność* relacji) określa *ograniczenia klucza* relacji. Trzeba koniecznie dobrze przemyśleć kwestię liczebności relacji. Istnieje wiele relacji, które na początku wydają się być typu „jeden do jednego", a tak naprawdę okazują się znacznie bardziej złożone. Na przykład zdarza się, że ktoś zmienia imię. W niektórych aplikacjach, takich jak policyjne bazy danych, to będzie bardzo ważna informacja. Dlatego też może być konieczne modelowanie relacji typu „wiele do wielu" między encją osoby a encją imienia. W razie przyjęcia założenia o istnieniu prostszej relacji modyfikacja bazy danych może się okazać zadaniem kosztownym i czasochłonnym.

Na diagramie ER zbiór relacji jest przedstawiony za pomocą nazwanego rombu, obok którego bardzo często jest podawana także liczebność relacji. Taki styl został zastosowany w książce. (Innym powszechnie spotykanym stylem jest narysowanie linii ze strzałką łączącą stronę „1" encji z rombem relacji). Na rysunku 2.4 pokazaliśmy relację zachodzącą między encjami klienta i produktu, razem z atrybutami relacji sprzedaży.



Rysunek 2.4. Diagram ER przedstawiający encje klienta i produktu oraz zachodzącą między nimi relację sprzedaży

Udział pełny i częściowy

Relacje między encjami mogą być opcjonalne lub obowiązkowe. W naszym przykładzie możemy zdecydować, czy osoba zostanie uznana za klienta tylko wtedy, gdy kupi produkt. Z drugiej strony można uznać, że klient to osoba, na której temat mamy pewne informacje i po której spodziewamy się zakupu. Przy takim założeniu w bazie danych mogą znajdować się osoby uznane za klientów, które jednak nigdy nic nie kupiły. W pierwszym przypadku encja klienta ma *udział pełny* w relacji zakupu (każdy klient dokonał zakupu, w bazie danych nie mamy informacji o osobach, które tego nie zrobiły). Natomiast w drugim przypadku encja klienta ma *udział częściowy* (klient może dokonać zakupu). To są tzw. *ograniczenia udziału* relacji. Na diagramie ER udział pełny jest wskazywany za pomocą podwójnej linii między elementem encji i rombem relacji.

Encja czy atrybut?

Od czasu do czasu zdarza się sytuacja, w której zastanawiamy się, czy dany element powinien być atrybutem czy raczej oddzielną encją. Na przykład adres e-mail można modelować jako oddzielną encję. W razie wątpliwości warto rozważyć wymienione tutaj kwestie.

Czy element ma duże znaczenie dla bazy danych?

Obiekt o dużym znaczeniu dla bazy danych powinien być encją, a opisujące go informacje powinny być przechowywane w atrybutach. W przypadku naszej bazy danych firmy handlowej duże znaczenie mają klienci, a nie adresy e-mail, więc najlepiej, aby adres e-mail był modelowany jako atrybut encji przedstawiającej klienta.

Czy element ma własne komponenty?

Jeżeli tak, konieczne jest znalezienie sposobu na przedstawienie tych komponentów. Najlepszym rozwiązaniem może się okazać oddzielna encja. W przedstawionym na początku rozdziału przykładzie ocen studentów przechowywane były m.in. następujące informacje: nazwa przedmiotu, rok i semestr dla każdego przedmiotu, na który uczęszcza student. Zwięźlejszym rozwiązaniem byłoby traktowanie przedmiotów jako oddzielnych encji oraz utworzenie identyfikatora klasy, co pozwoliłoby na identyfikację poszczególnych zajęć dostępnych dla studentów („oferta").

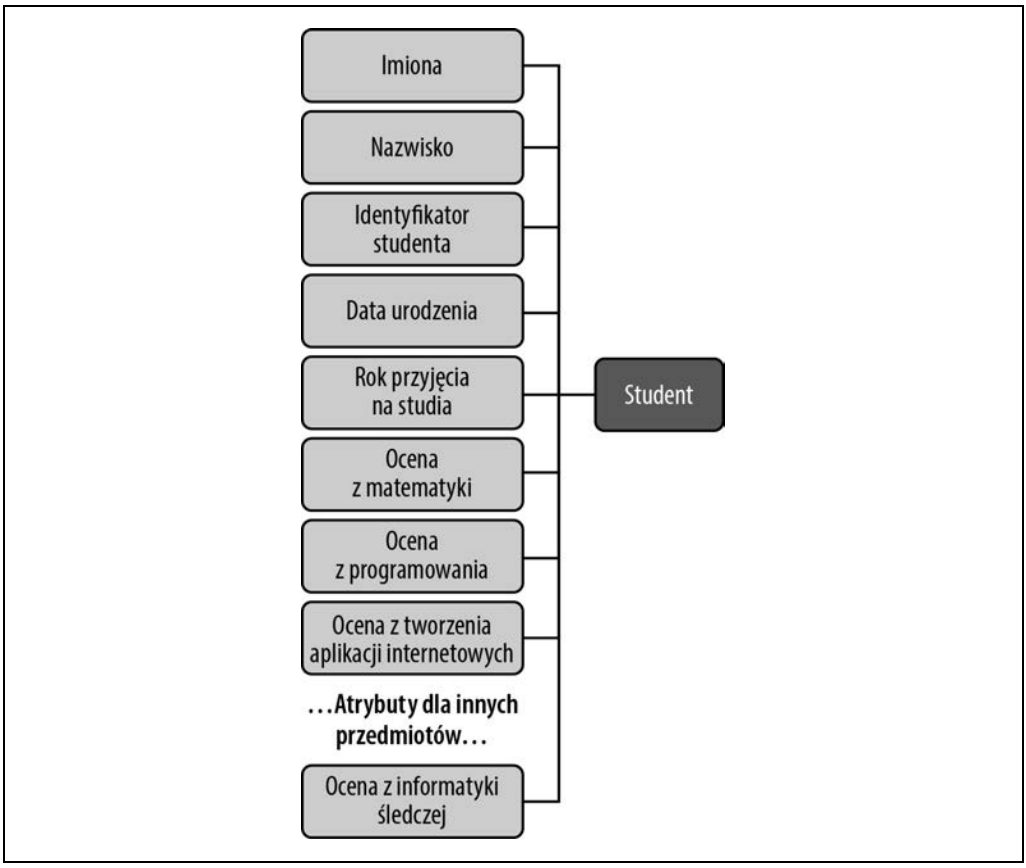
Czy może istnieć wiele egzemplarzy obiektu?

Jeżeli tak, konieczne jest znalezienie sposobu na przechowywanie danych w poszczególnych egzemplarzach. Najbardziej eleganckim rozwiązaniem będzie przedstawienie obiektu jako oddzielnej encji. W przykładzie sprzedaży trzeba odpowiedzieć sobie na pytanie, czy klient może mieć więcej niż tylko jeden adres e-mail. Jeżeli tak, wówczas adres e-mail powinien być modelowany jako oddzielna encja.

Czy obiekt często nie istnieje bądź jest nieznany?

Jeżeli tak, to mamy do czynienia z atrybutem jedynie pewnych encji. Najlepszym rozwiązaniem będzie jego modelowanie jako oddzielnej encji zamiast atrybutu, który często pozostaje pusty. Rozważ prosty przykład. W celu przechowywania otrzymywanych przez studentów ocen z różnych przedmiotów można utworzyć atrybut dla oceny z każdego możliwego przedmiotu,

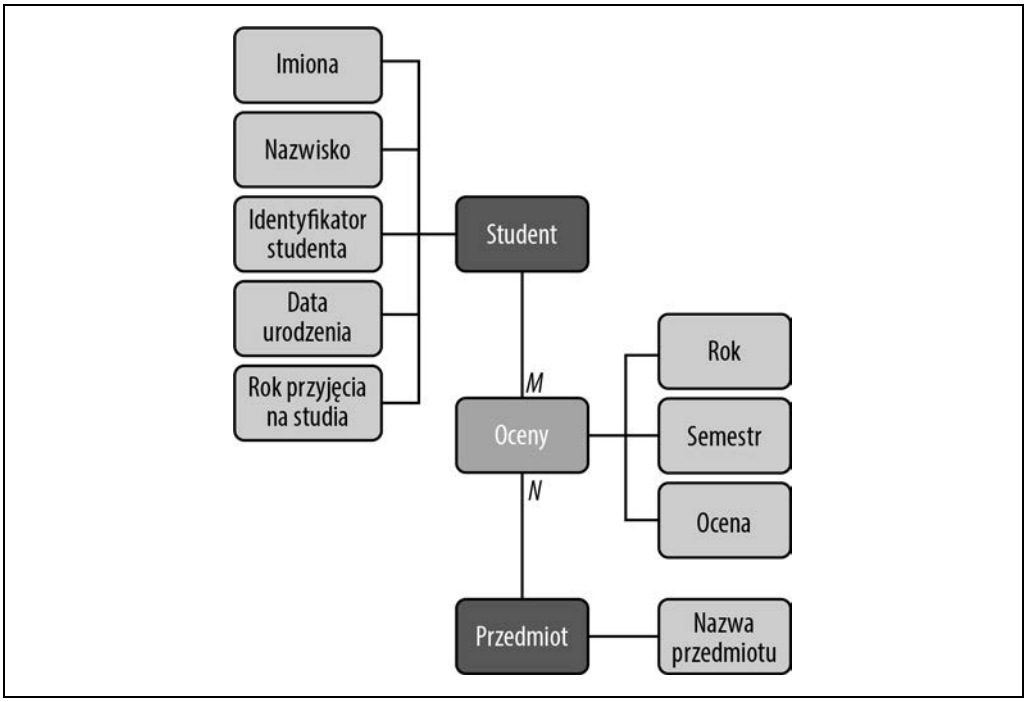
jak pokazaliśmy na rysunku 2.5. Skoro jednak większość studentów będzie miała oceny jedynie z niektórych przedmiotów, lepszym rozwiązaniem jest przedstawienie ocen jako oddzielnej encji, jak pokazaliśmy na rysunku 2.6.



Rysunek 2.5. Diagram ER przedstawiający oceny studenta jako atrybuty encji oznaczającej studenta

Encja czy relacja?

Łatwym sposobem na ustalenie roli danego obiektu, encja lub relacja, jest mapowanie — rzeczowników w tekście wymagań na encje, czasowników zaś — na relacje. Na przykład w zdaniu „Program przechowujący oceny z jednego bądź więcej przedmiotów" możemy zidentyfikować encje „program" i „przedmiot", a relacją jest „przechowujący". Podobnie w zdaniu „Student zapisujący się na przedmiot" możemy zidentyfikować encje „student" i „przedmiot", relacją zaś jest „zapisujący się". Oczywiście można wybierać inne pojęcia dla encji i relacji od tych pojawiających się w relacji, choć dobrze będzie nie oddalać się zbytnio od nazewnictwa użytego podczas definiowania wymagań, aby przygotowany projekt można było łatwo porównać z wymaganiami. Wszystkie pozostałe kwestie wciąż mają zastosowanie — projekt powinien być prosty i należy unikać zbędnych encji. Nie trzeba więc tworzyć oddzielnej encji dotyczącej przyjęcia studenta na wydział, wystarczające jest modelowanie tego jako relacji między istniejącymi encjami studenta i przedmiotów.



Rysunek 2.6. Diagram ER przedstawiający oceny studenta jako oddzielną encję

Encje pośrednie

Bardzo często istnieje możliwość koncepcyjnego uproszczenia relacji typu „wiele do wielu" przez zastąpienie jej nową encją *pośrednią* (czasami nazywaną encją asocjacyjną) oraz połączenie pierwotnych encji za pomocą relacji typu „wiele do jednego" i „jeden do wielu".

Rozważ następujące zdanie „Pasażer ma rezerwację w samolocie". Mamy tutaj relację typu „wiele do wielu" między encjami „pasażer" i „samolot". Diagram ER dla tej sytuacji przedstawiliśmy na rysunku 2.7.

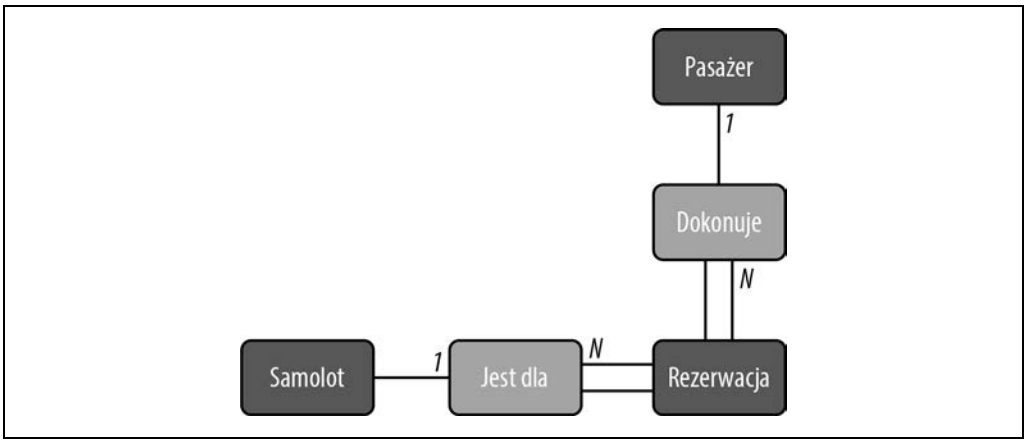
Rysunek 2.7. Pasażer uczestniczy w relacji M:N z samolotem

Spójrzmy na tę sytuację z obu stron relacji:

- W danym samolocie może znajdować się wielu pasażerów z rezerwacjami.
- Dany pasażer może zarezerwować miejsca w wielu samolotach.

W omawianym przykładzie relacja typu „wiele do wielu" może być w rzeczywistości dwiema relacjami typu „jeden do wielu". To prowadzi do istnienia ukrytej encji pośredniej, rezerwacji, między encjami

samolotu i pasażera. Wymagania mogłyby zostać zdefiniowane lepiej za pomocą następujących słów: „Pasażer dokonuje rezerwacji miejsca w samolocie”. Uaktualniony diagram ER zamieściliśmy na rysunku 2.8.



Rysunek 2.8. Encja pośrednia znajdująca się między encjami pasażera i samolotu

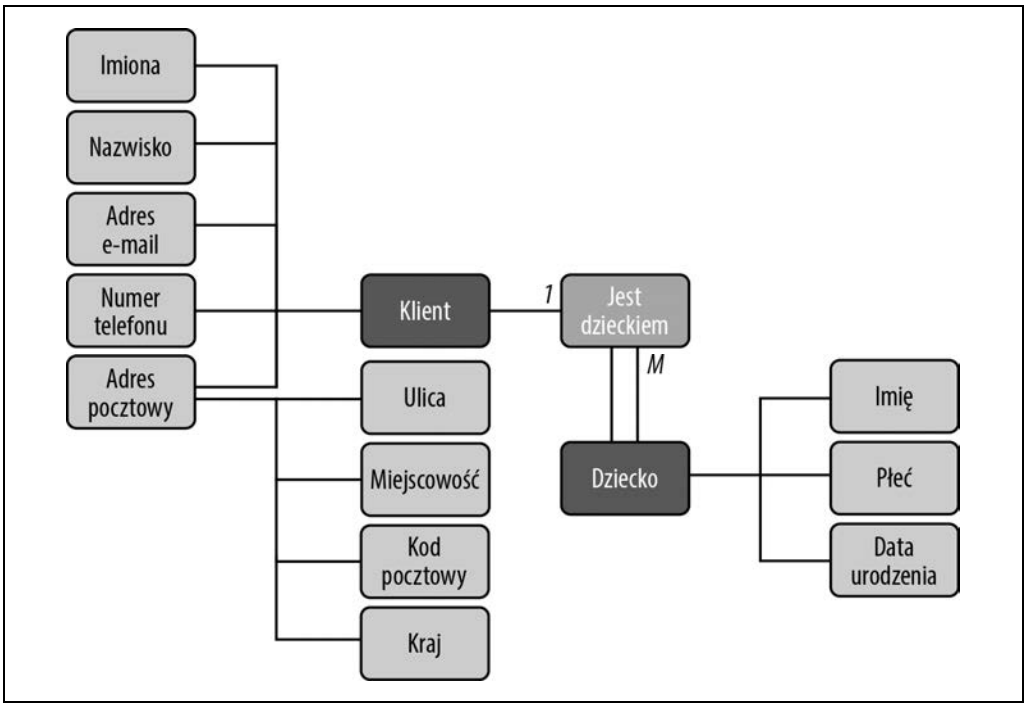
Każdy pasażer może mieć wiele rezerwacji, przy czym każda z nich należy tylko do jednego pasażera, więc liczebność tej relacji wynosi 1:N. Podobnie może istnieć wiele rezerwacji na lot w samolocie, przy czym każda rezerwacja dotyczy pojedynczego lotu, więc liczebność tej relacji wynosi 1:N. Skoro każda rezerwacja musi być powiązana z konkretnym pasażerem i lotem, encja rezerwacji ma udział całkowity w relacji zachodzącej między tymi encjami (więcej informacji na temat typów udziałów przedstawiliśmy we wcześniejszej części rozdziału). Ten udział całkowity nie może być efektownie pokazany na diagramie zamieszczonym na rysunku 2.7.

Encje słabe i silne

Kontekst ma bardzo duże znaczenie w codziennej pracy. Jeżeli kontekst jest znany, pracę można wykonać, mając do dyspozycji znacznie mniejszą ilość informacji. Na przykład do członków rodziny zwracamy się po imieniu bądź przezwisku. W razie niejasności można dodać kolejne informacje, takie jak nazwisko, aby tym samym jasno wyrazić intencje. W projekcie bazy danych można w przypadku encji pominąć pewne ważne informacje, które są zależne od innych encji. Na przykład, jeśli chcemy przechowywać imiona dzieci klientów, możemy utworzyć nową encję i umieszczać w niej tylko ilość informacji wystarczającą do identyfikacji dziecka w kontekście jego rodzica. Można po prostu przechowywać imię dziecka i bezpiecznie przyjąć założenie, że klient nigdy nie będzie miał więcej niż tylko jedno dziecko o danym imieniu. Dlatego też encja przedstawiająca dziecko jest *słaba*, a jego relacja z encją klienta jest określana mianem *relacji identyfikującej*. Słaba encja ma udział pełny w relacji identyfikującej, ponieważ ta encja nie może istnieć w bazie danych niezależnie od encji będącej jej właścicielem.

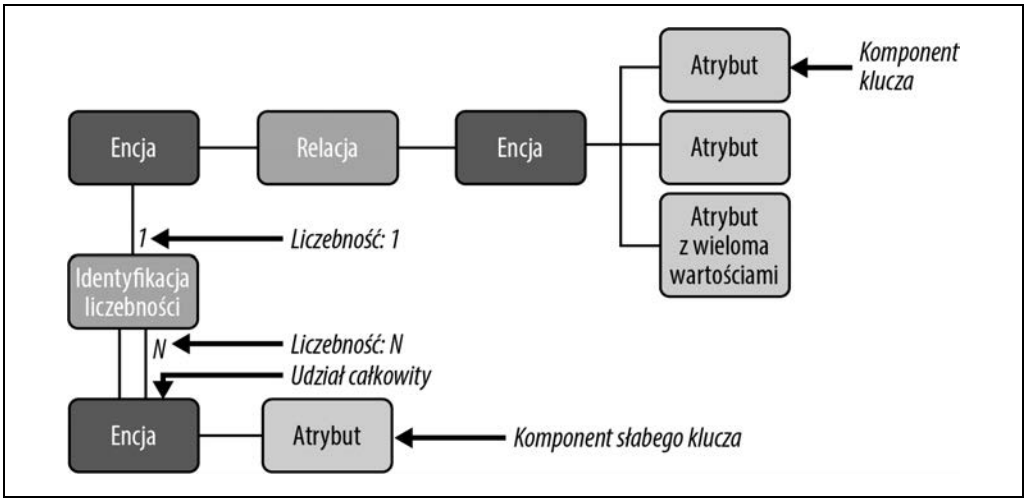
Na diagramie ER encje słabe i relacje identyfikujące są przedstawiane za pomocą linii podwójnych, a klucz częściowy encji słabej jest oznaczony linią przerywaną, jak pokazaliśmy na rysunku 2.9. Encja słaba jest unikatowo identyfikowana w kontekście encji będącej jej właścicielem (czyli *silnej*).

