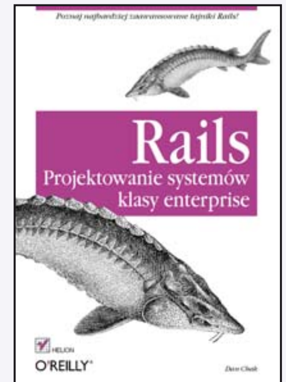
helion.pl

Rails. Projektowanie systemów klasy enterprise



Autor: Dan Chak
Tłumaczenie: Andrzej Grażyński
ISBN: 978-83-246-2198-9
Tytuł oryginału: Enterprise Rails
Format: 168×237, stron: 328

Poznaj najbardziej zaawansowane tajniki Rails!

- Jak zorganizować kod, wykorzystując system wtyczek lub moduły?
- Jakie zalety posiada architektura SOA?
- Jak zwiększyć wydajność Rails?

Rynek szkieletów aplikacji internetowych jest niezwykle urozmaicony. Wśród wielu dostępnych opcji można znaleźć tu rozwiązania przeznaczone dla projektów o różnej skali złożoności, zarówno te mniej, jak i bardziej popularne. Warto jednak sięgnąć po rozwiązanie absolutnie unikatowe i wyjątkowe – Rails. Szkielet ten świetnie sprawdza się zarówno w projektach małych, jak i tych klasy enterprise, a ponadto znany jest ze swoich możliwości, wydajności oraz elastyczności. Warto także podkreślić, że w pakiecie razem z nim dostaniemy liczną, chętną do pomocy społeczność użytkowników!

Autor książki porusza interesujące kwestie związane z budową zaawansowanych systemów informatycznych opartych o Rails. W trakcie lektury dowiesz się, jak wykorzystać system wtyczek jako środek organizujący Twój kod oraz jak w tej roli sprawdzą się moduły. Kolejne rozdziały przyniosą solidny zastrzyk wiedzy na temat tworzenia rozbudowanego i bezpiecznego modelu danych, dziedziczenia wielotabelarycznego oraz wykorzystania wyzwalaczy jako narzędzia kontroli skomplikowanych zależności w danych. Dan Chak duży nacisk kładzie na zagadnienia związane z SOA (skrót od ang. Service Oriented Architecture) oraz wydajnością. Jest to genialna pozycja dla wszystkich programistów i projektantów uczestniczących w projekcie wytwarzanym z wykorzystaniem Rails.

- Komponenty aplikacji
- Organizacja kodu z wykorzystaniem wtyczek
- Rola modułów w porządkowaniu kodu
- Budowa solidnego modelu danych
- Normalizacja modelu
- Obsługa danych dziedzinowych
- Wykorzystanie wyzwalaczy w celu kontroli zależności w danych
- Dziedziczenie jedno- i wielotabelaryczne
- Zastosowanie modeli widokowych
- Architektura SOA
- Dostarczanie usług typu XML-RPC
- Usługi typu REST
- Zwiększenie wydajności Rails

Obowiązkowa pozycja dla wszystkich programistów i projektantów korzystających z Rails!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

© Helion 1991-2010

Spis treści

Wstęp ... 9

1. Widok z góry ... 19

Co to znaczy „enterprise?” 19
Powolny wzrost 21
Komponenty aplikacji 24
Warstwa danych 24
Warstwa aplikacyjna 26
Warstwa cache’owania 29
System komunikacyjny 32
Serwer WWW 33
Zapora sieciowa 33

2. Wtyczki jako środek organizacji kodu ... 35

Korzyści 36
Tworzenie własnej wtyczki 37
Rozszerzanie klas wbudowanych 38
Rozszerzenia uniwersalne 40
Wdrażanie 45
svn:externals 45

3. Organizacja kodu za pomocą modułów ... 47

Pliki i katalogi 47
Granice modułu wyznaczają przestrzeń nazw 49
Międzymodułowe skojarzenia klas modelowych 50
Relacje wzajemne 51
Modularyzacja jako wstęp do architektury usługowej 51
Wymuszenie prawidłowej kolejności ładowania definicji klas 53
Ćwiczenia 54

Refaktoring 54
Wyodrębnianie modułów fizycznych 54
Uwalnianie metod użytkowych 55

4. Baza danych jak forteca ... 57
Baza danych jako część aplikacji 58
„Jedno środowisko wyznacza reguły” 58
„Nasi programiści nie popełniają błędów” 58
„Baza danych moja i tylko moja” 59
Siadając na ramionach gigantów 59
Wybór właściwego systemu bazy danych 59
À propos migracji 60
Obalając mity… 62
Raporty, raporty… 63

5. Budowanie solidnego modelu danych ... 67
Bilety do kina 67
Na początek bardzo prosto 68
Ograniczenia 70
Obalamy mity 78
Integralność referencyjna 78
Wprowadzenie do indeksowania 85

6. Refaktoryzacja bazy do trzeciej postaci normalnej ... 87
Trzecia postać normalna 87
Zacznij od normalizacji 91
Dziedziczenie tabel i domieszki 92
Ćwiczenia 95
Refaktoryzacja 96

7. Dane dziedzinowe ... 97
Kody pocztowe i geograficzne dane dziedzinowe 99
Wzorzec projektowy — strategia dla tabel dziedzinowych 101
Refaktoryzacja od samego początku 104

8. Klucze złożone i postać normalna DKNF ... 107
Klucze naturalne — korzyści i kłopoty 108
Wybór kluczy naturalnych 111
Siedząc już na ramionach giganta… 112
Migracja do postaci normalnej DKNF 113
Klucze wielokolumnowe i ich implementacja w Rails 116
Odroczona kontrola integralności referencyjnej 120
Coś za coś… 122

Ćwiczenia 123
Refaktoryzacja 124
Klucz jednokolumnowy 124
Klucz wielokolumnowy 125

9. Wyzwalacze jako narzędzia kontroli skomplikowanych zależności wewnątrz danych 127
Kontrola ograniczeń za pomocą wyzwalaczy 127
Anatomia funkcji w języku PL/pgSQL 130
To tylko łańcuchy… 131
Zmienne lokalne i przypisywanie im wartości 131
Bloki 132
Dodatkowe cechy wyzwalacza 132
Wyzwalacz — łagodna zapora lub bezpiecznik 132
Instrukcje warunkowe 133

10. Dziedziczenie wielotabelaryczne 135
O co chodzi? 135
Polimorfizm — co to jest? 137
Dziedziczenie a dane fizyczne 138
Dziedziczenie jednotabelaryczne 140
Dziedziczenie wielotabelaryczne 140
Alternatywa wyłączająca dla zbioru kolumn 143
Implementacja MTI w Rails 145
Klasy-fabryki 151
Ćwiczenia 152
Refaktoryzacja 152
Z STI do MTI 152
Z :polymorphic => true do MTI 153

11. Modele widokowe 155
Widoki 156
Definiowanie widoku 156
Definiowanie klasy modelowej na bazie widoku 157
Specyfika widoków 158
Dodawanie, modyfikowanie i usuwanie rekordów 159
Ograniczenia i klucze obce 159
Indeksowanie 160
Ćwiczenia 161
Refaktoryzacja 161

12. Widoki zmaterializowane ... 163
Reguły rządzące widokami zmaterializowanymi 164
Widok źródłowy 165
Formatowanie widoku 166
Tabela docelowa 168
Funkcje odświeżające i unieważniające 169
Zarządzanie zależnościami czasowymi 171
Kto za to płaci? 172
Odświeżanie i unieważnianie sterowane wyzwalaczami 175
Tabela movie_showtimes 176
Tabela movies 178
Tabela theatres 178
Tabela orders 179
Tabela purchased_tickets 180
Ukrycie implementacji dzięki widokowi uzgadniającemu 181
Periodyczne odświeżanie 183
Indeksowanie widoku zmaterializowanego 184
To się naprawdę opłaca… 185
Kaskadowe cache'owanie widoków 186
Ćwiczenia 187

13. SOA — zaczynamy ... 189
Czym jest SOA? 189
Dlaczego SOA? 192
Współdzielenie zasobów 193
Redukcja obciążenia baz danych 196
Skalowalność i cache'owanie 202
Lokalna redukcja złożoności 202
Podsumowanie 205
Ćwiczenia 205

14. Specyfika SOA ... 207
Specyfika usług 207
Ukryta implementacja 207
Przystępne API 210
Projektowanie API 211
Nie rozdrabniaj się 211
Ogranicz kontakty 213
Korzystaj ze współbieżności 214
Tylko to — i nic więcej 215

REST, XML-RPC i SOAP 217
XML-RPC 217
SOAP 219

15. Usługi typu XML-RPC ... 221
ActionWebService w Rails 2.0 221
Definiowanie bariery abstrakcji 222
ActiveRecord jako warstwa modelu fizycznego 222
Warstwa modelu logicznego 224
Definiowanie API 229
Więcej testów… 233
Wtyczka kliencka 235
Współdzielony kod 236
Kliencka klasa-singleton 237
Testy integracyjne 238

16. Przechodzimy na SOA ... 241
Usługa zamówień — OrdersService 242
Integracja z usługą MoviesService 252
Konsekwencje… 254
Model obiektowy usługi MoviesService 256
Podsumowanie 265

17. Usługi typu REST ... 267
Podstawy REST 267
Zasoby i polecenia 267
Sprzęt jest częścią aplikacji 269
REST a SOA 270
REST a CRUD 270
Uniwersalny interfejs 271
HTTP+POX 273
Definiowanie kontraktu usługi 274
Klient REST w Rails 276
REST czy XML-RPC? 276

18. Usługi webowe typu RESTful ... 279
Sformułowanie zadania 279
Narzędzia 281
ROXML 281
Net::HTTP 283

Usługa MoviesWebService 284
Implementacja zasobów serwera 284
Implementacja akcji serwera 287
Implementacja klienta 288

19. Cache’owanie 295
Dla przypomnienia — cache’owanie w warstwie fizycznej 296
Migawki 296
Funkcja odświeżająca 297
Wyzwalacze unieważniające 297
Indeksowanie 298
Cache’owanie modeli logicznych 298
Uwarunkowania 304
Pułapka nieaktualnych danych 307
Indeksowanie cache 310
Inne aspekty cache’owania 311
Cache’owanie planu realizacji 311
Cache’owanie żądań 312
Cache’owanie w Rails 313
Cache’owanie fragmentów, akcji i stron 314

Skorowidz 315

ROZDZIAŁ 8.

