John L. Viescas, Douglas J. Steele, Ben G. Clothier

Mistrzowski SQL

61 technik
pisania wydajnego kodu SQL

Helion

Tytuł oryginału: Effective SQL: 61 Specific Ways to Write Better SQL

Tłumaczenie: Jakub Hubisz

ISBN: 978-83-283-3563-9
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

1 Projektowanie modelu danych

„Nie zrobisz jedwabnej sakiewki z ucha maciory". Ten cytat, przypisywany żyjącemu w XVI w. angielskiemu satyrykowi Stephenowi Gossonowi, zdecydowanie ma przełożenie na bazy danych. Nie możesz pisać efektywnych zapytań SQL, pracując ze złym modelem danych. Gdy Twój model danych nie jest poprawnie znormalizowany i nie ma zdefiniowanych relacji, wyciąganie danych przy wykorzystaniu języka SQL będzie trudne lub nawet niemożliwe. W tym rozdziale omówimy podstawy dobrego projektu relacyjnego. Jeżeli Twoja baza łamie którąś z podanych tutaj zasad, musisz znaleźć problem i go rozwiązać.

Jeżeli to nie Ty kontrolujesz projekt, dowiesz się przynajmniej, co powoduje problemy, a to pozwoli Ci przedstawić potencjalne usprawnienia osobom odpowiedzialnym za projekt. Możesz wykorzystać informacje z tego rozdziału do wyjaśnienia, dlaczego trudne lub wręcz niemożliwe jest napisanie jakiegoś zapytania, o które zostałeś poproszony. Jeżeli nie możesz naprawić projektu bazy, np. ze względu na brak uprawnień lub dlatego, że baza należy do kogoś innego, zapoznaj się z rozdziałem 3. „Gdy nie możesz zmienić projektu", który przedstawia kilka technik SQL pozwalających obejść niektóre problemy.

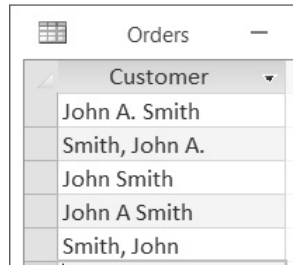
Nie jesteśmy w stanie omówić wszystkich niuansów projektowania baz danych, przedstawiamy jedynie podstawy. Jeżeli chcesz dokładniej zgłębić temat tworzenia relacyjnych baz danych, zapoznaj się z dobrą książką na ten temat, np. *Projektowanie baz danych dla każdego. Przewodnik krok po kroku* Michaela J. Hernandeza (Helion, 2014).

Zagadnienie 1: Sprawdzenie, czy wszystkie tabele posiadają klucz główny

Ponieważ przestrzeganie modelu relacyjnego wymaga, aby Twój system bazodanowy był w stanie odróżnić pojedynczy rekord od wszystkich innych, każda z tabel powinna mieć kolumnę lub zestaw kolumn zdefiniowany jako klucz główny. Zawartość klucza głównego musi być unikalna dla każdego rekordu i nie może mieć wartości null (spójrz na zagadnienie 10, „Podczas

tworzenia indeksów weź pod uwagę wartości NULL"). Bez klucza głównego nie ma możliwości zapewnienia, że podczas filtrowania znajdziesz dokładnie zero lub jeden z rekordów. Problemem jest jednak to, że utworzenie tabeli bez klucza głównego jest możliwe. Co więcej, samo posiadanie kolumn nieprzyjmujących wartości `null` i unikalnych w ramach tabeli nie oznacza jeszcze, że baza będzie w stanie wykorzystać kolumny efektywnie. Musisz jawnie wskazać klucz główny. Bez zdefiniowanego klucza głównego zazwyczaj niemożliwe (lub niewskazane) jest również modelowanie relacji między tabelami.

Gdy w tabeli brakuje klucza głównego, występować może wiele rodzajów problemów, w tym powtarzające się i niespójne dane, wolno działające zapytania i niedokładne informacje w raportach. Rozważmy przykład tabeli zawierającej zamówienia (`Orders`), przedstawionej na rysunku 1.1.



Rysunek 1.1. *Przykład niespójnych danych*

Z punktu widzenia komputera wszystkie wartości na rysunku 1.1 są unikalne, ale może być tak, że wszystkie reprezentują tę samą osobę — z pewnością rekordy 1, 2 i 4 są takie same (wariacje nazwiska John A. Smith). Wprawdzie komputery przetwarzają dane znacznie szybciej niż ludzki umysł, ale bez specjalistycznych programów nie są zbyt dobre w określaniu, kiedy dane powinny być uznawane za takie same. Tak więc, jeżeli nawet kolumnę `Customer` (klient) oznaczymy jako klucz główny tabeli, nie oznacza to, że jest to dobry wybór, ponieważ nie gwarantuje wystarczającej unikalności.

Jak zatem określić dobrego kandydata na klucz główny? Kolumna lub kolumny powinny posiadać następujące cechy:

- Muszą przechowywać unikalne wartości.
- Nie mogą przyjmować wartości `null`.
- Powinny być stabilne, to znaczy nigdy nie może wystąpić konieczność aktualizacji wartości.
- Powinny być możliwie proste (np. wykorzystana powinna być liczba całkowita zamiast zmiennoprzecinkowej, preferowana jest jedna kolumna zamiast wielu itp.).

Powszechna metoda osiągnięcia powyższych atrybutów to wykorzystanie jako klucza głównego sztucznie generowanych danych numerycznych, które nic nie znaczą. W zależności od wykorzystywanego systemu zarządzania danymi

(RSZBD) taki mechanizm jest różnie nazywany, np. `IDENTITY` w IBM DB2, Microsoft SQL Server i Oracle 12c, `AutoNumber` w Microsoft Access, `AUTO_INCREMENT` w MySQL i `serial` w PostgreSQL. We wcześniejszych wersjach Oracle'a do osiągnięcia podobnego wyniku konieczne było wykorzystanie obiektu sekwencji (`Sequence`), ale był to osobny obiekt, a nie atrybut kolumny. DB2, SQL Server i PostgreSQL również wspierają obiekt `Sequence`.

Integralność referencyjna **RI** (ang. *referential integrity*) to bardzo ważna koncepcja w bazach relacyjnych. Wspieranie integralności referencyjnej oznacza, że w przypadku klucza obcego nieakceptującego wartości `null` dla każdego rekordu z tabeli podrzędnej musi istnieć rekord w tabeli nadrzędnej.

W dobrze zaprojektowanej tabeli zamówień informacje o kliencie zapisane by były w postaci klucza obcego do tabeli klientów (`Customers`) wskazującego na klucz główny tej tabeli. Jeżeli faktycznie istnieje wielu klientów o nazwisku John Smith, każdy z tych klientów będzie posiadał swój własny unikalny klucz.

Aby zapewnić integralność referencyjną pomiędzy tabelami, wszelkie zmiany wartości klucza głównego muszą być kaskadowo rozpropagowane we wszystkich tabelach, które się do niego odwołują. W trakcie takich kaskadowych aktualizacji na tabelach odwołujących się zakładane są blokady, które mogą prowadzić do problemów w bazach wymagających równoległego dostępu wielu użytkowników. Rozważmy przykład z rysunku 1.2 przedstawiającego tabelę klientów (`Customers`) z bazy Northwind będącej częścią pakietu Microsoft Access 2003.

Results | Messages

	CustomerID	CompanyName	ContactName	ContactTitle
1	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative
2	ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner
3	ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner
4	AROUT	Around the Horn	Thomas Hardy	Sales Representative
5	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator
6	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative

Rysunek 1.2. *Przykładowe dane z tabeli Customers*

W tym przykładzie zakładamy, że regułą biznesową jest, aby tekstowy klucz główny, `CustomerId`, był powiązany z nazwą firmy. Gdyby któraś z firm zmieniła nazwę, identyfikator klienta również zostałby zmieniony. Wymagałoby to wprowadzenia kaskadowych zmian w powiązanych tabelach. Jeżeli wykorzystujesz nieznaczący klucz, unikasz konieczności aktualizowania jego wartości, a na potrzeby wyświetlania w osobnej kolumnie możesz przechowywać identyfikator generowany na podstawie nazwy.

Częstym argumentem za tekstowymi kluczami głównymi jest to, że zapobiegają one wstawianiu zduplikowanych wartości. Na przykład gdyby kolumna `CompanyName` (nazwa firmy) została ustawiona jako klucz główny, nie moglibyśmy wprowadzić dwóch takich samych nazw. Utworzenie unikalnego indeksu na kolumnie

CompanyName tabeli Customers jest proste i również zapewnia unikalność danych w kolumnie. Integralność jest zapewniona i nadal można korzystać z wygenerowanej wartości numerycznej jako klucza głównego. Wykorzystując wskazówki z zagadnienia 2, „Eliminacja nadmiarowego przechowywania danych" i zagadnienia 4, „Przechowywanie jednej właściwości w kolumnie", unikniesz problemu, który przedstawiliśmy na rysunku 1.1. Z drugiej strony prawdą jest, że wykorzystanie tekstowych kluczy głównych często pozwala uprościć zapytania SQL dzięki uniknięciu złączeń w celu pobrania wartości powiązanych z kluczem numerycznym (CompanyName w przykładzie z rysunku 1.2).

W środowisku profesjonalistów baz danych prowadzono wiele dyskusji związanych z tym, który klucz jest bardziej przydatny: czy numeryczny, czy tekstowy. Nie chcemy zajmować strony w tym sporze: najważniejsze jest zastosowanie we wszystkich tabelach unikalnego identyfikatora, który może być wykorzystany jako klucz główny.

Nie radzimy również korzystać ze złożonych kluczy głównych, ponieważ z dwóch powodów są mniej wydajne:

1. Gdy definiujesz klucz główny, większość baz danych wspiera definicję za pomocą unikalnego indeksu. Unikalny indeks na więcej niż jednej kolumnie to więcej pracy dla systemu zarządzania bazą danych.
2. Wykonywanie złączenia na kluczu głównym jest dość powszechne, ale robienie tego na większej liczbie kolumn jest bardziej skomplikowane i wolniejsze.

W niektórych przypadkach wykorzystanie kilku kolumn jako klucza głównego może mieć jednak swoje uzasadnienie. Rozważmy tabelę łączącą produkty i producentów, która składa się z kolumn VendorID (identyfikator producenta) i ProductID (identyfikator produktu) jako klucza głównego. Tabela może zawierać inne kolumny, np. identyfikator, informację o tym, czy producent jest głównym czy alternatywnym dostawcą produktu, albo cenę, jakiej producent żąda za produkt.

Mógłbyś utworzyć dodatkowo generowaną kolumnę z liczbami, która pełniłaby rolę sztucznego klucza głównego, ale jako klucz główny możesz również wykorzystać kombinację kolumn VendorID i ProductID. Złączenia z tą tabelą będą zawsze wykonywane za pomocą pojedynczych kolumn, dlatego zdefiniowanie klucza złożonego może być wydajniejsze niż wykorzystanie dodatkowej kolumny z kluczem. Powinieneś zdefiniować kolumny jako klucz złożony, a z dodatkowej kolumny zrezygnować. W zagadnieniu 8, „Gdy 3NF to za mało, normalizuj dalej", znajdziesz analizę przykładu, w którym pożądane mogą być złożone klucze główne.

Do zapamiętania

- Wszystkie tabele powinny posiadać kolumnę (lub zestaw kolumn) zaprojektowaną jako klucz główny.

- Jeżeli martwisz się zduplikowanymi wartościami w kolumnie nie będącej częścią klucza, możesz zdefiniować indeks unikalny na kolumnie, co zapewni integralność.
- Wykorzystuj jak najprostsze klucze zawierające wartości, których nie trzeba aktualizować.

Zagadnienie 2: Eliminacja nadmiarowego przechowywania danych

Nadmiarowo przechowywane dane powodują wiele problemów, np. niespójność danych, anomalie podczas wykonywania operacji wstawiania, aktualizacji i usuwania, a także marnotrawienie miejsca na dysku. Normalizacja to proces dzielenia informacji tematycznie w celu uniknięcia przechowywania duplikatów danych. Zwróć uwagę, że pisząc „nadmiarowe", nie mamy na myśli oczywistej duplikacji danych klucza głównego jako klucza obcego w innej tabeli. Taka nadmiarowość jest konieczna do zachowania połączenia między tabelami. Bardziej interesuje nas sytuacja, w której użytkownik wprowadza te same dane więcej niż jeden raz.

Ze względu na ograniczone miejsce nie możemy się zbytnio zagłębiać w kwestię normalizacji baz danych, jednak bardzo ważne jest, aby ludzie pracujący z bazami danych dokładnie rozumieli ten temat. Jest tu wiele doskonałych źródeł informacji.

Jednym z celów normalizacji jest zminimalizowanie konieczności powtarzania tych samych danych zarówno w obrębie tej samej tabeli, jak i we wszystkich tabelach w bazie danych. Kilka przykładów nadmiarowości danych obrazuje baza sprzedaży na rysunku 1.3.