Ponadto pełny klucz dla encji słabej jest połączeniem jej klucza (częściowego) i klucza encji silniej. Do unikatowej identyfikacji dziecka w omawianym przykładzie potrzebne jest imię dziecka i adres e-mail rodzica.



Rysunek 2.9. Diagram ER przedstawiający encję słabą

Na rysunku 2.10 pokazaliśmy podsumowanie elementów pojawiających się na diagramach ER.



Rysunek 2.10. Podsumowanie elementów używanych na diagramach ER

Normalizacja bazy danych

Normalizacja bazy danych to bardzo ważna koncepcja podczas projektowania relacyjnej struktury danych. Wprawdzie dr Edgar F. Codd, wynalazca modelu relacyjnej bazy danych, postaci normalizacji zaproponował we wczesnych latach 70. ubiegłego stulecia, ale wciąż są one powszechnie stosowane w informatyce. Pomimo pojawienia się baz danych typu NoSQL nie ma żadnych przesłanek wskazujących, że w dającej się przewidzieć przyszłości relacyjne bazy danych znikną z rynku bądź też postaci normalizacji przestaną być stosowane.

Podstawowym zadaniem postaci normalizacji jest zmniejszenie powielania się danych i podniesienie poziomu ich spójności. Normalizacja ułatwia także proces przeprojektowania i rozszerzania struktury bazy danych.

Oficjalnie mamy sześć postaci normalizacji, przy czym w większości baz danych stosowane są tylko trzy pierwsze. To wynika z tego, że proces normalizacji jest progresywny — nie można osiągnąć wyższego poziomu normalizacji bazy danych, o ile nie zostaną spełnione niższe poziomy. Zastosowanie wszystkich sześciu postaci normalizacji zawęziłoby model bazy danych tak bardzo, że ogólnie rzecz biorąc, jego implementacja byłaby bardzo skomplikowana.

W rzeczywistych projektach zwykle napotykamy problemy związane z wydajnością działania. Mamy konkretny powód istnienia zadań ETL (ang. *extract, transform, load*): denormalizacja danych w celu ich przetworzenia.

Zapoznaj się teraz z trzema pierwszymi postaciami normalizacji.

Pierwsza postać normalizacji, 1NF (ang. first normal form), ma następujące cele:

- Wyeliminowanie w poszczególnych tabelach powtarzających się grup.
- Utworzenie oddzielnej tabeli dla każdego zbioru powiązanych ze sobą danych.
- Identyfikowanie za pomocą klucza podstawowego poszczególnych zbiorów powiązanych ze sobą danych.

  Jeżeli relacja zawiera atrybuty złożone bądź mające wiele wartości, wówczas następuje złamanie pierwszej postaci normalizacji. Dlatego też można stwierdzić, że relacja będzie znajdowała się w pierwszej postaci normalizacji, gdy nie zawiera żadnych atrybutów złożonych lub mających wiele wartości. Jeżeli relacja jest w pierwszej postaci normalizacji, każdy jej atrybut ma pojedynczą wartość odpowiedniego typu.

Druga postać normalizacji, 2NF (ang. second normal form), ma następujące cele:

- Utworzenie oddzielnych tabel dla zbiorów wartości, które mają zastosowanie dla wielu rekordów.
- Powiązanie tych tabel z kluczem zewnętrznym.

  Rekordy nie powinny mieć innych zależności niż klucz podstawowy tabeli (lub klucz złożony, jeśli zachodzi taka potrzeba).

Trzecia postać normalizacji, 3NF (ang. third normal form), ma następujące cele:

- Wyeliminowanie pól niezależnych od klucza.

Wartości w rekordzie niebędące częścią klucza tego rekordu nie należą do tabeli. Ogólnie rzecz biorąc, za każdym razem, gdy zawartość grupy pól może mieć zastosowanie do więcej niż tylko jednego rekordu tabeli, należy rozważyć umieszczenie tych pól w oddzielnej tabeli.

W tabeli 2.1 wymieniliśmy postaci normalizacji od najmniej do najbardziej znormalizowanej. Postać nieznormalizowana, UNF (ang. *unnormalized form*), to model bazy danych niespełniający żadnych warunków normalizacji bazy danych. Wprawdzie istnieją jeszcze inne postaci normalizacji, ale wykraczają one poza zakres materiału prezentowanego w książce.

Tabela 2.1. Postaci normalizacji (od najmniej do najbardziej znormalizowanej)

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	4NF (1977)	5NF (1979)	6NF (2003)
Klucz podstawowy (brak powielających się krotek)	Być może	Tak	Tak	Tak	Tak	Tak	Tak
Brak powielających się grup	Nie	Tak	Tak	Tak	Tak	Tak	Tak
Kolumny niepodzielne (komórki mają pojedyncze wartości)	Nie	Tak	Tak	Tak	Tak	Tak	Tak
Każda bardziej złożona zależność funkcyjna, która nie rozpoczyna się odpowiednim podzbiorem klucza kandydata lub kończy się atrybutem podstawowym (brak częściowych zależności funkcyjnych atrybutów niepodstawowych dla kluczy kandydatów)	Nie	Nie	Tak	Tak	Tak	Tak	Tak
Każda bardziej złożona zależność funkcyjna, która rozpoczyna się od superklucza lub kończy atrybutem podstawowym (brak przechodnich zależności funkcyjnych atrybutów niepodstawowych dla kluczy kandydatów)	Nie	Nie	Nie	Tak	Tak	Tak	Tak
Każda bardziej złożona zależność funkcyjna, która rozpoczyna się od superklucza lub kończy atrybutem podstawowym	Nie	Nie	Nie	Nie	Tak	Tak	nd.
Każda bardziej złożona zależność funkcyjna, która rozpoczyna się od superklucza	Nie	Nie	Nie	Nie	Tak	Tak	nd.
Każda bardziej złożona zależność z wieloma wartościami, która rozpoczyna się od superklucza	Nie	Nie	Nie	Nie	Tak	Tak	nd.
Każda zależność złączenia, która ma komponent superklucza	Nie	Nie	Nie	Nie	Nie	Tak	nd.
Każda zależność złączenia, która ma jedynie komponenty superklucza	Nie	Nie	Nie	Nie	Nie	Tak	nd.
Każde ograniczenie jest wynikiem zastosowania ograniczeń domeny i klucza	Nie	Nie	Nie	Nie	Nie	Nie	nd.
Każda zależność złączenia jest prosta	Nie	Nie	Nie	Nie	Nie	Nie	Tak

Normalizacja przykładowej tabeli

Aby dogłębniej zrozumieć omówione wcześniej koncepcje, zapoznaj się z przykładem normalizacji fikcyjnej tabeli przechowującej dane studentów.

Na początek mamy tabelę w postaci nieznormalizowanej:

```
Student#   Advisor   Adv-Room   Class1   Class2   Class3
1022       Jones     412        101-07   143-01   159-02
4123       Smith     216        201-01   211-02   214-01
```

Pierwsza postać normalizacji — brak powtarzających się grup

Tabela powinna mieć tylko pojedyncze pole dla każdego atrybutu. Skoro student może uczęszczać na wiele przedmiotów, powinny być one wymienione w oddzielnej tabeli. Pola `Class1`, `Class2` i `Class3` w naszej nieznormalizowanej tabeli wskazują na nie najlepszą jakość projektu.

Arkusze kalkulacyjne bardzo często mają wiele pól dla tego samego atrybutu (np. `address1`, `address2` i `address3`), natomiast tabele nie powinny ich mieć. Oto inne spojrzenie na ten problem: w przypadku relacji typu „jeden do wielu" obu jej stron nie umieszczamy w tej samej tabeli. Zamiast tego tworzymy kolejną tabelę w pierwszej postaci znormalizowanej przez wyeliminowanie powtarzających się grup — np. `Class#`, jak pokazaliśmy w kolejnym fragmencie kodu:

```
Student#   Advisor   Adv-Room   Class#
1022       Jones     412        101-07
1022       Jones     412        143-01
1022       Jones     412        159-02
4123       Smith     216        201-01
4123       Smith     216        211-02
4123       Smith     216        214-01
```

Druga postać normalizacji — wyeliminowanie zbędnych danych

Zwróć uwagę na wiele wartości `Class#` dla poszczególnych wartości `Student#` w poprzedniej tabeli. Wartość `Class#` nie jest funkcjonalnie zależna od `Student#` (klucz podstawowy), więc ta relacja nie jest w drugiej postaci normalizacji.

Dwie kolejne tabele pokazują konwersję na drugą postać normalizacji. Oto obecna postać tabeli `Students`:

```
Student#    Advisor   Adv-Room
1022        Jones     412
4123        Smith     216
```

Z kolei tabela `Registration` prezentuje się następująco:

```
Student#  Class#
1022      101-07
1022      143-01
1022      159-02
4123      201-01
4123      211-02
4123      214-01
```

Trzecia postać normalizacji — wyeliminowanie danych niezależnych od klucza

W poprzednim przykładzie kolumna `Adv-Room` (numer gabinetu wykładowcy) jest funkcjonalnie zależna od atrybutu `Advisor`. Rozwiązaniem jest przeniesienie tego atrybutu z tabeli `Students` do tabeli `Faculty`, jak pokazaliśmy w kolejnym fragmencie kodu.

Oto obecna postać tabeli `Students`:

```
Student#  Advisor
1022      Jones
4123      Smith
```

Z kolei tabela `Faculty` prezentuje się następująco:

```
Name  Room  Dept
Jones 412 42
Smith 216 42
```

Przykłady modelowania relacji encji

W poprzednich punktach przedstawiliśmy hipotetyczne przykłady, których zadaniem była pomoc w zrozumieniu podstaw projektowania bazy danych, diagramów ER i normalizacji. Teraz przejdziemy do analizy wybranych przykładów ER pochodzących z baz danych standardowo dostępnych dla serwera MySQL. Do zwizualizowania diagramów ER posłużymy się oprogramowaniem MySQL Workbench (*https://dev.mysql.com/downloads/workbench/*).

MySQL Workbench korzysta z fizycznej reprezentacji ER. Fizyczny model diagramu ER jest znacznie bardziej szczegółowy i pokazuje procesy niezbędne w celu dodania informacji do bazy danych. Na diagramie ER zamiast symboli będą użyte tabele, co przybliży projekt do rzeczywistej bazy danych. Oprogramowanie MySQL Workbench pozwala pójść o krok dalej i skorzystać z *diagramów EER* (ang. *enhanced entity-relationship*). Tego rodzaju diagramy są rozszerzonymi wersjami diagramów ER.

Nie zamierzamy zagłębiać się w szczegóły związane z diagramami EER. Mimo to warto wspomnieć, że zapewniają one obsługę wszystkich elementów diagramów ER, a ponadto:

- dziedziczenie atrybutów i relacji,
- obsługę typów kategorii i unii,
- specjalizację i generalizację,
- obsługę podklas i superklas.

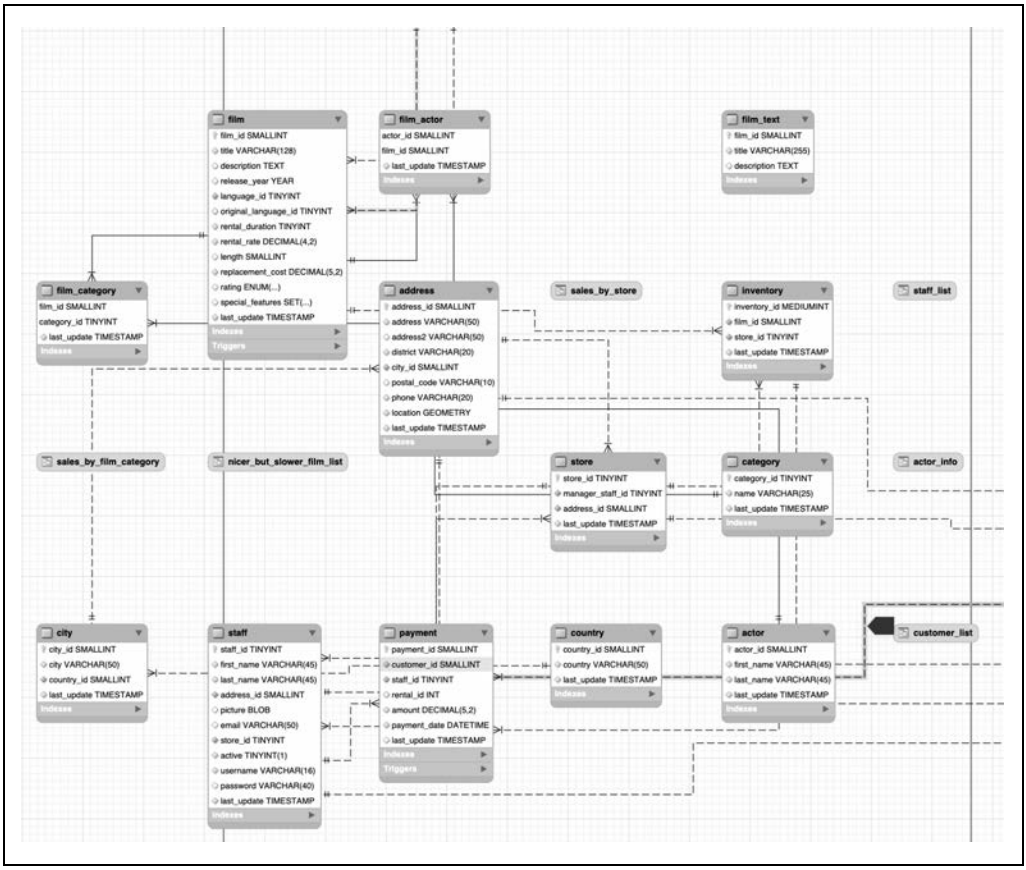
Zaczynamy od procesu pobrania przykładowych baz danych, a później przejdziemy do wizualizacji ich diagramów EER w oprogramowaniu MySQL Workbench.

Na początek wykorzystamy bazę danych `sakila`. Prace nad nią rozpoczęły się w 2005 roku. Wczesne wersje zostały oparte na bazie danych użytej w dokumencie *Three Approaches to MySQL Applications on Dell PowerEdge Servers* (*https://web.archive.org/web/20210228195710/http://www.dell.com/downloads/global/solutions/mysql_apps.pdf*), w której znajdowały się informacje używane w sklepie

internetowym zajmującym się sprzedażą płyt DVD. Podobnie baza danych `sakila` została zaprojektowana do przedstawienia wypożyczalni płyt DVD, przy czym tytuły filmów oraz dane aktorów zostały zapożyczone z przykładowej bazy danych firmy Dell. Wymienione tutaj polecenia pozwalają na pobranie bazy danych `sakila` i jej zaimportowanie do egzemplarza MySQL:

```
# wget https://downloads.mysql.com/docs/sakila-db.tar.gz
# tar -xvf sakila-db.tar.gz
# mysql -uroot -pmsandbox < sakila-db/sakila-schema.sql
# mysql -uroot -pmsandbox < sakila-db/sakila-data.sql
```

Baza danych `sakila` udostępnia model EER — znajdziesz go w pliku *sakila.mwb*, który możesz otworzyć za pomocą oprogramowania MySQL Workbench, jak pokazaliśmy na rysunku 2.11.



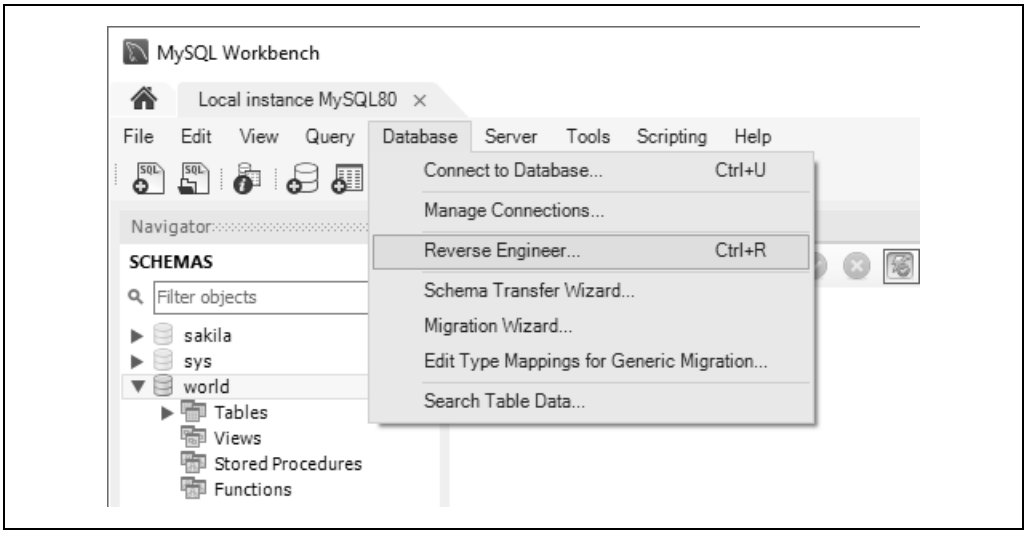
Rysunek 2.11. Model EER w bazie danych sakila. Zwróć uwagę na fizyczną reprezentację encji zamiast użycia symboli

Następnie mamy bazę danych `world`, wykorzystującą przykładowe dane statystyczne dotyczące Finlandii (*https://tilastokeskus.fi/tup/kvportaali/index_en.html*).

Wymienione tutaj polecenia pozwalają na pobranie bazy danych `world` i jej zaimportowanie do egzemplarza MySQL:

```
# wget https://downloads.mysql.com/docs/world-db.tar.gz
# tar -xvf world-db.tar.gz
# mysql -uroot -plearning_mysql < world-db/world.sql
```

Wprawdzie baza danych `world` nie jest dostarczana razem z plikiem EER, ale model EER można wygenerować samodzielnie na podstawie bazy danych z użyciem MySQL Workbench. W tym celu z menu *Database* wybierz opcję *Reverse Engineer* (zobacz rysunek 2.12).



Rysunek 2.12. Samodzielne generowanie modelu EER w bazie danych world

MySQL Workbench nawiąże połączenie z bazą danych (jeżeli nie nastąpiło to już wcześniej) i pozwoli na wybór schematu, na podstawie którego będzie wygenerowany model EER (zobacz rysunek 2.13).

Po wybraniu schematu kliknij przycisk *Continue*, a w kolejnym kroku kliknij przycisk *Execute*, jak pokazaliśmy na rysunku 2.14.

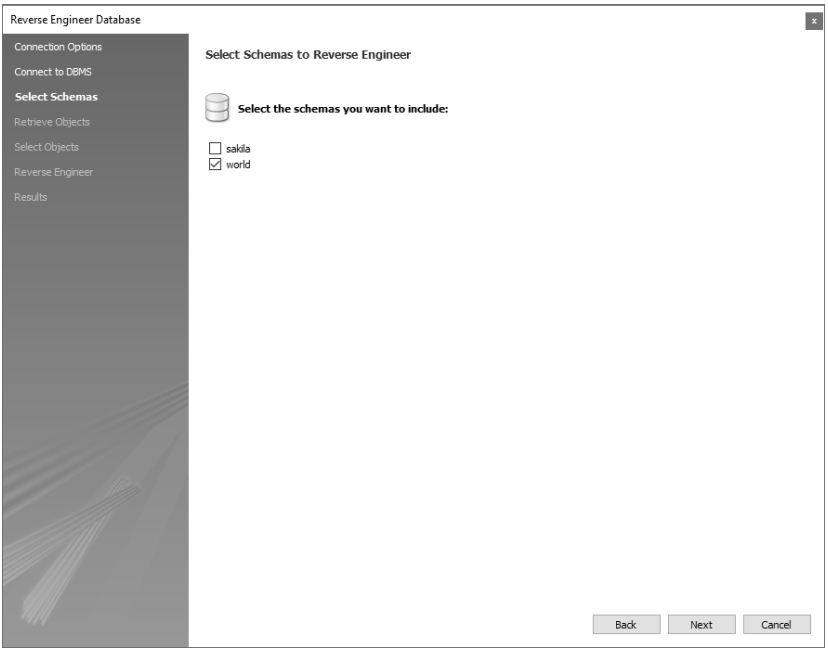
To spowoduje wygenerowanie modelu ER dla bazy danych `world` (zobacz rysunek 2.15).

