Spis treści

Podziękowania

Wiele osób przyczyniło się do powstania tej książki, czy to bezpośrednio, czy pośrednio i wszystkim im należą się serdeczne podziękowania i uznanie. Mogło się naturalnie zdarzyć, że kogoś nieintencjonalnie pominąłem i z góry za to przepraszam.

Dla Lilach: Jesteś tym kimś, kto sprawia, że jestem dobry w tym, co robię. Oprócz tego, że jesteś nieustającą inspiracją, zawsze przyjmujesz aktywną rolę w powstawaniu moich książek, pomagając w pierwszej korekcie tekstu. W tej książce objęłaś bardziej oficjalną rolę redaktora technicznego i nie mogę wystarczająco docenić tego, jak wiele błędów znikło dzięki twojej spostrzegawczości. Dziękuję również za wiele pomysłów i sugestii ulepszeń.

Dla mojego rodzeństwa, Mickey'a i Iny: Dziękuję za ciągłe wsparcie i akceptowanie faktu, że jestem nieobecny.

Dla Davida Mauri, Herberta Alberta, Gianluca Hotza i Dejana Sarka: Dzięki za wasze cenne rady podczas planowania i pisania tej książki.

Dla zespołu redakcyjnego w wydawnictwie Pearson. Loretta Yates, dziękuję za profesjonalizm i pozytywne nastawienie! Dziękuję też Charvi Arora za ciężką pracę i wysiłek. Podziękowania należą się też Songlin Qiu, Scoutowi Festa, Karthikowi Orukaimani i Tracey Croom za sczytanie całego tekstu i upewnieniu się, że został wygładzony.

Dla moich przyjaciół z firmy Lucient, Fernando G. Guerrero, Herberta Alberta, Fritza Lechnitza i wielu innych. Pracowaliśmy wspólnie przez ponad dwie dekady i była to niezła jazda!

Dla członków zespołu programistycznego Microsoft SQL Server: Umachandara Jayachandrana (UC), Conora Cunninghama, Kevina Farlee, Craiga Freedmana, Kendala Van Dyke, Dereka Wilsona, Davida Mauri, Boba Warda, Bucka Woody'ego, a na pewno jeszcze wielu innych. Dziękuję za stworzenie tak wspaniałego produktu i za czas, jaki poświęciliście na spotkania ze mną i odpowiadanie na moje emaile, wyjaśnianie wątpliwości i reagowanie na prośby o ujednoznacznianie.

Dla Aarona Bertranda, który poza tym, że jest jednym z najaktywniejszych i wpływowych profesjonalistów SQL Server, jakich znam, wykonuje wspaniałą robotę, edytując zawartość portalu sqlperformance.com – w tym moje własne artykuły.

Dla Data Platform MVP, dawnych i obecnych: Erlanda Sommarskoga, Aarona Bertranda, Hugo Kornelisa, Paula White'a, Alejandro Mesa, Tibora Karaszi, Simona Sabina, Denisa Reznika, Tony'ego Rogersona i wielu innych, a nade wszystko dla ich

lidera, Rie Merritta. To wspaniała grupa, za którą jestem wdzięczny i dumny z tego, że mogę być jej częścią. Poziom wiedzy tej grupy jest zadziwiający i zawsze ogarnia mnie radość, gdy możemy się spotkać, zarówno by dzielić się pomysłami, jak i po prostu poznać się bliżej.

Na koniec, podziękowania kieruję do moich studentów: nauczanie SQL jest moją siłą napędową, moją pasją. Dziękuję, że mogę spełniać moje powołanie i dziękuję za wszystkie interesujące pytania, które pozwalają pogłębiać wiedzę.

Itzik Ben-Gan

O autorze

Itzik Ben-Gan jest czołowym autorytetem w zakresie języka T-SQL, stale piszącym, nauczającym i prowadzącym wykłady na ten temat. Jest autorem licznych szkoleń koncentrujących się na takich zagadnieniach, jak zapytania w języku T-SQL, dostrajanie zapytań i programowanie. Jest też autorem wielu książek, w tym *Podstawy języka T-SQL*, *Zapytania w języku T-SQL* oraz *Funkcje okna w języku T-SQL*. Od roku 1999 ma tytuł Microsoft Data Platform MVP (Most Valuable Professional).

Wprowadzenie

Książka ta pełni rolę przewodnika dla osób podejmujących pierwsze kroki w języku T-SQL (nazywanym także Transact-SQL), który jest opracowanym w firmie Microsoft dialektem języka SQL, zdefiniowanego przez standardy ISO/IEC. Poznamy teorię konstruowania zapytań i programowania w języku T-SQL oraz sposoby projektowania kodu T-SQL w celu uzyskiwania i modyfikowania danych, a także skrótowy przegląd obiektów programowalnych.

Pomimo że książka pomyślana jest dla Czytelników początkujących, nie jest jedynie zbiorem procedur, według których mają postępować – wykracza poza elementy składni T-SQL i wyjaśnia logikę działającą w tle języka i jego elementów.

Od czasu do czasu w książce pojawiają się zagadnienia, które mogą być uważane za tematykę zaawansowaną – z tego też względu zapoznawanie się z tymi fragmentami jest opcjonalne. Jeśli Czytelnik pewnie czuje się w omówionym do tej pory materiale, może przejść do tematów bardziej zaawansowanych; w przeciwnym razie spokojnie może opuścić te fragmenty i powrócić do nich, gdy już nabierze większego doświadczenia.

Wiele aspektów SQL jest unikatowych dla tego języka i znacznie odbiega od innych języków programowania. Książka ta ułatwi przyswojenie sobie właściwego sposobu myślenia i pozwoli dobrze poznać elementy języka. Czytelnik będzie mógł nauczyć się myśleć w kategoriach relacyjnych i postępować zgodnie z najlepszymi zaleceniami praktycznymi programowania w języku SQL.

Książka nie jest związana z konkretną wersją oprogramowania SQL Server; obejmuje jednak elementy języka, które zostały wprowadzone w ostatnich wersjach SQL Server, w tym SQL Server 2022. Przy omawianiu ostatnio wprowadzonych elementów języka wskazuję wersję produktu, w której dany element został dodany.

SQL Server, oprócz klasycznego rozwiązania instalowanego na lokalnym komputerze (serwerze), jest także dostępny w odmianach chmurowych o nazwach Azure SQL Database oraz Azure SQL Managed Instance. Przykłady kodu przytaczane w książce były testowane zarówno w lokalnych instalacjach SQL Server, jak i w chmurze.

W celu usprawnienia procesu nauczania książka zawiera ćwiczenia, które pozwalają poznane informacje utrwalić w praktyce. Nie muszę chyba podkreślać, jak ważne jest praktyczne poznanie jakiegoś zagadnienia – nie powinno się pomijać tych ćwiczeń, ale przeciwnie, wykonać je starannie, a być może poszukać również jakiegoś alternatywnego rozwiązania. W T-SQL, jak niemal zawsze w programowaniu, zazwyczaj istnieje więcej niż jedno poprawne rozwiązanie jakiegoś problemu!

Dla kogo przeznaczona jest ta książka

Niniejsza książka skierowana jest do programistów korzystających z języka T-SQL, administratorów baz danych, osób zajmujących się rozwiązaniami BI, autorów raportów, analityków, architektów baz danych i zaawansowanych użytkowników, którzy dopiero rozpoczynają pracę z SQL Server i muszą tworzyć kwerendy albo kod przy użyciu języka T-SQL.

Zgodnie z tytułem, książka przedstawia podstawy języka T-SQL. Przeznaczona jest głównie dla praktyków, którzy mają jedynie niewielkie doświadczenia w tym zakresie. Tym niemniej, wielu czytelników poprzedniego wydania tej książki uważa, że pomimo doświadczeńzdobytych w kolejnych latach pracy książka ta nadal jest przydatna i uzupełnia brakującą wiedzę.

Zakładam też znajomość podstawowych koncepcji zarządzania relacyjnymi bazami danych.

Organizacja książki

Książka rozpoczyna się od przedstawienia teoretycznych podstaw konstruowania zapytań i programowania w języku T-SQL (rozdział 1), co stanowi fundament dla pozostałej części książki, a także dla procesów tworzenia tabel i definiowania integralności danych. W rozdziałach 2 do 8 poruszane są różnorodne aspekty uzyskiwania i modyfikowania danych, a omówienie transakcji i współbieżności zostało zawarte w rozdziale 10. Rozdziały 9 i 11 obejmują specjalistyczne tematy, takie jak tabele temporalne i SQL Graph. Na koniec rozdział 12 stanowi skrótowy przegląd obiektów programowalnych. Poniżej przedstawiono listę rozdziałów wraz z krótkim ich opisem:

- Rozdział 1 „Podstawy zapytań i programowania T-SQL” – teoretyczne podstawy SQL, teoria zbiorów i logika predykatów; analizy modelu relacyjnego; opisy architektury SQL Server; wyjaśnienie sposobów tworzenia tabel i definiowania integralności danych.
- Rozdział 2 „Zapytania do pojedynczej tabeli” – różnorodne aspekty konstruowania zapytań dotyczących pojedynczej tabeli przy użyciu polecenia `SELECT`.
- Rozdział 3 „Złączenia” – opis zapytań dotyczących wielu tabel przy użyciu złączeń (`join`), w tym złączenia krzyżowe (*cross join* – iloczyn kartezjański), złączenia wewnętrzne i zewnętrzne.
- Rozdział 4 „Podzapytania” – omówienie zapytań zawartych wewnątrz innych zapytań, czyli *podzapytań*.
- Rozdział 5 „Wyrażenia tablicowe” – omówienie tabel pochodnych, wyrażeń CTE (Common Table Expression), widoków, wbudowanych funkcji zwracających tabele (iTVF) i operatora `APPLY`.