CustomerSales

SalesID	CustFirstName	CustLastName	Address	City	Phone	
1	Amy	Bacock	111 Dover Lane	Chicago	312-222-1111	...
2	Tom	Frank	7453 NE 20th St.	Bellevue	425-888-9999	...
3	Debra	Smith	3223 SE 12th Pl.	Seattle	206-333-4444	...
4	Barney	Killjoy	4655 Rainier Ave.	Auburn	253-111-2222	...
5	Homer	Tyler	1287 Grady Way	Renton	425-777-8888	...
6	Tom	Frank	7435 NE 20th St.	Bellevue	425-888-9999	...

	PurchaseDate	ModelYear	Model	SalesPerson
...	2/14/2016	2016	Mercedes R231	Mariam Castro
...	3/15/2016	2016	Land Rover	Donald Ash
...	1/20/2016	2016	Toyota Camry	Bill Baker
...	12/22/2015	2016	Subaru Outback	Bill Baker
...	11/10/2015	2016	Ford Mustang GT Convertible	Mariam Castro
...	5/25/2015	2015	Cadillac CT6 Sedan	Jessica Robin

Rysunek 1.3. *Przykład nadmiarowego przechowywania danych w jednej tabeli*

Przykładem niespójnych danych jest adres klienta Toma Franka. W drugim rekordzie wartość numeryczna jego adresu to 7453, natomiast w szóstym 7435. Podobna niespójność może występować we wszystkich kolumnach.

Anomalia wstawiania może występować wtedy, gdy na przykład nie można wprowadzić informacji dla modelu samochodu, dopóki nie wystąpi jego sprzedaż, którą wprowadza się wraz z danymi klienta. Projekt wymaga również powtarzania większości danych w momencie zakupu przez klienta nowego samochodu. Te dodatkowo wprowadzane dane powodują marnotrawienie miejsca na dysku, pamięci, zasobów sieciowych, a nawet czasu użytkownika, który wprowadza te dane. Co więcej, ponowne wpisywanie danych znacząco zwiększa ryzyko błędów, takich jak zamiana miejscami numerów w adresie klienta, co obrazuje rysunek 1.3.

Może występować również anomalia aktualizacji. Jeśli na przykład sprzedawca zmienia swój stan cywilny, a w konsekwencji nazwisko, konieczna jest aktualizacja wszystkich rekordów, w których występuje nazwisko tej osoby. Może to być problematyczne, gdy masz do czynienia z dużą liczbą rekordów w bazie, z której wspólnie korzysta wielu użytkowników. Dodatkowo taka aktualizacja będzie wykonana poprawnie tylko wtedy, gdy wszystkie wystąpienia nazwiska są napisane dokładnie w ten sam sposób (dane są spójne) i gdy nie ma dwóch osób o tym samym nazwisku.

Występuje również anomalia usuwania, ponieważ możesz wtedy stracić dane, których nie chciałeś usuwać z bazy.

Dane sprzedażowe przedstawione na rysunku 1.3 mogą być logicznie podzielone na cztery tabele:

1. `Customers` — tabela z klientami (nazwa, adres itd.)
2. `Employees` — tabela z pracownikami (nazwa sprzedawcy, data zatrudnienia itd.)
3. `AutomobileModels` — tabela z modelami samochodów (rok modelowy, model itd.)
4. `SalesTransactions` — tabela z transakcjami sprzedaży

Taki projekt pozwala wprowadzać dane klientów, pracowników i modeli samochodów tylko raz do tabel dla nich przeznaczonych. Wszystkie tabele zawierają unikalny identyfikator, który może zostać ustawiony jako klucz główny. Tabela `SalesTransactions` wykorzystuje klucze obce do przechowywania szczegółów każdej transakcji. Zobacz rysunek 1.4.

Uważny czytelnik mógł zauważyć, że w tym procesie został wyeliminowany jeden zduplikowany rekord klienta. Wymagało to określenia, który rekord adresu jest poprawny dla klienta Tom Frank.

Możemy utworzyć relacje (określane również jako więzy integralności klucza głównego), łącząc klucz główny z trzech tabel nadrzędnych (`Customers`, `AutomobileModels` i `Employees`) i klucze obce z tabeli `SalesTransactions` (patrz rysunek 1.5). Stworzyliśmy przykład na podstawie rysunku, korzystając z edytora relacji zawartego w Microsoft Access. Każda baza relacyjna posiada inny sposób na reprezentowanie relacji pomiędzy tabelami.

Customers

CustomerID	CustFirstName	CustLastName	Address	City	Phone
1	Amy	Bacock	111 Dover Lane	Chicago	312-222-1111
2	Tom	Frank	7453 NE 20th St.	Bellevue	425-888-9999
3	Debra	Smith	3223 SE 12th Pl.	Seattle	206-333-4444
4	Barney	Killjoy	4655 Rainier Ave.	Auburn	253-111-2222
5	Homer	Tyler	1287 Grady Way	Renton	425-777-8888

Employees

EmployeeID	SalesPerson
1	Mariam Castro
2	Donald Ash
3	Bill Baker
4	Jessica Robin

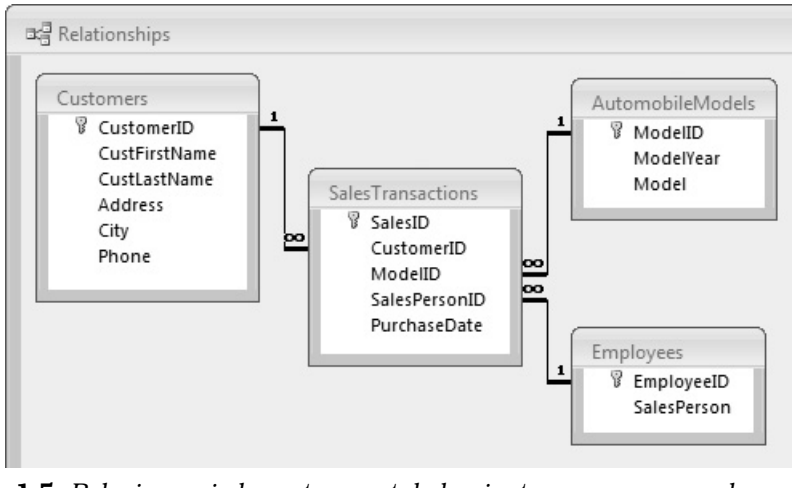
AutomobileModels

ModelID	ModelYear	Model
1	2016	Mercedes R231
2	2016	Land Rover
3	2016	Toyota Camry
4	2016	Subaru Outback
5	2016	Ford Mustang GT Convertible
6	2015	Cadillac CT6 Sedan

SalesTransactions

SalesID	CustomerID	ModelID	SalesPersonID	PurchaseDate
1	1	1	1	2/14/2016
2	2	2	2	3/15/2016
3	3	3	3	1/20/2016
4	4	4	3	12/22/2015
5	5	5	1	11/10/2015
6	2	6	4	5/25/2015

Rysunek 1.4. *Przykład dzielenia danych na tabele zgodnie z tematyką*



Rysunek 1.5. *Relacje pomiędzy czterema tabelami, utworzone przez połączenie kluczy głównych i kluczy obcych*

Możesz łatwo odtworzyć przedstawione na rysunku 1.3 oryginalne dane, konstruując tabelę wirtualną (zapytanie) zgodnie z listingiem 1.1 i unikając jednocześnie problemów związanych z przechowywaniem nadmiarowych danych. (Tworzenie tabeli wirtualnej to doskonały przykład wykorzystania wspólnych wyrażeń tabelarycznych, które omówione zostały w zagadnieniu 42, „Wykorzystanie wspólnych wyrażeń tabelarycznych zamiast podzapytań").

Listing 1.1. *Zapytanie SQL zwracające oryginalne dane*

```
SELECT st.SalesID, c.CustFirstName, c.CustLastName, c.Address, c.City, c.Phone,
➥st.PurchaseDate, m.ModelYear, m.Model, e.SalesPerson
FROM SalesTransactions st
    INNER JOIN Customers c
        ON c.CustomerID = st.CustomerID
    INNER JOIN Employees e
        ON e.EmployeeID = st.SalesPersonID
    INNER JOIN AutomobileModels m
        ON m.ModelID = st.ModelID;
```

Do zapamiętania

- Celem normalizacji baz danych jest eliminacja nadmiarowych danych i minimalizacja wykorzystania zasobów wykorzystywanych podczas przetwarzania danych.
- Poprzez usunięcie nadmiarowych danych eliminujesz anomalie wstawiania, aktualizacji i usuwania.
- Poprzez eliminację nadmiarowych danych minimalizujesz występowanie niespójnych danych.

Odniesienia

Jeżeli chcesz dowiedzieć się więcej na temat poprawnych sposobów projektowania baz relacyjnych, skorzystaj z zamieszczonych poniżej dobrych źródeł wiedzy; pierwsze z nich jest przystępne dla początkujących i może być dobre na początek:

- Michael J. Hernandez, *Projektowanie baz danych dla każdego. Przewodnik krok po kroku*, Helion, 2014.
- Candace C. Fleming i Barbara von Halle, *Handbook of Relational Database Design*, Addison-Wesley, 1989.

Zagadnienie 3: Pozbywanie się powtarzających się grup

Często się zdarza, że arkusze kalkulacyjne zawierają powtarzające się grupy podobnych danych. Pracownicy pracujący z tymi danymi importują je do nowej bazy danych, nie biorąc w ogóle pod uwagę normalizacji. Przykład powtarzających się grup danych przedstawiony został na rysunku 1.6, gdzie numer rysunku (`DrawingNumber`) został powiązany z pięcioma poprzednikami (`Predecessor`). Tabela posiada relację „jeden do wielu" pomiędzy numerami rysunków i wartościami poprzedników.

Assignments

ID	DrawingNumber	Predecessor_1	Predecessor_2	Predecessor_3	Predecessor_4	Predecessor_5
1	LO542B2130	LS01847409	LS02390811	LS02390813	LS02390817	LS02390819
2	LO426C2133	LS02388410	LS02495236	LS02485238	LS02495241	LS02640008
3	LO329W2843-1	LS02388418	LS02640036	LS02388418		
4	LO873W1842-1	LS02388419	LS02741454	LS02741456	LS02769388	
5	LO690W1906-1	LS02742130				
6	LO217W1855-1	LS02388421	LS02769390			

Rysunek 1.6. *Powtarzające się grupy danych w jednej tabeli*

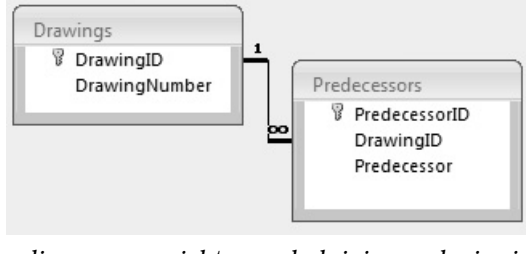
Przykład na rysunku 1.6 przedstawia pojedynczy atrybut, `Predecessor`, jako powtarzającą się grupę. Mamy też powtarzającą się wartość dla tego atrybutu, czyli dla `ID=3`, co nie jest zamierzone. Innym przykładem mogłyby być kolumny nazwane `Styczeń`, `Luty`, `Marzec` itd. Jednak powtarzające się grupy nie są ograniczone do pojedynczych atrybutów. Jeżeli na przykład zobaczysz kolumny takie jak `Ilosc1`, `Opis1`, `Cena1`, `Ilosc2`, `Opis2`, `Cena2`, ..., `IloscN`, `OpisN`, `CenaN`, możesz uznać je za wzorzec powtarzającej się grupy.

Powtarzające się grupy powodują problemy przy pisaniu zapytań i tworzeniu raportów grupujących po atrybutach. Jeśli do przykładu z rysunku 1.6 chciałbyś dodać wartości poprzedników (`Predecessor`) lub obniżyć liczbę dostępnych poprzedników, aktualny projekt wymagałby dodawania lub usuwania kolumn. Musiałbyś również zmodyfikować wszystkie zapytania (widoki), formularze i raporty zależne od danych w tej kolumnie. Należy zawsze pamiętać o ważnej zasadzie:

Kolumny są kosztowne.

Rekordy są tanie.

Powinieneś uważać zawsze, gdy projekt bazy wymaga dodawania lub usuwania kolumn, aby sprostać późniejszym wymaganiom danych. Dużo lepszym rozwiązaniem jest projekt pozwalający dodawać i usuwać wiersze. W tym przykładzie utworzymy tabelę `Predecessors` (poprzednicy) korzystającą z identyfikatora jako klucza obcego. Dla przejrzystości zmienimy również nazwę istniejącej kolumny `ID` na `DrawingID`, zgodnie z rysunkiem 1.7.