Klucze złożone i postać normalna DKNF

Nasz obecny model znacznie różni się od swego pierwowzoru z rysunku 5.1, doświadczył bowiem szeregu przeobrażeń, polegających na (przypomnijmy):

- rozszerzeniu definicji schematu o ograniczenia (*constraints*) narzucone na dopuszczalną postać danych,
- wymuszeniu kontroli integralności referencyjnej,
- skojarzeniu podstawowych indeksów z tabelami,
- usunięciu redundancji danych drogą dziedziczenia tabel i dziedziczenia klas modelowych oraz zamknięciu definicji tych ostatnich w formę wtyczek Rails,
- utworzeniu nowych tabel na bazie kolumn, których tematyczne rozszerzenie stwarzałoby ryzyko naruszenia reguł trzeciej postaci normalnej,
- magazynowaniu bazy wiedzy aplikacji w postaci tabel dziedzinowych i odpowiadających im klas modelowych i stałych Rails.

To bardzo wiele, jednak naszemu modelowi wciąż jeszcze trochę brakuje do tego, by uznać go za wystarczająco solidny dla aplikacji enterprise. W tym rozdziale zajmiemy się dwoma mechanizmami, dzięki którym można ów dystans wyraźnie zmniejszyć: pierwszym z nich są *klucze złożone*, drugim — *klucze naturalne* dla domen (zwane po prostu „kluczami domenowymi”), czyli sekwencje kolumn jednoznacznie identyfikujące rekordy w ramach tabeli.

W kwestii kluczy naturalnych Rails znacznie ułatwia zadanie programistom, przyjmując dla danej tabeli pojedynczą kolumnę `id` w charakterze jej klucza głównego. Z jednej strony, programiści nie muszą się więc martwić o definiowanie kluczy naturalnych zgodnych z *naturą danych* przechowywanych w tabeli, z drugiej jednak, pozbawiają się w ten sposób pewnych zalet sprawiających, że klucze takie przewyższają standardowe klucze oparte na kolumnach `id` (dla prostoty w dalszym ciągu będziemy je nazywać po prostu „kluczami `id`”). W rzeczywistości obydwa typy kluczy mają swoje słabe i mocne strony, właśnie im poświęcimy znaczącą część tego rozdziału. Pokażemy m.in., jak za pomocą wtyczek pogodzić można klucze naturalne z konwencjami Rails; następnie zademonstrujemy, jak można zjeść przysłowiowe ciastko i mieć je nadal, czyli jak pogodzić klucze definiowane przez programistów ze standardowymi kluczami opartymi na kolumnach `id`. Jak się ostatecznie okaże, wszystko to osiągnąć można za cenę umiarkowanego wysiłku programisty.

Przeanalizujmy zatem najpierw zalety, jakimi cechują się standardowe klucze `id`. Najbardziej oczywistą ich zaletą jest natychmiastowa dostępność — są zdefiniowane i czekają na to, by ich użyć; skojarzenia między tabelami, ustanawiane za pomocą metod `has_many`, `belongs_to`

oraz has_and_belongs_to_many, realizowane są właśnie za pośrednictwem kluczy id. Ma to niebagatelne znaczenie, gdy trzeba naprędce stworzyć niewyszukaną, ale jednak działającą aplikację.

Drugą kapitalną zaletą kluczy id jest fakt, że jako niewchodzące w skład „zasadniczej" treści przechowywanej w rekordzie pozostają bez związku z edytowaniem tegoż rekordu; innymi słowy, edycja rekordu nigdy nie powoduje zmiany klucza głównego. W rezultacie użytkownik otrzymuje możliwość nieskrępowanego edytowania wszystkich pól.

Naruszenie klucza głównego ma niebagatelne konsekwencje w kontekście integralności referencyjnej, wymaga bowiem zrewidowania wszystkich zależności rekordów w innych tabelach od rekordu właśnie zmodyfikowanego. Załóżmy na chwilę, że pole rating_name pełni rolę klucza głównego tabeli ratings; dla rekordu, w którym pole to równe jest PG-13, istnieją prawdopodobnie skorelowane rekordy w tabeli movies, zawierające w polu rating_id tenże łańcuch PG-13. Jeżeli w tabeli ratings zmienilibyśmy zawartość rzeczonego pola na PG13, zmuszeni bylibyśmy zrewidować również zawartość odpowiednich rekordów w tabeli movies. Używając kluczy id, jesteśmy wolni od tego problemu, bowiem wartość wpisana w pole id służy *wyłącznie* kojarzeniu rekordów i nie ma żadnego powodu, by ją w jakikolwiek sposób jawnie zmieniać — to jest trzecia zaleta wspomnianych kluczy.

Wreszcie, kluczom id bardzo łatwo zapewnić unikalność, bowiem generowanie „następnej" wartości dla nowo dodawanego rekordu odbywa się automatycznie, na bazie sekwencji definiowanej w schemacie oraz wbudowanego w klasy modelowe mechanizmu serializacji.

Dla kluczy naturalnych lista korzyści nie jest już tak oczywista, mniej oczywiste bowiem są zasady ich używania. W przeciwieństwie do pola id, którego obecność nie budzi żadnych wątpliwości, nie zawsze da się w schemacie tabeli zidentyfikować zestaw kolumn, których (łączna) zawartość z *natury* jest unikalna dla rekordów tej tabeli i może każdy z tych rekordów jednoznacznie identyfikować. Jeżeli jednak taki zestaw da się określić w sposób niebudzący wątpliwości, warto obsadzić go w roli klucza naturalnego, ze względu na wynikające z tego korzyści dotyczące zachowania integralności danych.

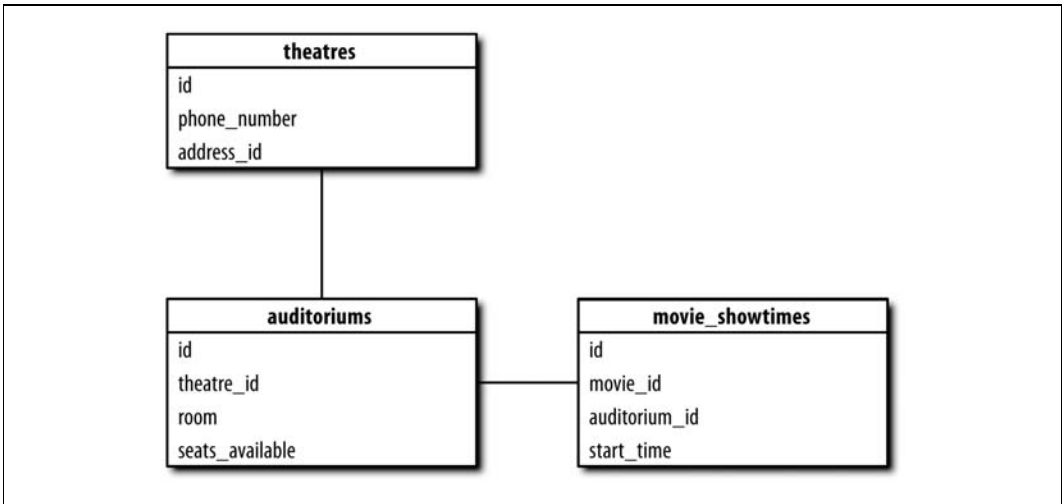
Mimo zatem pozornie większej wygody używania kluczy id, w pewnych sytuacjach stosowanie kluczy naturalnych jest wysoce uzasadnione, a niekiedy wręcz nieodzowne. Za chwilę pokażemy, jak zastąpienie klucza id kluczem naturalnym pomoże uchronić bazę danych przed powstaniem poważnej luki w integralności danych — luki niemożliwej do wykrycia na poziomie ograniczeń wpisanych w schemat bazy.