- Rozdział 6 „Operatory zbiorowe” – omówienie operatorów `UNION`, `INTERSECT` i `EXCEPT`.
- Rozdział 7 „Kod T-SQL dla analizowania danych” – omówienie funkcji okien, operatorów `PIVOT` i `UNPIVOT` oraz praca z operatorami `GROUPING SETS` i obsługa danych typu serii czasowych.
- Rozdział 8 „Modyfikowanie danych” – wstawianie, aktualizowanie, usuwanie i scalanie danych.
- Rozdział 9 „Tabele temporalne” – omówienie wersjonowanych przez system tabel temporalnych (czasowych).
- Rozdział 10 „Transakcje i współbieżność” – omówienie kwestii współdziałania połączeń użytkowników, którzy jednocześnie korzystają z tych danych; rozdział opisuje takie pojęcia, jak transakcje, blokady, poziomy izolacji czy zakleszczenia.
- Rozdział 11 „SQL Graph” omawia modelowanie danych za pomocą grafów, wykorzystujących takie koncepcje, jak węzły i krawędzie. Tematyka obejmuje tworzenie, modyfikowanie i odpytywanie danych bazujących na grafach.
- Rozdział 12 „Obiekty programowalne” zawiera skrótowe omówienie możliwości programowania przy użyciu T-SQL w SQL Server.
- W książce zamieszczono także dodatek „Rozpoczynamy”, który ułatwia skonfigurowanie środowiska, pobranie kodów źródłowych książki, zainstalowanie przykładowej bazy danych `TSQLV6`, rozpoczęcie pisania kodu dla SQL Server oraz poznanie sposobów uzyskania pomocy dzięki dokumentacji SQL Server.

Wymagania systemowe

Dodatek „Rozpoczynamy” zawiera informacje, których wersji produktu SQL Server 2022 można użyć do pracy z przykładowym kodem zamieszczonym w tej książce. Poszczególne wersje SQL Server mogą mieć różne wymagania systemowe i programowe; te wymagania są dokładnie opisane w dokumentacji produktu SQL Server w sekcji „Hardware and Software Requirements for Installing SQL Server 2022” pod adresem *https://learn.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server-2022*. W Dodatku wyjaśniono również, jak korzystać z dokumentacji produktu SQL Server.

Jeśli korzystamy z usługi Azure SQL Database albo Azure SQL Managed Instance, sprzęt i oprogramowanie jest utrzymywane przez firmę Microsoft, zatem wymagania te nie są istotne.

Jeśli chodzi o narzędzie klienckie dla uruchamiania przykładów kodu względem SQL Server, Azure SQL Database i Azure SQL Managed Instance, można używać albo SQL Server Management Studio (SSMS), albo Azure Data Studio (ADS). Narzędzie SSMS jest dostępne do pobrania pod adresem *https://learn.microsoft.com/en-us/sql/ssms*.

Program Azure Data Studio można pobrać ze strony *https://learn.microsoft.com/en-us/sql/azure-data-studio*.

Przykłady kodu

Większość rozdziałów książki zawiera ćwiczenia, które pozwalają interaktywnie wypróbować nowo poznany materiał zawarty w książce. Wszystkie przykłady kodu używane w książce, w tym ćwiczenia i rozwiązania dostępne są na poniższej stronie:

MicrosoftPressStore.com/TSQLFund4e/downloads

Dodatek „Rozpoczynamy” zawiera szczegółowe informacje na temat instalowania kodu źródłowego.

ROZDZIAŁ 1

Podstawy zapytań i programowania T-SQL

Zaczynamy podróż do krainy innej niż wszystkie – miejsca, które rządzi się swoim własnym zbiorem praw. Jeśli ta książka stanowi pierwszy krok na drodze do poznania języka T-SQL (Transact-SQL), powinniście się czuć tak jak Alicja przed rozpoczęciem jej przygód w Krainie Czarów. Dla mnie podróż ta nie ma końca, a po drodze stale odkrywam coś nowego. Zazdroszczę moim czytelnikom – niektóre z najbardziej ekscytujących odkryć wciąż są przed Wami!

Z językiem T-SQL związany jestem od wielu lat: nauczanie, seminaria, pisanie i konsultacje związane z tą tematyką. T-SQL to coś więcej niż tylko język programowania – to sposób myślenia. Często wykładałem i pisałem o (bardzo) zaawansowanych kwestiach, ale opis podstaw języka wciąż odkładałem na później. Nie dlatego, że fundamenty T-SQL są proste czy łatwe – wręcz przeciwnie: pozorna prostota języka jest bardzo myląca. Mógłbym skrótowo przedstawić elementy składni języka – to wystarczy, by w kilka minut zacząć pisanie zapytań. W praktyce jednak takie podejście utrudnia zrozumienie istoty języka i wydłuża proces jego poznawania.

Rola przewodnika osób podejmujących pierwsze kroki to duża odpowiedzialność. Chciałem mieć pewność, że poświęciłem wystarczająco dużo czasu i wysiłku na analizę i poznanie języka, zanim zabrałem się za opisywanie jego podstaw. Język T-SQL nie jest prosty; właściwe opanowanie podstaw to znacznie więcej, niż zrozumienie elementów składni i kodowania zapytań zwracających właściwe wyniki. W istocie trzeba wyrzucić z pamięci wszystko, co się wie o innych językach programowania i zacząć myśleć w kategoriach języka T-SQL.

Podstawy teoretyczne

SQL to akronim nazwy *Structured Query Language* (strukturalny język zapytań). Jest to standaryzowany język, opracowany do tworzenia zapytań i zarządzania danymi w systemach zarządzania relacyjnymi bazami danych (RDBMS). Akronim RDBMS oznacza system zarządzania bazą danych oparty na modelu relacyjnym (model semantyczny przedstawiania danych), który z kolei bazuje na dwóch działach matematyki: teorii zbiorów i logice predykatów. Wiele innych języków programowania i różnych

aspektów przetwarzania komputerowego powstało i rozwijało się w dużej mierze w oparciu o intuicję. Inaczej jest w przypadku SQL. W takim stopniu, w jakim SQL bazuje na modelu relacyjnym, wznosi się na solidnym fundamencie – matematyce stosowanej. Tak więc T-SQL opiera się na pewnych podstawach. Firma Microsoft udostępnia język T-SQL jako dialekt (lub rozszerzenie) języka SQL, wykorzystywany w Microsoft SQL Server – produkcie zarządzania relacyjnymi bazami danych (RDBMS), a także w Azure SQL i Azure Synapse Analytics – chmurowych systemach RDBMS.

Podrozdział ten zawiera krótki opis teoretycznych podstaw języka SQL, teorii zbiorów i logiki predykatów, modelu relacyjnego i typów systemów bazodanowych. Ponieważ książka ta nie jest ani podręcznikiem matematycznym, ani książką o modelowaniu danych i projektowaniu, przedstawione w niej informacje teoretyczne przedstawiam w swobodnej formie, a tym samym nie są one pełne. Celem jest opisanie kontekstu języka T-SQL i przedstawienie kluczowych punktów, które są niezbędne dla prawidłowego pojmowania działania języka T-SQL w dalszej części książki.

Niezależność języka

Model relacyjny jest niezależny od języka. Oznacza to, że można zaimplementować model relacyjny przy użyciu innego języka niż SQL, na przykład przy użyciu C# i modelu obiektowego. Obecnie typowym rozwiązaniem są systemy RDBMS obsługujące także inne języki programowania, niż tylko pewien dialekt SQL – na przykład integracja CLR, języków Java, Python i R w SQL Server – pozwalająca obsłużyć zadania historycznie realizowane w SQL przy użyciu innych języków programowania.

Dodatkowo już od początku trzeba zdawać sobie sprawę, że SQL pod wieloma względami odbiega od modelu relacyjnego. Niektórzy nawet twierdzą, że SQL powinien zostać zastąpiony nowym językiem (takim, który byłby bardziej zgodny z modelem relacyjnym). Na razie jednak to SQL jest de facto standardem, językiem używanym przez wszystkie wiodące systemy RDBMS.

Zobacz także Informacje szczegółowe na temat niezgodności SQL z modelem relacyjnym, a także na temat metod używania SQL zgodnie z tym modelem, znaleźć można w następującej książce: *SQL and Relational Theory: How to Write Accurate SQL Code*, Third Edition, C. J. Date (O'Reilly Media, 2015).

SQL

SQL to definiowany przez standardy ANSI i ISO, bazujący na modelu relacyjnym język programowania, opracowany pod kątem tworzenia zapytań i zarządzania danymi w systemach RDBMS.

Na początku lat siedemdziesiątych firma IBM opracowała język SEQUEL (Structured English QUEry Language) na potrzeby swojego systemu RDBMS nazwanego System R. Nazwa języka została później zmieniona na SQL ze względu na problemy dotyczące zastrzeżonego znaku towarowego. Język SQL stał się najpierw standardem ANSI (w roku 1986), a następnie standardem ISO (w roku 1987). Od roku 1986 instytucje ANSI (American National Standards Institute) i ISO (International Organization for Standardization) co kilka lat publikują kolejne wersje standardu SQL. Do tej pory opublikowane zostały następujące standardy: SQL-86 (1986), SQL-89 (1989), SQL-92 (1992), SQL:1999 (1999), SQL:2003 (2003), SQL:2006 (2006), SQL:2008 (2008) i SQL:2011 (2011). Standard SQL składa się z kilku rozdziałów. Część 1 (*Framework* – platforma) oraz Część 2 (*Foundation* – podstawy) dotyczą języka SQL, podczas gdy pozostałe części definiują rozszerzenia standardu, takie jak SQL for XML lub integracja SQL i Java.

Co interesujące, składnia języka SQL przypomina naturalny język angielski i jest również bardzo logiczna. Inaczej niż w przypadku wielu języków programowania, które stosują imperatywne wzorce programowania, SQL używa wzorców deklaratywnych. Oznacza to, że SQL wymaga określenia, *co* chcemy uzyskać, a nie *jak* ma to być wykonane, pozostawiając systemowi RDBMS wybór mechanizmów fizycznych niezbędnych do przetworzenia żądania użytkownika.