Rysunek 1.7. *Znormalizowany projekt uwzględniający relację „jeden do wielu”*

Podczas pracy z powtarzającymi się grupami użyteczne są zapytania `UNION`. Możemy wykorzystać takie zapytanie do „znormalizowania” naszych danych w widoku tylko do odczytu, jeżeli nie mamy możliwości utworzenia poprawnie

znormalizowanego schematu. Możemy również wykorzystać podobne zapytanie jako źródło zapytania dodające rekordy do nowej tabeli Predecessors, zgodnie z listingiem 1.2.

Listing 1.2. *Zapytanie UNION normalizujące dane*

```
SELECT ID AS DrawingID, Predecessor_1 AS Predecessor
FROM Assignments WHERE Predecessor_1 IS NOT NULL
UNION
SELECT ID AS DrawingID, Predecessor_2 AS Predecessor
FROM Assignments WHERE Predecessor_2 IS NOT NULL
UNION
SELECT ID AS DrawingID, Predecessor_3 AS Predecessor
FROM Assignments WHERE Predecessor_3 IS NOT NULL
UNION
SELECT ID AS DrawingID, Predecessor_4 AS Predecessor
FROM Assignments WHERE Predecessor_4 IS NOT NULL
UNION
SELECT ID AS DrawingID, Predecessor_5 AS Predecessor
FROM Assignments WHERE Predecessor_5 IS NOT NULL
ORDER BY DrawingID, Predecessor;
```

> **UWAGA**
> Jeśli musielibyśmy pobrać wszystkie dane razem, włącznie z duplikatami w ramach wiersza, moglibyśmy dodać słowo kluczowe ALL po każdym wystąpieniu słowa UNION, czyli operować wyrażeniem UNION ALL. W tym przypadku jednak chcemy wyeliminować z tabeli Predecessor zduplikowane wartości, ponieważ są one błędnie wprowadzone, tak jak w przypadku rekordu z identyfikatorem 3.

Zapytanie UNION wymaga, aby kolumny były tego samego typu i w tej samej kolejności dla każdego polecenia SELECT. Oznacza to, że umieszczanie AS DrawingID czy AS Predecessor po pierwszej instancji nie jest konieczne, ponieważ zapytanie UNION pobierze nazwy kolumn z pierwszego polecenia SELECT.

Każde polecenie SELECT może mieć różne predykaty w klauzuli WHERE. Zależnie od danych możemy również wykluczyć łańcuchy znaków o zerowej długości (ZLS — ang. *zero length strings*) i/lub białe znaki, takie jak pojedyncza spacja.

Zapytanie UNION może wykorzystywać pojedynczą klauzulę ORDER BY na końcu. Możemy definiować referencje porządkowe, czyli ORDER BY 1, 2. Byłoby to równoznaczne z poleceniem ORDER BY DrawingID, Predecessor z listingu 1.2.

Do zapamiętania

- Celem normalizacji baz danych jest eliminacja powtarzających się grup danych i minimalizacja zmian schematu.
- Eliminując powtarzające się grupy danych, możesz wykorzystać indeksowanie do uniknięcia przypadkowej duplikacji danych oraz znacząco ułatwiasz ewentualne tworzenie zapytań.

- Usuwanie powtarzających się grup danych sprawia, że projekt bazy jest bardziej elastyczny, ponieważ dodanie nowej grupy wymaga po prostu dodania kolejnego rekordu danych, a nie zmiany schematu i dodania nowych kolumn.

Zagadnienie 4: Przechowywanie jednej właściwości w kolumnie

W terminologii relacyjnej relacja (tabela) powinna opisywać jeden i tylko jeden przedmiot akcji. Atrybuty (kolumny) zawierają dane odpowiadające tylko jednej właściwości opisującej podmiot definiowany przez relację (takie dane nazywane są danymi „atomowymi"). Atrybut może również być kluczem obcym zawierającym atrybut z innej relacji, a z kolei ten klucz obcy może zawierać połączenie z innym tupletem (wierszem) z innej relacji.

Przechowywanie więcej niż jednej wartości właściwości w jednej kolumnie nie jest dobrym pomysłem, ponieważ ciężko wtedy wyizolować wartość właściwości podczas wykonywania wyszukiwania lub agregowania wartości. Powinieneś starać się umieszczać ważne pojedyncze właściwości w ich własnych kolumnach. Przykład tabeli zawierającej wiele właściwości w jednej kolumnie znajdziesz w tabeli 1.1. (Przy okazji: adresy z tej tabeli są prawdziwe, ale nie są to prawdziwe adresy wspomnianych autorów).

Tabela 1.1. *Tabela zawierająca wiele atrybutów w pojedynczych kolumnach*

AuthID	AuthName	AuthAddress
1	John L. Viescas	144 Boulevard Saint-Germain, 75006, Paris, France
2	Douglas J. Steele	555 Sherbourne St., Toronto, ON M4X 1W6, Canada
3	Ben Clothier	2015 Monterey St., San Antonio, TX 78207, USA
4	Tom Wickerath	2317 185th Place NE, Redmond, WA 98052, USA

W takiej tabeli napotykamy kilka problemów:

- Trudne, a nawet wprost niemożliwe, jest wyszukiwanie po nazwisku. Zakładając, że tabela zawiera więcej niż tylko cztery przykładowe rekordy, a Ty chcesz wyszukać kogoś o nazwisku `Smith`, wyszukiwanie `LIKE` z maskowaniem wyszuka również nazwiska `Smithson` lub `Blacksmith`.
- Możesz szukać po imieniu, ale wtedy musisz użyć mniej wydajnej klauzuli `LIKE` lub wyciąć imię z ciągu znaków. Klauzula `LIKE` z maską na końcu może być wykonywana efektywnie, ale ze względu na zwroty grzecznościowe (takie jak *Mr.*) musisz użyć maski również na początku, aby mieć pewność, że odnajdziesz szukane imię, a to z kolei spowoduje skanowanie danych.
- Nie możesz łatwo wyszukiwać po nazwie ulicy, mieście, stanie/województwie czy kodzie pocztowym.

- Mimo prób łączenia danych w grupy (być może złączonych z inną tabelą zawierającą przypisane rozdziały i liczby stron) ciężko jest wyekstrahować stan/województwo, kod pocztowy czy kraj na potrzeby grupowania.

Najczęściej tego typu dane spotkasz w przypadku importów z innych źródeł danych, takich jak arkusze kalkulacyjne. Tak źle zaprojektowaną tabelę nierzadko spotkasz w środowiskach produkcyjnych.

Bardziej poprawnym rozwiązaniem będzie utworzenie tabeli podobnej do tej z listingu 1.3.

Listing 1.3. *Listing tworzący tabelę autorów (Authors) z oddzielnymi atrybutami*

```
CREATE TABLE Authors (
    AuthorID int IDENTITY (1,1),
    AuthFirst varchar(20),
    AuthMid varchar(15),
    AuthLast varchar(30),
    AuthStNum varchar(6),
    AuthStreet varchar(40),
    AuthCity varchar(30),
    AuthStProv varchar(2),
    AuthPostal varchar(10),
    AuthCountry varchar(35)
);

INSERT INTO Authors (AuthFirst, AuthMid, AuthLast, AuthStNum, AuthStreet,
↪AuthCity, AuthStProv, AuthPostal, AuthCountry)
    VALUES ('John', 'L.', 'Viescas', '144', 'Boulevard Saint-Germain',
    ↪'Paris', ' ', '75006', 'France');

INSERT INTO Authors (AuthFirst, AuthMid, AuthLast, AuthStNum, AuthStreet,
↪AuthCity, AuthStProv, AuthPostal, AuthCountry)
    VALUES ('Douglas', 'J.', 'Steele', '555', 'Sherbourne St.', 'Toronto', 'ON',
    ↪'M4X 1W6', 'Canada');
-- … kolejne rekordy.
```

Zwróć uwagę, że również dla numeru domu użyliśmy klucza typu znakowego, ponieważ niekiedy „numer" zawiera również litery i inne znaki (np. *8a*, a we Francji numery zawierają dodatkowo końcówkę *bis*). To samo dotyczy kodów pocztowych, które w Stanach Zjednoczonych są numeryczne, ale w Kanadzie czy Wielkiej Brytanii zawierają litery i spacje.

Używając zasugerowanej tabeli, dane można podzielić tak, aby każdy atrybut przechowywany był w odrębnej kolumnie zgodnie z tabelą 1.2.

Teraz łatwo możemy wykonywać wyszukiwanie i grupowanie na pojedynczych lub wielu właściwościach, ponieważ każda właściwość przechowywana jest w odrębnej kolumnie.

Tabela 1.2. *Poprawnie zaprojektowana tabela Authors przechowująca właściwości w odrębnych kolumnach*

Auth ↪ID	Auth ↪Name	Auth ↪Mid	Auth ↪Last	AuthSt ↪Num	Auth ↪Street	Auth ↪City	AuthSt ↪Prov	Auth ↪Postal	Auth ↪Country
1	John	L.	Viescas	144	Boule--vard Saint-Germain	Paris		75006	France
2	Douglas	J.	Steele	555	Sherbo--urne St.	Toronto	ON	M4X 1W6	Canada
3	Ben		Clot--hier	2015	Monte--rey St.	San Antonio	TX	78207	USA
4	Tom		Wicke--rath	2317	185th Place NE	Redmont	WA	98052	USA

Jeżeli musisz powtórnie połączyć właściwości, aby na przykład utworzyć listę mailingową, wystarczy wykorzystać w zapytaniu SQL konkatenację. Listing 1.4 pokazuje, jak to zrobić.

Listing 1.4. *Powtórne połączenie oryginalnych danych przy wykorzystaniu konkatenacji*

```
SELECT AuthorID AS AuthID, CONCAT(AuthFirst,
    CASE
        WHEN AuthMid IS NULL
        THEN ' '
        ELSE CONCAT(' ', AuthMid, ' ')
        END, AuthLast) AS AuthName,
    CONCAT(AuthStNum, ' ', AuthStreet, ' ', AuthCity, ', ', AuthStProv, ' ',
    ↪AuthPostal, ', ', AuthCountry)
        AS AuthAddress
FROM Authors;
```

> **UWAGA**
> IBM DB2, Microsoft SQL Server, MySQL, Oracle i PostgreSQL wspierają funkcję `CONCAT()`, ale w DB2 i Oracle funkcja ta przyjmuje tylko dwa argumenty, musisz więc zagnieździć funkcję `CONCAT()`, aby złączyć wiele ciągów znaków. Standard ISO do konkatenacji definiuje jedynie operator `||`. DB2, Oracle i PostgreSQL akceptują ten operator, a MySQL akceptuje go, jeżeli ustawienie `sql_mode` ma wartość `PIPES_AS_CONCAT`. W SQL Server jako operator konkatenacji wykorzystywany jest operator +. Microsoft Access nie wspiera funkcji `CONCAT()`, ale pozwala łączyć ciągi znaków przy wykorzystaniu operatorów & lub +.

Wcześniej wspominaliśmy, że listing 1.3 przedstawia jeden z kilku możliwych „w miarę poprawnych" projektów. Możesz się więc teraz zastanawiać, dlaczego rekomendujemy oddzielenie numeru domu od nazwy ulicy. Prawdę powiedziawszy, w większości zastosowań połączenie nazwy ulicy z numerem domu będzie

wystarczające. Musisz jednak zawsze dokładnie rozpatrzyć potrzeby aplikacji. W przypadku bazy urzędu geodezyjnego oddzielenie numeru domu od nazwy ulicy (być może nawet dodanie typu adresu, np. ulica, aleja, osiedle) może być niezbędne. W innych aplikacjach ważne może być rozdzielenie kodu kraju, numeru kierunkowego i lokalnego numeru telefonu. Musisz zdecydować, które części są wystarczająco ważne, aby ustalić, jak bardzo będą rozdzielone atrybuty.

Oczywiste jest, że rozdzielenie właściwości na osobne kolumny ułatwia wyszukiwanie i grupowanie poszczególnych elementów danych. Powtórne złożenie danych na potrzeby raportów lub list wyświetlanych jest również bardzo proste.

Do zapamiętania

- Poprawny projekt bazy przypisuje pojedyncze właściwości do własnej kolumny. Gdy kolumna zawiera wiele właściwości, wyszukiwanie i grupowanie staje się trudne lub wręcz niemożliwe.
- W przypadku niektórych aplikacji konieczność filtrowania danych na podstawie kolumn takich jak adres czy numer telefonu może dyktować szczegółowość danych.
- W przypadku konieczności powtórnego złączenia właściwości na potrzeby raportu lub list wyświetlanych wykorzystaj konkatenację.

Zagadnienie 5: Dlaczego przechowywanie danych wyliczeniowych zazwyczaj nie jest dobrym pomysłem

Czasami możesz poczuć pokusę przechowania danych wyliczeniowych, szczególnie gdy wynik uzależniony jest od danych w tabeli powiązanej. Rozważmy przykład z listingu 1.5.