Klucze naturalne — korzyści i kłopoty

O wartości kluczy naturalnych niech przekona czytelników konkretny przykład, integralnie związany z naszym serwisem obsługującym internetową sprzedaż biletów. Na rysunku 8.1 widzimy jego fragment — tabelę movie_showtimes, skorelowaną z tabelą auditoriums, która z kolei skorelowana jest z tabelą theatres. Korelacje te oparte są na standardowych dla Rails kluczach id, zgodnie z poniższymi definicjami w schemacie:

```
movie_showtimes(auditorium_id) references auditorium(id)
auditoriums(theatre_id) references theatres(id)
```

Istniejące niegdyś bezpośrednie powiązanie tabeli movie_showtimes z tabelą theatres (poprzez klucz obcy theatre_id) zostało (jak pamiętamy z rozdziału 6.) usunięte ze względu na zachowanie zgodności z regułami trzeciej postaci normalnej i pewną anomalię spowodowaną



Rysunek 8.1. Pośrednie powiązanie tabeli projekcji z tabelą kin

brakiem tej zgodności. Robiąc krok w niewątpliwie dobrym kierunku, jednocześnie doprowadziliśmy do trochę dziwnej sytuacji. Otóż, odwołanie do kina związanego z konkretną projekcją musi teraz następować pośrednio poprzez tabelę reprezentującą sale projekcyjne; gdy chcemy uzyskać nieskomplikowaną w gruncie rzeczy informację na temat łącznej liczby seansów w kinie, w którym wyświetlany jest film reprezentowany przez bieżący rekord z tabeli `movie_showtimes`, nie możemy już napisać po prostu, jak niegdyś:

```
select count(*)
  from movie_showtimes
where theatre_id = ?
```

lecz musimy trochę się pogimnastykować:

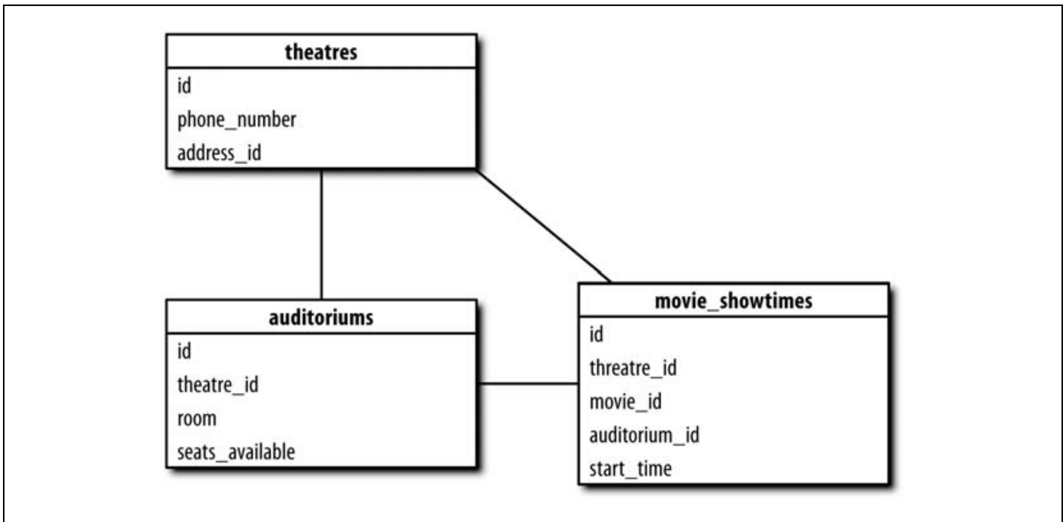
```
select count(*)
  from auditoriums a,
       movie_showtimes ms,
where ms.auditorium_id = a.id
  and a.theatre_id = ?
```

Może więc usunięcie bezpośredniego powiązania tabel `movie_showtimes` i `theatres` było decyzją zbyt pochopną? Przywróćmy je więc (jak na rysunku 8.2), dodając do definicji schematu kolejną klauzulę:

```
movie_showtimes(theatre_id) references theatres(id)
```

A teraz zobaczmy, jak tym drobnym posunięciem uczyniliśmy potężną wyrwę w spójności przechowywanych danych. Przypomnijmy mianowicie opisaną w rozdziale 6. anomalię polegającą na tym, że oba odwołania do tabeli `theatres` — bezpośrednie oraz poprzez tabelę `auditoriums` — prowadzą do dwóch różnych kin. Mimo iż jest to ewidentna anomalia, z punktu widzenia definicji zawartych w schemacie wszystko jest w porządku — próba zapisania „anormalnych" danych nie spowoduje wystąpienia wyjątku.

Ale to jeszcze nie koniec. Uważny czytelnik z pewnością zauważył, że mając możliwość *niezależnego* ustanawiania powiązań tabeli `movie_showtimes` z tabelami `auditoriums` i `theatres`, można doprowadzić do sytuacji, w której odnośna sala projekcyjna nie będzie częścią odnośnego kina! Spójrzmy na poniższe dane:



Rysunek 8.2. Przywrócone pomocnicze powiązanie tabel movie_showtimes i theatres

```
movies_development=# select id, name from theatres;
 id |       name
----+-------------------
  1 | Steller Theatre
  2 | Old Towne Theatre
(2 rows)

movies_development=# select * from auditoriums;
 id | theatre_id | room | seats_available
----+------------+------+-----------------
  1 |          1 | A    |             150
  2 |          2 | B    |             150
(2 rows)

movies_development=#
select id, movie_id, theatre_id, auditorium_id * from movie_showtimes;
 id | movie_id | theatre_id | auditorium_id
----+----------+------------+---------------
  1 |        1 |          1 |             2
(1 row)
```

Jak widać, rekord tabeli `movie_showtimes` odwołuje się do sali `B` (`auditorium_id=2`) w kinie *Steller Theatre* (`theatre_id=1`). Z punktu widzenia poprawności odwołań wszystko jest w najlepszym porządku, sęk jednak w tym, że w kinie *Steller Theatre* nie ma sali `B`! Zgodnie z zawartością tabeli sala `B` jest częścią *Old Towne Theatre*:

```
>> t = Theatre.find_by_name('Steller Theatre')
>> puts t.movie_showtimes.first.auditorium.theatre.name
=> Old Towne Theatre
```

Ponownie, tej nieadekwatnej do stanu faktycznego sytuacji nie jest w stanie zapobiec kontrola integralności danych na poziomie schematu bazy. Zaprezentowana anomalia nie wynika bowiem z jakiegoś błędu w systemie korelacji tabel, lecz z niefortunnego wyboru klucza naturalnego. Innymi słowy, mimo iż formalnie zachowana została integralność referencyjna „surowych" danych, o integralności dziedziny problemowej nie ma mowy. Wszystko dlatego, że klucz `id` tabeli `auditorium` nie zawiera informacji wystarczających dla zachowania adekwatności ze stanem rzeczywistym — niezbędne zatem staje się użycie klucza naturalnego.

Wybór kluczy naturalnych

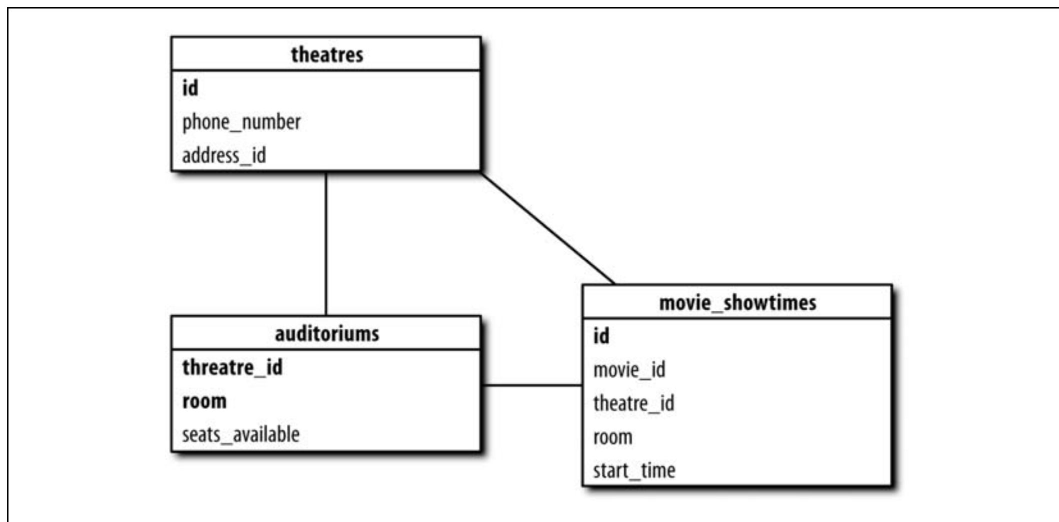
Klucz złożony to — mówiąc najprościej — klucz utworzony z kilku kolumn. Określenie, które kolumny kwalifikują się jako kluczowe dla tabeli, już takie proste nie jest.

Gdy zastanowimy się nad warunkami, jaki musi spełniać klucz główny, natychmiast oczywisty staje się jeden: klucz ten musi jednoznacznie identyfikować rekordy tabeli, czyli musi być inny dla każdego rekordu. Zasada ta działa również w drugą stronę: jeżeli w modelu danych istnieje zestaw kolumn, na który (zgodnie z dziedziną problemową) narzucono ograniczenie unikalności, zestaw taki kwalifikuje się do roli klucza głównego. Jeśli ponadto unikalność ta wynika wprost z natury samych danych, klucz ten nazywamy *kluczem naturalnym*.

Gdy przyjrzymy się schematowi tabeli `auditoriums`, szybko spostrzeżemy, że taki unikalny zestaw tworzą dwie kolumny: `theatre_id` i `room`. Istotnie, nie ma większego sensu istnieje kilku identycznie nazwanych sal projekcyjnych w tym samym kinie. Ów zestaw stanowi jednocześnie znakomitą podstawę odwoływania się do danej sali projekcyjnej w innych tabelach: odwołanie „sala o nazwie `A` w kinie identyfikowanym przez `id`=1" brzmi nieco bardziej komunikatywnie niż „sala identyfikowana przez `id`=47" . To pierwsze, jako zawierające bardziej naturalne określenie sali projekcyjnej, lepiej nadaje się do kontrolowania integralności danych od tego drugiego, identyfikującego salę projekcyjną wyłącznie w sposób wewnętrzny, wynikający z sekwencji związanej z tabelą.

Na rysunku 8.3 widzimy zatem kolejne przeobrażenie naszego schematu — z tabeli `auditoriums` usunięta została kolumna `id`, jako już niepotrzebna; spełnianą przez nią dotąd rolę klucza głównego przejęła para kolumn (`theatre_id`, `room`). W konsekwencji z tabeli `movie_showtimes` zniknąć musi kolumna `auditorium_id` — pełnioną dotąd przez nią rolę klucza obcego przejmuje teraz para (`theatre_id`, `room`). Oczywiście, zmiana ta musi znaleźć swe odzwierciedlenie w definicji schematu — zależność między tabelami `theatres`, `auditoriums` i `movie_showtimes` przedstawia się teraz następująco:

```
movie_showtimes(theatre_id) references theatres(id)
movie_showtimes(theatre_id, room) references auditoriums(theatre_id, room)
auditoriums(theatre_id) references theatres(theatre_id)
```



Rysunek 8.3. Klucz id tabeli auditoriums zastąpiony przez klucz naturalny

Uaktualniona definicja schematu tabel `auditoriums` i `movie_showtimes` widoczna jest na listingu 8.1.