Ostatnią importowaną bazą danych jest `employees`. Pierwotne dane zostały utworzone przez Fushenga Wanga i Carlo Zaniolo w Siemens Corporate Research (*http://timecenter.cs.aau.dk/software.htm*). Giuseppe Maxia utworzył schemat relacyjny, Patrick Crews zaś wyeksportował dane w formacie relacyjnym.

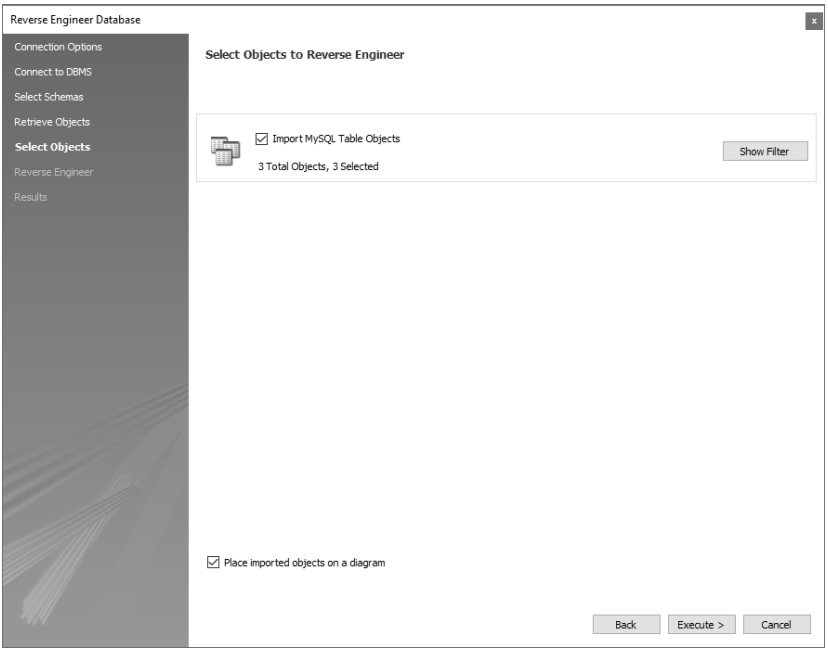
W celu zaimportowania tej bazy danych trzeba zacząć od sklonowania repozytorium Git:

```
# git clone https://github.com/datacharmer/test_db.git
# cd test_db
# cat employees.sql | mysql -uroot -psekret
```

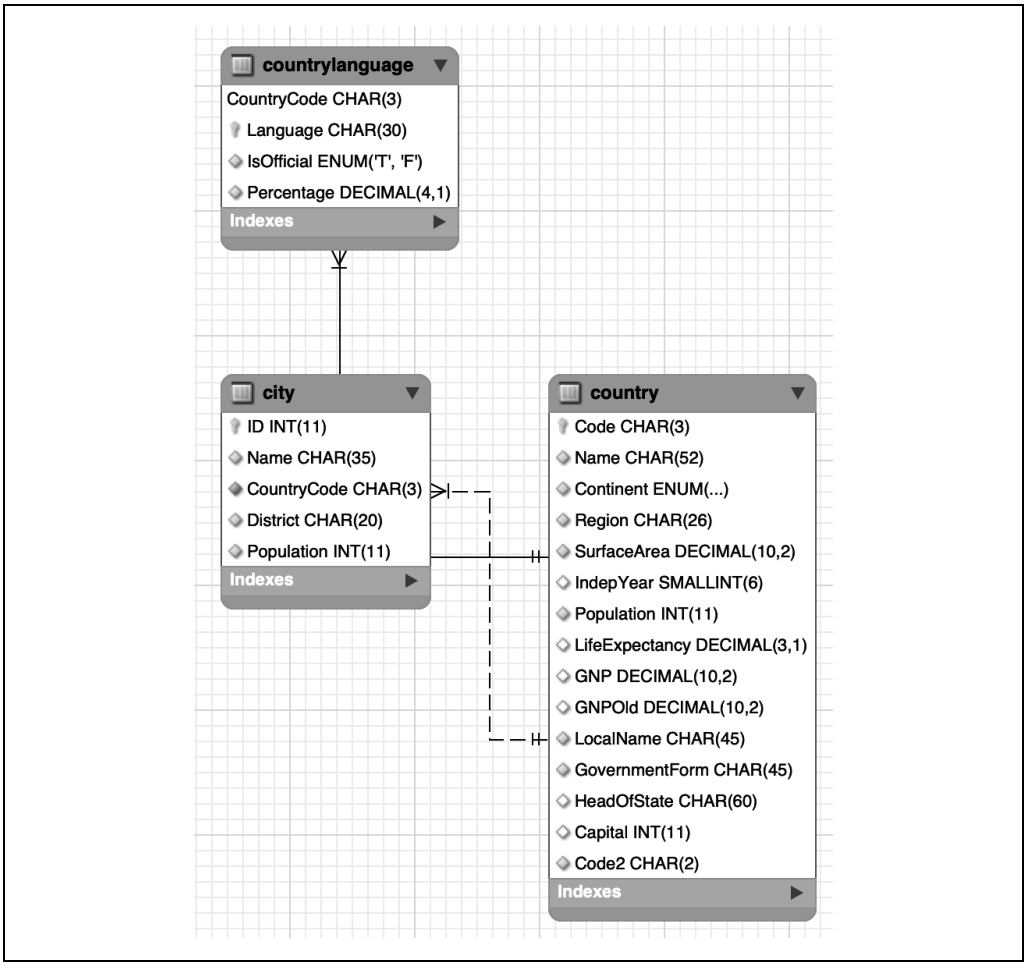
Następnie można ponownie skorzystać z procedury inżynierii odwrotnej w oprogramowaniu MySQL Workbench i utworzyć model ER dla bazy danych `employees`, jak pokazaliśmy na rysunku 2.16.



Rysunek 2.13. Wybór schematu bazy danych



Rysunek 2.14. Kliknięcie przycisku Execute rozpocznie proces inżynierii odwrotnej

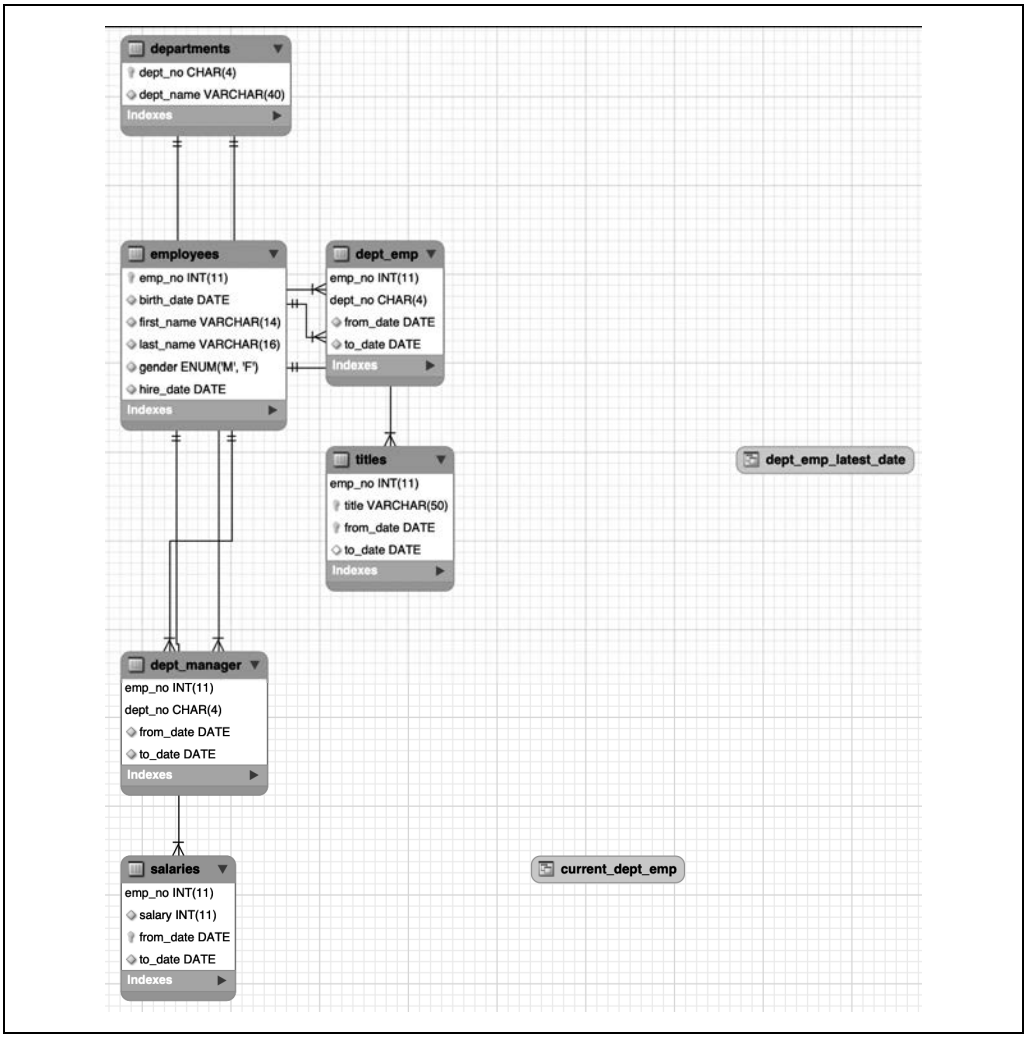


Rysunek 2.15. Model ER dla bazy danych world

Bardzo duże znaczenie ma dokładne przeanalizowanie przedstawionych modeli ER, aby dobrze zrozumieć relacje zachodzące między encjami a ich atrybutami. Po opanowaniu koncepcji można przystąpić do eksperymentów, które zaprezentujemy w następnym podrozdziale. W rozdziale 4. dowiesz się zaś, jak utworzyć bazę danych w serwerze MySQL.

Używanie modelu relacji encji

W tym podrozdziale przedstawimy procedurę utworzenia modelu ER i jego wdrożenia w tabelach bazy danych. Wcześniej wyjaśniliśmy, że oprogramowanie MySQL Workbench pozwala na przeprowadzanie inżynierii odwrotnej istniejącej bazy danych. Być może zastanawiasz się, jak modelować nową bazę danych i jak ją wdrożyć? Ten proces można zautomatyzować za pomocą narzędzia MySQL Workbench.



Rysunek 2.16. Model ER dla bazy danych employees

Mapowanie encji i relacji na tabele bazy danych

Podczas konwersji modelu ER na schemat bazy danych pracujemy z poszczególnymi encjami i następnie z poszczególnymi relacjami, zgodnie z regułami omówionymi w kolejnych punktach. Procedura trwa aż do przetworzenia całego zbioru tabel bazy danych.

Mapowanie encji na tabele bazy danych

Dla każdej silnej encji tworzona jest tabela składająca się z atrybutów tej encji oraz następuje zdefiniowanie klucza podstawowego. Uwzględniane są wszelkie elementy atrybutów złożonych.

W przypadku encji słabej następuje utworzenie tabeli składającej się z atrybutów tej encji i dołączany jest klucz podstawowy encji będącej właścicielem tej encji słabej. Ten klucz podstawowy będzie tutaj pełnił funkcję klucza zewnętrznego, ponieważ znajduje się w innej tabeli. Kluczem podstawowym encji słabej jest połączenie klucza zewnętrznego i klucza częściowego encji słabej. Jeżeli relacja z encją nadrzędną ma jakiekolwiek atrybuty, zostaną one dodane do tabeli.

Dla każdego atrybutu encji składającego się z wielu wartości zostaje utworzona tabela zawierająca klucz podstawowy encji i atrybut.

Mapowanie relacji do tabel bazy danych

Każda relacja typu „jeden do jednego" między dwiema encjami zawiera klucz podstawowy jednej z nich, który staje się kluczem zewnętrznym w tabeli należącej do drugiej encji. Jeżeli jedna encja ma udział całkowity w relacji, klucz zewnętrzny należy umieścić w jej tabeli. Natomiast jeśli obie encje mają udział całkowity w relacji, wówczas warto rozważyć ich połączenie w jedną tabelę.

Dla każdej nieidentyfikującej relacji typu „jeden do wielu" między dwiema encjami klucz podstawowy encji należy po stronie „jeden" relacji dołączyć jako klucz zewnętrzny w tabeli dla encji znajdującej się po stronie „wielu" relacji. Wszelkie atrybuty relacji należy umieścić w tabeli razem z kluczem zewnętrznym. Zwróć uwagę na to, że identyfikujące relacje typu „jeden do wielu" (między encją słabą a jej encją nadrzędną) są przechwytywane jako część etapu mapowania encji.

Jeśli relacja między dwiema encjami relacji jest typu „wiele do wielu", należy utworzyć nową tabelę, zawierającą klucze podstawowe tych encji jako klucz podstawowy tabeli, oraz dodać wszystkie atrybuty relacji. Ten krok pomoże w wykryciu encji pośrednich.

W przypadku każdej relacji obejmującej więcej niż dwie encje należy utworzyć tabelę z kluczami podstawowymi wszystkich tych encji oraz dodać wszystkie atrybuty relacji.

Utworzenie modelu ER bazy danych banku

We wcześniejszej części rozdziału omówiliśmy modele baz danych przechowujących informacje o ocenach studentów i dane klientów. Ponadto zaprezentowaliśmy trzy dostępne jako oprogramowanie typu open source modele EER dla serwera MySQL. W tym punkcie dowiesz się, jak modelować bazę danych banku. Zakładamy, że zebrane zostały wszystkie wymagania i zostały określone wymagania dla systemu bankowego działającego w internecie. Ustaliliśmy, że trzeba będzie utworzyć wymienione tutaj encje:

- Employees,
- Branches,
- Customers,
- Accounts.

Stosując się do przedstawionych wcześniej reguł mapowania, przystępujemy do utworzenia tabel i atrybutów dla każdej z tabel. Zdefiniowaliśmy klucze podstawowe, aby mieć pewność, że każda tabela otrzyma kolumnę unikatowego identyfikatora dla jej rekordów. Następnym krokiem jest zdefiniowanie relacji zachodzących między tabelami.

Relacje typu „wiele do wielu" (N:M)

Ten typ relacji został zdefiniowany między encjami Branches i Employees, a także między encjami Accounts i Customers. Pracownik banku może pracować w wielu oddziałach, a w każdym z nich może być zatrudnionych wiele osób. Podobnie klient może mieć wiele kont, konto zaś może być współdzielone przez więcej niż dwóch klientów.

Aby można było modelować te relacje, konieczne są dwie kolejne encje pośrednie. Tworzymy więc dwie następujące encje:

- account_customers,
- branch_employees.

Nowe encje stanowią pomosty między encjami, odpowiednio, Account i Customer oraz Branch i Employees. Relację typu N:M konwertujemy na dwie relacje typu 1:N. W następnym punkcie pokażemy, jak przedstawia się ich projekt.

Relacja typu „jeden do wielu" (1:N)

Ten typ relacji zachodzi między encjami Branches i Accounts oraz między Customers i account_customers. Mamy tutaj przykład zastosowania koncepcji *relacji nieidentyfikującej*. Na przykład w tabeli `accounts` kolumna `branch_id` nie jest częścią klucza podstawowego (jednym z powodów jest możliwość przeniesienia konta bankowego do innego oddziału). Obecnie powszechne jest stosowanie klucza surogata jako klucza podstawowego w tabeli. Dlatego też rzadkością jest prawdziwa relacja identyfikacyjna, w której klucz zewnętrzny jest również częścią klucza podstawowego w modelu danych.

Skoro tworzymy fizyczny model EER, definiujemy także klucze podstawowe. Powszechnym i zalecanym rozwiązaniem jest stosowanie dla klucza podstawowego pola zapewniającego obsługę automatycznej inkrementacji.

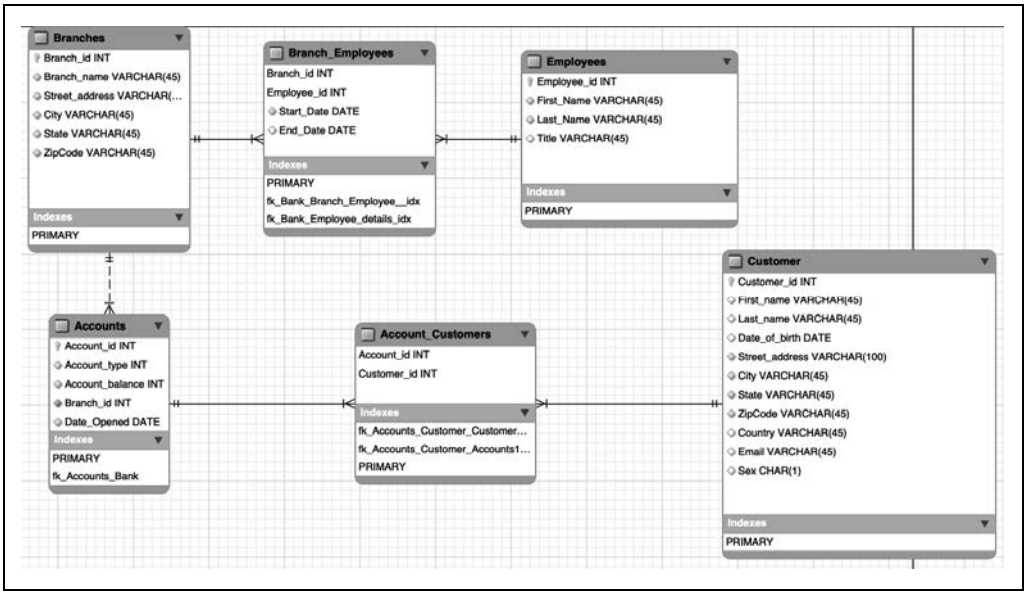
Na rysunku 2.17 pokazaliśmy ostateczną postać modelu bazy danych banku.

Zauważ, że mamy pewne elementy nieuwzględnione w tym modelu. Dlatego też nasz model nie zapewnia obsługi klienta z wieloma adresami e-mail (np. domowym i firmowym). Celowo zdecydowaliśmy się na takie rozwiązanie, aby podkreślić wagę przygotowania wymagań przed przystąpieniem do wdrażania bazy danych.

Ten model znajdziesz w archiwum materiałów przygotowanych dla książki (możesz je pobrać pod adresem *ftp://ftp.helion.pl/przyklady/troya.zip*). Plik zawierający omówiony model nosi nazwę *bank_model.mwb*.