Listing 1.5. *Przykładowa definicja tabeli w SQL*

```
CREATE TABLE Orders (
    OrderNumber int NOT NULL,
    OrderDate date NULL,
    ShipDate date NULL,
    CustomerID int NULL,
    EmployeeID int NULL,
    OrderTotal decimal(15,2) NULL
);
```

Na pierwszy rzut oka zawarcie w tabeli `Orders` (zamówienia) kolumny `OrderTotal` (suma zamówienia) zawierającej najprawdopodobniej sumę iloczynów *liczby produktów* i *ceny jednostkowej* wszystkich powiązanych rekordów z tabeli przechowującej szczegóły zamówienia może wydawać się dobrym pomysłem,

ponieważ nie będzie trzeba pobierać powiązanych rekordów i wykonywać obliczeń za każdym razem, gdy będziemy chcieli pobrać zamówienia wraz z ich wartością. Takie pole może być dobre w hurtowni danych, ale może negatywnie odbić się na wydajności bazy aktywnie użytkowanego systemu (patrz również zagadnienie 9, „Wykorzystanie denormalizacji w magazynach danych"). Trudne może okazać się utrzymanie integralności danych, ponieważ wciąż musiałbyś pamiętać o tym, aby dane w kolumnie były zmieniane za każdym razem, gdy któryś z powiązanych rekordów zostanie zaktualizowany, wstawiony lub usunięty.

Dobrą wiadomością jest natomiast to, że wiele współczesnych systemów zarządzania bazami zapewnia sposoby na takie ustawienie kolumny, aby kod działający na serwerze wykonywał kalkulacje za Ciebie. Najbardziej prymitywnym sposobem na dopilnowanie, aby kolumna miała zawsze aktualne wartości, jest utworzenie wyzwalacza dla tabeli zawierającej dane źródłowe dla obliczeń. Wyzwalacz to kod uruchamiany wtedy, gdy do tabeli wstawiane są dane lub gdy istniejące dane są aktualizowane bądź usuwane. W przykładzie z listingu 1.5 wyzwalacz zostałby utworzony dla tabeli `Order_Details` (szczegóły zamówienia) i przeliczałby wartość kolumny `OrderTotal`. Wyzwalacze mogą jednak być kosztowne, a ich poprawne utworzenie może nie być proste (patrz również zagadnienie 13, „Nie przesadź z wyzwalaczami").

Niektóre systemy bazodanowe dają potencjalnie lepsze rozwiązanie problemu, czyli możliwość zdefiniowania kolumny wyliczeniowej podczas definiowania tabeli. Uważamy, że to rozwiązanie jest lepsze, ponieważ pozwala uniknąć konieczności tworzenia skomplikowanego kodu często wymaganego w wyzwalaczach. Niektóre systemy RSZBD (szczególnie w nowszych wersjach) wspierają już definiowanie kolumn wyliczeniowych. Na przykład Microsoft SQL Server udostępnia słowo kluczowe `AS`, po którym należy podać wyrażenie definiujące wymagane obliczenia. Gdy w obliczeniach wykorzystywane są jedynie kolumny z tej samej tabeli, możesz po prostu wykorzystać nazwy kolumn w definicji wyrażenia. Jeżeli do obliczenia konieczne jest wykorzystanie wartości z tabeli powiązanej, niektóre systemy pozwalają zdefiniować funkcję wykonującą obliczenia, a następnie wykorzystać funkcję w klauzuli `AS` podczas tworzenia lub modyfikacji tabeli docelowej. Listing 1.6 przedstawia przykładową funkcję i definicję tabeli w systemie Microsoft SQL Server. Zwróć uwagę, że ponieważ funkcja korzysta z danych z innej tabeli, jest ona niedeterministyczna, nie można więc utworzyć indeksu dla kolumny wyliczeniowej.

Listing 1.6. *Przykładowa funkcja i definicja tabeli dla Microsoft SQL Server*

```
CREATE FUNCTION dbo.getOrderTotal(@orderId int)
RETURNS money
AS
BEGIN
    DECLARE @r money
    SELECT @r = SUM(Quantity * Price)
    FROM Order_Details WHERE OrderNumber = @orderId
    RETURN @r;
END;
```

```
GO
CREATE TABLE Orders (
    OrderNumber int NOT NULL,
    OrderDate date NULL,
    ShipDate date NULL,
    CustomerID int NULL,
    EmployeeID int NULL,
    OrderTotal money AS dbo.getOrderTotal(OrderNumber)
);
```

Deterministyczny kontra niedeterministyczny

Funkcja deterministyczna to taka, która zawsze zwróci ten sam wynik dla jednego zestawu parametrów wsadowych. Funkcja niedeterministyczna może zwracać różne rezultaty dla każdego wywołania z tym samym zestawem parametrów. Na przykład funkcja wbudowana bazy SQL Server `DATEADD()` jest deterministyczna, ponieważ zawsze zwróci taki sam wynik dla takiego samego zestawu trzech parametrów, natomiast funkcja `GETDATE()` jest niedeterministyczna, gdyż zawsze wywoływana jest z tym samym argumentem, jednak wartość zwracana może się zmienić przy każdym jej uruchomieniu. (Zakładamy, że trzy parametry funkcji `DATEADD()` również są deterministyczne. To znaczy, że nie możesz użyć funkcji `GETDATE()` jako jednego z argumentów). W dodatku „Typy, operatory i funkcje dla dat i czasu" znajdziesz dokładniejsze informacje dotyczące funkcji dat i czasu w Twoim systemie bazodanowym.

Zrobienie tego w ten sposób jest jednak bardzo złym pomysłem. Ponieważ funkcja jest niedeterministyczna, kolumna nie może zostać zapisana na stałe (`PERSISTED`) jak prawdziwa kolumna. Na kolumnie nie można zbudować indeksu, a za każdym razem gdy odwołasz się do tej kolumny, serwer będzie musiał wykonać dużo pracy, ponieważ funkcja będzie wywoływana dla każdego rekordu. Znacznie efektywniejsze byłoby połączenie tabel za pomocą podzapytania wykonującego obliczenia z grupowaniem po kolumnie `OrderID` za każdym razem, gdy potrzebne będą dane.

W systemie IBM DB2 istnieje podobna funkcjonalność, ale słowo kluczowe to `GENERATED`. W DB2 niemożliwe jest jednak utworzenie kolumny wyliczeniowej z wykorzystaniem funkcji, która wywołuje zapytanie, ponieważ czyni to funkcję niedeterministyczną. Można natomiast utworzyć kolumnę z wykorzystaniem funkcji lub wyrażenia deterministycznego. Listing 1.7 pokazuje, jak zdefiniować wyrażenie obliczające liczbę produktów pomnożonych przez cenę i zwracające całkowitą cenę, które może być wykorzystane do utworzenia kolumny w tabeli `Order_Details`.

Listing 1.7. *Przykładowa definicja kolumny w DB2, wykorzystująca wyrażenie*

```
-- Wyłączenie więzów integralności, aby możliwa była modyfikacja tabeli
SET INTEGRITY FOR Order_Details OFF;
-- Utworzenie kolumny wyliczeniowej z wykorzystaniem wyrażenia
```

```
ALTER TABLE Order_Details
    ADD COLUMN ExtendedPrice decimal(15,2)
        GENERATED ALWAYS AS (QuantityOrdered * QuotedPrice);
-- Powtórne włączenie więzów integralności
SET INTEGRITY FOR Order_Details
IMMEDIATE CHECKED FORCE GENERATED;
-- Założenie indeksu na kolumnie wyliczeniowej
CREATE INDEX Order_Details_ExtendedPrice
    ON Order_Details (ExtendedPrice);
```

Ponieważ wyrażenie jest teraz zawsze deterministyczne, możesz utworzyć kolumnę oraz indeks. Listing 1.7 pokazuje przykład dla DB2, ale w plikach z listingami zawarliśmy również przykłady dla innych baz danych.

Jeżeli będziesz chciał utworzyć w Oracle'u kolumnę wyliczeniową (nazywaną *kolumną wirtualną*), wykorzystaj `GENERATED [ALWAYS] AS`. SQL tworzący kolumnę `ExtendedPrice` (suma pozycji zamówienia) dla tabeli `Order_Details` w bazie Oracle mógłby wyglądać tak, jak na listingu 1.8.

Listing 1.8. *Przykładowa definicja tabeli z osadzonym wyrażeniem dla bazy Oracle*

```
CREATE TABLE Order_Details (
    OrderNumber int NOT NULL,
    OrderNumber int NOT NULL,
    ProductNumber int NOT NULL,
    QuotedPrice decimal(15,2) DEFAULT 0 NULL,
    QuantityOrdered smallint DEFAULT 0 NULL,
    ExtendedPrice decimal(15,2)
        GENERATED ALWAYS AS (QuotedPrice * QuantityOrdered)
);
```

W tym miejscu pewnie się zastanawiasz, dlaczego to zagadnienie zatytułowane zostało „Dlaczego przechowywanie danych wyliczeniowych zazwyczaj nie jest dobrym pomysłem", chociaż właśnie pokazaliśmy, jak to zrobić. Czas teraz na złe wieści: jeżeli ta tabela ma być intensywnie użytkowaną tabelą dla systemu czasu rzeczywistego, dodanie takiej kolumny może spowodować dodatkowe obciążenie serwera, co z kolei negatywnie przełoży się na czas odpowiedzi serwera.

Jeżeli korzystasz z IBM DB2, Microsoft SQL Server lub Oracle, być może będziesz mógł również utworzyć indeks dla kolumny wyliczeniowej, co pomoże podczas wykonywania zapytań wykorzystujących tę kolumnę. Pamiętaj, że w SQL Server dla przykładu z listingu 1.6 nie będziesz w stanie założyć indeksu (nie mógłbyś też w żadnym innym systemie bazodanowym), ponieważ wynik działania funkcji jest niedeterministyczny — uzależniony jest on od wyniku pobrania danych z innej tabeli (patrz również zagadnienie 17, „Kiedy wykorzystywać wartości wyliczane w indeksach").

W SQL Server musisz wykonać jeszcze jeden krok polegający na podaniu dla wyrażenia słowa kluczowego `PERSISTED`, natomiast w DB2 takie kolumny zapisywane są domyślnie po utworzeniu indeksu dla kolumny.

W przypadku listingu 1.7 dodatkowe operacje mają miejsce za każdym razem, gdy wartość wywoływanej funkcji mogłaby ulec zmianie — gdy aktualizujesz, wstawiasz lub usuwasz rekord z tabeli `Order_Details`. Ktoś siedzący przy terminalu i wpisujący wiele elementów zamówień może doświadczać nieakceptowalnych czasów reakcji systemu, ponieważ za każdym razem należy wykonać funkcję, a jej wynik musi zostać zapisany w indeksie. Na listingu 1.6 lub listingu 1.8 nadmiarowe operacje wykonywane byłyby podczas pobierania danych z tabeli i tutaj nieakceptowalny czas odpowiedzi może występować podczas wykonywania operacji `SELECT`, zawierającej kolumnę wyliczeniową i pobierającej wiele rekordów.

Do zapamiętania

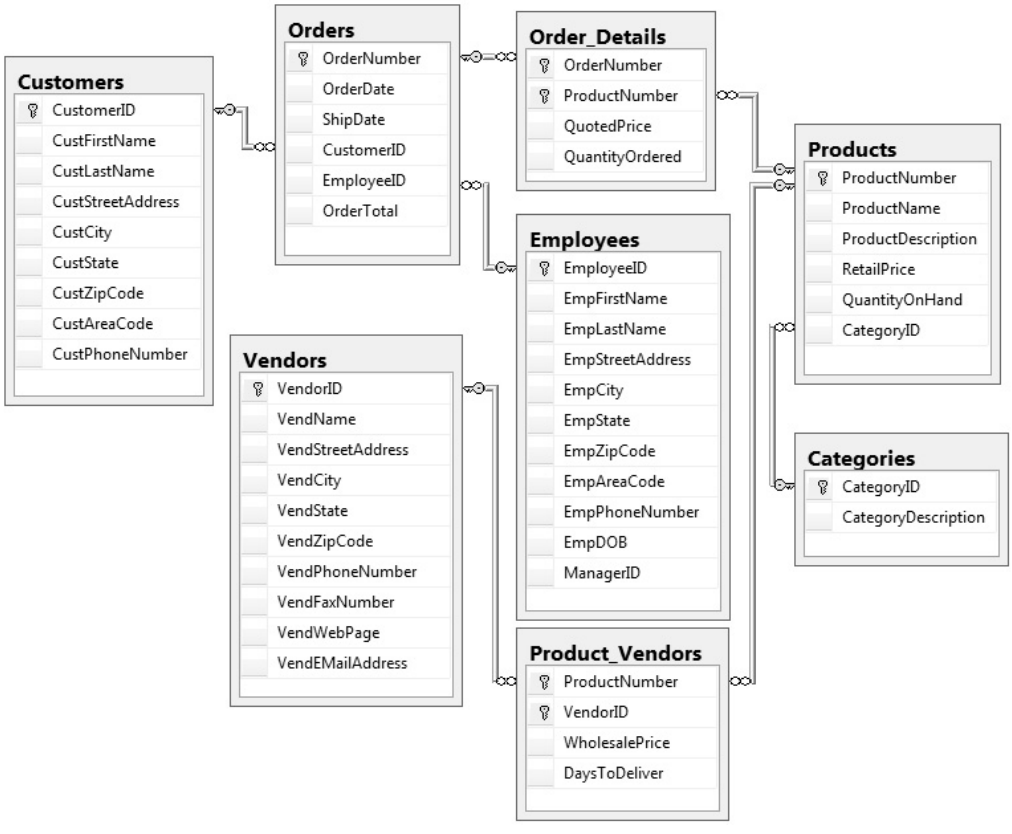
- Wiele systemów pozwala Ci definiować kolumny wyliczeniowe podczas definiowania tabel, ale musisz być świadomy implikacji wydajnościowych (szczególnie w przypadku wyrażeń i funkcji niedeterministycznych).
- Możesz również definiować kolumny wyliczeniowe jako standardowe kolumny, a następnie dbać o wartości za pomocą wyzwalaczy, ale potrzebny do tego kod może być skomplikowany.
- Kolumny wyliczeniowe powodują konieczność wykonywania dodatkowych operacji w systemie bazodanowym, używaj ich więc tylko wtedy, gdy korzyści przewyższają koszty.
- Zazwyczaj będziesz chciał tworzyć indeksy na kolumnach wyliczeniowych, aby zyskać pewne korzyści w zamian za zwiększenie wykorzystania przestrzeni dyskowej i wolniejszą aktualizację.
- Wykorzystanie widoków do definiowania obliczeń jest często dobrą alternatywą dla przechowywania obliczeń w tabeli tam, gdzie nie jest konieczne indeksowanie.