Język SQL obejmuje kilka kategorii instrukcji, w tym DDL (Data Definition Language – język definicji danych), DML (Data Manipulation Language – język manipulacji danymi) i DCL (Data Control Language – język sterowania danymi). Kategoria DDL dotyczy definicji obiektów i obejmuje takie polecenia, jak `CREATE`, `ALTER` czy `DROP`. Kategoria DML umożliwia tworzenie zapytańoraz modyfikowanie danych i obejmuje takie instrukcje, jak `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` i `MERGE`. Typowym nieporozumieniem jest opinia, że DML zawiera tylko polecenia służące do modyfikowania danych; jak już wspomniałem, zawiera także instrukcję `SELECT` (wybierz). Innym częstym nieporozumieniem jest traktowanie instrukcji `TRUNCATE` (obetnij) jako polecenia DDL, kiedy w rzeczywistości jest poleceniem z kategorii DML. Kategoria DCL dotyczy uprawnień i obejmuje takie polecenia, jak `GRANT` czy `REVOKE`. Niniejsza książka skupia się na poleceniach DML.

T-SQL opiera się na standardowym języku SQL, ale wprowadza także pewne własne, niestandardowe rozszerzenia. Co więcej, T-SQL nie implementuje wszystkich elementów standardu SQL. Przy opisie elementu języka zazwyczaj zaznaczam, czy jest to element standardu, czy też nie.

Teoria zbiorów

Teoria zbiorów, znana też pod nazwą teorii mnogości, której twórcą był matematyk Georg Cantor, to jeden z działów matematyki, na której bazuje model relacyjny. Definicja zbioru sformułowana przez Cantora jest następująca:

> *Zbiorem jest spojenie w całość określonych rozróżnialnych podmiotów naszej poglądowości czy myśli, które nazywamy elementami danego zbioru.*
>
> *– Wikipedia (https://pl.wikipedia.org/wiki/Georg_Cantor)*

Każde słowo tej definicji ma głębokie i kluczowe znaczenie. Definicje zbioru i członkostwa w zbiorze są aksjomatami, które nie podlegają dowodzeniu (są to *pojęcia pierwotne* teorii mnogości). Każdy element jest częścią wszechświata i należy lub nie należy do zbioru.

Rozpocznijmy od słowa *całość* w definicji Cantora. Zbiór powinien być traktowany jako pojedyncza jednostka, a nie kolekcja elementów, które go tworzą. Powinniśmy się skupić na zestawie obiektów, a nie na poszczególnych obiektach, tworzących zestaw. Później, kiedy będziemy pisać zapytania T-SQL odwołujące się do tabeli w bazie danych (jak na przykład tabela pracowników), powinniśmy myśleć o zbiorze pracowników jako o całości, a nie o poszczególnych pracownikach. Może się to wydawać oczywiste i bardzo proste, jednak wielu programistów ma trudności z przyswojeniem sobie takiego sposobu myślenia.

Słowo *rozróżnialny* oznacza, że każdy element zbioru musi być niepowtarzalny. Wybiegając w przód do pojęcia tabeli w bazie danych, możemy wymusić unikatowość wierszy w tabeli, definiując ograniczenia klucza. Bez klucza nie będziemy mogli w sposób jednoznaczny identyfikować wierszy i dlatego tabela nie będzie mogła być kwalifikowana jako zbiór, a będzie raczej *wielozbiorem* czy *pojemnikiem*.

Wyrażenie *naszej myśli czy postrzegania* sugeruje subiektywność definicji zbioru. Weźmy pod uwagę salę lekcyjną: jedna osoba może dostrzegać zbiór osób, a inna zbiór uczniów oraz zbiór nauczycieli. Z tego względu w definiowaniu zbiorów mamy pokaźną dawkę swobody. Gdy projektujemy model dla naszej bazy danych, proces ten powinien uważnie uwzględniać subiektywne potrzeby aplikacji, by określić właściwe definicje występujących tam jednostek.

Jeśli chodzi o słowo *obiekt*, definicja zbioru nie jest ograniczona tylko do bytów fizycznych, takich jak samochody czy pracownicy, ale odnosi się także do tworów abstrakcyjnych, takich jak linie czy liczby pierwsze.

To, czego brakuje w definicji Cantora, jest zapewne równie ważne jak to, co zawiera. Zwróćmy uwagę, że definicja ta nie wspomina o jakimkolwiek uporządkowaniu elementów zbioru. Kolejność elementów zbioru nie jest istotna. Używany w matematyce formalny zapis wymieniający elementy zbioru korzysta z nawiasów klamrowych: {a, b, c}. Ponieważ kolejność nie ma znaczenia, ten sam zbiór można przedstawić jako

{b, a, c} lub {b, c, a}. Wyprzedzając opis, w zestawie atrybutów (nazywanych w SQL *kolumnami*) tworzących nagłówek relacji (w SQL nazywany *tabelą*) zakłada się, że element (w tym przypadku *atrybut*) jest identyfikowany przez nazwę, a nie przez numer porządkowy.

Podobnie rozważmy zbiór krotek (w SQL nazywanych *wierszami*), który stanowi treść relacji; element (w tym przypadku krotka) jest identyfikowany przez jego wartości klucza, a nie na podstawie położenia. Wielu programistów ma trudności z przyswojeniem sobie tego sposobu myślenia, że z punktu widzenia zapytań dotyczących tabel nie istnieje żadna określona czy preferowana kolejność wierszy. Inaczej mówiąc, zapytanie odwołujące się do tabeli może zwrócić jej wiersze w dowolnej kolejności, chyba że jawnie zażądamy określonego posortowania wynikowych danych, na przykład w celu ich określonego zaprezentowania.

Logika predykatów

Logika predykatów, korzeniami sięgająca starożytnej Grecji, jest innym działem matematyki, na którym opiera się model relacyjny. Edgar F. Codd tworząc model relacyjny dostrzegł możliwość połączenia logiki predykatów zarówno z zarządzaniem danymi, jak i tworzeniem zapytań. Mówiąc niezbyt ściśle, *predykat* jest pewną właściwością lub wyrażeniem, które albo jest spełnione, albo nie – mówiąc inaczej, albo jest prawdą, albo fałszem. Model relacyjny polega na predykatach w celu utrzymywania logicznej integralności danych i definiowania struktury danych. Przykładem predykatu użytego do wymuszenia integralności jest ograniczenie zdefiniowane w tabeli nazwanej `Employees` (pracownicy), które zezwala na umieszczenie w tabeli jedynie takich danych pracowników, których pensja jest większa od zera. Predykatem jest tu wyrażenie „pensja większa niż 0” (wyrażenie T-SQL: `pensja > 0`).

Predykaty używane są również podczas filtrowania danych w celu zdefiniowania podzbiorów. Jeśli na przykład zachodzi potrzeba odpytania tabeli `Employees` i zwrócenia jedynie tych wierszy, które dotyczą pracowników działu sprzedaży, w naszym filtrze możemy użyć predykatu „działem jest (równa się) dział sprzedaży” (wyrażenie T-SQL: `dział = 'sprzedaż'`).

W teorii mnogości predykatów można używać do definiowania zbiorów. Metoda taka jest przydatna, ponieważ nie zawsze możemy zdefiniować zbiór poprzez wymienienie wszystkich jego elementów (przykładem mogą być zbiory nieskończone), a często wygodniej jest zdefiniować zbiór w oparciu o właściwość. Przykładem zbioru nieskończonego zdefiniowanego za pomocą predykatu może być zbiór wszystkich liczb pierwszych, który można zdefiniować przy użyciu następującego wyrażenia: „x jest dodatnią liczbą całkowitą większą niż 1, która podzielna jest tylko przez 1 i samą siebie”. Dla dowolnej wyspecyfikowanej wartości predykat jest albo prawdą, albo fałszem. Zbiorem wszystkich liczb pierwszych jest zbiór wszystkich elementów, dla których predykat jest prawdziwy. Przykładem skończonego zbioru zdefiniowanego

za pomocą predykatu może być zbiór `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`, definiowany jako zbiór wszystkich elementów, dla których następujący predykat ma wartość prawda: „x jest liczbą całkowitą równą lub większą niż 0 i równą lub mniejszą niż 9".

Model relacyjny

Model relacyjny jest modelem semantycznym zarządzania i modyfikowania danych, opartym na teorii zbiorów i logice predykatów. Jak już wspomniałem wcześniej, model ten został stworzony przez Edgara F. Codda, a następnie rozbudowany i opracowany przez Chrisa Date, Hugha Darwena i innych. Pierwsza wersja modelu relacyjnego została zaproponowana przez Codda w roku 1969 w postaci raportu badawczego firmy IBM pod tytułem „Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks". Poprawiona wersja przedstawiona została przez Codda w roku 1970 w dokumencie zatytułowanym „A Relational Model of Data for Large Shared Data Banks", opublikowanym w czasopiśmie *Communications of the* `ACM`.

Celem modelu relacyjnego jest udostępnienie spójnej reprezentacji danych przy minimalnej redundancji lub jej braku i bez utraty kompletności oraz zdefiniowanie integralności danych (wymuszanie spójności danych) jako części modelu. System RDBMS powinien implementować model relacyjny i dostarczać środków do realizowania zapytań oraz do przechowywania, zarządzania i wymuszania integralności danych. Model relacyjny opiera się na silnych podstawach matematycznych, a nie na intuicji, co oznacza, że mając pewną instancję modelu danych (na podstawie których zostanie później wygenerowana fizyczna baza danych) możemy dokładnie określić, czy projekt ten jest podatny na wady, nie polegając wyłącznie na intuicji.