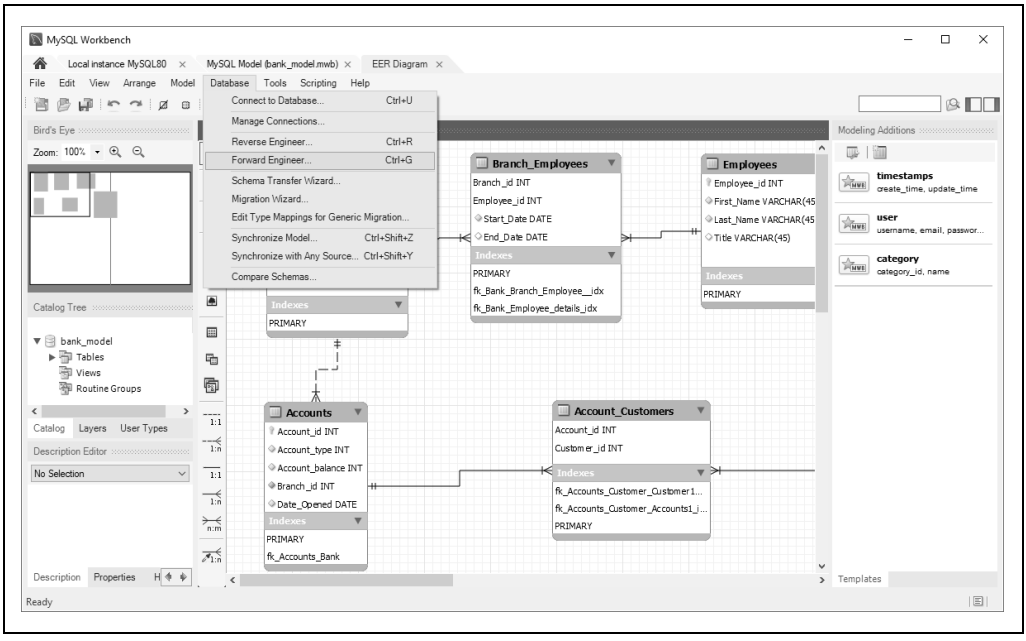
Konwersja modelu EER na bazę danych MySQL za pomocą oprogramowania MySQL Workbench

Do przygotowywania diagramów ER warto stosować odpowiednie narzędzia. Dzięki temu diagramy można łatwo edytować i modyfikować aż do chwili, gdy ich postać stanie się ostateczna i przejrzysta. Gdy model jest zadowalający, można go wdrożyć. Aplikacja MySQL Workbench



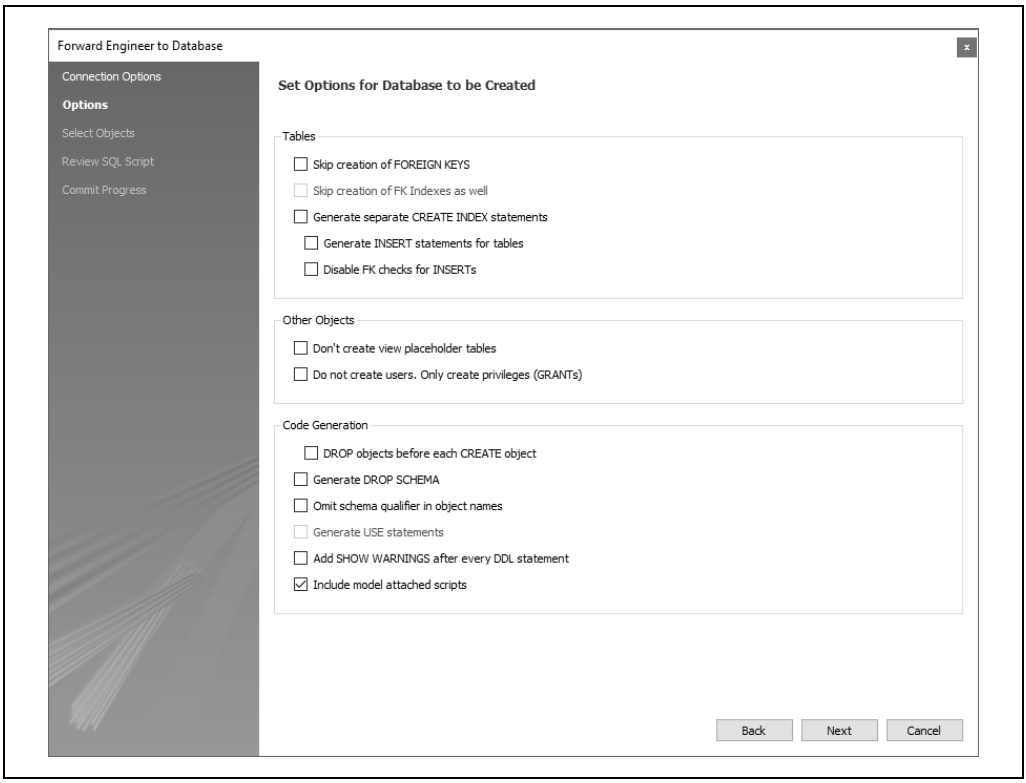
Rysunek 2.17. Model ERR dla bazy danych banku

pozwala skonwertować model EER na polecenia DDL (ang. *data definition language*) przeznaczone do utworzenia bazy danych MySQL za pomocą opcji *Forward Engineer* w menu *Database*, jak pokazaliśmy na rysunku 2.18.



Rysunek 2.18. Opcja Forward Engineer w menu Database w oprogramowaniu MySQL Workbench

Konieczne będzie podanie danych uwierzytelniających w celu nawiązania połączenia z bazą danych. Następnie MySQL Workbench wyświetli pewne opcje. W tym modelu zamierzamy skorzystać z opcji standardowych (zobacz rysunek 2.19), czyli wybrana jest jedynie ostatnia opcja z dostępnych.



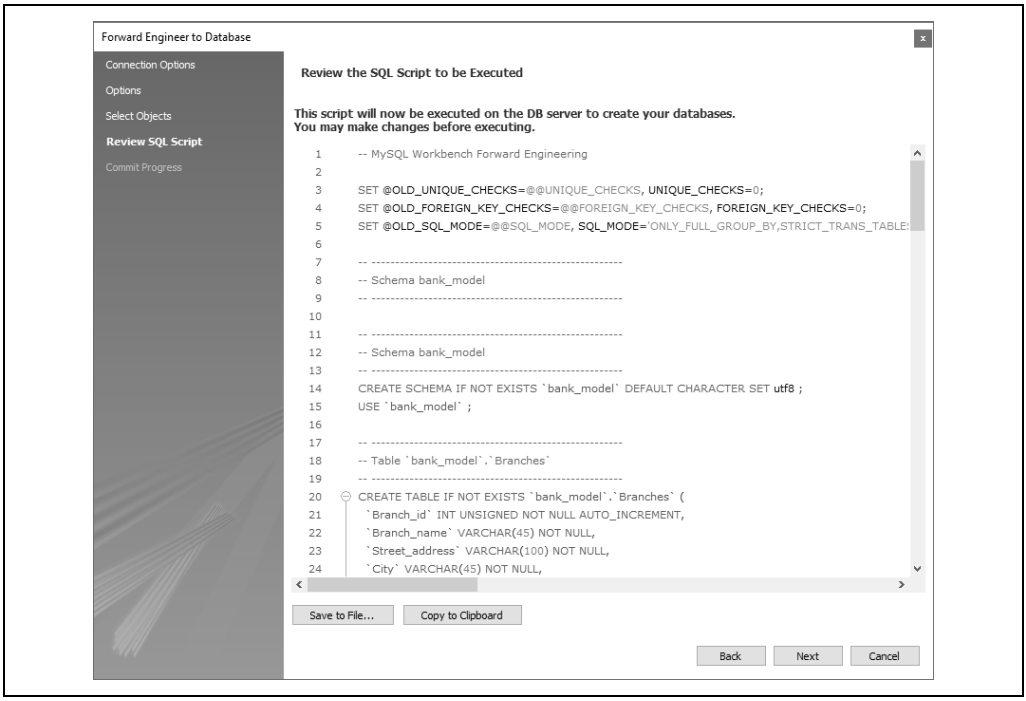
Rysunek 2.19. Opcje dostępne podczas tworzenia bazy danych

W trakcie kolejnego kroku można wskazać, które elementy modelu mają być wygenerowane. Ponieważ nie mamy żadnych funkcjonalności specjalnych, takich jak wyzwalacze, procedury składowane, konta użytkowników itd., tworzymy jedynie obiekty tabeli i ich relacje. Pozostałe opcje są niezaznaczone.

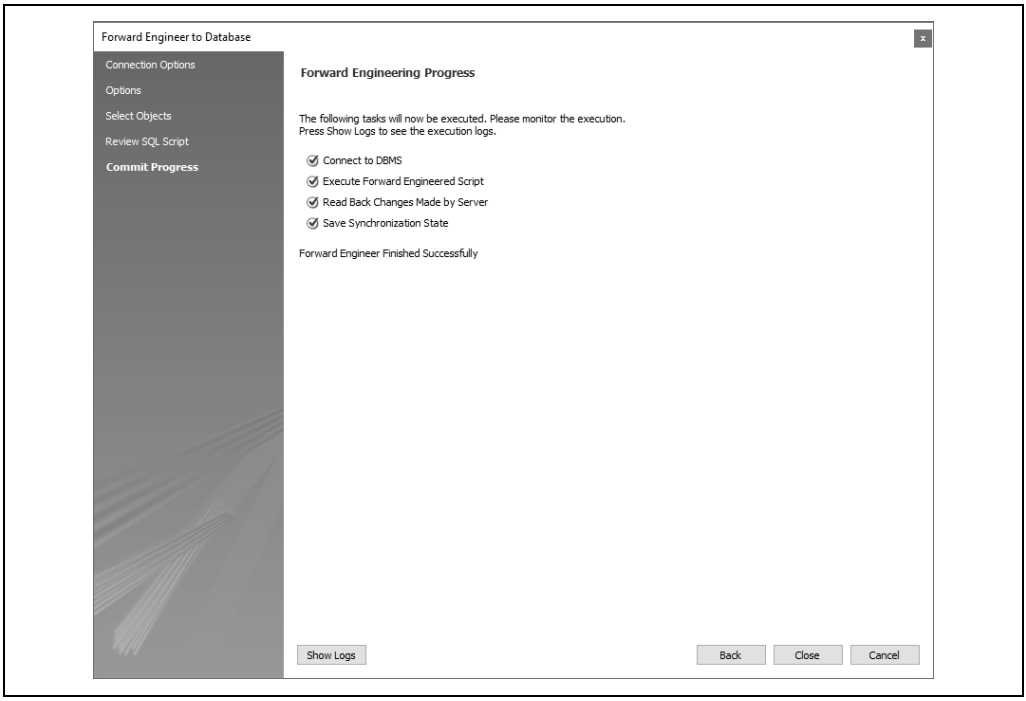
Aplikacja MySQL Workbench wyświetli teraz skrypt SQL, który zostanie wykonany w celu utworzenia bazy danych na podstawie naszego modelu, jak pokazaliśmy na rysunku 2.20.

Po kliknięciu przycisku *Continue* MySQL Workbench wykona polecenia w serwerze MySQL (zobacz rysunek 2.21).

Szczegóły związane z poleceniami znajdującymi się w tym skrypcie wyjaśnimy w rozdziale 4.



Rysunek 2.20. Wygenerowany skrypt, który będzie użyty do utworzenia bazy danych



Rysunek 2.21. Aplikacja MySQL Workbench zakończyła sukcesem wykonywanie skryptu

ROZDZIAŁ 3.

Podstawy języka SQL

Jak wspomnieliśmy w rozdziale 2., we wczesnych latach 70. ubiegłego stulecia dr Edgar F. Codd opracował model relacyjnych baz danych i ich postaci znormalizowane. W 1974 roku naukowcy w laboratorium IBM w San Jose rozpoczęli pracę nad dużym projektem, nazwanym System R., który miał potwierdzić możliwość zastosowania modelu relacyjnego. Jednocześnie dr Donald Chamberlin i jego współpracownicy rozpoczęli prace nad zdefiniowaniem języka baz danych. Stworzyli SEQUEL (ang. *structured english query language*), czyli język pozwalający użytkownikom na wykonywanie zapytań do baz danych za pomocą jasno zdefiniowanych zdań w stylu języka angielskiego. Ze względów prawnych nazwa tego języka została później zmieniona na SQL (ang. *structured query language*).

Oparte na SQL pierwsze komercyjne systemy zarządzania bazami danych (ang. *database management systems*, DBMS) pojawiły się pod koniec lat 70. ubiegłego wieku. Wraz z intensyfikacją prac nad językami baz danych pojawiła się standaryzacja, która miała na celu uproszczenie wykonywania zadań, a społeczność zdecydowała się na SQL. W procesie standaryzacji wzięły udział organizacje zarówno amerykańskie, jak też międzynarodowe (ANSI i ISO) i w 1986 roku został zatwierdzony pierwszy standard SQL. Ten standard był później kilkukrotnie modyfikowany (SQL:1999, SQL:2003, SQL:2008 itd.), a nazwa zawierała rok wydania danej wersji standardu. Pojęcie *standard SQL* oznacza aktualną wersję standardu SQL.

MySQL rozszerza standard SQL i udostępnia pewną funkcjonalność dodatkową. Na przykład MySQL implementuje `STRAIGHT_JOIN`, czyli składnię nierozpoznawaną przez inne systemy DBMS.

W rozdziale zaprezentujemy implementację SQL w MySQL, która często jest określana mianem operacji CRUD: `create` (pol. *tworzenie*), `read` (pol. *odczytywanie*), `update` (pol. *uaktualnianie*), `delete` (pol. *usuwanie*). Pokażemy, jak za pomocą zapytania `SELECT` można odczytywać dane z bazy danych, a także jak wskazywać dane do pobrania i kolejność, w której mają być wyświetlone. Wyjaśnimy również podstawy modyfikowania baz danych za pomocą zapytań `INSERT` (dodawanie danych), `UPDATE` (zmiana danych) i `DELETE` (usuwanie danych). Na koniec omówimy używanie niestandardowych zapytań `SHOW TABLES` i `SHOW COLUMNS`, pozwalających na analizowanie bazy danych.

Używanie bazy danych sakila

W rozdziale 2. omówiliśmy zasady tworzenia diagramu bazy danych na podstawie modelu ER. Zaprezentowaliśmy także kroki pozwalające skonwertować model ER na format, który będzie miał sens podczas tworzenia relacyjnej bazy danych. W tym miejscu poznasz strukturę bazy danych `sakila` w MySQL, co pozwoli Ci poznać różne modele relacyjnych baz danych. Nie będziemy wyjaśniać zapytań SQL używanych do tworzenia bazy danych — to jest temat rozdziału 4.

Jeżeli baza danych `sakila` nie została jeszcze zaimportowana do serwera MySQL, należy to zrobić teraz za pomocą procedury omówionej w rozdziale 2.

Do wskazania `sakila` jako bieżącej bazy danych służy zapytanie `USE`. Wykonaj je po nawiązaniu połączenia z serwerem MySQL:

```
mysql> USE sakila;
Database changed
mysql>
```

To, jaka baza danych jest aktywna, można sprawdzić przez wykonanie zapytania `SELECT DATABASE()`:

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| sakila     |
+------------+
1 row in set (0.00 sec)
```

Zobaczmy teraz, z jakich tabel składa się baza danych `sakila`. W tym celu należy skorzystać z zapytania `SHOW TABLES`:

```
mysql> SHOW TABLES;
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| ...                        |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.00 sec)
```

Jak dotąd nie było żadnych niespodzianek. Dowiedzmy się więcej na temat poszczególnych tabel tworzących bazę danych `sakila`. Zaczniemy od zapytania `SHOW COLUMNS` w celu przeanalizowania tabeli `actor` (zauważ, że dane wyjściowe zostały zawężone, aby mieściły się na stronie drukowanej książki):

```
mysql> SHOW COLUMNS FROM actor;
+-------------+-------------------+------+-----+-------------------+...
| Field       | Type              | Null | Key | Default           |...
+-------------+-------------------+------+-----+-------------------+...
| actor_id    | smallint unsigned | NO   | PRI | NULL              |...
| first_name  | varchar(45)       | NO   |     | NULL              |...
| last_name   | varchar(45)       | NO   | MUL | NULL              |...
| last_update | timestamp         | NO   |     | CURRENT_TIMESTAMP |...
+-------------+-------------------+------+-----+-------------------+...
 ...+-----------------------------------------------+
 ...| Extra                                         |
 ...+-----------------------------------------------+
 ...| auto_increment                                |
 ...|                                               |
 ...|                                               |
 ...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
 ...+-----------------------------------------------+
4 rows in set (0.01 sec)
```

Słowo kluczowe `DESCRIBE` jest odpowiednikiem zapytania `SHOW COLUMN FROM` i może być skrócone do postaci po prostu `DESC`. Dlatego też poprzednie zapytanie można zapisać w następującej postaci:

```
mysql> DESC actor;
```

Wygenerowane dane wyjściowe będą identyczne. Przeanalizujmy nieco dokładniej strukturę tabeli `actor` — zawiera cztery kolumny: `actor_id`, `first_name`, `last_name` i `last_update`. Mamy również możliwość wyodrębnienia typów tych kolumn: `actor_id` — `smallint`, `first_name` i `last_name` — `varchar(45)`, `last_update` — `timestamp`. Żadna z tych kolumn nie akceptuje wartości `NULL` (oznaczającej brak wartości), kolumna `actor_id` jest kluczem podstawowym (`PRI`), `last_name` zaś to pierwsza kolumna indeksu nieunikatowego (`MUL`). W tym miejscu nie przejmuj się szczegółami, najważniejsze są nazwy kolumn, które będą używane w zapytaniach SQL.

Przechodzimy teraz do przeanalizowania tabeli `city` przez wykonanie zapytania `DESC`:

```
mysql> DESC city;
+-------------+-------------------+------+-----+-------------------+...
| Field       | Type              | Null | Key | Default           |...
+-------------+-------------------+------+-----+-------------------+...
| city_id     | smallint unsigned | NO   | PRI | NULL              |...
| city        | varchar(50)       | NO   |     | NULL              |...
| country_id  | smallint unsigned | NO   | MUL | NULL              |...
| last_update | timestamp         | NO   |     | CURRENT_TIMESTAMP |...
+-------------+-------------------+------+-----+-------------------+...
...+-----------------------------------------------+
...| Extra                                         |
...+-----------------------------------------------+
...| auto_increment                                |
...|                                               |
...|                                               |
```

```
...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
...+-----------------------------------------------+
4 rows in set (0.01 sec)
```

Wartość `DEFAULT_GENERATED` widoczna w kolumnie `Extra` wskazuje, że ta konkretna kolumna używa wartości domyślnej. To jest notacja stosowana w MySQL 8.0, nie znajdziesz jej w serwerach MySQL 5.7 ani MariaDB 10.5.

Najważniejsze jest zapoznanie się z kolumnami w poszczególnych tabelach, ponieważ będą one często używane w dalszej części rozdziału podczas omawiania zapytań.

W następnym podrozdziale dowiesz się, jak analizować dane przechowywane przez MySQL w bazie danych `sakila` i jej tabelach.

Zapytanie SELECT i podstawowe techniki wykonywania zapytań

W poprzednim rozdziale wyjaśniliśmy, jak zainstalować i skonfigurować MySQL, jak używać powłoki serwera MySQL, a także pokrótce omówiliśmy model ER. Teraz możesz rozpocząć poznawanie języka SQL, który przez wszystkich klientów MySQL jest wykorzystywany do analizowania i przetwarzania danych. Ten podrozdział zawiera wprowadzenie do najczęściej używanego słowa kluczowego SQL, czyli `SELECT`. Wyjaśnimy podstawowe elementy stylu i składni, funkcjonalność klauzuli `WHERE`, operatory boolowskie oraz sortowanie (większość tych informacji ma również zastosowanie do zapytań `INSERT`, `UPDATE` i `DELETE`, omówionych w dalszej części rozdziału). To nie jest pełne omówienie zapytania `SELECT` — więcej informacji na jego temat znajdziesz w rozdziale 5., w którym poznasz jego bardziej zaawansowaną funkcjonalność.

Zapytanie SELECT dotyczące pojedynczej tabeli

W swojej najprostszej postaci zapytanie `SELECT` pobiera dane z wszystkich rekordów i kolumn tabeli. Z poziomu powłoki nawiąż połączenie z serwerem MySQL i wybierz bazę danych `sakila`:

```
mysql> USE sakila;
Database changed
```

Zaczynamy od pobrania wszystkich danych z tabeli `language`:

```
mysql> SELECT * FROM language;
+-------------+----------+---------------------+
| language_id | name     | last_update         |
+-------------+----------+---------------------+
|           1 | English  | 2006-02-15 05:02:19 |
|           2 | Italian  | 2006-02-15 05:02:19 |
|           3 | Japanese | 2006-02-15 05:02:19 |
|           4 | Mandarin | 2006-02-15 05:02:19 |
|           5 | French   | 2006-02-15 05:02:19 |
|           6 | German   | 2006-02-15 05:02:19 |
+-------------+----------+---------------------+
6 rows in set (0.00 sec)
```

Wygenerowane dane wyjściowe składają się z sześciu rekordów, z których każdy zawiera wartości dla wszystkich kolumn w tabeli. Skoro wiemy o dostępności sześciu języków, możemy poznać związane z nimi szczegóły, np. identyfikator oraz datę i godzinę ostatniej aktualizacji danego języka.

Proste zapytanie `SELECT` składa się z czterech komponentów. Są to:

1. Słowo kluczowe `SELECT`.
2. Kolumny przeznaczone do wyświetlenia. Gwiazdka to znak wieloznaczny wskazujący wszystkie kolumny.
3. Słowo kluczowe `FROM`.
4. Nazwa tabeli.

W omawianym przykładzie pobieramy wszystkie kolumny z tabeli `language` i to są dane zwrócone przez MySQL.

Wypróbujemy teraz inne proste zapytanie `SELECT`. Tym razem zostaną pobrane wszystkie kolumny tabeli `city`:

```
mysql> SELECT * FROM city;
+---------+-----------------------+------------+---------------------+
| city_id | city                  | country_id | last_update         |
+---------+-----------------------+------------+---------------------+
|       1 | A Corua (La Corua)    |         87 | 2006-02-15 04:45:25 |
|       2 | Abha                  |         82 | 2006-02-15 04:45:25 |
|       3 | Abu Dhabi             |        101 | 2006-02-15 04:45:25 |
|     ...                                                            |
|     599 | Zhoushan              |         23 | 2006-02-15 04:45:25 |
|     600 | Ziguinchor            |         83 | 2006-02-15 04:45:25 |
+---------+-----------------------+------------+---------------------+
600 rows in set (0.00 sec)
```

Ta tabela zawiera rekordy dotyczące 600 miast, a wygenerowane dane wyjściowe mają tę samą podstawową strukturę jak w pierwszym przykładzie.

Ten przykład dostarcza pewnych informacji związanych z działaniem relacji między tabelami. Rozważ pierwszy rekord danych wyjściowych. Wartością kolumny `country_id` jest 87. Jak się dowiesz w dalszej części rozdziału, można sprawdzić tabelę `country` pod kątem podanej wartości i wówczas okaże się, że identyfikator 87 oznacza Hiszpanię. Tworzenie zapytań obejmujących relacje między tabelami zostanie omówione nieco dalej w rozdziale.

Jeżeli przeanalizujesz pełne dane wyjściowe, zobaczysz, że istnieje wiele różnych miast, które mają tę samą wartość kolumny `country_id`. Powtórzenie tej wartości nie stanowi problemu, ponieważ można się spodziewać, że w kraju może istnieć wiele miast (relacja typu „jeden do wielu").

W tym momencie prawdopodobnie nie masz problemów z wyborem bazy danych, wyświetlaniem jej tabel i pobieraniem wszystkich danych tabeli za pomocą zapytania `SELECT`. Aby utrwalić umiejętności, możesz eksperymentować z innymi tabelami w bazie danych `sakila`. Pamiętaj o dostępności zapytania `SHOW TABLES`, które wyświetla nazwy tabel.

Wybór kolumn

Wcześniej w rozdziale użyliśmy gwiazdki do pobrania wszystkich kolumn tabeli. Jeżeli nie chcesz wyświetlać wszystkich kolumn, bardzo łatwo możesz podać te, które Cię interesują, w wybranej kolejności i rozdzielone przecinkami. Na przykład w celu wyświetlenia jedynie kolumny `city` z tabeli `city` należy wykonać następujące zapytanie:

```
mysql> SELECT city FROM city;
+--------------------+
| city               |
+--------------------+
| A Corua (La Corua) |
| Abha               |
| Abu Dhabi          |
| Acua               |
| Adana              |
+--------------------+
5 rows in set (0.00 sec)
```

Natomiast jeśli chcesz wyświetlić kolumny `city` i `city_id`, w podanej kolejności, zapytanie powinno mieć postać:

```
mysql> SELECT city, city_id FROM city;
+--------------------+---------+
| city               | city_id |
+--------------------+---------+
| A Corua (La Corua) |       1 |
| Abha               |       2 |
| Abu Dhabi          |       3 |
| Acua               |       4 |
| Adana              |       5 |
+--------------------+---------+
5 rows in set (0.01 sec)
```

Daną kolumnę można nawet wyświetlić więcej niż tylko raz:

```
mysql> SELECT city, city FROM city;
+--------------------+--------------------+
| city               | city               |
+--------------------+--------------------+
| A Corua (La Corua) | A Corua (La Corua) |
| Abha               | Abha               |
| Abu Dhabi          | Abu Dhabi          |
| Acua               | Acua               |
| Adana              | Adana              |
+--------------------+--------------------+
5 rows in set (0.00 sec)
```

Wprawdzie to wydaje się bezcelowe, ale może się okazać użyteczne w połączeniu z aliasami w bardziej skomplikowanych zapytaniach, do czego powrócimy w rozdziale 5.

W zapytaniu `SELECT` można podać nazwy bazy danych, tabeli i kolumny. To pozwala uniknąć zapytania `USE` oraz pracować z bazą danych i tabelą bezpośrednio w zapytaniu `SELECT`. Takie podejście pomaga również w unikaniu niejasności, co dokładniej wyjaśnimy w dalszej części rozdziału. Na przykład załóżmy, że chcesz pobrać kolumnę `name` z tabeli `language` w bazie danych `sakila`. W tym celu możesz wykonać następujące zapytanie:

```
mysql> SELECT name FROM sakila.language;
+----------+
| name     |
+----------+
| English  |
| Italian  |
| Japanese |
| Mandarin |
| French   |
| German   |
+----------+
6 rows in set (0.01 sec)
```

Komponent `sakila.language` po słowie kluczowym `FROM` wskazuje bazę danych `sakila` i jej tabelę `language`. Przed wykonaniem tego zapytania nie trzeba już wykonywać `USE sakila;`. Taka składnia może być stosowana także z innymi zapytaniami SQL, m.in. `UPDATE`, `DELETE`, `INSERT` i `SHOW` — do tego jeszcze powrócimy w dalszej części rozdziału.

Wybieranie rekordów za pomocą klauzuli WHERE

W tym punkcie przedstawimy klauzulę `WHERE` i wyjaśnimy, jak używać operatorów do zdefiniowania wyrażeń. Z wyrażeniami spotykamy się w różnych zapytaniach `SELECT`, `UPDATE` i `DELETE`, jak to zobaczysz w dalszej części rozdziału.

Podstawy pracy z klauzulą WHERE

Klauzula `WHERE` to narzędzie o potężnych możliwościach, pozwalające na filtrowanie rekordów zwracanych przez zapytanie `SELECT`. Jest używane w celu zwrócenia rekordów spełniających zdefiniowany warunek, np. wartość kolumny dokładnie odpowiadająca danemu ciągowi tekstowemu, liczba większa lub mniejsza niż podana wartość, ciąg tekstowy z konkretnym prefiksem. Niemalże wszystkie przykłady w tym i w późniejszych rozdziałach zawierają klauzulę `WHERE`, którą dzięki temu doskonale poznasz.

Klauzula `WHERE` w najprostszej postaci dokładnie dopasowuje wartość. Rozważ przykład, w którym chcesz przeanalizować przechowywane w tabeli `language` szczegóły dotyczące języka angielskiego. Oto zapytanie, które należy wykonać:

```
mysql> SELECT * FROM sakila.language WHERE name = 'English';
+-------------+---------+---------------------+
| language_id | name    | last_update         |
+-------------+---------+---------------------+
|           1 | English | 2006-02-15 05:02:19 |
+-------------+---------+---------------------+
1 row in set (0.00 sec)
```

MySQL zwróci wszystkie rekordy dopasowane do kryteriów wyszukiwania — w omawianym przykładzie to będzie tylko jeden rekord z wszystkimi kolumnami.

Wypróbujemy teraz inny przykład dokładnego dopasowania. Przyjmujemy założenie, że chcesz poznać imię aktora, którego kolumna `actor_id` w tabeli `actors` ma wartość `4`. W takim przypadku należy wykonać następujące zapytanie:

```
mysql> SELECT first_name FROM actor WHERE actor_id = 4;
+------------+
| first_name |
+------------+
| JENNIFER   |
+------------+
1 row in set (0.00 sec)
```

W tym przykładzie została wskazana kolumna i rekord — po słowie kluczowym `SELECT` znalazła się nazwa kolumny `first_name` i później klauzula `WHERE` w postaci `WHERE actor_id = 4`.

Jeżeli warunek spowoduje dopasowanie więcej niż tylko jednego rekordu, wygenerowane dane wyjściowe będą zawierały wszystkie dopasowania. Załóżmy, że chcesz wyświetlić informacje o wszystkich miastach znajdujących się w Brazylii, której to identyfikator `country_id` w bazie danych `sakila` ma wartość 15. Oto zapytanie, które należy wykonać:

```
mysql> SELECT city FROM city WHERE country_id = 15;
+----------------------+
| city                 |
+----------------------+
| Alvorada             |
| Angra dos Reis       |
| Anpolis              |
| Aparecida de Goinia  |
| Araatuba             |
| Bag                  |
| Belm                 |
| Blumenau             |
| Boa Vista            |
| Braslia              |
| ...                  |
+----------------------+
28 rows in set (0.00 sec)
```

Dane wyjściowe zawierają 28 miast leżących w Brazylii. Jeżeli można byłoby połączyć informacje pobrane z tabeli `city` z informacjami pochodzącymi z tabeli `country`, wówczas można by wyświetlić nazwy miast i krajów, w których te miasta leżą. Omówienie tego rodzaju zapytań znajdziesz w dalszej części rozdziału.

Przechodzimy teraz do pobierania wartości pochodzących z pewnego zakresu. Pobieranie wielu wartości jest w przypadku zakresów liczbowych dość proste. Dlatego też zaczniemy od wyszukania nazw wszystkich miast, których identyfikator `city_id` ma wartość mniejszą niż 5. W tym celu należy wykonać następujące zapytanie:

```
mysql> SELECT city FROM city WHERE city_id < 5;
+--------------------+
| city               |
+--------------------+
| A Corua (La Corua) |
| Abha               |
| Abu Dhabi          |
| Acua               |
+--------------------+
4 rows in set (0.00 sec)
```

Podczas pracy z liczbami do często używanych operatorów zaliczamy równy (=), większy niż (>), mniejszy niż (<), mniejszy niż lub równy (<=), większy niż lub równy (>=) oraz nierówny (<> lub !=).

Rozważ jeszcze jeden przykład. Jeżeli chcesz wyszukać wszystkie języki o identyfikatorze `language_id` innym niż 2, musisz wykonać następujące zapytanie:

```
mysql> SELECT language_id, name FROM sakila.language
    -> WHERE language_id <> 2;
+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|           3 | Japanese |
|           4 | Mandarin |
|           5 | French   |
|           6 | German   |
+-------------+----------+
5 rows in set (0.00 sec)
```

Wygenerowane dane wyjściowe zawierają pierwszy, trzeci i wszystkie kolejne języki przechowywane w tabeli. Zwróć uwagę na możliwość użycia operatora <> lub != podczas definiowania warunku *nierówności*.

Te same operatory można zastosować w trakcie pracy z ciągami tekstowymi. Domyślnie podczas porównywania ciągu tekstowego wielkość liter nie ma znaczenia i jest używane bieżące kodowanie znaków. Spójrz na kolejne zapytanie:

```
mysql> SELECT first_name FROM actor WHERE first_name < 'B';
+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

Skoro wielkość liter nie ma znaczenia, znaki `B` i `b` są uznawane za ten sam filtr. Dlatego też następne zapytanie spowoduje wygenerowanie dokładnie tych samych danych wyjściowych:

```
mysql> SELECT first_name FROM actor WHERE first_name < 'b';
+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
```

```
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

Inne często wykonywane zadanie podczas pracy z ciągami tekstowymi to znalezienie dopasowania rozpoczynającego się od określonego prefiksu, zawierającego dany ciąg tekstowy lub kończącego się określonym sufiksem. Na przykład chcesz znaleźć wszystkie albumy, których nazwa rozpoczyna się od słowa *Retro*. W takim przypadku należy skorzystać z operatora `LIKE` w klauzuli `WHERE`. Spójrz na przykład zapytania, w którym szukany jest film o tytule zawierającym słowo `family`:

```
mysql> SELECT title FROM film WHERE title LIKE '%family%';
+----------------+
| title          |
+----------------+
| CYCLONE FAMILY |
| DOGMA FAMILY   |
| FAMILY SWEET   |
+----------------+
3 rows in set (0.00 sec)
```

Zapoznaj się teraz ze sposobem działania operatora `LIKE`. Jest on używany podczas pracy z ciągami tekstowymi i oznacza, że dopasowanie musi zawierać w ciągu tekstowym podany wzorzec. W omawianym przykładzie użyliśmy operatora w postaci `LIKE '%family%'`, co oznacza, że ciąg tekstowy musi zawierać słowo `family`, przed i po którym może znajdować się zero lub więcej znaków. Większość ciągów używanych razem z operatorem `LIKE` zawiera znak procentu (`%`), który tutaj jest znakiem wieloznacznym dopasowującym wszystkie możliwe ciągi tekstowe. Tego znaku można użyć do zdefiniowania ciągu tekstowego kończącego się określonym sufiksem, np. `"%yłem"`, lub ciągu tekstowego rozpoczynającego się wskazanym podciągiem tekstowym, np. `"Korupcja%"`.

Na przykład `"Jan%"` spowoduje dopasowanie wszystkich ciągów tekstowych rozpoczynających się od ciągu tekstowego `Jan`, np. `Jan Kowalski`, `Jan Adam Nowak` itd. Wzorzec `"%Nowak"` spowoduje zaś dopasowanie wszystkich ciągów tekstowych zawierających na końcu ciąg tekstowy `Nowak`. Z kolei wzorzec `"%Nowak%"` oznacza dopasowanie wszystkich ciągów tekstowych zawierających gdziekolwiek słowo `Nowak`.

Jeżeli w klauzuli `LIKE` chcesz dopasować dokładnie jeden znak, możesz użyć znaku wieloznacznego podkreślenie (_). Na przykład w celu pobrania tytułów wszystkich filmów, w których występuje aktor o imieniu rozpoczynającym się od trzech liter `NAT`, musisz wykonać następujące zapytanie:

```
mysql> SELECT title FROM film_list WHERE actors LIKE 'NAT_%';
+----------------------+
| title                |
+----------------------+
```

```
| FANTASY TROOPERS     |
| FOOL MOCKINGBIRD     |
| HOLES BRANNIGAN      |
| KWAI HOMEWARD        |
| LICENSE WEEKEND      |
| NETWORK PEAK         |
| NUTS TIES            |
| TWISTED PIRATES      |
| UNFORGIVEN ZOOLANDER |
+----------------------+
9 rows in set (0.04 sec)
```

Ogólnie rzecz biorąc, należy unikać stosowania znaku wieloznacznego % na początku wzorca, np. jak pokazaliśmy w kolejnym zapytaniu:

```
mysql> SELECT title FROM film WHERE title LIKE '%day%';
```

Wprawdzie otrzymasz żądany wynik, ale podczas jego generowania MySQL nie użyje indeksu. Podanie znaku wieloznacznego wymusza na MySQL odczytanie całej tabeli w celu przygotowania wyniku. Jeżeli tabela zawiera miliony rekordów, to będzie oznaczało ogromny spadek wydajności działania takiego zapytania.

Łączenie warunków za pomocą AND, OR, NOT i XOR

Dotychczas klauzuli `WHERE` używaliśmy do sprawdzania tylko jednego warunku i zwracania wszystkich spełniających go rekordów. Istnieje możliwość połączenia dwóch lub więcej warunków za pomocą operatorów boolowskich `AND`, `OR`, `NOT` i `XOR`.

Rozpoczynamy od przykładu. Załóżmy, że chcesz znaleźć tytuły wszystkich filmów zaliczanych do kategorii `sci-fi`, które zostały ocenione jako `PG`. Dzięki operatorowi `AND` to jest bardzo proste zadanie:

```
mysql> SELECT title FROM film_list WHERE category LIKE 'Sci-Fi'
    -> AND rating LIKE 'PG';
+----------------------+
| title                |
+----------------------+
| CHAINSAW UPTOWN      |
| CHARADE DUFFEL       |
| FRISCO FORREST       |
| GOODFELLAS SALUTE    |
| GRAFFITI LOVE        |
| MOURNING PURPLE      |
| OPEN AFRICAN         |
| SILVERADO GOLDFINGER |
| TITANS JERK          |
| TROJAN TOMORROW      |
| UNFORGIVEN ZOOLANDER |
| WONDERLAND CHRISTMAS |
+----------------------+
12 rows in set (0.07 sec)
```

Operacja `AND` w klauzuli `WHERE` powoduje ograniczenie wyników do tych rekordów, które spełniają oba kryteria.

Operator OR jest używany do wyszukiwania rekordów spełniających przynajmniej jeden warunek. Aby to zilustrować, załóżmy, że interesuje nas lista filmów zaliczanych do kategorii Children lub Family. W takim przypadku należy wykonać następujące zapytanie:

```
mysql> SELECT title FROM film_list WHERE category LIKE 'Children'
    -> OR category LIKE 'Family';
+-----------------------+
| title                 |
+-----------------------+
| AFRICAN EGG           |
| APACHE DIVINE         |
| ATLANTIS CAUSE        |
...
| WRONG BEHAVIOR        |
| ZOOLANDER FICTION     |
+-----------------------+
129 rows in set (0.04 sec)
```

Operator OR w klauzuli WHERE ogranicza rekordy do tych spełniających przynajmniej jeden ze zdefiniowanych warunków. Zwróć uwagę na uporządkowanie wygenerowanych wyników. Tutaj to tylko zbieg okoliczności — dane zostają wyświetlone w kolejności, w jakiej zostały dodane do bazy danych. Do tematu sortowania danych wyjściowych powrócimy nieco dalej w rozdziale.

Istnieje możliwość połączenia operatorów AND i OR, przy czym trzeba wyraźnie wskazać, który jako pierwszy ma być użyty podczas definiowania warunków. Umieszczenie fragmentu zapytania w nawiasie pomaga w zapewnieniu większej przejrzystości wyrażenia, przy czym nawiasy mogą być stosowanie podobnie jak w działaniach matematycznych. Załóżmy, że interesują nas filmy zaliczane do kategorii Sci-Fi lub Family oraz ocenione jako PG:

```
mysql> SELECT title FROM film_list WHERE (category like 'Sci-Fi'
    -> OR category LIKE 'Family') AND rating LIKE 'PG';
+-----------------------+
| title                 |
+-----------------------+
| BEDAZZLED MARRIED     |
| CHAINSAW UPTOWN       |
| CHARADE DUFFEL        |
| CHASING FIGHT         |
| EFFECT GLADIATOR      |
...
| UNFORGIVEN ZOOLANDER  |
| WONDERLAND CHRISTMAS  |
+-----------------------+
30 rows in set (0.07 sec)
```

Użycie nawiasu wyraźnie wskazuje kolejność w trakcie wykonywania zapytania: chcemy otrzymać filmy z wymienionych kategorii, ale ocenione jako PG.

Dzięki wykorzystaniu nawiasu można zmienić kolejność przetwarzania zapytania. Najłatwiej to pokazać na przykładzie obliczeń:

```
mysql> SELECT (2+2)*3;
+---------+
| (2+2)*3 |
+---------+
```

```
|      12 |
+---------+
1 row in set (0.00 sec)

mysql> SELECT 2+2*3;
+-------+
| 2+2*3 |
+-------+
|     8 |
+-------+
1 row in set (0.00 sec)
```

Jednym z trudniejszych problemów do wykrycia jest zapytanie wykonywane bez żadnych błędów składni i jednocześnie zwracające wartości inne niż oczekiwane. Wprawdzie nawias nie wpływa na operator AND, ale sytuacja przedstawia się zupełnie inaczej w przypadku operatora OR. Spójrz na kolejne zapytanie:

```
mysql> SELECT * FROM sakila.city WHERE city_id = 3
    -> OR city_id = 4 AND country_id = 60;
+---------+-----------+------------+---------------------+
| city_id | city      | country_id | last_update         |
+---------+-----------+------------+---------------------+
|       3 | Abu Dhabi |        101 | 2006-02-15 04:45:25 |
|       4 | Acua      |         60 | 2006-02-15 04:45:25 |
+---------+-----------+------------+---------------------+
2 rows in set (0.00 sec)
```

Po zmianie kolejności operacji otrzymamy zupełnie odmienny wynik:

```
mysql> SELECT * FROM sakila.city WHERE country_id = 60
    -> AND city_id = 3 OR city_id = 4;
+---------+------+------------+---------------------+
| city_id | city | country_id | last_update         |
+---------+------+------------+---------------------+
|       4 | Acua |         60 | 2006-02-15 04:45:25 |
+---------+------+------------+---------------------+
1 row in set (0.00 sec)
```

Użycie nawiasu ułatwia zrozumienie zapytania i zwiększa prawdopodobieństwo otrzymania wyniku zgodnego z oczekiwaniami. Zachęcamy do stosowania nawiasów wszędzie tam, gdzie istnieje niebezpieczeństwo, że MySQL błędnie odczyta intencję programisty. Nie ma żadnego powodu, dla którego należałoby opierać się na wybranej przez MySQL kolejności wykonywania operacji.

Operator jednoargumentowy NOT odwraca znaczenie polecenia boolowskiego. We wcześniejszej części rozdziału przedstawiliśmy przykład, w którym zostały wyświetlone rekordy wszystkich języków, poza tym o identyfikatorze language_id wynoszącym 2. To zapytanie można utworzyć także z użyciem operatora NOT:

```
mysql> SELECT language_id, name FROM sakila.language
    -> WHERE NOT (language_id = 2);
+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|           3 | Japanese |
```

```
|           4 | Mandarin |
|           5 | French   |
|           6 | German   |
+-------------+----------+
5 rows in set (0.01 sec)
```

Wyrażenie w nawiasie, (`language_id = 2`), określa warunek konieczny do dopasowania, a operator `NOT` odwraca go. W efekcie otrzymasz wszystkie rekordy, poza tym spełniającym dany warunek. Istnieje jeszcze wiele innych sposobów tworzenia klauzuli `WHERE` z zastosowaniem tego samego podejścia. W rozdziale 5. przedstawimy wybrane z nich, charakteryzujące się większą niż pozostałe wydajnością działania.

Przechodzimy teraz do przykładu zawierającego operator `NOT` i nawias. Załóżmy, że chcesz pobrać listę wszystkich filmów o identyfikatorze mniejszym niż 7, ale jednocześnie nie te o identyfikatorze 4 lub 6. W tym celu możesz wykonać następujące zapytanie:

```
mysql> SELECT fid,title FROM film_list WHERE FID < 7 AND NOT (FID = 4 OR FID = 6);
+------+------------------+
| fid  | title            |
+------+------------------+
|    1 | ACADEMY DINOSAUR |
|    2 | ACE GOLDFINGER   |
|    3 | ADAPTATION HOLES |
|    5 | AFRICAN EGG      |
+------+------------------+
4 rows in set (0.06 sec)
```

Zrozumienie kolejności wykonywania operatorów może być trudne i czasami administratorzy baz danych poświęcają dużo czasu na debugowanie zapytań i ustalanie, dlaczego wygenerowane dane wyjściowe są inne niż oczekiwane. Przedstawiona tutaj lista zawiera wszystkie operatory wymienione w kolejności od najwyższego priorytetu do najniższego. Operatory zamieszczone w jednym wierszu mają ten sam priorytet:

- `INTERVAL`,
- `BINARY`, `COLLATE`,
- `!`,
- `-` (jednoargumentowy minus), `~` (jednoargumentowy operator bitowy),
- `^`,
- `*`, `/`, `DIV`, `%`, `MOD`,
- `-`, `+`,
- `<<`, `>>`,
- `&`,
- `\|`,
- `=` (porównanie), `<=>`, `>=`, `>`, `<=`, `<`, `<>`, `!=`, `IS`, `LIKE`, `REGEXP`, `IN`, `MEMBER OF`,
- `BETWEEN`, `CASE`, `WHEN`, `THEN`, `ELSE`,
- `NOT`,

- AND, &&,
- XOR,
- OR, \|\|,
- = (przypisanie), := .

Istnieje możliwość łączenia tych operatorów i tym samym zmiany sposobów, na jakie jest otrzymywany żądany wynik. Na przykład możesz zdefiniować zapytanie pobierające tytuły filmów dostępnych w cenie od 2 do 4 dolarów, które są zaliczane do kategorii `Documentary` lub `Horror` oraz w których wystąpił aktor o imieniu `Bob`:

```
mysql> SELECT title
    -> FROM film_list
    -> WHERE price BETWEEN 2 AND 4
    -> AND (category LIKE 'Documentary' OR category LIKE 'Horror')
    -> AND actors LIKE '%BOB%';
+------------------+
| title            |
+------------------+
| ADAPTATION HOLES |
+------------------+
1 row in set (0.08 sec)
```

Zanim przejdziemy do sortowania, zwróć uwagę na możliwość wykonania zapytania niedopasowującego żadnych wyników. W takim przypadku wartością zwrotną będzie zbiór pusty:

```
mysql> SELECT title FROM film_list
    -> WHERE price BETWEEN 2 AND 4
    -> AND (category LIKE 'Documentary' OR category LIKE 'Horror')
    -> AND actors LIKE '%GRIPPA%';

Empty set (0.04 sec)
```

Klauzula ORDER BY

Wyjaśniliśmy, jak pobierać kolumny i wskazywać rekordy, które zostaną zwrócone w wyniku wykonania zapytania. Natomiast zupełnie pominęliśmy kwestie związane z kontrolowaniem sposobu wyświetlania danych wyjściowych zapytania. W relacyjnej bazie danych rekordy znajdujące się w tabeli tworzą zbiór. Nie istnieje żadna wewnętrzna kolejność stosowana dla tych rekordów, więc serwerowi MySQL trzeba nakazać sortowanie wyniku, jeśli dane wyjściowe mają być wyświetlone w konkretnej kolejności. W tym punkcie dowiesz się, jak do tego celu wykorzystać klauzulę `ORDER BY`. Trzeba koniecznie dodać, że sortowanie nie wpływa na *zwracane* dane, lecz jedynie na ich *kolejność* zastosowaną podczas wyświetlania rekordów.

Tabele InnoDB w MySQL mają indeks specjalny o nazwie *indeks klastrowany*, przechowujący dane rekordu. W trakcie definiowania klucza podstawowego w tabeli InnoDB używa tego indeksu klastrowanego. Załóżmy, że jest wykonywane zapytanie oparte na kluczu podstawowym. W takim przypadku zwrócone rekordy zostaną uporządkowane w kolejności rosnącej według klucza podstawowego. Jednak w celu wymuszenia określonej kolejności zawsze zaleca się użycie klauzuli `ORDER BY`.

Przyjmujemy założenie o konieczności wyświetlenia listy pierwszych 10 klientów zapisanych w bazie danych `sakila`, posortowanych alfabetycznie według imienia (kolumna `name`). W takim przypadku należy wykonać następujące zapytanie:

```
mysql> SELECT name FROM customer_list
    -> ORDER BY name
    -> LIMIT 10;
+------------------+
| name             |
+------------------+
| AARON SELBY      |
| ADAM GOOCH       |
| ADRIAN CLARY     |
| AGNES BISHOP     |
| ALAN KAHN        |
| ALBERT CROUSE    |
| ALBERTO HENNING  |
| ALEX GRESHAM     |
| ALEXANDER FENNELL |
| ALFRED CASILLAS  |
+------------------+
10 rows in set (0.01 sec)
```

Kolumna `ORDER BY` wskazuje na konieczność sortowania danych, a po niej znajduje się nazwa kolumny, która powinna zostać użyta jako klucz sortowania. W omawianym przykładzie dane są sortowane w kolejności rosnącej, alfabetycznie według imienia. Domyślnie podczas sortowania wielkość liter nie ma znaczenia i stosowana jest kolejność rosnąca. MySQL automatycznie stosuje sortowanie alfabetyczne, ponieważ kolumny zawierają wartości w postaci ciągów tekstowych. Sposób sortowania ciągów tekstowych jest określany na podstawie kodowania znaków i ich kolejności. Więcej informacji na ten temat znajduje się w rozdziale 4. W większości przykładów zamieszczonych w książce zostało przyjęte założenie o użyciu domyślnych ustawień sortowania.

Przechodzimy do kolejnego przykładu. Tym razem dane wyjściowe tabeli `address` są sortowane w kolejności rosnącej na podstawie wartości kolumny `last_update`. Zapytanie wyświetla tylko pięć pierwszych rekordów:

```
mysql> SELECT address, last_update FROM address
    -> ORDER BY last_update LIMIT 5;
+-----------------------------+---------------------+
| address                     | last_update         |
+-----------------------------+---------------------+
| 1168 Najafabad Parkway      | 2014-09-25 22:29:59 |
| 1031 Daugavpils Parkway     | 2014-09-25 22:29:59 |
| 1924 Shimonoseki Drive      | 2014-09-25 22:29:59 |
| 757 Rustenburg Avenue       | 2014-09-25 22:30:01 |
| 1892 Nabereznyje Telny Lane | 2014-09-25 22:30:02 |
+-----------------------------+---------------------+
5 rows in set (0.00 sec)
```

Jak możesz zobaczyć, mamy możliwość sortowania różnego typu kolumn. Co więcej, można przeprowadzać sortowanie na podstawie dwóch lub więcej kolumn. Załóżmy, że chcesz sortować adresy alfabetycznie i jednocześnie grupować je według dzielnicy:

```
mysql> SELECT address, district FROM address
    -> ORDER BY district, address;
+--------------------------------------+----------------------+
| address                              | district             |
+--------------------------------------+----------------------+
| 1368 Maracabo Boulevard              |                      |
| 18 Duisburg Boulevard                |                      |
| 962 Tama Loop                        |                      |
| 535 Ahmadnagar Manor                 | Abu Dhabi            |
| 669 Firozabad Loop                   | Abu Dhabi            |
| 1078 Stara Zagora Drive              | Aceh                 |
| 663 Baha Blanca Parkway              | Adana                |
| 842 Salzburg Lane                    | Adana                |
| 614 Pak Kret Street                  | Addis Abeba          |
| 751 Lima Loop                        | Aden                 |
| 1157 Nyeri Loop                      | Adygea               |
| 387 Mwene-Ditu Drive                 | Ahal                 |
| 775 ostka Drive                      | al-Daqahliya         |
| ...                                  |                      |
| 1416 San Juan Bautista Tuxtepec Avenue | Zufar              |
| 138 Caracas Boulevard                | Zulia                |
+--------------------------------------+----------------------+
603 rows in set (0.00 sec)
```

Sortowanie może odbywać się również w kolejności malejącej. Zachowujesz możliwość kontrolowania sposobu działania dla każdego klucza sortowania. Załóżmy, że chcesz sortować adresy w kolejności malejącej i dzielnice również w kolejności malejącej. W takim przypadku należy wykonać następujące zapytanie:

```
mysql> SELECT address,district FROM address
    -> ORDER BY district ASC, address DESC
    -> LIMIT 10;
+------------------------+-------------+
| address                | district    |
+------------------------+-------------+
| 962 Tama Loop          |             |
| 18 Duisburg Boulevard  |             |
| 1368 Maracabo Boulevard |            |
| 669 Firozabad Loop     | Abu Dhabi   |
| 535 Ahmadnagar Manor   | Abu Dhabi   |
| 1078 Stara Zagora Drive | Aceh       |
| 842 Salzburg Lane      | Adana       |
| 663 Baha Blanca Parkway | Adana      |
| 614 Pak Kret Street    | Addis Abeba |
| 751 Lima Loop          | Aden        |
+------------------------+-------------+
10 rows in set (0.01 sec)
```

W razie wystąpienia kolizji wartości i braku zdefiniowanego klucza sortowania kolejność sortowania będzie nieustalona. To może nie mieć dla Ciebie znaczenia, ponieważ np. nie interesuje Cię kolejność wyświetlenia dwóch klientów o identycznym imieniu i nazwisku. Jeżeli jednak w takim przypadku chcesz zastosować konkretną kolejność, w klauzuli `ORDER BY` musisz podać więcej kolumn, jak pokazaliśmy we wcześniejszym przykładzie.

Klauzula LIMIT

W kilku wcześniejszych zapytaniach została użyta klauzula `LIMIT`. To użyteczne niestandardowe polecenie SQL pozwalające na określanie liczby rekordów, które mają znaleźć się w danych wyjściowych. W najprostszej postaci ta klauzula pozwala na ograniczanie liczby rekordów zwracanych przez zapytanie `SELECT`. To przydaje się, gdy trzeba ograniczyć ilość danych przekazywanych przez sieć lub wyświetlanych na ekranie. Klauzuli `LIMIT` można użyć np. w celu pobrania próbki danych z tabeli, jak pokazaliśmy w kolejnym zapytaniu:

```
mysql> SELECT name FROM customer_list LIMIT 10;
+------------------+
| name             |
+------------------+
| VERA MCCOY       |
| MARIO CHEATHAM   |
| JUDY GRAY        |
| JUNE CARROLL     |
| ANTHONY SCHWAB   |
| CLAUDE HERZOG    |
| MARTIN BALES     |
| BOBBY BOUDREAU   |
| WILLIE MARKHAM   |
| JORDAN ARCHULETA |
+------------------+
10 rows in set (0.00 sec)
```

Klauzula `LIMIT` może mieć dwa argumenty. W omawianym przykładzie pierwszy wskazuje pierwszy rekord przeznaczony do zwrócenia, drugi zaś określa maksymalną liczbę zwracanych rekordów. Można się spotkać z określeniem pierwszego argumentu mianem *przesunięcia*. Załóżmy, że chcemy otrzymać pięć rekordów, przy czym pierwsze pięć rekordów wyniku ma być pominięte. W takim przypadku zostaną zwrócone rekordy począwszy od szóstego w wyniku. Jeżeli nie przesuwamy rekordów w klauzuli `LIMIT`, wówczas jej pierwszy argument ma wartość 0. Aby w omawianym przykładzie zastosować przesunięcie i zwrócić pięć rekordów począwszy od szóstego, trzeba wykonać przedstawione tutaj zapytanie:

```
mysql> SELECT name FROM customer_list LIMIT 5, 5;
+------------------+
| name             |
+------------------+
| CLAUDE HERZOG    |
| MARTIN BALES     |
| BOBBY BOUDREAU   |
| WILLIE MARKHAM   |
| JORDAN ARCHULETA |
+------------------+
5 rows in set (0.00 sec)
```

To spowoduje wyświetlenie przez zapytanie `SELECT` rekordów od 6. do 10.

Istnieje składnia alternatywna dla słowa kluczowego `LIMIT`: zamiast `LIMIT 10, 5` można użyć zapisu `LIMIT 10 OFFSET 5`. W tej składni `OFFSET` powoduje odrzucenie *N* podanych wartości.

Spójrz na przykład zapytania bez zastosowania przesunięcia:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10;
+----+------------------+
| ID | name             |
+----+------------------+
|  1 | MARY SMITH       |
|  2 | PATRICIA JOHNSON |
|  3 | LINDA WILLIAMS   |
|  4 | BARBARA JONES    |
|  5 | ELIZABETH BROWN  |
|  6 | JENNIFER DAVIS   |
|  7 | MARIA MILLER     |
|  8 | SUSAN WILSON     |
|  9 | MARGARET MOORE   |
| 10 | DOROTHY TAYLOR   |
+----+------------------+
10 rows in set (0.00 sec)
```

Oto to samo zapytanie, ale z zastosowaniem przesunięcia o 5:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10 OFFSET 5;
+----+----------------+
| ID | name           |
+----+----------------+
|  6 | JENNIFER DAVIS |
|  7 | MARIA MILLER   |
|  8 | SUSAN WILSON   |
|  9 | MARGARET MOORE |
| 10 | DOROTHY TAYLOR |
| 11 | LISA ANDERSON  |
| 12 | NANCY THOMAS   |
| 13 | KAREN JACKSON  |
| 14 | BETTY WHITE    |
| 15 | HELEN HARRIS   |
+----+----------------+
10 rows in set (0.01 sec)
```

Złączanie dwóch tabel

Dotychczas w zapytaniach `SELECT` używaliśmy tylko jednej tabeli. Jednak w większości przypadków będą potrzebne informacje pochodzące jednocześnie z więcej niż tylko jednej tabeli. Podczas przeglądania tabel znajdujących się w bazie danych `sakila` stało się jasne, że stosowanie relacji może pomóc w wykonaniu bardziej interesujących zapytań. Na przykład dobrze jest poznać nazwę kraju, w którym leży dane miasto. W tym punkcie dowiesz się, jak można tworzyć zapytania wykorzystujące złączenie dwóch tabel. Dokładniejsze wyjaśnienie tego tematu i bardziej zaawansowane przykłady znajdziesz w rozdziale 5.

W tym rozdziale przedstawimy tylko jedną składnię złączenia. Są jeszcze dwie inne (`LEFT` i `RIGHT JOIN`), zapewniające odmienny sposób pobierania danych z dwóch lub większej liczby tabel. Użytą tutaj składnią jest `INNER JOIN`, zaliczana do najczęściej stosowanej w codziennej pracy. Najpierw spójrz na przykład, a następnie wyjaśnimy sposób działania tego zapytania:

```
mysql> SELECT city, country FROM city INNER JOIN country
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;
+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.00 sec)
```

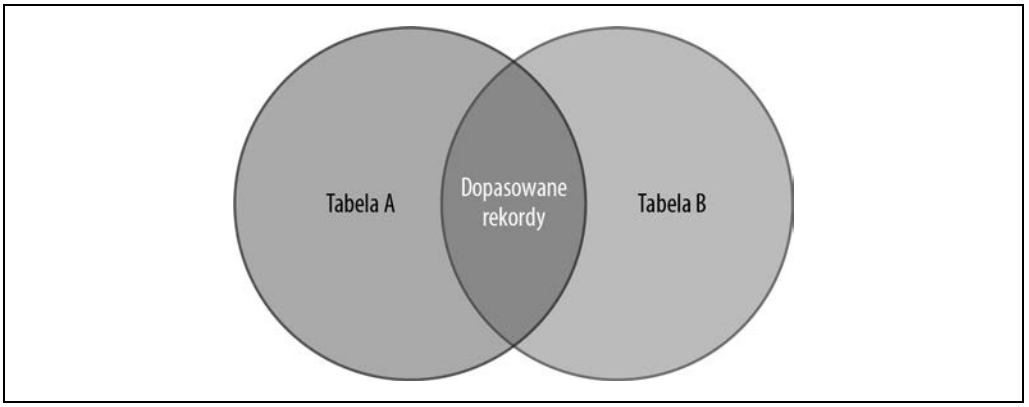
Dane wyjściowe zawierają nazwy miast leżących w krajach, których identyfikator `country_id` ma wartość mniejszą niż 5. Po raz pierwszy możesz wyświetlić nazwy krajów, w których leżą poszczególne miasta.

Jak działa złączenie `INNER JOIN`? To zapytanie składa się z dwóch części. Pierwsza to nazwy dwóch tabel rozdzielone słowami kluczowymi `INNER JOIN`. Druga to słowo kluczowe `ON` określające kolumny niezbędne do zdefiniowania warunku. W omawianym przykładzie zostaną złączone kolumny `city` i `country`, co zostało wyrażone w postaci `city INNER JOIN country`. (W przypadku zwykłego złączenia `INNER JOIN` kolejność kolumn nie ma znaczenia, więc zapis `country INNER JOIN city` będzie miał taki sam efekt). W klauzuli `ON` (`ON city.country_id = country.country_id`) wskazujemy MySQL kolumny przechowujące relację zachodzącą między tabelami. Przypomnij to sobie z projektu bazy danych i wcześniejszej analizy zamieszczonej w rozdziale 2.

Jeżeli w warunku złączenia nazwy kolumn w obu tabelach użytych podczas dopasowania są takie same, wówczas można skorzystać z klauzuli `USING`:

```
mysql> SELECT city, country FROM city
    -> INNER JOIN country using (country_id)
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;
+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.01 sec)
```

Złączenie typu `INNER JOIN` w sposób graficzny pokazaliśmy na diagramie Venna na rysunku 3.1.



Rysunek 3.1. Diagram Venna dla złączenia typu INNER JOIN

Zanim zakończymy to krótkie omówienie zapytania `SELECT`, chcemy jeszcze zaprezentować wprowadzenie do funkcji pozwalających na agregowanie wartości. Załóżmy, że chcesz sprawdzić, ile miast włoskich znajduje się w naszej bazie danych. W tym celu można przeprowadzić złączenie dwóch tabel, a następnie ustalić liczbę rekordów, których identyfikator `country_id` ma wartość wskazującą Włochy. Oto zapytanie, które należy wykonać:

```
mysql> SELECT COUNT(1) FROM city INNER JOIN country
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id = 49
    -> ORDER BY country, city;
+----------+
| count(1) |
+----------+
|        7 |
+----------+
1 row in set (0.00 sec)
```

Więcej informacji na temat funkcjonalności zapytania `SELECT` i jego funkcji agregacji, w tym użytej tutaj `COUNT()`, znajdziesz w rozdziale 5.

Zapytanie INSERT

Zapytanie `INSERT` jest używane podczas wstawiania nowych danych w bazie danych. W podrozdziale wyjaśnimy podstawową składnię zapytania `INSERT` i przedstawimy proste przykłady pokazujące dodawanie nowych rekordów do bazy danych `sakila`. W rozdziale 4. zaś dowiesz się, jak wczytywać dane z istniejących tabel lub zewnętrznych źródeł danych.

Podstawy zapytania INSERT

Wstawianie informacji do bazy danych zwykle następuje w dwóch sytuacjach: podczas przekazywania ogromnej ilości danych na etapie tworzenia bazy, a także podczas wstawiania informacji w trakcie normalnej pracy z bazą. W serwerze MySQL zostały wbudowane różne optymalizacje gotowe do zastosowania w obu wymienionych sytuacjach. Co ważne, w obu mamy do dyspozycji

różne składnie SQL, które ułatwiają pracę z serwerem. W tym punkcie omówimy podstawową składnię zapytania `INSERT` oraz pokażemy przykłady wstawiania danych hurtowo lub pojedynczo.

Rozpoczynamy od prostego zadania, jakim jest wstawienie nowego rekordu do tabeli `language`. To wymaga zrozumienia struktury tabeli. Jak już wyjaśniliśmy w poprzednim rozdziale, strukturę tabeli można wyświetlić za pomocą zapytania `SHOW COLUMNS`:

```
mysql> SHOW COLUMNS FROM language;
+-------------+------------------+------+-----+-------------------+...
| Field       | Type             | Null | Key | Default           |...
+-------------+------------------+------+-----+-------------------+...
| language_id | tinyint unsigned | NO   | PRI | NULL              |...
| name        | char(20)         | NO   |     | NULL              |...
| last_update | timestamp        | NO   |     | CURRENT_TIMESTAMP |...
+-------------+------------------+------+-----+-------------------+...
...+-----------------------------------------------+
...| Extra                                         |
...+-----------------------------------------------+
...| auto_increment                                |
...|                                               |
...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
...+-----------------------------------------------+
3 rows in set (0.00 sec)
```

Z wyświetlonych danych jasno wynika, że wartość kolumny `language_id` jest generowana automatycznie, a wartość kolumny `last_update` jest uaktualniana w trakcie każdej operacji `UPDATE`. Więcej na temat używania skrótu `AUTO_INCREMENT` w celu automatycznego przypisywania kolejnego dostępnego identyfikatora dowiesz się w rozdziale 4.

Dodamy teraz nowy rekord dla języka portugalskiego. Można to zrobić na dwa sposoby. Pierwszy jest najczęściej stosowany i pozostawia serwerowi MySQL zadanie wstawienia wartości domyślnej dla `language_id`, np. jak pokazaliśmy w kolejnym zapytaniu:

```
mysql> INSERT INTO language VALUES (NULL, 'Portuguese', NOW());
Query OK, 1 row affected (0.10 sec)
```

Jeżeli teraz wykonasz zapytanie `SELECT` w tabeli `language`, zobaczysz wstawiony przed chwilą rekord:

```
mysql> SELECT * FROM language;
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
+-------------+------------+---------------------+
7 rows in set (0.00 sec)
```

Zwróć uwagę na użycie funkcji `NOW()` do wygenerowania wartości dla kolumny `last_update`. Wartością zwrotną funkcji `NOW()` jest bieżąca data i godzina serwera MySQL.

Drugi sposób to ręczne wstawienie wartości kolumny `language_id`. Skoro obecnie w tabeli znajduje się siedem rekordów języków, następną wartością `language_id` jest 8. To można sprawdzić za pomocą przedstawionego tutaj zapytania SQL:

```
mysql> SELECT MAX(language_id) FROM language;
+------------------+
| max(language_id) |
+------------------+
|                7 |
+------------------+
1 row in set (0.00 sec)
```

Funkcja `MAX()` podaje wartość maksymalną kolumny wskazanej w parametrze funkcji. To jest znacznie bardziej eleganckie rozwiązanie niż wykonanie zapytania `SELECT language_id FROM language`, które wyświetla wszystkie rekordy i wymaga od Ciebie ich przeanalizowania w celu ustalenia wartości maksymalnej. Wprawdzie dodanie klauzul `ORDER BY` i `LIMIT` ułatwia zadanie, ale użycie funkcji `MAX()` jest znacznie prostsze niż wykonanie zapytania w stylu `SELECT language_id FROM language ORDER BY language_id DESC LIMIT 1`, zwracającego dokładnie ten sam wynik.

Teraz można już przystąpić do wstawienia rekordu. W omawianym przykładzie wartość kolumny `last_update` również będzie wstawiona ręcznie. Spójrz na zapytanie, które trzeba wykonać:

```
mysql> INSERT INTO language VALUES (8, 'Russian', '2020-09-26 10:35:00');
Query OK, 1 row affected (0.02 sec)
```

MySQL informuje, że zapytanie dotyczyło jednego rekordu (tutaj został dodany), co można potwierdzić przez ponowne wyświetlenie zawartości tabeli `language`:

```
mysql> SELECT * FROM language;
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
|           8 | Russian    | 2020-09-26 10:35:00 |
+-------------+------------+---------------------+
8 rows in set (0.00 sec)
```

Styl zapytania `INSERT` dotyczącego pojedynczego rekordu wychwytuje powielające się klucze podstawowe i natychmiast kończy działanie po wykryciu tego faktu. Załóżmy, że próbujemy wstawić inny rekord o takiej samej wartości identyfikatora `language_id` jak istniejąca już w bazie danych:

```
mysql> INSERT INTO language VALUES (8, 'Arabic', '2020-09-26 10:35:00');
ERROR 1062 (23000): Duplicate entry '8' for key 'language.PRIMARY'
```

Działanie zapytania `INSERT` zostało przerwane natychmiast po wykryciu powtarzającego się klucza. Wprawdzie można użyć klauzuli `IGNORE` w celu wyeliminowania tego komunikatu o błędzie, ale trzeba pamiętać, że mimo to rekord nie zostanie wstawiony:

```
mysql> INSERT IGNORE INTO language VALUES (8, 'Arabic', '2020-09-26 10:35:00');
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

W większości przypadków chcesz wiedzieć o wszelkich potencjalnych problemach (w końcu klucz podstawowy ma być unikatowy), więc składnia IGNORE rzadko jest stosowana.

Istnieje możliwość jednoczesnego wstawienia wielu wartości:

```
mysql> INSERT INTO language VALUES (NULL, 'Spanish', NOW()),
    -> (NULL, 'Hebrew', NOW());
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

Zwróć uwagę na to, że MySQL zupełnie inaczej informuje o wstawieniu wielu danych niż w sytuacji, gdy wstawiany jest pojedynczy rekord.

Pierwszy komunikat informuje o liczbie wstawionych rekordów. Pierwsza część drugiego komunikatu wskazuje zaś, ile rekordów zostało faktycznie przetworzonych. Jeżeli użyjesz zapytania INSERT IGNORE podczas próby wstawienia powtarzającego się rekordu (tzn. o takim samym kluczu podstawowym jak klucz w już istniejącym rekordzie), wówczas MySQL nie wstawi tego rekordu i zgłosi problem w drugiej części drugiego komunikatu.

```
mysql> INSERT IGNORE INTO language VALUES (9, 'Portuguese', NOW()),
    -> (11, 'Hebrew', NOW());
Query OK, 1 row affected, 1 warning (0.01 sec)
Records: 2  Duplicates: 1  Warnings: 1
```

Więcej informacji na temat ostrzeżeń w drugim komunikacie danych wyjściowych znajdziesz w rozdziale 4.

Składnie alternatywne

Istnieją składnie alternatywne dla zaprezentowanej w poprzednim punkcie składni VALUES. Zamierzamy je tutaj omówić, a także wyjaśnić ich wady i zalety. Jeżeli wolisz pozostać przy już przedstawionej składni, możesz przejść do następnego podrozdziału, dotyczącego zapytania DELETE.

Używana dotąd składnia VALUES ma pewne zalety: sprawdza się podczas wstawiania danych hurtowo i pojedynczo, powoduje wygenerowanie komunikatu o błędzie w razie pominięcia wartości dla którejkolwiek kolumny, a także nie wymaga wpisywania nazw kolumn. Zarazem ma pewne wady: wymaga zapamiętania kolejności kolumn, wymaga podania wartości dla każdej kolumny, ściśle wiąże składnię ze strukturą tabeli. Dlatego też zmiana struktury tabeli pociąga za sobą konieczność modyfikacji zapytań INSERT. Na szczęście można uniknąć tych wad przez zróżnicowanie składni.

Przyjmujemy następujące założenie: pamiętasz, że tabela actor ma cztery kolumny, pamiętasz ich nazwy, ale nie pamiętasz kolejności kolumn. Rekord możesz wstawić do tabeli za pomocą przedstawionego tutaj zapytania:

```
mysql> INSERT INTO actor (actor_id, first_name, last_name, last_update)
    -> VALUES (NULL, 'Vinicius', 'Grippa', NOW());
Query OK, 1 row affected (0.03 sec)
```

Nazwy kolumn zostały podane w nawiasie po nazwie tabeli, a wartości dla tych kolumn zostały wymienione w nawiasie po słowie kluczowym VALUES. W omawianym przykładzie będzie utworzony nowy rekord, w którym: wartość 201 zostaje wstawiona do kolumny actor_id (pamiętaj, że ta kolumna ma właściwość auto_increment), wartość Vinicius zostaje wstawiona do kolumny

first_name, wartość Grippa zostaje wstawiona do kolumny last_name, a w kolumnie last_update zostaje wstawiony bieżący znacznik czasu. Zaletami tej składni są czytelność i elastyczność (wyeliminowanie trzeciej wady, wymienionej nieco wcześniej) oraz niezależność od kolejności kolumn (wyeliminowanie pierwszej wady). Natomiast wadą takiego podejścia jest konieczność pamiętania nazw kolumn i wpisania ich w zapytaniu.

Nowa składnia może również wyeliminować drugą z wcześniej wymienionych wad i pozwala na wstawianie wartości dla jedynie wybranych kolumn. Aby się przekonać, jak użyteczne może być to podejście, przeanalizujemy teraz tabelę city:

```
mysql> DESC city;
+-------------+---------------------+------+-----+-------------------+...
| Field       | Type                | Null | Key | Default           |...
+-------------+---------------------+------+-----+-------------------+...
| city_id     | smallint(5) unsigned | NO   | PRI | NULL              |...
| city        | varchar(50)         | NO   |     | NULL              |...
| country_id  | smallint(5) unsigned | NO   | MUL | NULL              |...
| last_update | timestamp           | NO   |     | CURRENT_TIMESTAMP |...
+-------------+---------------------+------+-----+-------------------+...
...+-----------------------------------------------+
...| Extra                                         |
...+-----------------------------------------------+
...| auto_increment                                |
...|                                               |
...|                                               |
...| on update CURRENT_TIMESTAMP                   |
...|-----------------------------------------------+
4 rows in set (0.00 sec)
```

Zwróć uwagę, że kolumna last_update ma wartość domyślną CURRENT_TIMESTAMP. Dlatego też jeśli nie podasz wartości dla tej kolumny, MySQL wstawi bieżącą datę i godzinę. Dokładnie tego oczekujemy: podczas wstawiania nowego rekordu nie chcemy zajmować się sprawdzaniem bieżącej daty i godziny, aby następnie wpisać te dane w zapytaniu. Spójrz na próbę wstawienia rekordu niezawierającego wartości dla wszystkich kolumn:

```
mysql> INSERT INTO city (city, country_id) VALUES ('Bebedouro', 19);
Query OK, 1 row affected (0.00 sec)
```

Nie podaliśmy wartości kolumny city_id, więc serwer MySQL domyślnie użył kolejnej dostępnej wartości (ta kolumna ma zdefiniowaną właściwość auto_increment), kolumna last_update przechowuje zaś bieżącą datę i godzinę. Możesz to sprawdzić za pomocą następującego zapytania:

```
mysql> SELECT * FROM city where city like 'Bebedouro';
+---------+-----------+------------+---------------------+
| city_id | city      | country_id | last_update         |
+---------+-----------+------------+---------------------+
|     601 | Bebedouro |         19 | 2021-02-27 21:34:08 |
+---------+-----------+------------+---------------------+
1 row in set (0.01 sec)
```

Takie podejście można zastosować również podczas hurtowego wstawiania danych:

```
mysql> INSERT INTO city (city,country_id) VALUES
    -> ('Sao Carlos',19),
    -> ('Araraquara',19),
```

```
    -> ('Ribeirao Preto',19);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Poza koniecznością pamiętania nazw kolumn i wpisania ich w zapytaniu, wadą takiego podejścia jest także niebezpieczeństwo przypadkowego pominięcia wartości kolumn. W takich kolumnach MySQL wstawi wartości domyślne. Wszystkie kolumny w tabeli MySQL mają wartość domyślną `NULL`, o ile podczas tworzenia bądź modyfikowania tabeli nie zostanie wyraźnie zdefiniowana inna.

Gdy trzeba użyć wartości domyślnych dla kolumn tabeli, można skorzystać ze słowa kluczowego `DEFAULT` (obsługiwane przez serwer MySQL 5.7 i nowsze). Spójrz na przykład zapytania wstawiającego rekord do tabeli `country` z użyciem wymienionego słowa kluczowego:

```
mysql> INSERT INTO country VALUES (NULL, 'Uruguay', DEFAULT);
Query OK, 1 row affected (0.01 sec)
```

Słowo kluczowe `DEFAULT` nakazuje MySQL użycie wartości domyślnej dla danej kolumny, więc w omawianym przykładzie zostanie wstawiona bieżąca data i godzina. Zaletą takiego podejścia jest możliwość wykorzystania funkcjonalności hurtowego wstawiania danych z wartościami domyślnymi bez obawy, że jakaś kolumna zostanie przypadkowo pominięta.

Mamy jeszcze inną alternatywną składnię, `INSERT`. W tym podejściu podaje się nazwy kolumn i ich wartości, więc nie trzeba mentalnie mapować listy wartości na wcześniej zdefiniowaną listę kolumn. Spójrz na przykład wstawienia nowego rekordu do tabeli `country`:

```
mysql> INSERT INTO country SET country_id=NULL,
    -> country='Bahamas', last_update=NOW();
Query OK, 1 row affected (0.01 sec)
```

Ta składnia wymaga podania nazwy tabeli, słowa kluczowego `SET`, a następnie rozdzielonych przecinkami par: nazwa kolumny, znak równości, wartość. Jeżeli dla kolumny nie została podana wartość, wówczas będzie użyta wartość domyślna. Wadą także tego podejścia jest niebezpieczeństwo przypadkowego pominięcia wartości dla kolumn oraz konieczność pamiętania nazw kolumn i ich wpisania. Ponadto poważną wadą jest brak możliwości użycia tej metody podczas hurtowego wstawiania danych.

Istnieje również możliwość wstawiania wartości zwróconych przez zapytanie. Dokładne omówienie tego tematu znajdziesz w rozdziale 7.

Zapytanie DELETE

Zapytanie `DELETE` jest używane do usunięcia jednego lub więcej rekordów z tabeli. W tym podrozdziale omówimy usuwanie rekordów z pojedynczej tabeli, w rozdziale 7. zaś dowiesz się, jak za pomocą jednego zapytania usuwać dane z dwóch lub więcej tabel.

Podstawy pracy z zapytaniem DELETE

Najprostszy przykład użycia zapytania `DELETE` dotyczy usunięcia wszystkich rekordów w tabeli. Załóżmy, że chcesz opróżnić tabelę `rental`. Możesz to zrobić przez wykonanie przedstawionego tutaj zapytania:

```
mysql> DELETE FROM rental;
Query OK, 16044 rows affected (2.41 sec)
```

Ta składnia zapytania DELETE nie zawiera żadnych nazw kolumn, ponieważ zapytanie zostało wykonane w celu usunięcia wszystkich rekordów, a nie pewnych wartości z rekordu. Jeżeli chcesz wyzerować lub zmienić wartość w rekordzie, skorzystaj z zapytania UPDATE, które zostanie dokładnie omówione w następnym podrozdziale. Pamiętaj, że zapytanie DELETE nie powoduje usunięcia tabeli. Dlatego też po usunięciu wszystkich rekordów tabeli rental nadal można wykonywać do niej zapytania:

```
mysql> SELECT * FROM rental;
Empty set (0.00 sec)
```

Strukturę tej tabeli wciąż można analizować za pomocą zapytań DESCRIBE i SHOW CREATE TABLE, a ponadto można wstawiać do niej nowe rekordy z wykorzystaniem zapytania INSERT. Jeżeli chcesz usunąć tabelę, musisz wykonać zapytanie DROP, o którym więcej dowiesz się w rozdziale 4.

Jeżeli tabela ma relację z inną tabelą, wówczas próba usunięcia tabeli zakończy się niepowodzeniem ze względu na istnienie ograniczenia klucza zewnętrznego:

```
mysql> DELETE FROM language;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
constraint fails (`sakila`.`film`, CONSTRAINT `fk_film_language` FOREIGN KEY
(`language_id`) REFERENCES `language` (`language_id`) ON UPDATE CASCADE)
```

Używanie klauzul WHERE, ORDER BY i LIMIT

Jeżeli w poprzednim punkcie usunąłeś rekordy, ponownie wczytaj bazę danych sakila z wykorzystaniem procedury przedstawionej w poprzednim rozdziale. Przykłady zamieszczone w tym punkcie wymagają istnienia rekordów tabeli rental.

W celu usunięcia jednego lub więcej rekordów tabeli, ale nie wszystkich, należy użyć klauzuli WHERE. Działa ona w dokładnie taki sam sposób jak w przypadku zapytania SELECT. Na przykład załóżmy, że z tabeli rental chcesz usunąć wszystkie rekordy, których identyfikator rental_id ma wartość mniejszą niż 10. Można to zrobić za pomocą przedstawionego tutaj zapytania:

```
mysql> DELETE FROM rental WHERE rental_id < 10;
Query OK, 9 rows affected (0.01 sec)
```

W wyniku wykonania tego zapytania zostało usuniętych dziewięć rekordów spełniających podane kryteria.