Zagadnienie 6: Definiowanie kluczy obcych do ochrony integralności referencyjnej

Podczas tworzenia poprawnego projektu schematu bazy danych w wielu tabelach tworzone są klucze obce zawierające klucze główne z tabel nadrzędnych. Na przykład tabela `Orders` (zamówienia) w bazie danych sprzedaży powinna zawierać kolumnę `CustomerId` (identyfikator klienta) lub `CustomerNumber` (numer klienta) wskazujący na klucz główny tabeli `Customers` (klienci), dzięki czemu możliwe jest zidentyfikowanie tego, kto złożył dane zamówienie.

Rysunek 1.8 przedstawia możliwy układ „typowej” bazy sprzedaży.

> **UWAGA**
> Rysunek 1.8 został utworzony przy wykorzystaniu narzędzia do tworzenia diagramów, które jest częścią pakietu Microsoft SQL Server Management Studio. Podobne narzędzia dostępne są w DB2, MySQL, Oracle i Microsoft Access, a także w specjalizowanych narzędziach do modelowania, takich jak Erwin oraz Idera ER/Studio.



Rysunek 1.8. *Projekt typowej bazy danych sprzedaży*

Diagram jasno pokazuje relacje pomiędzy poszczególnymi tabelami. Symbol klucza na końcu linii relacji wskazuje, że relacja jest od klucza głównego, a nieskończoność po drugiej stronie oznacza relację „jeden do wielu" dla klucza obcego w drugiej tabeli.

System bazodanowy zna relacje pomiędzy tabelami, ponieważ zadeklarowaliśmy więzy deklaratywnej integralności referencyjnej (DRI — ang. *declarative referential integrity*). Te definicje relacji mają dwa cele:

1. Podczas tworzenia nowego widoku lub procedury przechowywanej za pomocą graficznego edytora zapytań dla bazy edytor wie, jak poprawnie konstruować złączenia.
2. System bazodanowy wie, jak wspierać integralność danych podczas wstawiania lub zmiany w tabeli po stronie „wielu", lub podczas zmiany lub usuwania danych po stronie „jeden".

Drugi punkt jest najważniejszy, ponieważ musisz zapewnić, że na przykład nie mogą być tworzone wiersze zamówienia dla nieistniejących identyfikatorów klientów. Jeżeli możliwa jest zmiana identyfikatora klienta w tabeli klientów, powinieneś mieć pewność, że nowa wartość zostanie rozpropagowana (można

to wyspecyfikować za pomocą klauzuli ON UPDATE CASCADE) do wszystkich powiązanych rekordów zamówień. Jeżeli natomiast użytkownik spróbuje usunąć rekord klienta, który posiada powiązane rekordy w tabeli zamówień, powinieneś zadbać o to, aby albo usunięcie nie było dozwolone, albo wszystkie powiązane rekordy w tabeli zamówień również zostały usunięte (można to wyspecyfikować za pomocą klauzuli ON DELETE CASCADE).

Aby włączyć tę ważną funkcję w swojej bazie danych, musisz dodać więzy klucza obcego (FOREIGN KEY) —albo podczas tworzenia tabeli po stronie „wielu" przy wykorzystaniu CREATE TABLE, albo później za pomocą ALTER TABLE. Zobaczmy, jak to zrobić dla tabeli klientów (Customers) i zamówień (Orders).

Najpierw utwórzmy tabelę Customers — jej definicja znajduje się na listingu 1.9.

Listing 1.9. *Utworzenie tabeli Customers*

```
CREATE TABLE Customers (
    CustomerID int NOT NULL PRIMARY KEY,
    CustFirstName varchar(25) NULL,
    CustLastName varchar(25) NULL,
    CustStreetAddress varchar(50) NULL,
    CustCity varchar(30) NULL,
    CustState varchar(2) NULL,
    CustZipCode varchar(10) NULL,
    CustAreaCode smallint NULL DEFAULT 0,
    CustPhoneNumber varchar(8) NULL
);
```

Teraz utwórzmy tabelę Orders (zamówienia), a potem wykonajmy polecenie ALTER TABLE, które zdefiniuje relację. Potrzebne polecenia znajdują się na listingu 1.10.

Listing 1.10. *Utworzenie tabeli Orders i zdefiniowanie relacji*

```
CREATE TABLE Orders (
    OrderNumber int NOT NULL PRIMARY KEY,
    OrderDate date NULL,
    ShipDate date NULL,
    CustomerID int NOT NULL DEFAULT 0,
    EmployeeID int NULL DEFAULT 0,
    OrderTotal decimal(15,2) NULL DEFAULT 0
);
ALTER TABLE Orders
    ADD CONSTRAINT Orders_FK99
        FOREIGN KEY (CustomerID)
            REFERENCES Customers (CustomerID);
```

Zauważ, że jeżeli najpierw utworzysz dwie tabele, dodasz do nich dane, a następnie postanowisz dodać więzy klucza obcego, wówczas próba modyfikacji tabeli Orders może zakończyć się niepowodzeniem, jeżeli dane w tabelach nie przejdą sprawdzenia integralności. W niektórych systemach bazodanowych dodanie relacji może zakończyć się powodzeniem, ale będzie ona uznana za

niewiarygodną i nie będzie wykorzystywana przez optymalizator, dlatego samo jej zdefiniowanie nie gwarantuje jeszcze integralności wstawionych danych, gdy relacja jeszcze nie istniała.

Możesz również zdefiniować więzy podczas tworzenia tabeli podrzędnej, tak jak na listingu 1.11.

Listing 1.11. *Definiowanie klucza obcego podczas tworzenia tabeli*

```
CREATE TABLE Orders (
    OrderNumber int NOT NULL PRIMARY KEY,
    OrderDate date NULL,
    ShipDate date NULL,
    CustomerID int NOT NULL DEFAULT 0
        CONSTRAINT Orders_FK98 FOREIGN KEY
            REFERENCES Customers (CustomerID),
    EmployeeID int NULL DEFAULT 0,
    OrderTotal decimal(15,2) NULL DEFAULT 0
);
```

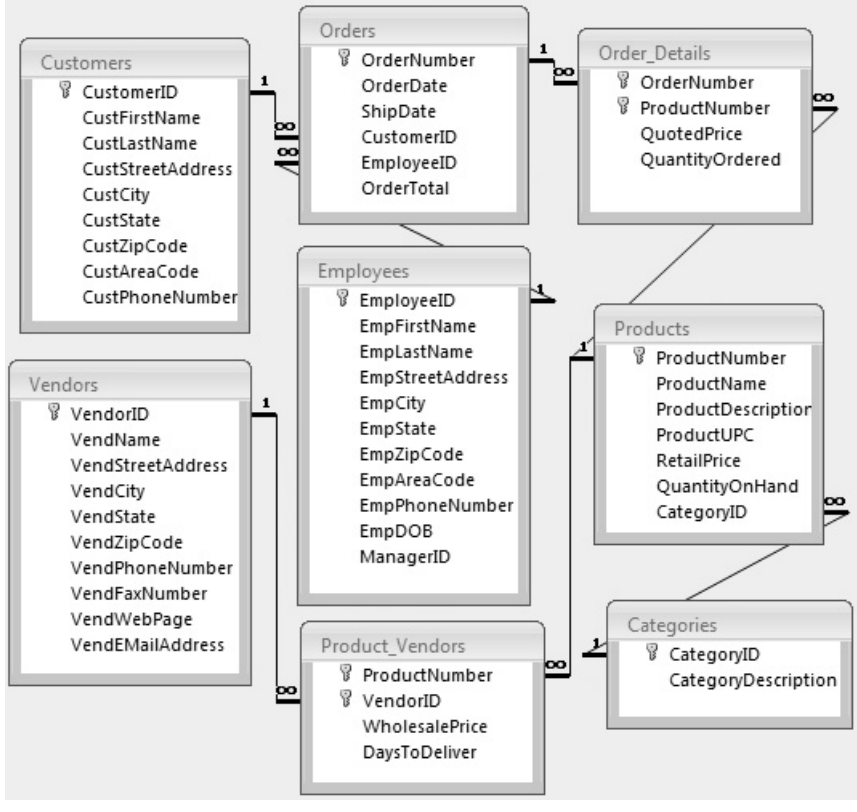
W niektórych systemach bazodanowych (np. Microsoft Access) zdefiniowanie więzów integralności referencyjnej automatycznie tworzy indeks na kolumnie (lub kolumnach) z kluczem obcym, więc podczas wykonywania złączeń wydajność może być lepsza. Dla tych baz, w których taki indeks nie jest tworzony automatycznie (np. DB2), dobrą praktyką jest jego utworzenie w celu optymalizacji sprawdzania więzów.

Do zapamiętania

- Jawne określenie klucza obcego pozwala zapewnić integralność danych pomiędzy połączonymi tabelami poprzez sprawdzenie, że nie istnieje żaden rekord w tabeli podrzędnej, dla którego nie istnieje odpowiadający rekord w tabeli nadrzędnej.
- Próba dodania więzów klucza obcego do tabel zawierających dane zakończy się niepowodzeniem, jeżeli istniejące dane pogwałcają dodawane więzy.
- W niektórych systemach wydajność złączeń może zostać poprawiona, ponieważ dodanie więzów klucza obcego automatycznie powoduje dodanie indeksu. W innych systemach powinieneś sam zadbać o utworzenie indeksu pokrywającego relację klucza obcego. Nawet bez indeksów optymalizatory niektórych systemów mogą traktować kolumny inaczej i tworzyć lepsze plany zapytań.

Zagadnienie 7: Upewnij się, że relacje między tabelami mają sens

Teoretycznie możesz utworzyć między dwiema tabelami taką relację, jaką tylko chcesz, o ile typy danych każdej pary połączonych kolumn będą takie same. Ale to, że *możesz* coś zrobić, nie oznacza jeszcze, że *powinieneś*. Rozważmy diagram schematu bazy zawierającej informacje o zamówieniach, przedstawiony na rysunku 1.9.



Rysunek 1.9. *Diagram schematu bazy danych zamówień*

Na pierwszy rzut oka wydaje się, że wszystko gra: jest kilka tabel, a każda z nich zawiera dane odrębnej encji. Skupmy się na trzech z nich: `Employees` (pracownicy), `Customers` (klienci) i `Vendors` (dostawcy). Jeżeli przyjrzysz się tym trzem tabelom, zobaczysz, że posiadają wiele podobnych pól. Często nie jest to uważane za problem, ponieważ dane w tabelach są zazwyczaj różne.

Jeżeli jednak ta firma miałaby dostawców lub pracowników, którzy są również klientami firmy, ten model pogwałcałby reguły dotyczące duplikacji danych omówione w zagadnieniu 2, „Eliminacja nadmiarowego przechowywania danych". Można próbować rozwiązać ten problem poprzez utworzenie jednej tabeli, nazwanej choćby `Contacts` (kontakty), w której wymienione byłyby wszystkie rodzaje kontaktów. Takie podejście nie jest jednak pozbawione wad.

Na przykład EmployeeID (identyfikator pracownika), CustomerID (identyfikator klienta) i VendorID (identyfikator dostawcy) pochodziłyby teraz z jednego klucza głównego, czyli ContactID (identyfikator kontaktu), który nie daje nam możliwości zweryfikowania, że dany identyfikator należy faktycznie do danego typu kontaktu.