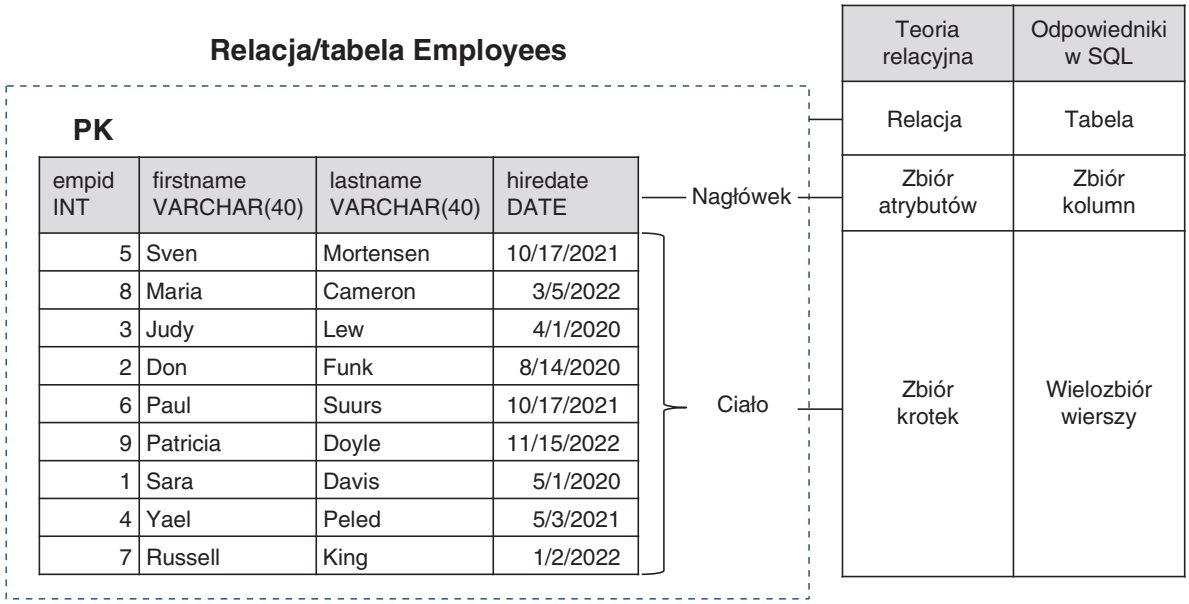
Model relacyjny korzysta z takich pojęć, jak tezy, predykaty, relacje, krotki, atrybuty i wiele innych. Dla osób nie zajmujących się matematyką pojęcia te mogą wydawać się początkowo przytłaczające, ale w kolejnych podrozdziałach przedstawię najważniejsze aspekty modelu w sposób przystępny i nie-matematyczny oraz wyjaśnię, jak pojęcia te odnoszą się do baz danych.

Tezy, predykaty i relacje

Dość powszechne przekonanie, że pojęcie *relacyjny* wywodzi się z zależności pomiędzy tabelami, nie jest poprawne. „Relacyjny" w rzeczywistości odnosi się do matematycznego pojęcia *relacji*. W teorii mnogości relacja jest przedstawieniem pewnego zbioru. W modelu relacyjnym relacją jest zbiór powiązanych informacji, którego odpowiednikiem w SQL jest tabela – aczkolwiek nie jest to dokładny odpowiednik. Kluczowy punkt modelu relacyjnego to fakt, że pojedyncza relacja powinna reprezentować pojedynczy zbiór (na przykład `Klienci`). Warto zwrócić uwagę na to, że operacje na relacjach (oparte na algebrze relacji) dają w wyniku relacje (na przykład przecięcie dwóch relacji). Jest to znane jako właściwość zamknięcia algebry relacyjnej ze względu na składanie operacji i jest tym, co umożliwia zagnieżdżanie wyrażeńrelacyjnych.

Uwaga Model relacyjny rozróżnia pojęcie *relacji* i *zmiennej relacji*, ale dla uproszczenia nie będę ich rozróżniać – w obu przypadkach będę używać pojęcia *relacji*. Dodatkowo, jak pokazuje rysunek 1-1, relacja zbudowana jest z nagłówka i treści. Nagłówek składa się ze zbioru atrybutów (nazywanych w SQL *kolumnami*), gdzie każdy element jest identyfikowany przez nazwę atrybutu i nazwę typu. Treść składa się ze zbioru krotek (w SQL nazywanych *wierszami*), gdzie każdy element jest identyfikowany przez klucz. Dla uproszczenia będę często odnosił się do tabeli jak do zbioru wierszy.

Rysunek 1-1 pokazuje przykład relacji o nazwie `Employees`. Porównuje on składniki relacji według teorii relacyjnej z elementami tabeli w SQL.



Rysunek 1-1 Przykładowa relacja Employees

Trzeba mieć na uwadze, że utworzenie prawdziwie adekwatnej reprezentacji wizualnej dowolnej relacji w praktyce jest bardzo trudne, gdyż zbiór atrybutów budujących nagłówek relacji nie ma uporządkowania i to samo dotyczy zbioru krotek tworzących treść (ciało) relacji. W powyższej ilustracji może się wydawać, że elementy mają jakiś porządek, choć go nie mają. Po prostu trzeba o tym pamiętać.

Podczas projektowania modelu danych dla bazy danych przedstawiamy wszystkie dane za pomocą relacji (tabel). Rozpoczynamy od zidentyfikowania tez (ang. *proposition*), które powinny być reprezentowane w bazie danych. Teza jest twierdzeniem, które może być prawdziwe lub fałszywe. Na przykład tezą jest stwierdzenie „Pracownik Jiru Ben-Gan urodził się 22 czerwca 2003 roku i pracuje w dziale żywności dla zwierząt". Jeśli ta teza jest prawdą, sama uwidoczni się w postaci wiersza tabeli `Employees` (pracownicy). Fałszywa teza po prostu się nie uwidoczni. Takie wstępne założenie nazywane jest *założeniem o świecie zamkniętym* – CWA (*Close World Assumption*).

Następnym krokiem jest sformalizowanie tez. W tym celu bierzemy rzeczywiste dane (treść relacji) i definiujemy strukturę (nagłówek relacji) – na przykład poprzez tworzenie predykatów. Możemy traktować predykaty jako sparametryzowane tezy. Nagłówek relacji zawiera zbiór atrybutów. Zwróćmy uwagę na użycie pojęcia „zbiór"; w modelu relacyjnym atrybuty są nieuporządkowane i unikatowe. Atrybut jest identyfikowany przez nazwę atrybutu i nazwę typu. Na przykład nagłówek relacji `Employees` może składać się z następujących atrybutów (wyrażonych jako pary – nazwa atrybutu i nazwa typu): `employeeid integer` (identyfikator pracownika, liczba całkowita), `firstname character string` (imię, ciąg znaków), `lastname character string` (nazwisko, ciąg znaków), `birthdate date` (data urodzin, data), `departmentid integer` (identyfikator działu, liczba całkowita).

Typ jest jednym z najbardziej fundamentalnych bloków konstrukcyjnych relacji. Typ ogranicza atrybut do pewnego zestawu możliwych lub poprawnych wartości. Na przykład typ `int` jest zbiorem wszystkich liczb całkowitych z zakresu od –2 147 483 648 do 2 147 483 647. Typ jest w bazie danych jedną z najprostszych form predykatu, ponieważ ogranicza dopuszczane wartości atrybutu. Na przykład baza danych nie będzie akceptować tezy, w której data urodzin pracownika to 31 luty 2003 (nie wspominając o dacie podanej jako coś w rodzaju „abc!"). Zwróćmy uwagę, że nie jesteśmy ograniczeni do typów podstawowych, takich jak liczby całkowite czy ciąg znaków. Typ może być również wyliczeniem możliwych wartości, jak na przykład lista możliwych stanowisk pracy. Typ może być bardzo złożoną konstrukcją. Dla osób, które znają inne języki programowania, prawdopodobnie najlepszym sposobem myślenia o typie jest przyrównanie go do klasy – kapsułkującej zarówno dane, jak i sposób ich obsługi. Przykładem typu złożonego jest typ `geometry`, który obsługuje wielokąty.

Brakujące wartości

Istnieje pewien aspekt modelu relacyjnego, który jest źródłem wielu emocjonujących dyskusji – czy należy jakoś obsługiwać brakujące wartości, stosując jakiegoś rodzaju logikę trójwartościową, czy też predykaty powinny być ograniczone do logiki dwuwartościowej. W przypadku dwuwartościowej logiki predykat może mieć tylko wartość prawda albo fałsz. Jeśli predykat nie jest prawdą, musi być fałszem. Używanie dwuwartościowej logiki predykatu jest zgodne z prawem matematycznym zwanym „zasadą wyłączonego środka". Niektórzy jednak twierdzą, że istnieje przestrzeń dla stosowania logiki trójwartościowej (czy nawet logiki o czterech wartościach), co pozwoliłoby uwzględniać przypadki, w których brakuje wartości. Predykat zastosowany do brakującej wartości nie generuje prawdy lub fałszu, ale wartość *nieznaną*.

Dla przykładu weźmy atrybut `mobilephone` relacji `Employees`. Załóżmy, że brak jest informacji o numerach telefonów niektórych pracowników. W jaki sposób przedstawić ten fakt w bazie danych? W przypadku zaimplementowania trójwartościowej logiki atrybut telefonu komórkowego powinien pozwalać na stosowanie specjalnego znaku

dla brakującej wartości. Następnie predykat porównując atrybut telefonu z pewnym określonym numerem wygeneruje wartość *nieznany* dla przypadków, kiedy brak tej wartości. Trójwartościowa logika predykatu generuje jedną z trzech możliwych wartości logicznych – *prawda*, *fałsz* i *nieznany*.