Listing 8.1. Efekt zastąpienia klucza id kluczem naturalnym w tabeli auditoriums

```
create table auditoriums (
  room varchar(64) not null
    check (length(room) >= 1),
  theatre_id integer not null
    references theatres(id),
  seats_available integer not null,
  primary key (room, theatre_id)
);

create sequence movie_showtimes_id_seq;
create table movie_showtimes (
  id integer not null
    default nextval('movie_showtimes_id_seq')
  movie_id integer not null
    references movies(id),
  theatre_id integer not null
    references theatres(id),
  room varchar(64) not null,
  primary key (id),
  foreign key (theatre_id, room)
    references auditoriums(theatre_id, room) initially deferred
);
```

W tych warunkach wystąpienie opisanej wcześniej anomalii jest niemożliwe, bo po pierwsze, zawartość pól `theatre_id` musi być identyczna w rekordach obu tabel — `movie_showtimes` i `auditoriums`, wykluczone jest więc wskazanie dwóch różnych kin; po drugie, jako że kolumny `theatre_id` i `room` występują teraz łącznie jako identyfikacja sali projekcyjnej, nie jest możliwe odwołanie się do sali nieistniejącej danym kinie.

Siedząc już na ramionach giganta...

W rozdziale 4. przedstawialiśmy już technologie bazodanowe jako swoistego giganta, stanowiącego solidny fundament dla tworzonego kodu aplikacji — aplikacji sadowiącej się na ramionach owego giganta. „Giganta", bo dzisiejszy stan wiedzy w tej dziedzinie stanowi dziedzictwo kilkudziesięciu lat dociekań teoretycznych i badań eksperymentalnych. Problematyka normalizacji danych oraz odpowiedniego wyboru kluczy naturalnych przewija się przez szereg publikacji od ponad 25 lat, jest więc problematyką doskonale rozpoznaną i jako taka stanowi doskonały punkt odniesienia dla poczynań programistycznych.

W roku 1981 Ronald Fagin z IBM Research Laboratories sformułował ideę *postaci normalnej klucza domenowego* (DKNF — *Domain Key Normal Form*); w swej publikacji zatytułowanej *A Normal Form for Relational Databases That Is Based on Domains and Keys* udowodnił w sposób formalny, że można całkowicie wyeliminować rozmaite anomalie w danych (np. takie, jak opisana wcześniej w tym rozdziale), obsadzając w charakterze klucza tabeli taki najmniejszy zestaw kolumn, którego unikalność dla poszczególnych rekordów zagwarantowana jest przez naturę danych. Ów „zestaw" może jednak mieć niekiedy postać pojedynczej kolumny (i często faktycznie ma), może też zawierać kilka kolumn, jedno wszak jest pewne: nie istnieje uniwersalny przepis na wybór „dobrego" klucza naturalnego. Wybór ten musi być wynikiem dogłębnej analizy danych przechowywanych w tabeli, a także analizy sposobu, w jaki tabela ta wpisuje się w ogólny schemat bazy.

Najlepsze spośród dostępnych dziś systemy zarządzania bazami danych stworzone zostały na bazie wieloletniego dorobku badawczego; nawet koncepcje wydające się dziś nowatorskimi — jak wybór wielokolumnowego klucza naturalnego — datują się na wiele lat wstecz, czego przykładem cytowana publikacja Fagina. I choć nie wydaje się to niczym niezwykłym, nieco zaskakujący jawi się zupełny brak wsparcia ze strony Rails dla wielu kluczowych koncepcji w tej dziedzinie.

W konsekwencji wielu programistów, dla których Rails jest podstawowym (lub jedynym) środowiskiem pracy, skłonnych jest uważać te koncepcje za niezbyt istotne, bo skoro brak ich obsługi w tak popularnym środowisku, to widocznie nie są specjalnie potrzebne. Co więcej, jeśli Rails stanowi dla nich pierwszą okazję kontaktu z bazami danych w ogóle, być może nie wyobrażają sobie innych kluczy głównych niż klucze `id` — szczególnie kluczy domenowych. Mamy nadzieję, że przeczytanie tego rozdziału pomoże uchronić czytelników przed tą niewiedzą.

Migracja do postaci normalnej DKNF

Normalizowanie schematu do postaci DKNF może być uciążliwym zadaniem. Z naszym schematem będzie trochę łatwiej, bo sprowadziliśmy go już do trzeciej postaci normalnej. Przeanalizujmy zatem naturę danych w poszczególnych tabelach i charakter powiązań między tymi tabelami.

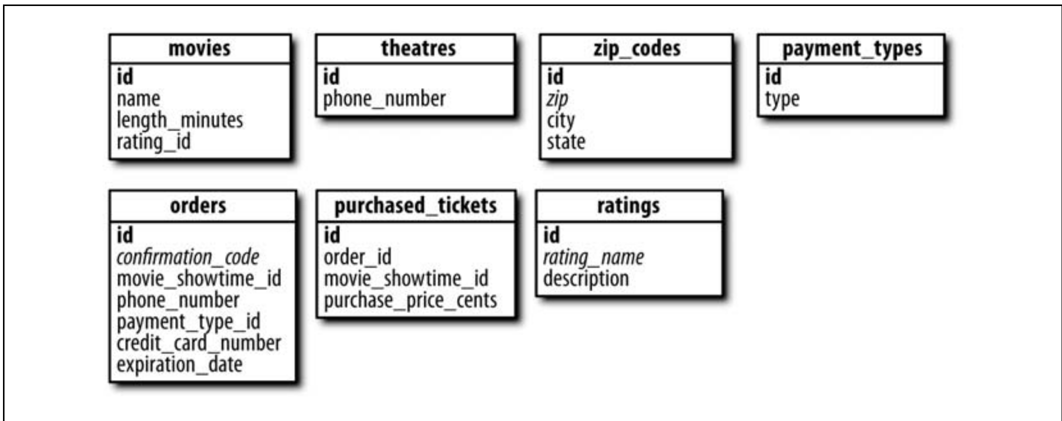
W tabeli `auditoriums` zastąpiliśmy już klucz `id` kluczem naturalnym. Kolejny krok to zdecydowanie, które z tabel kwalifikują się do posiadania *jednokolumnowych* kluczy naturalnych; będą wśród nich takie, w których uzasadnione będzie pozostawienie standardowego klucza `id`, oraz takie, w których klucz ten tworzyć będzie kolumna zawierająca „rzeczywiste" dane. Później zajmiemy się kluczami złożonymi (wielokolumnowymi) i ich implementacją w Rails za pomocą dedykowanej wtyczki. Na przykładzie tabeli `movie_showtimes` pokażemy także, jakie nowe problemy mogą pojawić się w związku z używaniem kluczy naturalnych i jak można je złagodzić, tworząc swoiste rozwiązanie hybrydowe, sprowadzające się do współegzystencji tych kluczy ze standardowymi dla Rails kluczami `id`.

Klucze jednokolumnowe

Jednokolumnowe klucze główne zdefiniowane zostały dla tabel `movies`, `payment_types`, `orders`, `purchased_tickets`, `zip_codes` i `ratings`. Klucze jednokolumnowe posiada więc *większość* tabel i jest to sytuacja typowa dla większości schematów — być może okoliczność ta tłumaczy fakt, że w Rails jedynie takim kluczom zapewniono standardową obsługę.

Żeby zdecydować, czy dla danej tabeli wystarczający jest jednokolumnowy klucz będący w istocie arbitralnie wybieraną liczbą całkowitą, należy spróbować znaleźć w schemacie tej tabeli kolumnę, której charakter decyduje o jej unikalności, zaś unikalność z kolei kwalifikuje kolumnę do roli klucza naturalnego. W tabelach widocznych na rysunku 8.4 wyróżniono takie kolumny kursywą (pogrubioną czcionką oznaczone są kolumny `id` jako bazowe dla istniejących kluczy naturalnych).

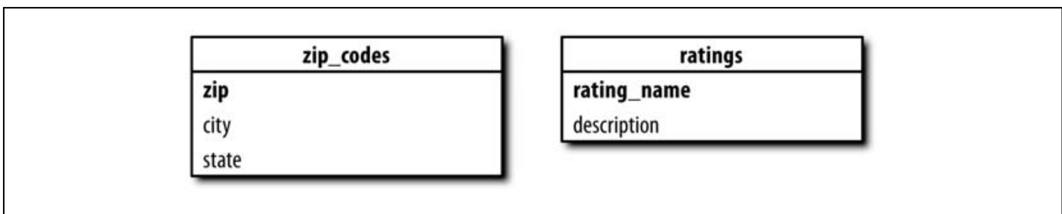
Ostatecznie więc okazuje się, że tabele `zip_codes`, `ratings` i `orders` posiadają takie „unikalne" kolumny, odpowiednio, `zip`, `rating_name` i `confirmation_code`. Oznacza to, że rekordy tych tabel mogą być jednoznacznie identyfikowane *na dwa różne sposoby*, choć — prawdę mówiąc — nie jest to fakt zbyt istotny; ważne jest natomiast to, iż każda z owych unikalnych



Rysunek 8.4. Tabele posiadające standardowe klucze id

kolumn (posiadająca intuicyjną nazwę) może po prostu zastąpić odnośną kolumnę `id`. Z perspektywy Rails typ danych przechowywanych w kolumnie klucza naturalnego jest w zasadzie obojętny, jeżeli jednak nie jest to arbitralnie generowana liczba całkowita, sami musimy zadbać o generowanie unikalnych wartości dla nowo tworzonych rekordów. Jeżeli ponadto nazwa kolumny tworzącej klucz naturalny jest inna niż `id`, musimy nazwę tę jawnie wskazać w klasie modelowej jako parametr wywołania metody `set_primary_key`.