Można rozwiązać ten problem, dodając tabele Customers, Vendors i Employees zawierające relacje „jeden do jednego" do tabeli Contacts. Zaletą tego podejścia jest to, że dane wyjątkowe dla encji, takie jak ManagerID (identyfikator menadżera) czy VendWebPage (strona internetowa dostawcy), pozwala trzymać oddzielnie od innych rekordów, które nie potrzebują tych informacji. Oznacza to również, że aplikacja wykorzystująca schemat będzie bardziej rozbudowana, ponieważ musi zawierać logikę sprawdzającą, czy encja istnieje, a jeżeli tak, czy posiada wypełnione wymagane dane. Dodatkowe pola nie miałyby sensu, gdyby aplikacja mogła bezmyślnie wstawiać dane bez sprawdzenia istnienia duplikatów. Co zrozumiałe, nie wszystkie firmy chcą wydawać więcej pieniędzy i poświęcać więcej czasu na dodatkowe utworzenie bardziej złożonego i rozbudowanego rozwiązania (zawierającego więcej tabel). Bardziej prawdopodobne jest, że firma sprzedająca produkty nie ma klientów, którzy są jednocześnie dostawcami lub pracownikami, więc sporadyczna duplikacja w tych rzadkich przypadkach jest niewielką ceną do zapłacenia za uproszczenie schematu bazy.

Rozważmy scenariusz, w którym musimy przypisać terytoria sprzedaży do pracowników i w konsekwencji mapować klientów do pracowników znajdujących się na tych terytoriach. Jednym ze sposobów może być utworzenie relacji pomiędzy kolumną CustZipCode (kod pocztowy klienta) w tabeli Customers (klienci) i kolumną EmpZipCode (kod pocztowy pracownika) w tabeli Employees (pracownicy). Obie kolumny mają ten sam typ danych i zawierają dane z tej samej domeny. Zamiast tworzyć relacje pomiędzy tabelami, mógłbyś wykonać złączenie na tych kolumnach, aby sprawdzić, którzy klienci mieszkają blisko których pracowników.

Możliwe jest utworzenie po prostu klucza obcego EmployeeID (identyfikator pracownika) w tabeli Customers (klienci) i połączenie w ten sposób klienta z pracownikiem. Powoduje to jednak nowe problemy. Po pierwsze, załóżmy, że klient przeprowadza się do innego terytorium sprzedażowego. Osoba wprowadzająca dane może poprawnie zmienić adres klienta, ale nie zdawać sobie sprawy lub nie pamiętać o tym, że konieczna jest również aktualizacja pracownika. Może to być źródłem nowych błędów.

Lepszym rozwiązaniem byłoby utworzenie tabeli SalesTerritory (terytorium sprzedażowe), posiadającej klucz obcy EmployeeID, a rekordy w tej tabeli identyfikowałyby kody pocztowe (TerrZIP) przypisane do danego pracownika. Każdy kod pocztowy byłby unikalny w ramach tabeli, ponieważ niepożądane by było przypisanie kodu pocztowego do więcej niż jednego pracownika. Teraz poprawne byłoby utworzenie relacji od kolumny TerrZIP do tabeli klientów, dzięki czemu pracownik mógłby sprawdzić, którzy klienci znajdują się w jego terytorium sprzedażowym.

Gdyby natomiast pracownicy byli przypisani do klientów za pomocą jakiegoś kryterium innego niż terytorium, posiadanie klucza obcego `EmployeeID` w tabeli klientów mogłoby być lepszym rozwiązaniem, ponieważ odzwierciedlałoby luźniejszy sposób przypisywania klientów do pracowników. Takie przypisanie będzie działało nawet wtedy, gdy terytorium jest domyślnym kryterium, ale klienci mogą poprosić o innego pracownika. Podobnie jak w poprzednim przykładzie, to podejście zakłada, że program będzie napisany w sposób minimalizujący błędy wynikające z wprowadzania danych.

Podobny problem występuje, gdy firma musi wylistować wszystkie produkty, jakie sprzedaje, wraz z ich dokładnymi danymi i wszystkimi właściwościami. Dla firmy sprzedającej drewno sensowne może być posiadanie kolumn przechowujących długość, wysokość, szerokość i rodzaj drewna — w końcu firma sprzedaje drewno. Jeżeli jednak firma jest sklepem detalicznym sprzedającym szeroki wachlarz towarów, dodawanie wielu kolumn, które będą rzadko wykorzystywane, nie wydaje się być dobrym pomysłem. Nie chcielibyśmy też tworzyć po jednej tabeli dla każdej kategorii produktu, w których przechowywalibyśmy dane specyficzne dla danej kategorii. W takiej sytuacji ktoś mógłby chcieć utworzyć tabelę `Attribute` (atrybuty) przechowującą dokumenty XML lub JSON. Może to być dobre rozwiązanie w przypadku braku reguł biznesowych wymagających możliwości udostępniania atrybutów produktu w bazie relacyjnej. Jeżeli jednak konieczne jest wykonywanie zapytań na atrybutach, cel pozwoliłoby osiągnąć utworzenie tabeli `ProductAttributes` (atrybuty produktu), a tym samym przekształcenie kolumn w rekordy i połączenie ich z tabelą `Products` (produkty)[1]. Listing 1.12 przedstawia możliwy projekt tabel.

Listing 1.12. *Tworzenie relacji pomiędzy tabelami Products i ProductAttributes*

```
CREATE TABLE Products (
    ProductNumber int NOT NULL PRIMARY KEY,
    ProdDescription varchar(255) NOT NULL
);
CREATE TABLE ProductAttributes (
    ProductNumber int NOT NULL,
    AttributeName varchar(255) NOT NULL,
    AttributeValue varchar(255) NOT NULL,
    CONSTRAINT PK_ProductAttributes
        PRIMARY KEY (ProductNumber, AttributeName)
);
ALTER TABLE ProductAttributes
    ADD CONSTRAINT FK_ProductAttributes_ProductNumber
        FOREIGN KEY (ProductNumber)
            REFERENCES Products (ProductNumber);
```

Chociaż może się wydawać, że poprzez przechowanie atrybutów jako rekordów rozwiązaliśmy problem, to zapytania do wyszukania konkretnych produktów z konkretnymi atrybutami staną się teraz znacznie bardziej skomplikowane, szczególnie jeżeli musimy pracować z wieloma atrybutami.

[1] Takie rozwiązanie jest często nazywane modelem „encja-atrybut-wartość": EAV (ang. *entity-attribute-value*)

Przez przypadek przykład z atrybutami ilustruje, dlaczego projektanci muszą odróżniać dane ustrukturyzowane od danych częściowo ustrukturyzowanych. W modelu relacyjnym dane muszą być od razu dobrze zdefiniowane, a wszystkie możliwe tabele i kolumny wyszczególnione, zanim będzie możliwe dodanie samych danych. Odwrotnie jest z danymi częściowo ustrukturyzowanymi takimi jak dokumenty XML lub JSON, gdzie dokumenty nie muszą mieć identycznego schematu, nawet na poziomie rekordu. Jeżeli więc się okaże, że masz problem ze zdefiniowaniem relacji, może warto zapytać, czy przypadkiem nie masz do czynienia z danymi częściowo ustrukturyzowanymi i czy rzeczywiście potrzebujesz, aby były dostępne bezpośrednio z modelu relacyjnego. Standardy SQL wspierają teraz wykorzystanie formatów XML i JSON bezpośrednio w SQL, co daje jeszcze więcej opcji, ale dyskusja na ten temat wykracza poza zakres tej książki.

Powyższe rozważania powinny udowodnić, że to biznes dyktuje, czy model jest poprawny, a Ty musisz zapewnić odpowiedni projekt. Jest to często trudne, ponieważ niekiedy łatwiej jest pozwolić, aby to aplikacja determinowała model danych, podczas gdy w rzeczywistości powinno być odwrotnie. W praktyce wybór jednego modelu kosztem innego powoduje znaczące zmiany w podejściu do projektowania aplikacji opartej na tym modelu. Te zmiany mogą mieć wpływ na czas i koszt dostarczenia produktu na rynek.

Do zapamiętania

- Przyjrzyj się dokładnie, czy połączenie utworzone w celu uproszczenia relacji tabel zawierających podobne kolumny jest faktycznie sensowne.
- Możesz utworzyć złączenie pomiędzy kolumnami w dwóch tabelach, o ile typy danych w tych kolumnach są zgodne (lub mogą być niejawnie zrzutowane), ale relacja jest poprawna tylko wtedy, gdy kolumny są z tej samej domeny. Optymalne jest posiadanie tego samego typu danych po obu stronach złączenia.
- Przed umieszczeniem danych w modelu danych sprawdź, czy faktycznie masz do czynienia z danymi ustrukturyzowanymi. Jeżeli dane są częściowo ustrukturyzowane, dostosuj swoje podejście.
- Zazwyczaj warto jasno zdefiniować cele modelu danych, aby ustalić, czy dany projekt usprawiedliwia dodatkową złożoność lub anomalie mogące powstawać ze względu na uproszczenie oraz projekt aplikacji korzystającej z danych.

Zagadnienie 8: Gdy 3NF to za mało, normalizuj dalej

Powszechnym mitem jest przekonanie, że w większości przypadków trzecia postać normalna jest zazwyczaj wystarczająca. Wielu praktyków słyszało i cytowało, że „trzecia postać normalna jest wystarczająca” albo „normalizuj do bólu, potem denormalizuj, aż zacznie działać”. Problem z tymi stwierdzeniami jest taki, że imputują one, iż wyższe postacie normalne wymagają więcej

modyfikacji. W rzeczywistości w większości modeli danych encja już w trzeciej postaci normalnej spełnia kryteria również wyższych postaci normalnych. W praktyce wiele współczesnych tabel referencyjnych jest w 5NF lub nawet 6NF, chociaż ludzie nazywają to 3NF. Szukać zatem trzeba tabel, które znajdują się w trzeciej postaci normalnej i nie spełniają wymagań postaci wyższych. Jest to niewielki zbiór, ale takie tabele się zdarzają, a jeśli się zdarzą, łatwo popełnić błędy projektowe powodujące anomalie nawet wtedy, gdy tabela wydaje się spełniać warunki trzeciej postaci normalnej.

Sygnałem ostrzegawczym, że projekt jest w 3NF, ale nie spełnia wymagań wyższych form normalnych, jest relacja tabeli z więcej niż jedną inną tabelą. Dzieje się tak, gdy tabela bierze udział w więcej niż jednej relacji „wiele do wielu". Innym symptomem jest sytuacja, gdy tabela zawiera klucze złożone, które mogą pogwałcać zasady wyższych postaci normalnych. Uważaj, jeśli używasz kluczy sztucznych, a analizujesz klucze naturalne, co zostanie przedstawione w późniejszych przykładach.

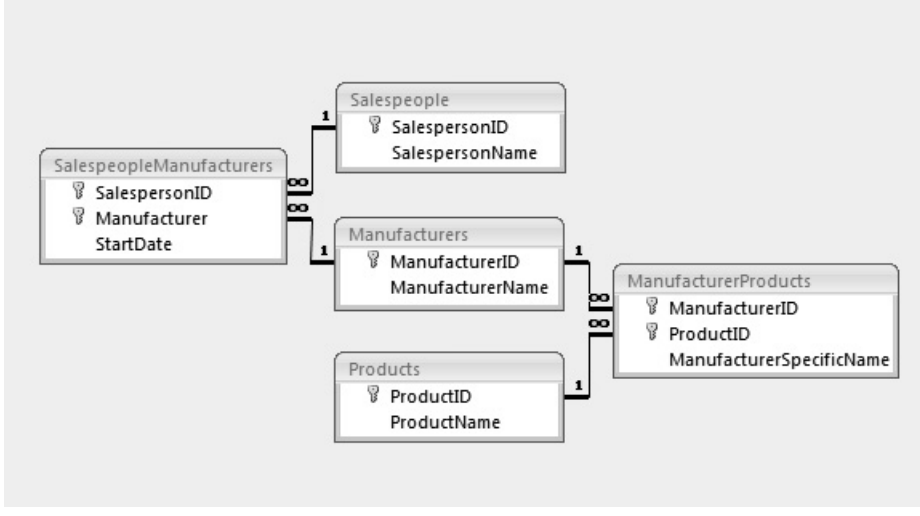
W ramach szybkiego przypomnienia: trzy pierwsze postacie normalne (jak również postać normalna Boyce'a-Codda) skupiają się na zależności funkcjonalnej pomiędzy atrybutami relacji. Przez zależność funkcjonalną rozumiemy, że atrybut jest zależny od klucza relacji. Na przykład kolumna przechowująca numer telefonu zawierający wartość `466 315 0072` jest funkcjonalnie zależna od kolumny przechowującej wartość `Douglas J. Steele`, czyli ten numer telefonu należy do niego, a inne atrybuty nie wpływają na tę zależność. Jeżeli numer telefonu zależny jest również od innych atrybutów, niebędących częścią klucza, model podatny jest na anomalie danych.