Niektóre osoby są przekonane, że `NULL`-*e* i trójwartościowa logika predykatu jest nie-relacyjna, podczas gdy inni są zupełnie odmiennego zdania. W rzeczywistości Codd był zwolennikiem czterowartościowej logiki predykatu, twierdząc, że istnieją dwa różne przypadki wartości brakujących: brakująca, ale mająca zastosowanie (*A-Mark*, od *applicable*) oraz brakująca, ale i bez zastosowania (*I-Mark*, od *inapplicable*). Przykładem wartości A-Mark jest sytuacja, kiedy pracownik ma telefon komórkowy, ale nie znamy numeru tego telefonu. Z wartością I-Mark będziemy mieli do czynienia w sytuacji, kiedy pracownik w ogóle nie posiada telefonu. Według Codda do obsługi tych dwóch przypadków wartości brakujących powinny być używane dwa znaczniki specjalne. Język SQL implementuje trójwartościową logikę predykatu, obsługując znacznik `NULL` dla oznaczania ogólnego pojęcia wartości brakującej. Obsługa znaczników `NULL` i trójwartościowej logiki predykatu w SQL jest źródłem różnych pomyłek i złożoności, chociaż można się upierać, że brakujące wartości to nieusuwalna cecha świata rzeczywistego. Ponadto podejście alternatywne, czyli stosowanie wyłącznie dwuwartościowej logiki predykatu i reprezentowanie brakujących wartości w jakiś własny, niestandardowy sposób wcale nie jest mniej problematyczne.

Uwaga Jak wspomniałem, *NULL* nie jest wartością, ale znacznikiem wskazującym brak wartości. Tym samym, choć nieszczęśliwie często spotykane, sformułowanie w rodzaju „wartość *NULL*" jest błędem logicznym. Właściwa terminologia to „znacznik *NULL*" lub po prostu samo „*NULL*". W tej książce posługuję się tym drugim wariantem, gdyż jest powszechnie używany w społeczności SQL.

Ograniczenia

Jedną z największych zalet modelu relacyjnego jest możliwość definiowania integralności danych jako części modelu. Integralność danych jest osiągana poprzez reguły nazywane *ograniczeniami* (*constraint*), które są definiowane w modelu danych i wymuszane przez system RDBMS. Najprostsza metoda wymuszania integralności danych to przypisanie wymaganego typu atrybutu oraz określenie możliwości obsługi znaczników `NULL` lub brakiem tej obsługi. Ograniczenia są wymuszane również poprzez sam model; na przykład relacja `Orders(orderid, orderdate, duedate, shipdate)` [Zamówienia(id, data, data płatności, data wysyłki)] dla jednego zamówienia dopuszcza trzy różne daty, podczas gdy relacje `Employees(empid)` [Pracownicy(id)] i `EmployeeChildren(empid, childname)` [DzieciPracownika(id pracownika, imię dziecka)] zezwalają na liczbę dzieci z zakresu od 0 do (przeliczalnej) nieskończoności.

Innym przykładem ograniczeń jest wymuszanie *kluczy kandydujących* (*candidate keys*), zwanych też *potencjalnymi*, które zapewniają integralność krotki* (*wiersza* w terminologii SQL), oraz *klucze obce* (*foreign keys*), które zapewniają integralność referencyjną. Klucz kandydujący to klucz zdefiniowany w oparciu o jeden lub kilka atrybutów, uniemożliwiający pojawianie się w relacji więcej niż jednej instancji tej samej krotki (wiersza). Predykat oparty na kluczu kandydującym jednoznacznie identyfikuje wiersz (na przykład pracownika). W relacji można definiować wiele kluczy kandydujących. Na przykład w relacji `Employees` możemy zdefiniować klucze kandydujące w oparciu o atrybut `employeeid` (id pracownika), `SSN` (numer ubezpieczenia) i inne. Zazwyczaj arbitralnie wybieramy jeden z kluczy kandydujących jako *klucz główny* (*primary key*), na przykład `employeeid` w relacji `Employees` i używamy tego klucza jako preferowanej metody identyfikowania wiersza. Wszystkie pozostałe klucze kandydujące nazywane są *kluczami alternatywnymi*.

Klucze obce są używane do wymuszania integralności referencyjnej. Klucz obcy definiowany jest w oparciu o jeden lub kilka atrybutów relacji (nazywanej *relacją referencyjną*, czyli odwołującą się) i odnosi się do klucza kandydującego w innej (lub niekiedy w tej samej) relacji. Ta więź ogranicza wartości mogące wystąpić w atrybutach klucza obcego relacji odwołującej się do wartości, które już występują w atrybutach klucza kandydującego relacji, do której następuje odwołanie. Załóżmy na przykład, że relacja `Employees` ma klucz obcy zdefiniowany w oparciu o atrybut `departmentid` (id działu), który odwołuje się do atrybutu `departmentid` klucza głównego w relacji `Departments`. Oznacza to, że wartości `Employees.departmentid` zostaną ograniczone do wartości, które występują w `Departments.departmentid`.

Normalizacja

Model relacyjny definiuje również *reguły normalizacji*, nazywane również *postaciami normalnymi* (*normal form* – NF). Normalizacja to formalny proces matematyczny, który gwarantuje, że każda jednostka będzie reprezentowana przez pojedynczą relację. W znormalizowanej bazie danych zapobiegamy powstawaniu nieprawidłowości i utrzymujemy redundancję na minimalnym poziomie bez utraty danych. Jeśli postępujemy zgodnie z modelowaniem ERM (Entity Relationship Modeling) i przedstawiamy każdą jednostkę i jej atrybuty, normalizacja prawdopodobnie nie będzie potrzebna; normalizację będziemy wtedy stosować jedynie do wzmocnienia poprawności działania modelu. Formalną definicję ERM można znaleźć w następującym artykule Wikipedii: *https://en.wikipedia.org/wiki/Entity–relationship_model*.

* Krotka (ang. *tuple*) – struktura danych będąca odzwierciedleniem matematycznej *n*-ki, tj. uporządkowanego ciągu wartości. Rekordy relacyjnych baz danych są krotkami, gdyż poszczególne pola rekordów mogą zawierać dane różnych typów. W bazach SQL rekordy są nazywane wierszami (ang. *row*), zaś zbiory krotek definiowane przez relacje – tabelami.

W kolejnych podpunktach skrótowo przedstawię trzy postaci normalne (1NF, 2NF i 3NF) wprowadzone przez Codda.

1NF Pierwsza postać normalna określa, że krotka (wiersz) w relacji (tabeli) musi być unikatowa, a atrybuty powinny być niepodzielne na mniejsze wartości (atomowość danych). Jest to nadmiarowa definicja relacji; inaczej mówiąc, jeśli tabela rzetelnie odzwierciedla relację, spełnia już pierwszą postać normalną.

Niepowtarzalność wierszy osiągamy definiując dla tabeli unikatowy klucz.

Na atrybutach można wykonywać tylko takie operacje, które są zdefiniowane dla typu atrybutu. Atomowość atrybutów jest cechą subiektywną, podobnie jak subiektywna jest definicja zbioru. Powstaje na przykład pytanie, czy nazwa pracownika w relacji `Employees` powinna być wyrażana za pomocą jednego atrybutu (nazwisko), dwóch atrybutów (imię i nazwisko) czy trzech atrybutów (imię, drugie imię i nazwisko)? Odpowiedź zależna jest od zastosowań. Jeśli aplikacja musi oddzielnie operować na częściach nazwy użytkownika (na przykład w celu wyszukiwania), sensownie jest je rozdzielać; w przeciwnym razie – nie.

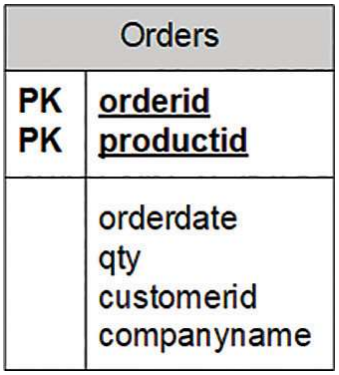
Podobnie jak atrybut może nie być wystarczająco „atomowy" z punktu widzenia potrzeb aplikacji, może również być „podatomowy". Jeśli na przykład dla danej aplikacji adresu jest traktowany jak niepodzielny, niedołączenie miasta jako części adresu będzie niespełnieniem warunku pierwszej postaci normalnej.

Ta postać normalna jest często rozumiana nieprawidłowo. Niektórzy uważają, że próba naśladowania tablic arkuszy kalkulacyjnych narusza pierwszą postać normalną. Przykładem może być zdefiniowanie relacji `YearlySales` (sprzedaż roczna) za pomocą następujących atrybutów: `salesperson` (sprzedawca), `qty2020` (ilość w 2020), `qty2021` i `qty2022`. W tym przykładzie jednak tak naprawdę nie naruszamy pierwszej postaci normalnej; po prostu nakładamy ograniczenie – ograniczamy dane do trzech określonych lat: 2020, 2021 i 2022.

2NF Druga postać normalna zakłada dwie reguły. Pierwsza reguła określa, że dane muszą spełniać wymagania pierwszej postaci normalnej. Druga reguła dotyczy zależności pomiędzy atrybutami klucza kandydującego a atrybutami, które nie są częścią klucza. Dla każdego klucza kandydującego, każdy atrybut niebędący częścią klucza musi być w pełni funkcjonalnie zależny od całego klucza kandydującego. Inaczej mówiąc, atrybut niebędący częścią klucza nie może być w pełni funkcjonalnie zależny od części klucza kandydującego. Aby to trochę uprościć, jeśli potrzebujemy uzyskać wartość pewnego atrybutu niebędącego częścią klucza, trzeba dostarczyć wartości wszystkich atrybutów klucza kandydującego tej samej krotki. Możemy wyszukać dowolną wartość dowolnego atrybutu dowolnej krotki, jeśli znamy wszystkie wartości atrybutów klucza kandydującego.