Jak już wspominaliśmy w rozdziale 7., w tabeli `zip_codes` taką kolumną jest kolumna `zip`. W tabeli `ratings` funkcję klucza naturalnego mogłaby pełnić kolumna `rating_name`. Obie te tabele są o tyle łatwiejsze w obsłudze — pod względem niestandardowych kluczy głównych — że są tabelami dziedzinowymi: ich zawartość prawdopodobnie nie będzie się zmieniać, a jeżeli nawet, to na pewno bardzo rzadko. Dotyczy to szczególnie tabeli `ratings`, dla której w klasie modelowej zdefiniowano zestaw stałych reprezentujących poszczególne wartości potencjalnego klucza. Dla obu tabel nie definiowaliśmy żadnego interfejsu umożliwiającego operowanie danymi, więc wszelkie modyfikacje i dodawanie nowych rekordów odbywać się będą *poza warstwą aplikacyjną*, ergo — nie istnieje problem generowania ad hoc unikalnych wartości dla klucza w nowych rekordach. Ostatecznie eliminujemy z tabeli `ratings` pole `id`, obsadzając w roli klucza głównego kolumnę `rating_name`, tak jak na rysunku 8.5.



Rysunek 8.5. Tabele dziedzinowe z niestandardowymi kluczami głównymi

W tabeli `orders` zawartość pola `confirmation_code`, będącą w istocie kodem autoryzacyjnym transakcji, również można przyjąć jako niezmienną. Jeśli obsadzimy ją w roli klucza głównego, zamiast kolumny `id`, przejmiemy na siebie obowiązek generowania jej unikalnej zawartości — *która i tak jest generowana dla każdej transakcji*, problem więc rozwiązuje się sam. Najbardziej odpowiednim miejscem dla dokonywania tego generowania jest metoda `before_create` klasy modelowej `Order`; nowa wartość będzie po prostu odwzorowaniem mieszającym (*hash*)

kolejnej wartości sekwencyjnej, jaka zostałaby przypisana polu `id` w nowym rekordzie[1]. Przy okazji zwracamy uwagę na kolejny, niezwykle istotny fakt: mimo że kolumna tworząca klucz główny nie nosi już nazwy `id`, *na poziomie klasy modelowej nadal jest ona reprezentowana przez właściwość* `id`, stąd wyrażenie `self.id` w poniższym fragmencie, stanowiące w istocie odwołanie do kolumny `confirmation_code`, nie jest pomyłką.

```
class Order < ActiveRecord::Base
  set_primary_key :confirmation_code
  has_many :purchased_tickets, :foreign_key => 'order_confirmation_code'

  def before_create
    next_ordinal_id = Order.connection.select_value(
      "select nextval('orders_id_seq')"
    )
    self.id = next_ordinal_id.crypt("CONF_CODE")
  end
end
```

Łatwo się przekonać, że dodawanie nowych rekordów do tabeli `orders` odbywa się całkowicie poprawnie — w polu `confirmation_code` zapisywany jest mało czytelny, lecz unikalny kod:

```
>> o = Order.create({:movie_showtime_id => 1,
                     :purchaser_name => 'Jaś Fasola' })
=> #<Order:0x2553af0>
>> o.id
=> "CotW6pp1X6z7o"
```

Tworzenie rekordów zależnych także nie stanowi problemu:

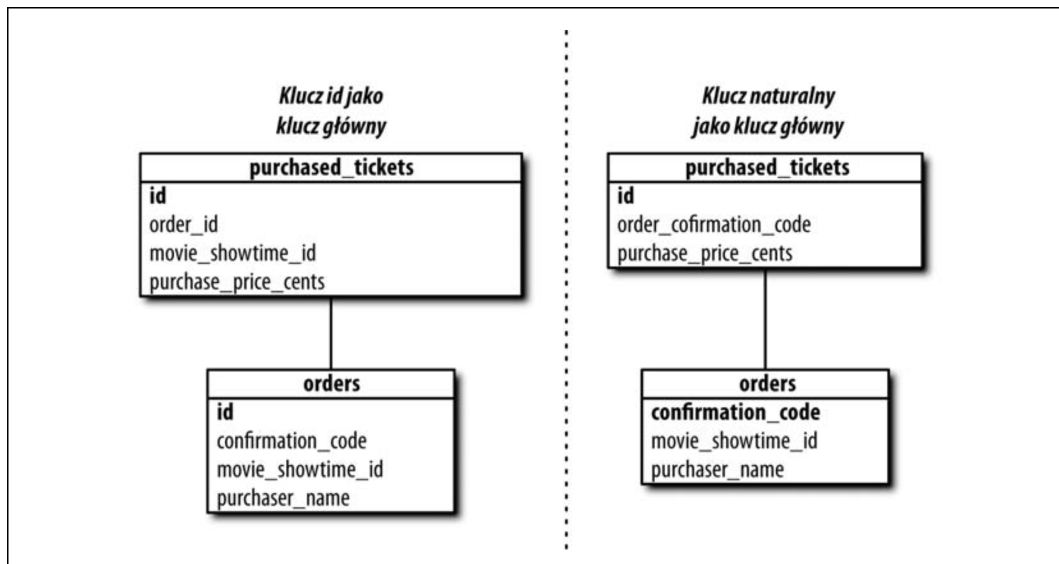
```
>> o.purchased_tickets << PurchasedTicket.new(:purchase_price_cents => 650)
=> #<PurchasedTicket:0x25166c8>
o.confirmation_code
=> "CotW6pp1X6z7o"
```

Zdefiniowanie klucza głównego jako klucza naturalnego, zamiast standardowego klucza `id`, daje wiele dodatkowych korzyści, z których jedną wyjaśnimy na przykładzie konkretnego powiązania. Na rysunku 8.6 widzimy tabele `orders` i `purchased-tickets`, powiązane na dwa różne sposoby: za pomocą klucza `id` tabeli `orders` (w lewej części) oraz przy użyciu klucza naturalnego tej tabeli (z prawej). W tym drugim przypadku klucz główny tej tabeli jest nie tylko unikalną, beznamiętną wartością zapewniającą jednoznaczne identyfikowanie rekordów, lecz również niesie ze sobą konkretną informację, jaką jest kod autoryzacyjny transakcji. Ponieważ klucz główny tabeli `orders` ma swój odpowiednik w postaci klucza obcego, jakim jest pole `order_confirmation_code` tabeli `purchased_tickets`, więc owa konkretna informacja „przemycona" została mimowolnie do tejże tabeli. W efekcie dla konkretnego rekordu reprezentującego sprzedany bilet informacja o kodzie autoryzacyjnym transakcji sprzedaży obecna jest wprost w tymże rekordzie; w układzie z lewej strony rysunku informacja ta dostępna jest tylko pośrednio, poprzez klucz obcy `order_id` odsyłający do odpowiedniego rekordu w tabeli `orders`.

Dla tabel `theatres` i `movies` nie istnieją żadne przesłanki do definiowania kluczy naturalnych, dla nich pozostawiamy zatem standardowe klucze główne `id`.

---

[1] Zwracamy uwagę, że klauzula `nextval`, wyznaczająca kolejną wartość wynikającą ze zdefiniowanej sekwencji, jest klauzulą specyficzną dla PostgreSQL. Na poziomie klasy modelowej generowanie kolejnych wartości sekwencyjnych wykonywane jest przez metodę `next_sequence_value`, działającą niezależnie od konkretnego systemu bazy danych; niestety, poprawne działanie tej metody w kontekście PostgreSQL wymaga zainstalowania poprawki do Rails, dostępnej na stronie *http://dev.rubyonrails.org/ticket/9178*. W kontekście Oracle funkcja ta natomiast spisuje się bezproblemowo.



Rysunek 8.6. Dwa różne powiązania tabel orders i purchased-tickets, na podstawie klucza id oraz na podstawie klucza naturalnego

Klucze wielokolumnowe i ich implementacja w Rails

Mimo iż Rails nie zapewnia standardowo obsługi głównych kluczy wielokolumnowych, istnieją dwa sposoby osiągnięcia korzyści wynikających z używania takich kluczy. Pierwszy z nich sprowadza się do wykorzystania pewnej dedykowanej wtyczki, drugi zasadza się na współistnieniu głównych kluczy wielokolumnowych ze standardowymi kluczami `id`.