W przypadku czwartej postaci normalnej interesuje nas zależność wielowartościowa. Ma to miejsce wtedy, gdy dwa atrybuty nie są od siebie zależne, ale oba zależą od tego samego klucza relacji. Tworzy to wiele możliwych kombinacji pomiędzy tymi dwoma atrybutami. Istnieje przypadek specjalny, w którym możemy pogwałcić reguły czwartej postaci normalnej. Rozważmy produkty przedstawione w tabeli 1.3, które mogą zostać sprzedane przez sprzedawcę.

Tabela 1.3. *Produkty sprzedawane przez sprzedawców*

Sprzedawca	Producent	Produkt
Jay Ajurap	Acme	Slicer
Jay Ajurap	Acme	Dicer
Jay Ajurap	Ace	Dicer
Jay Ajurap	Ace	Whomper
Sheila Nyu	A-Z Inc.	Slicer
Sheila Nyu	A-Z Inc.	Whomper

Z tej tabeli nie wynika, że każdy wytwórca produkuje tylko dwa produkty, a sprzedawca, który sprzedaje produkty wytwórcy, musi sprzedawać wszystkie produkty przez niego wytwarzane. Zatem, jeżeli `Sheila Nyu` decyduje się na sprzedaż produktów `Ace`, musimy wstawić dwa rekordy — jeden dla produktu `Slicer` i drugi dla produktu `Whomper`. Nieprawidłowe zasilenie tabeli może prowadzić do anomalii. Aby uniknąć takiej ewentualności, powinniśmy rozbić tę tabelę na kilka tabel, tak jak na rysunku 1.10.



Rysunek 1.10. *Diagram schematu bazy danych produktów sprzedawanych*

Dzięki temu modelowi potrzebujemy tylko listy wszystkich produktów, które dowolny sprzedawca może sprzedać. Następnie mapujemy je na producentów, którzy faktycznie wytworzyli dany produkt. Listę produktów, które sprzedaje dany sprzedawca, otrzymujemy, łącząc tabelę `SalespeopleManufacturers` (producenci sprzedawców) z tabelą `ManufacturerProducts` (produkty producentów) — wynik będzie identyczny jak w tabeli 1.3. Należy zaznaczyć, że wspieramy regułę biznesową mówiącą, iż sprzedawca musi sprzedawać wszystkie produkty producenta. W rzeczywistości jednak bardziej prawdopodobne jest to, że sprzedawca sprzedaje tylko część produktów danego producenta. W takim przypadku dane z tabeli 1.3 nie pogwałcają czwartej postaci normalnej. To ilustruje, dlaczego wyższe postaci normalne są rzadkie; większość reguł biznesowych sprawia, że model od razu spełnia zasady wyższych postaci normalnych.

Piąta postać normalna wymaga, aby potencjalne klucze implikowały wszystkie zależności złączeń. Rozważmy nieznormalizowane dane z tabeli 1.4, zawierające listę gabinetów, sprzętu i lekarzy.

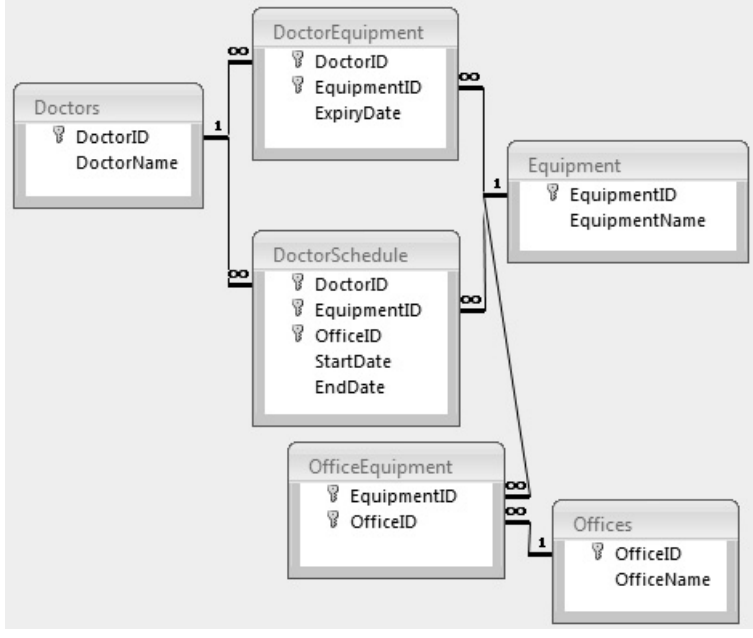
Tabela 1.4. *Tabela zawierająca wiele atrybutów w kolumnach*

Gabinet	Lekarz	Sprzęt
Southside	Salazar	X-Ray Machine
Southside	Salazar	CAT Scanner
Southside	Salazar	MRI Imaging
Eastside	Salazar	CAT Scanner
Eastside	Salazar	MRI Imaging
Northside	Salazar	X-Ray Machine
Southside	Chen	X-Ray Machine
Southside	Chen	CAT Scanner
Eastside	Chen	CAT Scanner
Northside	Chen	X-Ray Machine
Southside	Smith	MRI Imaging
Eastside	Smith	MRI Imaging

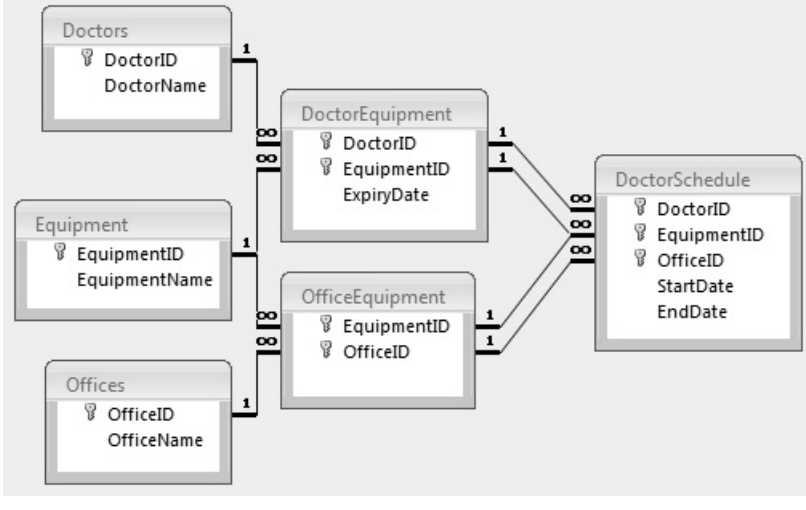
W tym modelu danych musimy przypisywać lekarzy do gabinetów, w których mogą wykonywać pracę na danym sprzęcie. Zakładamy, że lekarze mają kwalifikacje do pracy z tym sprzętem, dlatego nie ma sensu wysyłać ich do gabinetów niezawierających sprzętu, do którego obsługi posiadają kwalifikacje. Jednak nie wszyscy lekarze posiadają te same kwalifikacje; niektórzy mogą być relatywnie nowi albo mogą mieć inną specjalizację, więc nie wszyscy mają ten sam zestaw umiejętności.

Mamy więc lokalizacje gabinetów i mamy sprzęt. Te dwie encje krzyżują się, ale są raczej niezależne; gabinet z danym sprzętem nie ma żadnego związku z tym, czy lekarz potrafi z tego sprzętu korzystać. Próba utworzenia tego modelu danych przy wykorzystaniu sześciu tabel została przedstawiona na rysunku 1.11.

Zauważ, że są trzy tabele bazowe: `Doctors` (lekarze), `Equipment` (sprzęt) i `Offices` (biura) oraz tabela łącząca dla każdej z możliwych par `DoctorEquipment` (sprzęt lekarza) dla {`Doctors`, `Equipment`}, `OfficeEquipment` (sprzęt w gabinecie) dla {`Offices`, `Equipment`} i `DoctorSchedule` (harmonogram lekarza) dla {`Doctors`, `Offices`} wraz z referencją `Equipment` wynikającą z gabinetu. Jeżeli zatem otwarty zostanie nowy gabinet, do istniejącego gabinetu zostanie dodany nowy sprzęt lub lekarz zdobędzie nowe kwalifikacje, będziemy mogli wprowadzać zmiany niezależnie i unikniemy anomalii danych dla każdej z par. Istnieje natomiast ryzyko powstawania anomalii w tabeli `DoctorSchedule`. Możliwe jest utworzenie pary lekarza i gabinetu, w której albo lekarz nie posiada wyszkolenia do korzystania ze sprzętu, albo gabinet nie posiada odpowiedniego sprzętu. Jest to problematyczne i tym samym łamie zasady piątej postaci normalnej. Aby rozwiązać ten problem, musimy zmienić model danych zgodnie z rysunkiem 1.12.



Rysunek 1.11. *Diagram schematu dla bazy lekarzy, sprzętu i harmonogramowania gabinetów*



Rysunek 1.12. *Poprawiony schemat bazy lekarzy, sprzętu i harmonogramowania gabinetów*

Zauważ, że tabela `DoctorSchedule` (harmonogram lekarza) ma dwa klucze obce nakładające się na kolumnę `EquipmentID` (identyfikator sprzętu). Te dwa więzy klucza obcego zapewniają, że dla dowolnego sprzętu może zostać wybrana tylko poprawna kombinacja lekarza i gabinetu, możemy więc pominąć dbającą o to logikę aplikacji, a jednocześnie nie ma ryzyka wystąpienia anomalii danych.

Należy zauważyć, że gdybyśmy nie wymagali kolumny EquipmentID w tabeli DoctorSchedule, schemat z rysunku 1.11 byłby już w 5NF. Jeżeli więc chcielibyśmy tylko przypisać lekarza do gabinetu, nie określając, czy w ramach harmonogramu lekarz przypisany jest również do sprzętu, schemat z rysunku 1.11 byłby wystarczający.

Kolejną ważną rzeczą, na którą należy zwrócić uwagę, jest wykorzystanie kluczy złożonych. Gdybyśmy w tabelach łączących OfficeEquipment i DoctorTraining wykorzystali klucze sztuczne, pominęlibyśmy EquipmentID i wówczas model z rysunku 1.12 nie byłby możliwy. Jeżeli więc domyślnie wykorzystujesz klucze sztuczne, musisz się szczególnie uważnie przyglądać, czy przypadkiem nie powodują one ukrycia istotnych informacji. Zwracaj uwagę na wszystkie klucze obce w relacjach „wiele do wielu" i analizuj je pod kątem ich wpływu na relacje.

Dekompozycja bezstratna to metoda, którą możesz wykorzystać do analizy zgodności z wyższymi postaciami normalnymi. Niezależnie od tego, jak dużą masz tabelę, powinieneś rozłożyć część kolumn, jak gdybyś wykonał SELECT DISTINCT na podzbiorze, a następnie sprawdzić, czy wyniki mogą być połączone z powrotem z tabelą macierzystą przy wykorzystaniu LEFT OUTER JOIN i czy zwracają te same wyniki. Jeżeli rozłożone tabele po złączeniu nie stracą informacji, będziesz wiedzieć, że oryginalna tabela pogwałca jakieś zasady wyższych postaci normalnych i że wymaga dalszej analizy, czy nie spowoduje to wystąpienia anomalii danych. Tabela 1.5 przedstawia zdekomponowane dane z tabeli 1.3.

Tabela 1.5. *Dekompozycja tabeli 1.3*

Salesperson	Manufacturer
Jay Ajurap	Acme
Jay Ajurap	Ace
Sheila Nyu	A-Z Inc.

Manufacturer	Product
Acme	Slicer
Acme	Dicer
Ace	Dicer
Ace	Whomper
A-Z Inc.	Slicer
A-Z Inc.	Whomper

Jeżeli wrócisz do przykładów, które wykorzystaliśmy do przedstawienia łamania reguł 4NF i 5NF, zobaczysz, że gdybyśmy z tabeli 1.3 zabrali jeden rekord, złączenie pomiędzy tabelami SalespeopleManufacturers i ManufacturerProducts byłoby „stratne", ponieważ złączenie tabel z tabeli 1.5 nie zgadzałoby się ze zmodyfikowaną tabelą 1.3. W takim przypadku zmodyfikowana tabela 1.3

nie naruszałaby już reguł 4NF. Podobnie, gdyby w tabeli `DoctorSchedule` nie było kolumny `EquipmentID`, znów mielibyśmy straty, czyli reguły 5NF nie byłyby naruszone[2]. Zauważ, że analiza zakłada, iż w tabelach mamy na tyle danych, aby można było ustalić, czy dekompozycja będzie bezstratna.