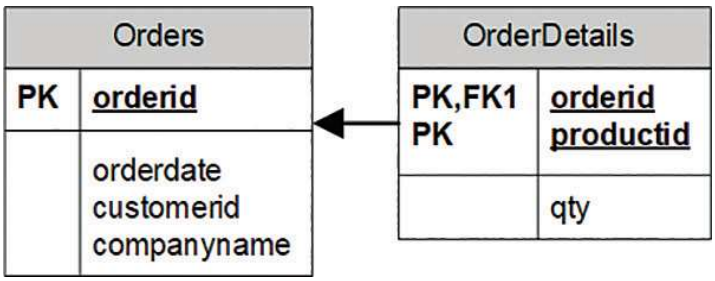
Dla przykładu naruszenia drugiej postaci normalnej załóżmy, że zdefiniowaliśmy relację nazwaną `Orders` (zamówienia), która reprezentuje informacje o zamówieniach i jego pozycjach (rysunek 1-2). Relacja `Orders` zawiera następujące atrybuty: `orderid`,

`productid`, `orderdate`, `qty`, `customerid` i `companyname` (identyfikator zamówienia, identyfikator produktu, data zamówienia, ilość, identyfikator klienta, nazwa firmy). Klucz główny zdefiniowany jest w oparciu o atrybuty `orderid` i `productid`.



Rysunek 1-2 Model danych przed zastosowaniem postaci 2NF

Relacja pokazana na rysunku 1-2 nie spełnia drugiej postaci normalnej, ponieważ istnieją atrybuty niebędące częścią klucza, które są zależne tylko od części klucza kandydującego (w tym przykładzie od klucza głównego). Na przykład, możemy znaleźć atrybut `orderdate` zamówienia, a także atrybuty `customerid` i `companyname`, bazując jedynie na samym atrybucie `orderid`. Aby spełniona była druga postać normalna, trzeba rozdzielić pierwotną relację na dwie relacje: `Orders` oraz `OrderDetails` (rysunek 1-3). Relacja `Orders` będzie zawierać atrybuty `orderid`, `orderdate`, `customerid` oraz `companyname`, z kluczem głównym zdefiniowanym za pomocą atrybutu `orderid`. Relacja `OrderDetails` będzie zawierać atrybuty `orderid`, `productid` oraz `qty`, wraz z kluczem głównym zdefiniowanym za pomocą atrybutów `orderid` i `productid`.

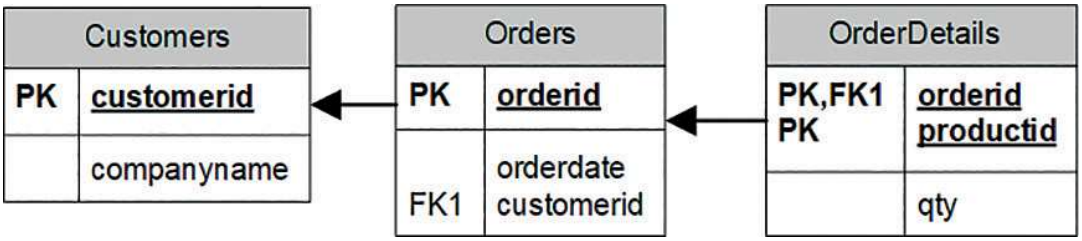


Rysunek 1-3 Model danych po zastosowaniu postaci 2NF, a przed zastosowaniem postaci 3NF

3NF Trzecia postać normalna również kieruje się dwoma regułami. Dane muszą spełniać drugą postać normalną. Ponadto wszystkie atrybuty niebędące częścią klucza muszą być zależne od kluczy kandydujących w sposób nieprzechodni. Upraszczając, reguła ta oznacza, że wszystkie atrybuty niebędące częścią klucza muszą być wzajemnie niezależne. Inaczej mówiąc, jeden atrybut niebędący częścią klucza nie może być zależny od innego takiego atrybutu.

Opisane poprzednio relacje `Orders` i `OrderDetails` obecnie spełniają drugą postać normalną. Pamiętamy, że w tym momencie relacja `Orders` zawiera atrybuty `orderid`,

`orderdate`, `customerid` i `companyname`, wraz z kluczem głównym zdefiniowanym w oparciu o atrybut `orderid`. Oba atrybuty `customerid` i `companyname` zależne są od całego klucza głównego – `orderid`. Na przykład, aby wyszukać atrybut `customerid` reprezentujący klienta, który złożył zamówienie, potrzebny jest cały klucz główny. Podobnie potrzebny jest cały klucz główny, by wyszukać nazwę firmy klienta, który złożył zamówienie. Atrybuty `customerid` i `companyname` są jednak również zależne od siebie samych. Aby spełnione były wymagania trzeciej postaci normalnej, trzeba dodać relację `Customers` (rysunek 1-4) wraz z atrybutami `customerid` (klucz główny) i `companyname`. Następnie możemy usunąć atrybut `companyname` z relacji `Orders`.



Rysunek 1-4 Model danych po zastosowaniu postaci 3NF

Nieformalnie postaci 2NF i 3NF są często podsumowywane następującym stwierdzeniem: „Każdy atrybut niebędący częścią klucza jest zależny od klucza, od całego klucza i od niczego poza kluczem – tak mi dopomóż Codd".

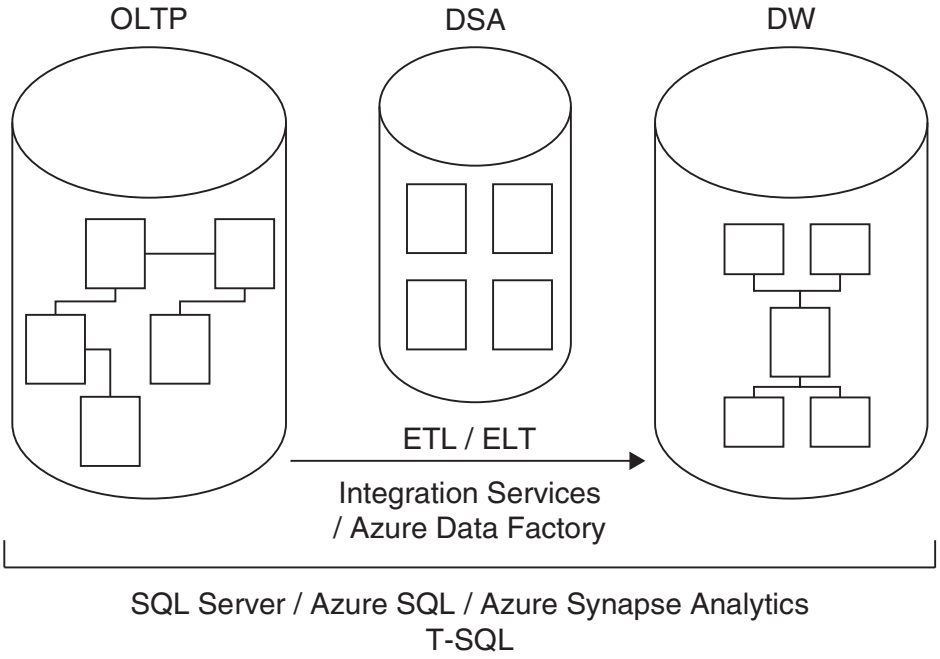
Istnieją wyższe postaci normalne, poza pierwszymi trzema opracowanymi przez Codda, które obejmują złożone klucze główne i tymczasowe bazy danych, ale tematyka ta wykracza poza zakres książki.

Uwaga Warto zauważyć, że SQL, a także T-SQL, pozwalają na naruszanie wszystkich form normalnych w rzeczywistych tabelach. To na twórcy modelu danych spoczywa odpowiedzialność za zaprojektowanie modelu znormalizowanego.

Typy obciążeń bazodanowych

Dwa główne typy obciążeń, które wykorzystują platformę RDBMS firmy Microsoft i język T-SQL do zarządzania i manipulowania danymi, to przetwarzanie transakcyjne w trybie online (*online transactional processing* – OLAP) oraz hurtownie danych (*data warehouse* – DW). Pierwsze mogą być implementowane w SQL Server albo Azure SQL. Te drugie można implementować w instalacjach SQL Server lub Azure SQL, które wykorzystują architekturę wieloprocesorową (*symmetric multiprocessing* – SMP). W przypadku bardziej wymagających obciążeń można je wdrażać w Synapse Azure Analytics, które wykorzystują architekturę masowo równoległą (*massively parallel processing* – MPP). Rysunek 1-5 ilustruje te systemy i procesy transformacji, które zwykle zachodzą pomiędzy nimi. Użyte akronimy mają następujące znaczenie.

- OLTP: Online Transactional Processing (przetwarzanie transakcyjne w trybie online)
- DSA: Data Staging Area (obszar przygotowywania danych)
- DW: Data Warehouse (hurtownia danych)
- ETL: Extract, Transform, Load (ekstrakcja, transformacja i ładowanie)



Rysunek 1-5 Klasy systemów bazodanowych

Online Transactional Processing

Dane są wprowadzane początkowo do systemu przetwarzania transakcji w trybie online. System OLTP skupia się na wprowadzaniu danych, a nie na raportowaniu – transakcje dotyczą głównie wstawiania, aktualizowania i usuwania danych. Model relacyjny jest ukierunkowany głównie na systemy OLTP, gdzie znormalizowany model zapewnia zarówno dobrą wydajność wprowadzania danych, jak i ich spójność. W znormalizowanym środowisku każda tabela reprezentuje pojedynczą jednostkę i minimalizuje redundancję. Jeśli zachodzi potrzeba zmodyfikowania faktu, trzeba go zmienić tylko w jednym miejscu. Dzięki temu optymalizowana jest wydajność modyfikowania danych i zmniejszane prawdopodobieństwo wystąpienia błędu.

Środowisko OLTP nie jest jednak odpowiednie do raportowania, ponieważ znormalizowany model zazwyczaj korzysta z wielu tabel (jedna dla każdej jednostki) o złożonych powiązaniach. Nawet prosty raport wymaga złączenia wielu tabel, co generuje złożone zapytania o niskiej wydajności.