Obsługa kluczy złożonych za pomocą dedykowanej wtyczki

Tytułowa wtyczka dostępna jest na stronie *http://compositekeys.rubyforge.org*, wraz z niezbędną dokumentacją. Zainstalowanie wtyczki jako gemu odbywa się w zwykły sposób:

```
ruby gem install composite_primary_keys
```

Należy jeszcze dołączyć na końcu pliku *config/environment.rb* instrukcję:

```
require 'composite_primary_keys'
```

Na gruncie klasy modelowej metoda definiująca złożony klucz główny nosi nazwę `set_primary_keys` — co jest liczbą mnogą `set_primary_key`:

```
class Auditorium < ActiveRecord::Base
  # musimy jawnie zdefiniować nazwę tabeli, bo reguły infleksji Rails
  # zawodzą w tym przypadku
  set table_name 'auditoriums'
  set_primary_keys :room :theatre_id

  belongs_to :theatre
  has_many :movie_showtimes, :dependent => :destroy
end
```

W powiązanej klasie modelowej wielokolumnowy klucz obcy reprezentowany jest w postaci tablicy kolumn, przekazywanej jako wartość parametru `:foreign_key`:

```
class MovieShowtime < ActiveRecord::Base
  belongs_to :movie
  belongs_to :theatre
  belongs_to :auditorium, :foreign_key => [:room, :theatre_id]
end
```

Klasy modelowe korzystające z wielokolumnowych kluczy głównych nie wymagają żadnego specjalnego traktowania. Tak jak w poniższym przykładzie, tworzenie obiektu klasy `Auditorium` odbywa się w zwykły sposób, podobnie pisząc obiekt klasy `MovieShowtime`, nie musimy jawnie specyfikować poszczególnych elementów klucza obcego, wystarczy powołanie się na powiązany obiekt, resztę załatwią mechanizmy zainstalowanej wtyczki.

```
m = Movie.create!(
  :name => 'Casablanca',
  :length_minutes => 120,
  :rating => Rating::PG13)

t = Theatre.create!(
  :name => 'Kendall Cinema'
  :phone_number => '5555555555')

a = Auditorium.create!(
  :theatre => t,
  :room => '1'
  :seats_available => 100)

ms = MovieShowtime.create!(
  :movie -> m,
  :theatre => t,
  :auditorium => a,
  :start_time => Time.new)
```

Model hybrydowy „id-DKNF"

Zajmijmy się teraz tabelą `movie_showtimes`. Po przeanalizowaniu przeznaczenia tabeli (każdy rekord reprezentuje jedną projekcję filmu) i znaczenia poszczególnych kolumn dochodzimy do wniosku, że kolumny (`movie_id`, `theatre_id`, `room` i `start_time`) tworzą minimalny[2] zestaw unikalności, który tym samym kwalifikuje się do roli klucza głównego. Teoretycznie, unikalność tego zestawu nie wyczerpuje możliwości ograniczeń narzuconych na zawartość tabeli, nie wynika z niej bowiem oczywisty fakt, że w konkretnej sali projekcyjnej wyświetlanie kolejnego filmu może zacząć się dopiero po zakończeniu emisji poprzedniego; tego rodzaju ograniczeniami zajmiemy się dopiero w następnym rozdziale.

Unikalność pewnego zestawu kolumn stanowi niewątpliwie warunek konieczny, by można było obsadzić ów zestaw w roli klucza głównego, nie zawsze jednak jest to warunek wystarczający; jak za chwilę zobaczymy, korzystanie z kluczy naturalnych może być przyczyną poważnych problemów, niwelujących ewentualne korzyści i sprawiających, że przysłowiowa skórka staje się niewarta wyprawki.

Jeżeli przyjmiemy wspomniany zestaw kolumn jako klucz główny tabeli `movie_showtimes`, w powiązanej z nią tabeli `orders` w polu `start_time`, wchodzącym w skład klucza obcego, pojawi się informacja o czasie rozpoczęcia projekcji. Zdarza się, że (z różnych przyczyn) czas ten ulega zmianie; klienci, którzy zakupili już bilety na konkretną godzinę, z reguły akceptują taką zmianę, a ci, którym ona nie odpowiada, mogą bilet zwrócić.

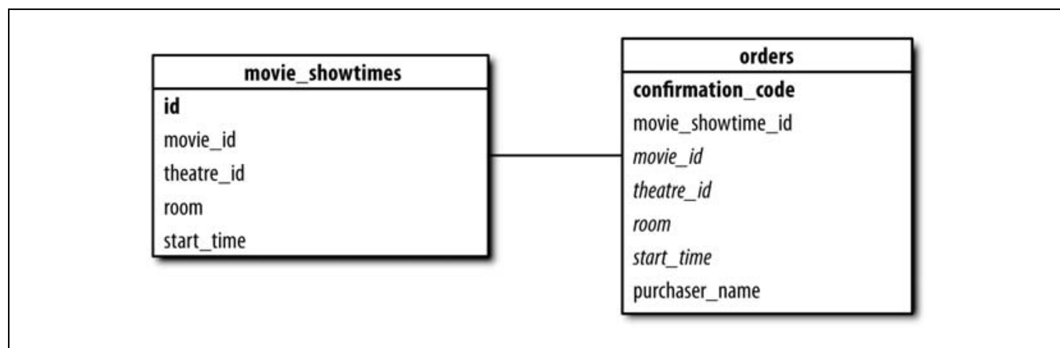
[2] „Minimalny", bo usunięcie którejkolwiek kolumny z zestawu pozbawi go cechy unikalności — *przyp. tłum.*

Z perspektywy integralności referencyjnej danych zgromadzonych w bazie sprawa jednak nie wygląda tak prosto, bowiem nie należy modyfikować klucza rekordu, dla którego w innych tabelach istnieją rekordy zależne. Dla rekordu z tabeli `movie_showtimes` mogą istnieć rekordy w tabeli `orders`, te zaś mogą odwoływać się do swych rekordów zależnych w tabeli `purchased_tickets`. Niezbyt spektakularne wydarzenie, jakim jest zmiana rozpoczęcia emisji filmu, w przełożeniu na konkretne operacje bazodanowe musiałoby obejmować kolejno:

1. Usunięcie zależnych rekordów z tabeli `purchased_tickets`.
2. Usunięcie zależnych rekordów z tabeli `orders`.
3. Usunięcie rekordu z tabeli `movie_showtimes`.
4. Utworzenie nowego rekordu w tabeli `movie_showtimes`, z nową wartością w polu `start_time`.
5. Odtworzenie rekordów usuniętych w punkcie 2., z nową wartością w polu `start_time`.
6. Odtworzenie rekordów usuniętych w punkcie 1.

To jeszcze nie wszystko. Otóż, `ActiveRecord` nie daje standardowo żadnej możliwości modyfikowania klucza głównego rekordu, modyfikację taką można jednak przeprowadzić za pomocą bezpośredniego odwołania do języka SQL. Istnieją jednak programiści „wysokopoziomowi", których sam skrót „SQL" przyprawia o palpitację serca, więc z myślą o nich zaproponujemy teraz rozwiązanie kompromisowe.

Nie byłoby całego zamieszania, gdyby kluczem głównym tabeli `movie_showtimes` był standardowy klucz `id`. Pozostawimy go zatem w roli klucza głównego, jednocześnie zrzucając na barki warstwy aplikacyjnej (czyli klas modelowych) zadanie utrzymywania spójności między tabelami `movie_showtimes` i `orders` na poziomie klucza naturalnego tej ostatniej, czyli (mówiąc po prostu) nadawania odpowiednich wartości polom `movie_id`, `theatre_id`, `room` i `start_time` nowo tworzonych rekordów tabeli `orders`; sytuację tę przedstawiono na rysunku 8.7. Z punktu widzenia integralności referencyjnej, zmiana wartości pola `start_time` w rekordzie tabeli `movie_showtimes` nie stanowi w tym stanie rzeczy ingerencji w klucz główny i może być wykonana na poziomie Rails w zwykły sposób; oczywiście, na poziomie klasy modelowej należy jednak zadbać o stosowną modyfikację pola `start_time` w powiązanym rekordzie tabeli `orders`. Tak oto udało nam się pogodzić dwie (pozorne, jak widać) sprzeczności: zapewnienie korzyści wynikających z używania kluczy naturalnych oraz zachowanie możliwości nieskrępowanego modyfikowania tych pól rekordu, które zawierają „rzeczywistą" informację.



Rysunek 8.7. Współistnienie dwóch powiązań między tabelami movie_showtimes i orders, poprzez klucz id oraz klucz naturalny

Zwracamy uwagę na jeszcze jeden istotny fakt: przy definiowaniu skojarzeń między tabelami na poziomie schematu bazy danych, w instrukcji `foreign key` zestaw kolumn kluczowych tabeli docelowej (w parametrze `references`) musi być w definicji tej tabeli objęty klauzulą unikalności. Brak tej klauzuli może stwarzać ryzyko niejednoznacznego wiązania — wspomniany zestaw identyfikować może nie jeden, lecz kilka rekordów. Ten ewidentny błąd logiczny jest jednak tolerowany przez MySQL, próba jego popełnienia w PostgreSQL powoduje natomiast sygnalizację błędu:

```
movies_development=# alter table orders
  add constraint movie_showtimes_movie_theatre_room_start_time_fkey
  foreign key (movie_id, theatre_id, room, start_time)
  references movie_showtimes(movie_id, theatre_id, room, start_time)
ERROR:  there is no unique constraint matching given keys for
referenced table "movie_showtimes"
```

Ponadto zdefiniowanie pewnego zestawu kolumn jako unikalnego spowoduje, że PostgreSQL automatycznie założy indeks na bazie tego zestawu, dzięki czemu poszukiwanie rekordu o danym kluczu naturalnym odbywać się będzie niemal błyskawicznie.

Uzupełnijmy zatem definicję tabel `movie_showtimes` i `orders` o niezbędne elementy — wspomnianą klauzulę unikalności i definicję klucza obcego:

```
create sequence movie_showtimes_id_seq;
create table movies_showtimes (
  id integer not null
    default nextval('movie_showtimes_id_seq'),
  movie_id integer not null
    references movies(id),
  theatre_id integer not null
    references theatres(id),
  room varchar(64) not null,
  start_time timestamp with time zone not null,
  primary key (id),
  unique (movie_id, theatre_id, room, start_time),
  foreign key (theatre_id, room)
    references auditoriums(theatre_id, room) intially deferred
);

create sequence orders_id_seq;
create table orders (
  confirmation_code varchar(16) not null
    check(length(confirmation_code) > 0),
  movie_showtime_id integer not null
    references movie_showtimes(id),
  movie_id integer not null,
  room varchar(64) not null,
  start_time timestamp with time zone,
  purchaser_name varchar(128) not null
    check (length(purchaser_name) > 0),
  primary key (confirmation_code),
  foreign key (movie_id, theatre_id, room, start_time)
    references movie_showtimes(movie_id, theatre_id, room, start_time)
) inherits (addresses);
```

Uproszczenie dzięki nowej metodzie create!