Do zapamiętania

- Wyższe postacie normalne prawdopodobnie są już osiągnięte w większości modelów danych. Musisz zatem uważać na sytuacje, gdzie wyższe postacie normalne są jawnie pogwałcone. Jest to bardziej prawdopodobne w tabelach posiadających klucze złożone lub biorących udział w kilku relacjach „wiele do wielu".
- Czwarta postać normalna może być pogwałcona w przypadku, gdy wszystkie możliwe kombinacje dwóch niepowiązanych atrybutów encji muszą być wymienione dla tej encji.
- Piąta postać normalna ma zapewnić, że wszystkie złączane zależności są pochodnymi kluczy kandydujących. Oznacza to, że powinieneś być w stanie ograniczyć zbiór poprawnych wartości klucza kandydującego na podstawie indywidualnych elementów. Może to mieć miejsce tylko wtedy, gdy klucz jest złożony.
- Szósta postać normalna sprowadza się do redukcji relacji do posiadającej tylko jeden atrybut niebędący kluczem. Tym samym owocuje znacznym wzrostem liczby tabel, ale sprawia, że nigdy nie ma konieczności definiowania kolumny przyjmującej wartości `NULL`.
- Testowanie dekompozycji bezstratnej może być efektywnym narzędziem pozwalającym wykryć, czy tabele naruszają zasady wyższych postaci normalnych.

Zagadnienie 9: Wykorzystanie denormalizacji w magazynach danych

W środowisku twórców oprogramowania wciąż jest podkreślana waga normalizowania tabel. Tabele znormalizowane są zazwyczaj mniejsze i zajmują mniej miejsca niż tabele nieznormalizowane. Ponieważ dane są podzielone na wiele tabel, wydajność zazwyczaj jest wyższa, ponieważ tabele są na tyle małe, aby zmieścić się w buforze. Ponieważ dane są zlokalizowane w jednym miejscu, operacje aktualizacji i wstawiania są szybkie. Ponieważ dane nie są zduplikowane, zapotrzebowanie na ciężkie operacje `GROUP BY` lub `DISTINCT` w zapytaniach jest mniejsze.

[2] Jeżeli wszystkie pola schematu z rysunku 1.12 byłyby oznaczone jako nieprzyjmujące wartości `NULL`, schemat byłby w 6NF.

Te argumenty są poprawne, ponieważ aplikacje zazwyczaj intensywnie zapisują dane, a obciążenie bazy operacjami zapisu jest większe niż operacjami odczytu. W przypadku magazynów danych jest jednak inaczej: pomiędzy operacjami ładowania danych może nie być w ogóle operacji zapisu, ale nawet jeżeli będą, obciążenie tymi operacjami jest zazwyczaj znacznie mniejsze niż operacjami odczytu. Problem z w pełni znormalizowanymi danymi jest taki, że wymagają one złączeń pomiędzy tabelami. Im więcej złączeń, tym trudniej optymalizatorowi znaleźć najlepszy możliwy plan zapytania, co może mieć negatywny wpływ na wydajność odczytu.

Zdenormalizowane bazy danych dobrze radzą sobie z intensywnymi operacjami odczytu, ponieważ dane znajdują się w mniejszej liczbie tabel, a potrzeba złączania tabel jest mniejsza lub zupełnie nie istnieje. Pojedyncza tabela ze wszystkimi wymaganymi danymi pozwala również na znacznie wydajniejsze wykorzystanie indeksów. Jeżeli kolumny zostaną poprawnie zaindeksowane, wyniki mogą być szybko filtrowane i sortowane z wykorzystaniem tych indeksów bez konieczności odczytu danych bezpośrednio z tabeli. Ponieważ zapisy nie są częste, nie ma obawy, że zbyt wiele indeksów wpłynie negatywnie na szybkość zapisu. Możesz, jeżeli to konieczne, zaindeksować każdą kolumnę w tabeli, co drastycznie przyśpieszy operacje filtrowania i sortowania.

Aby efektywnie zdenormalizować dane, musisz dobrze rozumieć same dane oraz to, jak zazwyczaj będą one wykorzystywane.

Jednym z najprostszych typów denormalizacji jest replikacja pól identyfikatorów w tabelach w celu uniknięcia złączeń. Na przykład znormalizowana baza danych może mieć kolumnę `EmployeeID` (identyfikator pracownika) jako klucz obcy w tabeli `Customers` (klienci), tak aby możliwe było połączenie klientów z ich opiekunami. Jeżeli istnieje potrzeba raportowania faktur wraz z opiekunami konta, będziesz musiał złączyć trzy tabele: `Invoices` (faktury), `Customers` (klienci) i `Employees` (pracownicy). Możesz jednak osiągnąć ten sam cel, jeżeli zreplikujesz kolumnę `EmployeeID` w tabeli `Invoices`. Teraz musisz złączyć jedynie tabele `Invoices` i `Employees`. Oczywiście nie byłoby żadnego zysku z tej operacji, jeżeli potrzebowałbyś również danych z tabeli `Customers`.

Taką denormalizację można posunąć o krok dalej. Na przykład jeżeli wiesz, że w magazynie informacje o fakturach często będą wyszukiwane za pośrednictwem nazwy klienta, opłacalne może być przechowywanie w tabeli `Invoices` (faktury) nie tylko identyfikatora klienta (`CustomerID`), ale również jego nazwy, a także utworzenie dla niej indeksu. Tak, narusza to reguły normalizacji, ponieważ będziesz przechowywał informacje o wielu podmiotach (fakturach i klientach) w jednej tabeli, a nazwę klienta będziesz powtarzał w wielu rekordach. Należy jednak pamiętać, że głównym celem posiadania magazynów danych jest udostępnianie informacji w sposób łatwy i szybki. Uniknięcie złączenia w celu pobrania nazwy klienta pozwala zaoszczędzić wiele wartościowych zasobów.

Innym powszechnym podejściem jest dodanie kolumn pomocniczych do innych tabel. Może to nie tylko poprawić wydajność, ale również pomóc w zachowaniu historii. W pełni znormalizowany schemat zazwyczaj zawiera jedynie aktualny stan. W tabeli `Customers` (klienci) przechowywany jest tylko aktualny adres klienta. Jeżeli klient się przeprowadzi, adres zmieniany jest w tym jedynym rekordzie. Przez to wydrukowanie identycznej kopii faktury w późniejszym czasie może okazać się niemożliwe bez zachowania historii dotyczącej adresów klienta. Jeżeli jednak w tabeli `Invoices` (faktury) zachowasz kopię adresu klienta z momentu wystawienia faktury, wszystko stanie się prostsze.

Przechowywanie wartości wyliczonych lub pochodnych jest kolejną powszechną techniką denormalizacji. Zapisanie całkowitej sumy w tabeli `Invoices` (faktury) zamiast podliczania wszystkich poszczególnych rekordów z tabeli `InvoiceDetails` (szczegóły faktury) nie tylko redukuje liczbę tabel, które trzeba odpytać, ale również eliminuje konieczność wykonywania powtarzających się obliczeń. Kolejna zaleta przechowywania przetworzonych wartości uwidacznia się, gdy jest kilka możliwych sposobów na wykonanie obliczenia. Gdy wartość zostanie zapisana w tabeli, wszystkie zapytania zakończone zostaną tym samym wynikiem.

Kolejna opcja jest związana z wykorzystaniem powtarzających się grup. Jeżeli częstym wymogiem jest porównanie wydajności w kolejnych miesiącach, to zapisanie wszystkich 12 miesięcy w jednym rekordzie redukuje liczbę rekordów, które muszą zostać pobrane.

Pamiętaj, że wymagania co do tego, jak pobierane i przeglądane są dane w magazynie, są różne. Ralph Kimball, ekspert z dziedziny magazynów danych, opisuje trzy najważniejsze cechy magazynów danych jako drążenie w dół, drążenie w poprzek i czas obsługi[3]. Mówi o „tabelach faktów" jako o „fundamentalnej mierze przedsiębiorstwa" i „ostatecznym celu większości zapytań w magazynie danych", ale podkreśla również, że tylko w przypadku, gdy „zostały utworzone, aby odzwierciedlać ważne priorytety biznesowe, zostały dokładnie przetestowane i otoczone wymiarami dającymi wiele możliwości ograniczania i grupowania"[4].

Kimball opisuje trzy następujące typy tabel faktów:

1. Tabele faktów transakcji, odpowiadające miarom zebranym w danej chwili.
2. Tabele faktów z okresową migawką, podsumowujące aktywność w trakcie lub na końcu określonego okresu, na przykład okresu raportowania finansowego.
3. Tabele faktów akumulujące migawki, raportujące możliwe do przewidzenia procesy z dokładnie określonym początkiem i końcem, takie jak przetwarzanie zamówień, przetwarzanie żądań, rozwiązywanie zgłoszeń serwisowych i podania o przyjęcie na uniwersytet.

[3] *www.kimballgroup.com/2003/03/the-soul-of-the-data-warehouse-part-one-drilling-down/*

[4] *www.kimballgroup.com/2008/11/fact-tables/*

Kolejną ważną koncepcją wprowadzoną przez Kimballa jest ta o wolno zmieniających się wymiarach. Jak to ujął, większość najważniejszych miar przechowywanych w tabelach faktów zawiera stemple czasowe i klucze obce łączące je do wymiarów z datami, ale efektów czasu jest więcej niż tylko stemple czasowe oparte na aktywności. Wszystkie inne wymiary połączone z tabelą faktów, włączając w to tak fundamentalne jak klient, produkt, usługa, zasady, lokalizacja czy pracownik, również są naznaczone upływem czasu. Czasami zmiana opisu jest po prostu poprawą błędu, ale może również oznaczać prawdziwą zmianę w czasie opisu danego składnika miary, np. klienta czy produktu. Ponieważ te zmiany występują znacznie rzadziej niż w przypadku miar z tabel faktów, określane są mianem wolno zmieniających się miar (SCD — ang. *slowly changing dimensions*)[5]. Zrozumienie tych koncepcji jest kluczowe podczas tworzenia wydajnych i efektywnych magazynów danych.

Jeżeli postanowisz denormalizować dane, dokładnie udokumentuj tę denormalizację. Opisz szczegółowo logikę stojącą za denormalizacją oraz kroki, jakie podjąłeś. Jeżeli Twoja organizacja będzie chciała w przyszłości znormalizować dane, ta dokumentacja posłuży osobom, które będą musiały wykonać tę pracę.

Do zapamiętania

- Zdecyduj, jakie dane zduplikujesz i dlaczego.
- Zaplanuj, jak synchronizować dane.
- Zrefaktoryzuj zapytania, tak aby wykorzystywały zdenormalizowane pola.

[5] *www.kimballgroup.com/2008/08/slowly-changing-dimensions/*

Skorowidz

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Bazy danych umożliwiają bezpieczne przechowywanie i użytkowanie danych; dobrze napisane, pracują szybko i wydajnie. Najlepsze relacyjne bazy danych są nierozłącznie związane z SQL, zatem aby profesjonalnie do nich podejść, trzeba dobrze opanować ten język. SQL może wydawać się trudny i złożony, a co więcej, dla poszczególnych baz istnieją specyficzne dialekty SQL — tak czy owak, wspaniałe zalety najlepszych baz danych dostępne są tylko dla tych, którzy opanują sztukę pisania efektywnego kodu SQL.

Niniejsza książka jest świetnym kompendium przeznaczonym dla osób z podstawową znajomością SQL-a. Dzięki niej poznasz najlepsze współczesne techniki pisania w tym języku. Każdą z nich poparto realistycznymi (i przydatnymi!) przykładami. Innymi słowy, znajdziesz tu zrozumiale objaśnione sztuczki ekspertów i mnóstwo użytecznego kodu. Poza zagadnieniami składni omówiono tematykę optymalizacji projektu bazy, a także zarządzania hierarchiami i metadanymi. Wyjątkowość tej książki polega na tym, że zawarty w niej materiał bez trudu zastosujesz do baz: IBM DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle Database czy PostgreSQL.

Spośród 61 zagadnień ujętych w książce warto wspomnieć o:

- **zasadach projektowania modelu danych**
- **sposobach na efektywne wykorzystanie indeksów i więzów integralności**
- **metodach szybkiego wyszukiwania danych z zastosowaniem algebry relacyjnej**
- **stosowaniu podzapytań i złączeń**
- **tajnikach tabel kalkulacyjnych**
- **zbiorach zagnieżdżonych i domknięciach podległości**

John L. Viescas — zajmuje się bazami danych od ponad 45 lat. Rozwiązywał problemy baz Access i SQL Server w firmach każdej wielkości. W Applied Data Research prowadził zespół rozwijający systemy bazodanowe IBM dla komputerów mainframe.

Douglas J. Steele — od ponad 40 lat specjalizuje się w bazach danych i modelowaniu danych. Przez 17 lat otrzymywał tytuł MVP nadawany przez Microsoft.

Ben G. Clothier — MVP od 2009 roku, jest architektem rozwiązań w znanej firmie programistycznej IT Impact.

Wszyscy trzej są autorami i współautorami cenionych książek o bazach danych.

Przekonaj się, jak szybki i wydajny może być Twój kod SQL!

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-3563-9

Informatyka w najlepszym wydaniu

cena: 59,00 zł