Bazę danych OLTP można zaimplementować w SQL Server lub Azure SQL i zarówno zarządzać nimi, jak i wykonywać zapytania przy użyciu języka T-SQL.

Data Warehouse

Hurtownie danych (Data Warehouse) to środowisko opracowane do odczytywania danych i raportowania. Jeśli środowisko to służy całej organizacji, nazywane jest hurtownią danych; jeśli służy jedynie części organizacji (na przykład określonemu działowi) lub tematycznemu obszarowi w organizacji, jest nazywane *data mart* (*minihurtowania, magazyn danych*). Model danych hurtowni danych został tak opracowany i zoptymalizowany, by przede wszystkim zapewnić obsługę funkcji odczytywania danych. Model zawiera zamierzoną redundancję, mniejszą liczbę tabel i uproszczone powiązania, co powoduje, że w porównaniu do środowiska OLTP zapytania są prostsze i działają efektywniej.

Najprostszy projekt hurtowni danych nazywany jest *schematem gwiazdy* (*star schema*). Schemat gwiazdy obejmuje tabelę faktów i kilka tabel wymiarów. Każda tabela wymiarów reprezentuje podmiot, który ma być podstawą analizy danych. Na przykład w systemie obsługi zamówień i sprzedaży będziemy prawdopodobnie chcieli analizować dane według klientów, produktów, pracowników, czasu i temu podobnych wymiarów. W schemacie gwiazdy każdy wymiar jest implementowany w postaci pojedynczej tabeli z nadmiarowymi danymi. Na przykład wymiar produktu mógłby zostać zaimplementowany w postaci pojedynczej tabeli `ProductDim`, zamiast trzech znormalizowanych tabel: `Products`, `ProductSubCategories` i `ProductCategories` [Produkty, Podkategorie_produktu i Kategorie_produktu]. Jeśli normalizujemy tabelę wymiarów, co spowoduje powstanie wielu tabel reprezentujących ten wymiar, otrzymamy strukturę, która nazywana jest wymiarem *płatka śniegu* (*snowflake dimension*). Schemat zawierający wymiary płatka śniegu nazywany jest *schematem płatka śniegu* (w odróżnieniu do schematu gwiazdy).

W tabeli faktów przechowywane są fakty i miary, takie jak ilość czy wartość każdego istotnego połączenia kluczy wymiarów. Na przykład dla każdego istotnego połączenia klienta, produktu, pracownika i dnia tabela faktów może zawierać wiersz zawierający ilość i wartość. Zwróćmy uwagę, że dane w hurtowni danych są zazwyczaj wstępnie przetworzone (posumowane) do pewnego poziomu rozdrobnienia (na przykład z podziałem na dni), inaczej niż w przypadku danych w środowisku OLTP, które zwykle są rejestrowane na poziomie transakcji.

Historycznie rzecz biorąc, wczesne wersje SQL Server były głównie ukierunkowane na środowiska OLTP, jednak ostatecznie SQL Server zaczął również obsługiwać systemy hurtowni danych i zapewniać odpowiednią analizę. Możemy implementować hurtownię danych jako bazę danych SQL Server lub Azure SQL, które używają architektury SMP. Bardziej wymagające obciążenia (większe zbiory danych) można implementować w Azure Synapse Analytics, opartym na architekturze MPP. W obu przypadkach możemy zarządzać danymi i wykonywać zapytania za pomocą T-SQL.

Proces wyciągania danych z systemów źródłowych (OLTP i inne), opracowywania danych i ładowania ich do hurtowni danych nazywany jest procesem ETL (Extract,

Transform, Load). Niektóre z rozwiązań integracyjnych – szczególnie te oparte na chmurze – zmieniają kolejność: wykonują ekstrakcję danych, ich ładowanie, a następnie transformację. W takim przypadku proces określany jest akronimem ELT. Firma Microsoft udostępnia narzędzie o nazwie Microsoft SQL Server Integration Services (SSIS) dla zastosowań w siedzibie, które pozwala zaspokoić potrzeby ETL/ELT. Narzędzie to jest zawarte w licencji SQL Server. Microsoft udostępnia również bezserwerową usługę chmurową dla rozwiązań ETL/ELT o nazwie Azure Data Factory.

Proces ETL będzie często korzystał z warstwy danych DSA (Data Staging Area – DSA), znajdującej się pomiędzy OLTP a DW. Warstwa DSA umieszczona jest zazwyczaj w relacyjnej bazie danych, takiej jak SQL Server lub Azure SQL albo Azure Data Lake Storage Gen2 i jest używana jako obszar czyszczenia danych.

Architektura SQL Server

Niniejszy podrozdział jest wprowadzeniem do architektury SQL Server, rozwiązania RDBMS w odmianie w siedzibie (*on-premises*) i chmurowej, jego odmian i używanych jednostek – instancji SQL Server, baz danych, schematów i obiektów danych – oraz opisuje przeznaczenie wszystkich tych elementów.

Wersje RDBMS w siedzibie i chmurowe

Przez wiele lat firma Microsoft oferowała tylko jeden produkt RDBMS klasy przedsiębiorstwa – SQL Server w siedzibie. Obecnie jednak Microsoft oferuje szeroką gamę odmian, które nieustannie ewoluują. W ramach tej oferty dostępne są zarówno rozwiązania instalowane w siedzibie, jak i chmurowe.

On-premises

Odmiana systemu RDBMS w siedzibie nosi nazwę Microsoft SQL Server albo po prostu SQL Server. Jest to tradycyjny sposób używania produktu, zazwyczaj instalowanego w siedzibie klienta. Klient jest odpowiedzialny za wszystko – uzyskanie sprzętu, zainstalowanie oprogramowania i obsługę aktualizacji, zapewnienie wysokiego poziomu dostępności i przywracania po awariach (HADR), bezpieczeństwo i wszystko inne.

Klient może instalować wiele instancji produktu na tym samym serwerze (więcej na ten temat w kolejnym podrozdziale) i może pisać zapytania, które współdziałają z wieloma bazami danych. Istnieje również możliwość przełączania się pomiędzy bazami danych, chyba że jedna z nich jest typu *contained* (zawarta) – definicję tego terminu podam później.

Podstawowym językiem zapytań jest T-SQL. Wszystkie przykłady kodu i ćwiczenia przytoczone w tej książce możemy uruchamiać na implementacji SQL Server

w siedzibie. W Dodatku znaleźć można informacje szczegółowe na temat uzyskania i instalowania SQL Server, a także na temat tworzenia przykładowej bazy danych.

Cloud

Przetwarzanie w chmurze zapewnia zasoby obliczeniowe i magazynowe na żądanie z współużytkowanej puli zasobów. Technologie RDBMS firmy Microsoft mogą być udostępniane zarówno jako usługi chmury prywatnej, jak i publicznej. *Chmura prywatna* to infrastruktura chmurowa obsługująca pojedynczą organizację i zazwyczaj wykorzystuje technologię wirtualizacji. Zwykle jest hostowana w siedzibie klienta i utrzymywana przez personel IT organizacji. Można patrzeć na to jak na samoobsługową elastyczność, pozwalającą użytkownikom na wdrażanie zasobów w miarę potrzeb. Zapewnia też standaryzację i pomiary wykorzystania. Silnikiem bazodanowym jest zazwyczaj instalacja w siedzibie, w której język T-SQL jest używany do zarządzania i manipulowania danymi. SQL Server może działać zarówno w systemie Windows, jak i Linux, a tym samym może zostać wdrożony w dowolnej chmurze prywatnej, niezależnie od leżącej w tle platformie systemu operacyjnego.

W przypadku *chmury publicznej* usługi udostępniane są poprzez sieć i dostępne dla każdego (gotowego zapłacić). Firma Microsoft udostępnia dwie formy publicznych usług chmurowych RDBMS: infrastruktura jako usługa (IaaS) oraz platforma jako usługa (PaaS). W przypadku IaaS wynajmujemy maszynę wirtualną (VM) zlokalizowaną w infrastrukturze chmurowej firmy Microsoft. Oferta ta nosi nazwę SQL Server on Azure VM. Jako punkt wyjścia, możemy wybrać spośród wielu wstępnie skonfigurowanych maszyn wirtualnych, które już zawierają zainstalowaną określoną wersję i wydanie SQL Server. Sprzęt jest utrzymywany przez Microsoft, ale to użytkownik jest odpowiedzialny za obsługę oprogramowania i instalowanie poprawek. Zasadniczo nie różni się to od utrzymywania własnej instalacji SQL Server – tyle, że ta zlokalizowana jest na sprzęcie należącym do firmy Microsoft.

W przypadku PaaS firma Microsoft udostępnia platformę bazodanową jako usługę. Jest ona hostowana w centrach danych firmy Microsoft. Sprzęt, instalowanie i utrzymanie oprogramowania, zapewnienie wysokiej dostępności i odzyskiwania po awariach, a także instalowanie poprawek – wszystko to stanowi odpowiedzialność firmy Microsoft. Jednak to klient nadal odpowiada za indeksy i dostrajanie zapytań

Microsoft udostępnia kilka wariantów baz danych w wariancie PaaS. Na potrzeby systemów OLTP oraz hurtowni danych opartych na architekturze SMP oferowane są usługi Azure SQL Database oraz Azure SQL Managed Instance. Szczegółowe porównanie tych dwóch ofert PaaS można znaleźć w dokumencie dostępnym pod adresem *https://learn.microsoft.com/en-us/azure/ azure-sql/database/features-comparison*. Dowiemy się z niego na przykład, że przy użyciu tego pierwszego rozwiązania nie można realizować zapytań międzybazowych opartych na trzyczęściowych nazwach, ale jest to wykonalne w tym drugim. W ogólności druga wersja w działaniu jest bliższa rozwiązaniu zainstalowanemu w siedzibie.