Teraz przyjmujemy założenie, że z bazy danych chcesz usunąć informacje dotyczące wszystkich płatności klienta Mary Smith. Musisz zacząć od zapytania SELECT wykorzystującego złączenie INNER JOIN (więcej informacji na jego temat znajdziesz we wcześniejszej części rozdziału) do pobrania danych z tabel customer i payment:

```
mysql> SELECT first_name, last_name, customer.customer_id,
    -> amount, payment_date FROM payment INNER JOIN customer
    -> ON customer.customer_id=payment.customer_id
    -> WHERE first_name like 'Mary'
    -> AND last_name like 'Smith';
```

```
+------------+-----------+-------------+--------+---------------------+
| first_name | last_name | customer_id | amount | payment_date        |
+------------+-----------+-------------+--------+---------------------+
| MARY       | SMITH     |           1 |   2.99 | 2005-05-25 11:30:37 |
| MARY       | SMITH     |           1 |   0.99 | 2005-05-28 10:35:23 |
| MARY       | SMITH     |           1 |   5.99 | 2005-06-15 00:54:12 |
| MARY       | SMITH     |           1 |   0.99 | 2005-06-15 18:02:53 |
...
| MARY       | SMITH     |           1 |   1.99 | 2005-08-22 01:27:57 |
| MARY       | SMITH     |           1 |   2.99 | 2005-08-22 19:41:37 |
| MARY       | SMITH     |           1 |   5.99 | 2005-08-22 20:03:46 |
+------------+-----------+-------------+--------+---------------------+
32 rows in set (0.00 sec)
```

Następnie trzeba wykonać pokazane niżej zapytanie `DELETE`, które z tabeli `payment` usunie rekordy zawierające identyfikator `customer_id` o wartości 1:

```
mysql> DELETE FROM payment where customer_id=1;
Query OK, 32 rows affected (0.01 sec)
```

W zapytaniu `DELETE` można stosować klauzule `ORDER BY` i `LIMIT`. Zwykle używa się takiego rozwiązania do ograniczenia liczby usuwanych rekordów. Spójrz na poniższy przykład:

```
mysql> DELETE FROM payment ORDER BY customer_id LIMIT 10000;
Query OK, 10000 rows affected (0.22 sec)
```

Ze względu na potencjalne problemy z wydajnością działania gorąco zachęcamy do wykonywania zapytań dla małych zbiorów. Odpowiednia wielkość takiego zbioru zależy od sprzętu. Przyjęło się, że powinna się ona mieścić w przedziale od 20 000 do 40 000 dla każdej operacji.

Usuwanie wszystkich rekordów za pomocą zapytania TRUNCATE

Jeżeli chcesz usunąć wszystkie rekordy w tabeli, jest szybsza metoda niż pozbywanie się ich za pomocą zapytania `DELETE`. Gdy użyjesz zapytania `TRUNCATE TABLE`, MySQL zastosuje skrót polegający na usunięciu tabeli razem z jej strukturami, a następnie na ponownym utworzeniu tej tabeli. Gdy tabela zawiera wiele rekordów, to podejście będzie znacznie szybsze.

Warto w tym miejscu wspomnieć o błędzie w serwerze 5.6, który może wstrzymać działanie MySQL podczas operacji `TRUNCATE`, gdy serwer został skonfigurowany z ogromną pulą bufora InnoDB (o wielkości ponad 200 GB). Więcej informacji na temat tego błędu znajdziesz na stronie *https://bugs.mysql.com/bug.php?id=80060*.

Jeżeli chcesz usunąć wszystkie dane w tabeli `payment`, możesz skorzystać z następującego zapytania:

```
mysql> TRUNCATE TABLE payment;
Query OK, 0 rows affected (0.07 sec)
```

Zwróć uwagę na komunikat, zgodnie z którym to zapytanie wpłynęło na zero rekordów. W celu przyśpieszenia operacji MySQL nie sprawdza liczby usuniętych rekordów, więc wyświetlona w tym komunikacie liczba nie odzwierciedla faktycznej liczby usuniętych rekordów.

Zapytanie TRUNCATE TABLE pod wieloma względami różni się od DELETE. Warto wspomnieć przynajmniej o kilku z nich:

- Operacja TRUNCATE usuwa tabelę i tworzy ją od początku, co jest znacznie szybsze niż pojedyncze usuwanie rekordów, szczególnie w przypadku ogromnych tabel.
- Operacja TRUNCATE może prowadzić do niejawnej operacji COMMIT, więc nie można jej wycofać.
- Operacji TRUNCATE nie można przeprowadzić, jeśli sesja zawiera aktywną blokadę tabeli.

Typy tabel, transakcje i blokady będą dokładnie omówione w rozdziale 5. W praktyce wymienione ograniczenia nie mają wpływu na większość aplikacji, więc zapytanie TRUNCATE TABLE można stosować do przyśpieszenia operacji przetwarzania danych. Oczywiście usuwanie całych tabel w trakcie normalnej pracy nie zdarza się zbyt często. Wyjątkiem są tabele tymczasowe używane do przechowywania wyników zapytań wykonywanych przez danego użytkownika w sesji. Te tabele można spokojnie usunąć, nie tracąc przy tym oryginalnych danych.

Zapytanie UPDATE

Zapytanie UPDATE jest wykonywane w celu modyfikacji danych. W tym podrozdziale dowiesz się, jak można uaktualnić jeden lub więcej rekordów w pojedynczej tabeli. Natomiast bardziej złożone operacje uaktualniania zostaną omówione w rozdziale 7.

Jeżeli z bazy danych zostały usunięte jakiekolwiek rekordy, trzeba ją przywrócić przed lekturą tego podrozdziału.

Przykłady

W swojej najprostszej postaci zapytanie UPDATE uaktualnia wszystkie rekordy w tabeli. Załóżmy, że zachodzi potrzeba uaktualnienia kolumny amount tabeli payment przez zwiększenie wszystkich wartości o 10%. W tym celu można wykonać następujące zapytanie:

```
mysql> UPDATE payment SET amount=amount*1.1;
Query OK, 16025 rows affected, 16025 warnings (0.41 sec)
Rows matched: 16049  Changed: 16025  Warnings: 16025
```

Zwróć uwagę, że zapomnieliśmy o uaktualnieniu stanu kolumny last_update. Aby zapewnić spójność z oczekiwanym modelem bazy danych, to niedopatrzenie można naprawić za pomocą kolejnego zapytania:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53:00';
Query OK, 16049 rows affected (0.27 sec)
Rows matched: 16049  Changed: 16049  Warnings: 0
```

Istnieje możliwość użycia funkcji NOW() w celu uaktualnienia kolumny last_update bieżącym znacznikiem czasu. Spójrz na zapytanie wykorzystujące tę funkcję:

```
mysql> UPDATE payment SET last_update=NOW();
```

Drugi komunikat wygenerowany przez zapytanie UPDATE podaje ogólny efekt jego wykonania. W omawianym przykładzie ten komunikat ma następującą postać:

```
Rows matched: 16049  Changed: 16049  Warnings: 0
```

Pierwsza wartość to liczba dopasowanych rekordów. Ponieważ w omawianym przykładzie nie została użyta klauzula WHERE lub LIMIT, dopasowane są wszystkie rekordy tabeli. Druga wartość podaje liczbę rekordów wymagających modyfikacji. To z reguły będzie liczba mniejsza lub równa liczbie dopasowanych rekordów. Jeżeli ponownie wykonasz to zapytanie, otrzymasz nieco inne dane wyjściowe:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53:00';
Query OK, 0 rows affected (0.07 sec)
Rows matched: 16049  Changed: 0  Warnings: 0
```

Ponieważ znacznik czasu był już wcześniej zdefiniowany jako 2021-02-28 17:53:00, a ponadto nie ma warunku WHERE, tym razem zapytanie dopasowało wszystkie rekordy, choć żaden z nich nie został zmodyfikowany. Zauważ, że liczba zmienionych rekordów zawsze jest równa liczbie rekordów, których dotyczyło zapytanie — ta wartość jest wyświetlana w pierwszym komunikacie wygenerowanym przez zapytanie.

Używanie klauzul WHERE, ORDER BY i LIMIT

Bardzo często jest tak, że nie chcesz zmieniać wszystkich rekordów w tabeli. Zamiast tego chcesz uaktualnić jeden lub więcej rekordów dopasowanych do warunku. Podobnie jak w przypadku zapytań SELECT i DELETE, także w zapytaniu UPDATE można użyć do tego celu klauzuli WHERE. Ponadto podobnie jak w zapytaniu DELETE, także w UPDATE można używać razem klauzul ORDER BY i LIMIT, aby ograniczyć liczbę uaktualnianych rekordów.

Przechodzimy do przykładu pokazującego modyfikację jednego rekordu tabeli. Załóżmy, że aktorka Penelope Guiness zmieniła nazwisko. Jeżeli chcesz je uaktualnić w tabeli actor bazy danych, musisz wykonać następujące zapytanie:

```
mysql> UPDATE actor SET last_name= UPPER('cruz')
    -> WHERE first_name LIKE 'PENELOPE'
    -> AND last_name LIKE 'GUINESS';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Zgodnie z oczekiwaniami serwer MySQL dopasował jeden rekord i zmodyfikował go.

W celu kontrolowania liczby uaktualnianych rekordów można wykorzystać połączenie klauzul ORDER BY i LIMIT:

```
mysql> UPDATE payment SET last_update=NOW() LIMIT 10;
Query OK, 10 rows affected (0.01 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

Podobnie jak w przypadku zapytania DELETE, tę operację należy przeprowadzać na mniejszych zbiorach rekordów lub modyfikować jedynie wybrane. W omawianym przykładzie zapytanie dopasowało i zmodyfikowało 10 rekordów.

Poprzednie zapytanie ilustruje także ważny aspekt uaktualnień. Jak możesz zobaczyć, uaktualnienie składa się z dwóch etapów: dopasowania i modyfikacji. W trakcie pierwszego są wyszukiwane rekordy spełniające warunek zdefiniowany w klauzuli `WHERE`. Podczas drugiego zaś są uaktualniane rekordy wymagające modyfikacji.

Przeglądanie baz danych i tabel za pomocą zapytań SHOW i polecenia mysqlshow

Wyjaśniliśmy już, jak można używać zapytania `SHOW` do pobierania informacji dotyczących struktury bazy danych, jej tabel i kolumn tych tabel. Natomiast w tym podrozdziale zaprezentujemy najczęściej stosowane rodzaje zapytania `SHOW` na podstawie przykładów dotyczących informacji przechowywanych w bazie danych `sakila`. Polecenie `mysqlshow` oferuje tę samą funkcjonalność co wiele wariantów zapytania `SHOW`, ale bez konieczności uruchamiania klienta MySQL.

Zapytanie `SHOW DATABASES` wyświetla listę baz danych, do których możesz uzyskać dostęp. Po wykonaniu procedury instalacji przykładowych baz danych i wdrożeniu modelu banku (zobacz rozdział 2.) wygenerowane przez to zapytanie dane wyjściowe będą przedstawiały się następująco:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| bank_model         |
| employees          |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
8 rows in set (0.01 sec)
```

To jest lista baz danych, do których dostęp możesz uzyskać za pomocą zapytania `USE` (będzie dokładniej omówione w rozdziale 4.). Jeżeli masz uprawnienia dostępu do innych baz danych w serwerze, one również zostaną wymienione na tej liście. Zobaczysz jedynie bazy danych, do których masz uprawnienia, o ile nie dysponujesz uprawnieniem globalnym `SHOW DATABASES`. Dokładnie ten sam efekt można uzyskać w powłoce po wydaniu polecenia `mysqlshow`:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306
```

W zapytaniu `SHOW DATABASES` można użyć klauzuli `LIKE`. To użyteczne rozwiązanie, jeśli masz wiele baz danych i chcesz otrzymać krótszą listę w danych wyjściowych. Na przykład w celu wyświetlenia jedynie baz danych o nazwach rozpoczynających się na literę *s*, należy wykonać następujące zapytanie:

```
mysql> SHOW DATABASES LIKE 's%';
+---------------+
| Database (s%) |
+---------------+
| sakila        |
```

```
| sys           |
+---------------+
2 rows in set (0.00 sec)
```

Składnia klauzuli `LIKE` jest identyczna ze stosowaną w zapytaniu `SELECT`.

Jeżeli chcesz poznać zapytanie użyte do utworzenia bazy danych, skorzystaj z zapytania `SHOW CREATE DATABASE`. Dane wyjściowe wygenerowane przez poniższe zapytanie pokazują, jak została utworzona baza danych `sakila`:

```
mysql> SHOW CREATE DATABASE sakila;
*************************** 1. row ***************************
       Database: sakila
Create Database: CREATE DATABASE `sakila` /*!40100 DEFAULT CHARACTER SET
utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT ENCRYPTION='N' */
1 row in set (0.00 sec)
```

To prawdopodobnie jest najmniej interesujące zapytanie `SHOW`. Wyświetla ono jedynie zapytanie tworzące bazę danych. Zwróć uwagę na dodatkowe komentarze umieszczone między znakami /* i */:

```
40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
80016 DEFAULT ENCRYPTION='N'
```

Te komentarze zawierają słowa kluczowe ściśle związane z MySQL i dostarczające informacji, które prawdopodobnie nie będą zrozumiałe dla innych serwerów baz danych. Inne serwery zignorują te komentarze, więc składnia jest użyteczna dla zarówno MySQL, jak i innego oprogramowania serwerów baz danych. Parametr opcjonalny na początku komentarza określa minimalną wersję MySQL, która może przetworzyć tę konkretną instrukcję (np. `40100` wskazuje wersję 4.01.00). Starsze wersje MySQL po prostu zignorują te instrukcje. Więcej informacji na ten temat tworzenia baz danych znajdziesz w rozdziale 4.

Polecenie `SHOW TABLES` wyświetla listę tabel znajdujących się w bazie danych. Jeżeli chcesz wyświetlić tabele w bazie danych `sakila`, musisz wykonać następujące zapytanie:

```
mysql> SHOW TABLES FROM sakila;
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
```

```
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| ...                        |
+----------------------------+
23 rows in set (0.01 sec)
```

Gdy baza danych `sakila` została wcześniej wybrana za pomocą zapytania `USE sakila`, można skorzystać ze skróconej wersji zapytania:

```
mysql> SHOW TABLES;
```

Ten sam wynik otrzymasz po podaniu nazwy bazy danych jako argumentu polecenia `mysqlshow`, jak pokazaliśmy w kolejnym fragmencie kodu:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 sakila
```

Podobnie jak w przypadku zapytania `SHOW DATABASES`, nie zobaczysz tabel, do których nie masz uprawnień. To oznacza, że nie zobaczysz w bazie danych tabel, do których nie możesz uzyskać dostępu, nawet jeśli masz uprawnienie globalne `SHOW DATABASES`.

Zapytanie `SHOW COLUMNS` wyświetla listę kolumn w tabeli. Spójrz na przykład takiego zapytania dotyczącego tabeli `country`:

```
mysql> SHOW COLUMNS FROM country;
*************************** 1. row ***************************
  Field: country_id
   Type: smallint unsigned
   Null: NO
    Key: PRI
Default: NULL
  Extra: auto_increment
*************************** 2. row ***************************
  Field: country
   Type: varchar(50)
   Null: NO
    Key:
Default: NULL
  Extra:
*************************** 3. row ***************************
  Field: last_update
   Type: timestamp
   Null: NO
    Key:
Default: CURRENT_TIMESTAMP
  Extra: DEFAULT_GENERATED on update CURRENT_TIMESTAMP
3 rows in set (0.00 sec)
```

Wygenerowane dane wyjściowe zawierają nazwy wszystkich kolumn, ich typy, wielkość, informacje o tym, czy mogą mieć wartość `NULL`, czy są częścią klucza, jakie mają wartości domyślne, a także wszelkie inne informacje. W rozdziale 4. dowiesz się więcej o typach, kluczach, wartościach `NULL` i wartościach domyślnych. Jeżeli baza danych `sakila` nie została wcześniej wybrana za pomocą zapytania `USE`, wówczas jej nazwę możesz umieścić przed nazwą tabeli, np. `sakila.country`. W przeciwieństwie do wcześniejszych zapytań `SHOW` zawsze otrzymasz nazwy wszystkich kolumn, o ile masz dostęp do tabeli. Nie mają tutaj żadnego znaczenia określone uprawnienia dla wszystkich kolumn.

Dokładnie te same dane można otrzymać po wydaniu polecenia `mysqlshow` razem z nazwami bazy danych i tabeli:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 sakila country
```

Zapytanie `SHOW CREATE TABLE` wyświetla zapytanie użyte do utworzenia danej tabeli (tym tematem będziemy się zajmować w rozdziale 4.). Niektórzy użytkownicy preferują dane wyjściowe w tej postaci zamiast generowanej przez zapytanie `SHOW COLUMNS`, ponieważ format tych danych jest podobny do formatu danych wyświetlanych przez zapytanie `CREATE TABLE`. Spójrz na przykład zapytania `SHOW CREATE TABLE` dotyczącego tabeli `country`:

```
mysql> SHOW CREATE TABLE country\G
*************************** 1. row ***************************
       Table: country
Create Table: CREATE TABLE `country` (
  `country_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `country` varchar(50) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
  CURRENT_TIMESTAMP,
  PRIMARY KEY (`country_id`)
) ENGINE=InnoDB AUTO_INCREMENT=110 DEFAULT CHARSET=utf8mb4
  COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

MySQL: dane zawsze gotowe do akcji!

Wymagania względem baz danych stale rosną, co jest związane z dostępnością coraz większych ilości danych. Obserwujemy więc dynamiczny rozwój różnych systemów bazodanowych. Mimo że w tej branży pojawia się sporo nowych propozycji, relacyjne bazy danych wciąż cieszą się dużą popularnością. Spośród rozwiązań typu open source najbardziej znaną i lubianą bazą danych od lat pozostaje MySQL. Jest to oprogramowanie, które świetnie się sprawdza nawet w systemach operujących na dużych ilościach danych.

W tym gruntownie zaktualizowanym przewodniku znalazły się dokładne informacje dotyczące konfiguracji MySQL w takich systemach jak Linux, Windows i macOS, jak również w kontenerze Dockera. Przedstawiono tutaj zasady projektowania baz danych, a także modyfikowania już istniejących. Opisano techniki pracy w obciążonym środowisku produkcyjnym, pokazano też, jak stosować mechanizm transakcji i reguły zarządzania użytkownikami. Omówiono sposoby uzyskiwania wysokiej wydajności działania i dostępności serwera przy minimalnych kosztach. Zademonstrowano, jak dostrajać i zabezpieczać bazy, jak pracować z kopiami zapasowymi, wreszcie — jak używać plików konfiguracyjnych. W tym wydaniu pojawiły się nowe rozdziały poświęcone wysokiej dostępności serwera, mechanizmu równoważenia obciążenia i używania MySQL w chmurze.

W książce między innymi:

- gruntowne podstawy MySQL
- wdrażanie bazy danych MySQL, również w maszynach wirtualnych i w chmurze
- projektowanie bazy danych i tworzenie zapytań
- monitorowanie bazy danych i praca z kopiami zapasowymi
- optymalizacja kosztów używania bazy danych w chmurze
- koncepcje związane z bazami danych

Vinicius M. Grippa jest starszym inżynierem pomocy technicznej w firmie Percona i Oracle Ace Associate. Ma duże doświadczenie w rozwiązywaniu złożonych problemów związanych z bazą danych MySQL.

Sergey Kuzmichev jest starszym inżynierem pomocy technicznej w firmie Percona. Pasjonat pracy z bazami danych, skupia się na tworzeniu systemów o wysokim stopniu niezawodności. Przez wiele lat był administratorem baz danych i inżynierem DevOps.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8960-1
9 788328 389601
Cena: 129,00 zł