Jedną z uciążliwości opisanego modelu hybrydowego jest konieczność jawnego przypisywania wartości kolumnom wchodzącym w skład klucza naturalnego w nowo tworzonych rekordach zależnych. Pozornie poprawny kod:

```
o = Order.create!(
  :movie_showtime => ms,
  :purchaser_name => 'Jaś Fasola')
```

nie będzie funkcjonować jak należy, bo przypisanie:

```
:movie_showtime => ms
```

spowoduje zainicjowanie *jedynie* pola `movie_showtime_id`, ponieważ *nie* jest aktywna wtyczka `composite_primary_keys` i Rails honoruje wyłącznie klucze `id` w roli kluczy głównych. Pozostałe kolumny wchodzące w skład klucza naturalnego należy zainicjować explicite:

```
o = Order.create!(
    :movie_showtime => ms,
    :movie => ms.movie,
    :auditorium => ms.auditorium,
    :start_time => ms.start_time,
    :purchaser_name => 'Jaś Fasola')
```

Zauważmy, że nie jest konieczne przypisywanie wartości polu `theatre_id`, zostanie ono bowiem zainicjowane w ramach przypisania `:auditorium => ms.auditorium`, ze względu na postać klucza głównego tabeli `auditoriums`.

Na szczęście, można sobie nieco ułatwić programistyczny żywot, za pomocą drobnego zabiegu sprawiającego, że Rails wykona automatycznie rzeczone przypisania w ramach instrukcji `:movie_showtime => ms`, analogicznie jak w przypadku używania wtyczki `composite_primary_keys`. Jak się czytelnicy zapewne domyślają, należy w tym celu przedefiniować odpowiednio metodę `movie_showtime=`. Ponieważ jednak jej nowa wersja odwoływać się będzie do wersji dotychczasowej, należy tę ostatnią najpierw przemianować, definiując jej *alias*:

```
class Order < ActiveRecord::Base
  alias :old_movie_showtime= :movie_showtime=
  def movie_showtime=(ms)
    self.movie_id = ms.movie_id
    self.theatre_id = ms.theatre_id
    self.room = ms.room
    self.start_time = ms.start_time
    self.old_movie_showtime=(ms)
  end
end
```

Odroczona kontrola integralności referencyjnej

Ingerowanie w zawartość kolumny wchodzącej w skład klucza głównego może powodować zerwanie relacji między powiązanymi rekordami, naruszenie integralności referencyjnej i w konsekwencji zgłoszenie wyjątku przez system bazy danych. Może się tak stać np. wskutek zmiany godziny rozpoczęcia emisji filmu, na którą to emisję sprzedane zostały już bilety:

```
def setup
  @m = Movie.create!(
         :name => 'Casablanca',
         :length_minutes => 120,
         :rating => Rating::PG13)
  @t = Theatre.create!(
         :name => 'Kendall Cinema',
         :phone_number => '5555555555')

  @a = Auditorium.create!(
         :theatre => @t,
         :auditorium => @a,
         :seats_available => 100)

  @ms = MovieShowtime.create!(
         :movie => @m,
         :theatre => @t,
         :auditorium = @a,
         :start_time => Time.new)
```

```
  @o = Order.create!(
         :movie_showtime => @ms,
         :movie => @m
         :theatre => @t,
         :auditorium = @a,
         :start_time => @ms.start_time,
         :purchaser_name => 'Jaś Fasola')
end

def test_deferrable_constraints
  MovieShowtime.transaction do
    @ms.start_time = @ms.start_time + 1.hour
    @ms.save!
    Order.update_all(["start_time = ?", @ms.start_time],
                     ["movie_showtime_id = ?", @ms.id ])
  end
end
```

Oczywiście, powyższy test załamie się, ponieważ wyróżniona instrukcja spowoduje naruszenie integralności referencyjnej:

```
ChakBookPro:chapter-7-dknf chak$ ruby test/unit/movie_showtime_test_case.rb
Loaded suite test/unit/movie_showtime_test_case
Started
  ...
Finished in 0.657148 second.

     1) Error:
test_deferrable_constraints(MovieShowtimeTestCase):
ActiveRecord::StatementInvalid: PGError: ERROR: update or
delete on table "movie_showtimes" violates foreign key constraint
"orders_ovie_id_fkey" on table "orders"
DETAIL:  Key (movie_id,theatre_id,room,start_time)=
(20,20,1,2007-12-16 00:53:49.076398) is still referenced from table "orders".
: UPDATE movie_showtimes SET "start_time" = '2007-12-16 01:53:49.076398',
"theatre_id" = 20, "movie_id" = 20, "room" = '1' WHERE "id" = 20

1 tests, 0 assertions, 0 failures, 1 errors
```

Wynika stąd, że aby naruszenie klucza naturalnego było w ogóle możliwe, baza danych musi stać się nieco bardziej wyrozumiała pod względem kontroli integralności referencyjnej i pozwolić na odstępstwo od zasad tejże integralności choć na chwilę. Istotnie, możliwe jest uzyskanie takiej „wyrozumiałości" — „na chwilę" oznacza w tym przypadku „do momentu zatwierdzenia transakcji", wewnątrz transakcji kontrola integralności referencyjnej, wynikającej z określonego klucza, jest zawieszona. W celu uzyskania tego stanu rzeczy należy w instrukcji `foreign key` (w schemacie tabeli) umieścić klauzulę `initially deferred`:

```
create table orders (
  confirmation_code varchar(16) not null
    check(length(confirmation_code) > 0),
  movie_showtime_id integer not null
    references movie_showtimes(id),
  movie_id integer not null,
  room varchar(64) not null,
  start_time timestamp with time zone,
  purchaser_name varchar(128) not null
    check (length(purchaser_name) > 0),
  primary key (confirmation_code),
  foreign key (movie_id, theatre_id, room, start_time)
    references movie_showtimes(movie_id, theatre_id, room, start_time)
    initially deferred
) inherits (addresses);
```

Po tej drobnej, acz istotnej modyfikacji test zostanie zaliczony:

```
ChakBookPro:chapter-7-dknf chak$ ruby test/unit/movie_showtime_test_case.rb
Loaded suite test/unit/movie_showtime_test_case
Started
  ...
Finished in 0.657148 second.

1 tests, 0 assertions, 0 failures, 0 errors
```

Odroczenie kontroli integralności referencyjnej umożliwia więc chwilowe naruszenie reguł, konieczne do wykonania pewnych operacji; naruszenie to odbywa się w ramach *trwającej* transakcji, nie może zatem powodować trwałych skutków w istniejących danych. Przed zatwierdzeniem transakcji konieczne jest przywrócenie absolutnej zgodności ze wspomnianymi regułami.

Pewna trudność w testowaniu opisanego odroczenia wiąże się z faktem, że (ewentualne) związane z nim błędy ujawniają się dopiero w momencie zatwierdzania (*commit*) transakcji. I tu mamy problem, bowiem każdy przypadek testowy weryfikowany jest w ramach transakcji, która ostatecznie zostaje anulowana (*rollback*) — wszystko po to, by przypadek testowy nie powodował zmian w danych będących przedmiotem zainteresowania kolejnego przypadku testowego. W rezultacie, dla odroczonej kontroli integralności referencyjnej *nie da się skonstruować testów negatywnych*.

Coś za coś...

Opisaliśmy trzy różne sposoby zapewnienia integralności referencyjnej. Podstawą pierwszego z nich są standardowe dla Rails klucze `id`. Kolumna `id` pełni wyłącznie rolę kluczową i nie reprezentuje żadnych treści merytorycznych, wskutek czego relacje między rekordami powiązanych tabel, całkowicie poprawne z punktu widzenia zgodności kluczy, niekoniecznie są poprawne z perspektywy rozwiązywanego problemu, ergo — klucze `id` nie zawsze są wystarczające do zapewnienia integralności referencyjnej, co stanowi przesłankę do definiowania i używania kluczy naturalnych.

Opisaliśmy dwa sposoby implementacji kluczy naturalnych na gruncie Rails. Pierwszy z nich opiera się na zastosowaniu dedykowanej wtyczki o nazwie `composite_primary_keys`, istotą drugiego jest jawna obsługa (zdefiniowanych w schemacie bazy) kluczy naturalnych na poziomie klasy modelowej, z zachowaniem standardowego dla Rails wiązania tabel na podstawie kolumn `id`. Krótką charakterystykę wszystkich trzech rodzajów kluczy zamieszczamy w tabeli 8.1.

Tabela 8.1. Podstawowe cechy trzech implementacji kluczy głównych

	Wyłącznie klucze `id`	Wyłącznie klucze naturalne, obsługiwane za pośrednictwem wtyczki	Model hybrydowy — współistnienie kluczy `id` z kluczami naturalnymi
Obsługa standardowa	Tak		Częściowo
Zapewnienie integralności referencyjnej na poziomie dziedziny problemowej	Nie	Tak	Tak
Możliwość zmiany klucza głównego za pośrednictwem API Rails	Nie	Nie	Tak
Efektywne wykorzystywanie indeksów	Tak	Tak	Nie zawsze
Komplikacja kodu aplikacji	Nie	Nie	Tak

Poświęcimy nieco uwagi dwóm ostatnim z wymienionych cech: efektywnemu korzystaniu z indeksów oraz komplikacji warstwy aplikacyjnej.