Jak wspomniano wcześniej, Microsoft używa terminu Azure SQL jako zbiorczego określenia trzech ofert chmurowych w architekturze SMP: SQL Server on Azure VM, Azure SQL Database oraz Azure SQL Managed Instance.

Trzeba podkreślić, że Azure SQL Database oraz Azure SQL Managed Instance wykorzystują tę samą bazę kodu, co najnowsze wersje SQL Server. Tym samym niemal całe otoczenie języka T-SQL jest identyczne tak w środowisku w siedzibie, jak i chmurowym. Tak więc większość funkcjonalności języka T-SQL jest dostępna (ostatecznie) w obu środowiskach w taki sam sposób. Dlatego większość wiedzy o języku T-SQL, którą zawiera ta książka, ma zastosowanie w obydwu środowiskach. Różnice, które jednak istnieją – szczególnie te pomiędzy SQL Server a Azure SQL Database – zostały opisane w dokumencie dostępnym pod adresem *https://learn.microsoft.com/en-us/azure/azure-sql/database/ transact-sql-tsql-differences-sql-server*.

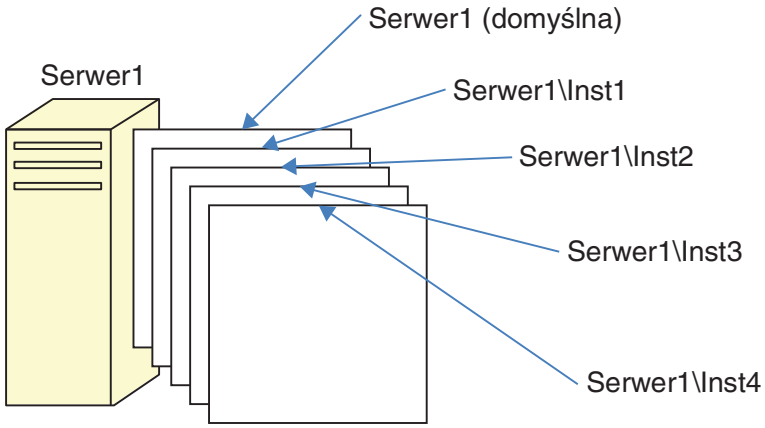
Warto też zauważyć, że tempo aktualizacji i wdrożeń nowych wersji chmurowych jest większe, niż w przypadku produktu SQL Server. Oznacza to, że niektóre nowe funkcje T-SQL mogą być dostępne w chmurze, zanim pojawią się w wersji SQL Server dla siedziby.

Jak napisano wcześniej, Microsoft udostępnia również ofertę typu PaaS dla systemów hurtowni danych o nazwie Azure Synapse Analytics jako chmurowe rozwiązanie wielkoskalowe z rozproszonym silnikiem przetwarzania, który można odpytywać i zarządzać nim za pomocą T-SQL.

Instancje produktu SQL Server

W przypadku odmiany w siedzibie *instancja* (wystąpienie) SQL Server jest pełną, odrębną instalacją silnika bazy danych lub usługi SQL Server, co ilustruje rysunek 1-6. Na tym samym komputerze (w tym samym systemie operacyjnym) możemy zainstalować wiele instancji produktu SQL Server w wersji dla siedziby. Każda instancja jest całkowicie niezależna od innych pod względem kontekstu zabezpieczeń, danych, którymi zarządza i w każdym innym aspekcie. Na poziomie logicznym dwie różne instancje umieszczone na tym samym komputerze nie mają więcej wspólnych elementów, niż dwie instancje umieszczone na różnych komputerach. Rzecz jasna, instancje umieszczone na tym samym komputerze współużytkują fizyczne zasoby serwera, takie jak CPU, pamięć czy dysk.

Jedną z wielu instancji na komputerze możemy skonfigurować jako *instancję domyślną*, natomiast pozostałe muszą być *instancjami nazwanymi*. To, czy instancja jest instancją domyślną, czy nazwaną, określane jest podczas instalacji; decyzji tej nie można później zmienić. W celu połączenia się z instancją domyślną aplikacja kliencka musi wskazać nazwę komputera lub jego adres IP. W celu połączenia się z instancją nazwaną program kliencki specyfikuje nazwę komputera lub adres IP, po których następuje odwrotny ukośnik (\) i nazwa instancji (określona podczas instalacji). Załóżmy dla przykładu, że mamy pięć instancji SQL Server zainstalowanych na komputerze *Serwer1*.



Rysunek 1-6 Wiele instancji SQL Server na tym samym komputerze

Jedna z tych instancji została zainstalowana jako domyślna, druga jako instancja nazwana – *Inst1*, trzecia jako *Inst2* i tak dalej W celu podłączenia się do instancji domyślnej trzeba podać jedynie *Serwer1* jako nazwę serwera, natomiast aby połączyć się z instancją nazwaną, trzeba użyć zarówno nazwy serwera, jak i nazwy instancji: *Serwer1\Inst1*.

Istnieją różne powody, dla których instalowanych jest wiele instancji SQL Server na tym samym komputerze, jednak wspomnę tylko o niektórych z nich. Jednym z nich, choć dziś głównie historycznym, jest chęć zmniejszenia kosztów obsługi. Na przykład w celu przetestowania działania funkcji w odpowiedzi na odebrane zgłoszenia lub odtworzenia napotkanych przez użytkowników błędów w środowisku produkcyjnym dla działu obsługi technicznej potrzebne są lokalne instalacje SQL Server, które imitują środowisko produkcyjne pod względem wersji, wydania i zainstalowanych pakietów serwisowych produktu SQL Server. Jeśli w organizacji jest wiele środowisk użytkowników, dział techniczny potrzebuje wielu instalacji SQL Server. Zamiast utrzymywania wielu komputerów, na których instalowane są różne instalacje SQL Server i które muszą być obsługiwane oddzielnie, można mieć jeden komputer z zainstalowanymi wieloma instancjami. Oczywiście obecnie ten sam efekt można uzyskać przy użyciu kontenerów albo maszyn wirtualnych. Po prostu instancje SQL Server były dostępne, zanim pojawiły się technologie wirtualizacji i kontenery.

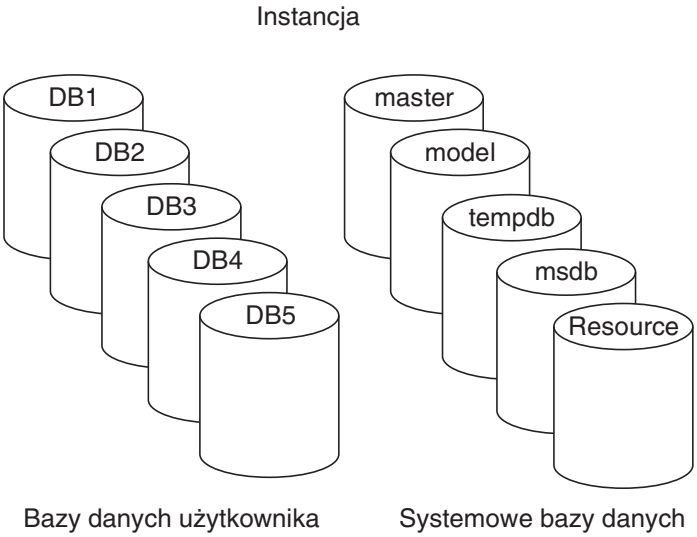
Innym przykładem są osoby, które mają zajęcia podobne do moich – uczą i prowadzą wykłady na temat SQL Server. Dla takich osób bardzo wygodna jest możliwość zainstalowania wielu instancji SQL Server na tym samym komputerze przenośnym. W ten sposób można na przykład prezentować różne wersje produktu, pokazując różnice w ich działaniu.

Jako ostatni przykład można przytoczyć sytuację dostawców usług bazodanowych, którzy zwykle potrzebują zagwarantować swoim klientom kompletne odseparowanie danych jednego klienta od danych innych klientów. Zamiast utrzymywania wielu mało wydajnych komputerów z zainstalowanymi odrębnymi instancjami, można utrzymywać wydajne centrum danych, w którym działa wiele instancji produktu SQL Server.

Obecnie rozwiązania chmurowe i zaawansowane technologie wirtualizacji również umożliwiają osiągnięcie tych samych celów.

Bazy danych

Bazę danych możemy traktować jak kontener zawierający takie obiekty, jak tabele, widoki, procedury składowane i inne. Każda instancja produktu SQL Server może zawierać wiele baz danych (rysunek 1-7). Podczas instalowania SQL Server program instalacyjny tworzy kilka systemowych baz danych przechowujących dane systemu, które mają wewnętrzne zastosowania. Po instalacji możemy tworzyć własne bazy danych użytkownika, w których przechowywane są dane aplikacji.



Rysunek 1-7 Przykład wielu baz danych w instancji SQL Server

Systemowe bazy danych utworzone przez program instalacyjny są następujące: *master*, *Resource*, *model*, *tempdb* oraz *msdb*.

- **master** Baza danych *master* przechowuje informacje metadanych dotyczące całej instancji, konfigurację serwera, informacje o wszystkich bazach danych instancji i informacje związane z procesem inicjalizacji.
- **model** Baza danych *model* jest używana jako szablon dla nowych baz danych. Każda nowa baza danych jest początkowo tworzona jako kopia bazy *model*. Tak więc, jeśli chcemy, by pewne obiekty (takie jak typy danych) występowały we wszystkich nowych bazach danych, które tworzymy, lub by pewne właściwości bazy danych były w określony sposób skonfigurowane we wszystkich nowych bazach danych, trzeba utworzyć te obiekty i skonfigurować te właściwości w bazie danych *model*. Zwróćmy uwagę, że zmiany zastosowane w bazie danych *model* nie wpłyną na istniejące bazy danych – tylko na nowe bazy danych, które utworzymy w przyszłości.