Efektywny użytek z indeksów

Jest oczywiste, że dla efektywnego działania modelu hybrydowego konieczne jest istnienie dwóch (co najmniej) indeksów, opartych na (odpowiednio) kolumnie `id` oraz zestawie kolumn tworzących klucz naturalny. Oba te indeksy muszą być aktualizowane po każdej operacji dodania, usunięcia lub zmodyfikowania rekordu. Weryfikacja integralności referencyjnej po wstawieniu lub zmodyfikowaniu rekordu w tabeli powiązanej wymaga sprawdzenia dwóch indeksów, o ile w ogóle dopuszczamy modyfikowanie klucza naturalnego — alternatywą dla modyfikacji kolumn wchodzących w skład klucza naturalnego rekordu jest usunięcie przedmiotowego rekordu i utworzenie nowego ze zmodyfikowanymi wartościami pól, co opisywaliśmy na przykładzie tabeli `movie_showtimes`. Czy ta alternatywa jest lepszym rozwiązaniem — zależne jest to od konkretnego problemu. Zmiana klucza naturalnego, w sytuacji gdy istnieją rekordy zależne od przedmiotowego rekordu, zawsze jest posunięciem ryzykownym, należy więc zastanowić się, czy faktycznie znajduje uzasadnienie w realiach problemu, z którym związane są przetwarzane dane.

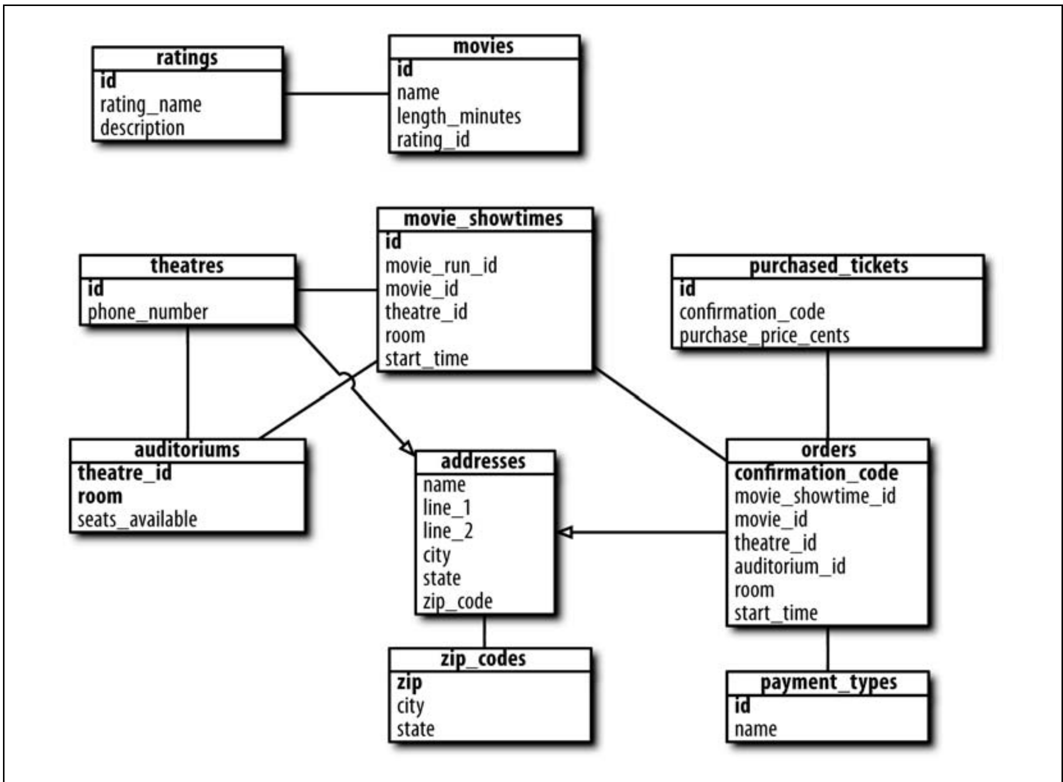
Komplikacja kodu aplikacji

Jedną z najistotniejszych cech języka Ruby, i w konsekwencji Rails, jest zwięzłość, czyli możliwość wyrażenia treściwej informacji w kilku zaledwie wierszach kodu. Wszystko, co odbywa się ze szkodą dla tej zwięzłości, traktowane bywa przez programistów jak przekleństwo. Jak ma się to do opisywanych implementacji kluczy naturalnych? Jak widzieliśmy, korzystanie z wtyczki `composite_primary_keys` wymaga niewielkiego narzutu ze strony kodowania — wszystko sprowadza się do wywołania metody `set_primary_keys` w przedmiotowej klasie modelowej i specyfikacji dodatkowego parametru (`:foreign_key`) w klasach powiązanych. W modelu hybrydowym nie pojawiają się żadne dodatkowe definicje, za to wobec faktu, że Rails nie jest w ogóle świadom istnienia klucza naturalnego, musimy we własnym zakresie implementować powiązania między rekordami na poziomie tego klucza. Wymaga to dodatkowego kodu, którego rozmiary można jednak zredukować, nadpisując metodę `create!` tworzącą obiekt klasy modelowej — co zaprezentowaliśmy na przykładzie klasy `Order`. I choć nie jest to naddatek spektakularny, warto wziąć go pod uwagę, decydując się na zastosowanie modelu hybrydowego zamiast „czystych" kluczy naturalnych.

Aktualna postać naszego modelu, po modyfikacjach opisanych w niniejszym rozdziale, widoczna jest na rysunku 8.8.

Ćwiczenia

1. Spróbuj doprowadzić do anomalii opisywanej na początku rozdziału (odwołanie do sali projekcyjnej nieistniejącej w konkretnym kinie) i postaraj się udowodnić, że przy odpowiednio wybranym kluczu naturalnym jej wystąpienie jest niemożliwe.
2. Podaj przykłady zapytań, których realizacja staje się efektywniejsza wskutek powiązania tabel `movie_showtimes` i `orders` za pomocą klucza naturalnego.



Rysunek 8.8. Rezultat wprowadzenia kluczy naturalnych do modelu

Refaktoryzacja

1. Przeanalizuj każdą tabelę i zastanów się, czy istnieją w niej kolumny niewchodzące w skład klucza głównego, dla których zdefiniowana jest klauzula unikalności bądź też klauzulę taką powinno się zdefiniować ze względu na charakter danych reprezentowanych przez te kolumny.
2. Jeżeli jakaś kolumna (lub zestaw kolumn) kwalifikuje się do opatrzenia klauzulą unikalności, zrób to:

```
create unique index concurrently <nazwa tabeli>,
 <nazwa kolumny1>_<nazwa kolumny2>_...<nazwa kolumnyN>_uniq_idx
 on nazwa tabeli(<nazwa kolumny1> , <nazwa kolumny2> , ... , <nazwa kolumnyN>);
```

3. Zależnie od tego, czy w punkcie 2. mowa jest o pojedynczej kolumnie, czy o zestawie kolumn, wybierz jeden z poniższych scenariuszy.

Klucz jednokolumnowy

1. W klasie modelowej reprezentującej tabelę wskaż kolumnę stanowiącą kandydata na nowy klucz główny:

```
set_primary_key  :<nazwa kolumny kluczowej>
```

2. W każdej z tabel powiązanych z przedmiotową tabelą dodaj nową kolumnę stanowiącą klucz obcy:

   ```
   alter table <nazwa tabeli zależnej>
     add column <nazwa tabeli głównej>_<nazwa kolumny kluczowej> <typ kolumny>;
   ```

3. Wypełnij nową kolumnę odpowiednimi wartościami we wszystkich rekordach każdej z tabel zależnych:

   ```
   update <nazwa tabeli zależnej>
     from <nazwa tabeli głównej> r
     set <nazwa tabeli głównej w liczbie pojedynczej>_<nazwa kolumny kluczowej> =
       r.<nazwa kolumny kluczowej>
   where <nazwa tabeli głównej w liczbie pojedynczej>_id = r.id;
   ```

4. Uzupełnij schemat każdej z tabel zależnych o definicję klucza obcego:

   ```
   alter table <nazwa tabeli zależnej>
   add constraint <nazwa tabeli głównej w liczbie pojedynczej>_<nazwa kolumny
     kluczowej>_fkey
   (<nazwa tabeli głównej w liczbie pojedynczej>_<nazwa kolumny kluczowej>)
   references <nazwa tabeli głównej> (<nazwa kolumny kluczowej>);
   ```

5. Usuń ze schematu tabeli przedmiotowej definicję kolumny `id`:

   ```
   alter table <nazwa tabeli głównej>
     drop column id;
   ```

6. Ze schematu każdej z tabel zależnych usuń definicję klucza obcego powołującą się na kolumnę, o której mowa w punkcie 5.:

   ```
   alter table <nazwa tabeli zależnej>
     drop column <nazwa tabeli głównej w liczbie pojedynczej>_id;
   ```

7. Obsadź nową kolumnę przedmiotowej tabel w roli jej klucza głównego:

   ```
   alter table <nazwa tabeli głównej>
   add primary key(<nazwa kolumny kluczowej>);
   ```

Klucz wielokolumnowy

1. Zainstaluj gem `composite_primary_keys`:

   ```
   ruby gem install composite_primary_keys
   ```

2. Dołącz gem do aplikacji, dopisując w pliku *environment.rb* wiersz:

   ```
   require 'composite_primary_keys'
   ```

3. W definicji klasy modelowej, reprezentującej przedmiotową tabelę, zmień definicję klucza głównego:

   ```
   set_primary_keys [:<nazwa kolumny1>, :<nazwa kolumny2>, ... , :<nazwa kolumnyN>]
   ```

4. Wykonaj czynności analogicznie jak w punktach od 2. do 7. scenariusza dla klucza jednokolumnowego.